ABSTRACT

Title of dissertation: MULTI-STAGE SEARCH ARCHITECTURES FOR STREAMING DOCUMENTS

Nima Asadi, Doctor of Philosophy, 2013

Dissertation directed by: Associate Professor Jimmy Lin
Department of Computer Science and
College of Information Studies

The web is becoming more dynamic due to the increasing engagement and contribution of Internet users in the age of social media. A more dynamic web presents new challenges for web search—an important application of Information Retrieval (IR). A stream of new documents constantly flows into the web at a high rate, adding to the old content. In many cases, documents quickly lose their relevance. In these time-sensitive environments, finding relevant content in response to user queries requires a real-time search service; immediate availability of content for search and a fast ranking, which requires an optimized search architecture. These aspects of today's web are at odds with how academic IR researchers have traditionally viewed the web, as a collection of static documents. Moreover, search architectures have received little attention in the IR literature. Therefore, academic IR research, for the most part, does not provide a mechanism to efficiently handle a high-velocity stream of documents, nor does it facilitate real-time ranking.

This dissertation addresses the aforementioned shortcomings. We present an efficient mechanism to index a stream of documents, thereby enabling immediate availability of content. Our indexer works entirely in main memory and provides a mechanism to control inverted list contiguity, thereby enabling faster retrieval. Additionally, we consider document ranking with a machine-learned model, dubbed "Learning to Rank" (LTR), and introduce a novel multi-stage search architecture that enables fast retrieval and allows for more design flexibility. The stages of our architecture include candidate generation (top k retrieval), feature extraction, and document re-ranking. We compare this architecture with a traditional monolithic architecture where candidate generation and feature extraction occur together. As we lay out our architecture, we present optimizations to each stage to facilitate low-latency ranking. These optimizations include a fast approximate top k retrieval algorithm, document vectors for feature extraction, architecture-conscious implementations of tree ensembles for LTR using predication and vectorization, and algorithms to train tree-based LTR models that are fast to evaluate. We also study the efficiency-effectiveness tradeoffs of these techniques, and empirically evaluate our end-to-end architecture on microblog document collections. We show that our techniques improve efficiency without degrading quality.

Multi-Stage Search Architectures for Streaming Documents

by

Nima Asadi

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2013

Dedication

To Katherine, Sina, Christina, and my parents.

Acknowledgements

It has been an incredible journey with lots of ups and downs, and I am grateful to have experienced every moment, bitter or sweet, of the past five years. This dissertation seemed so far-fetched even a year ago, and if it were not for Jimmy, I would not be writing these words today. Jimmy has been a great advisor and I am very lucky to have been part of his team.

I am also forever indebted to Katherine who has helped me to be the person I am today. I have learnt and continue to learn a tremendous amount from her. I dedicate this work to her, for words cannot express my gratitude for all her love and support throughout the past years.

I am thankful to my parents and Sina for the sacrifices they have made and for their encouragement from the beginning, in return for nothing but this very moment. I owe my success to my family and dedicate this work to them as well.

I am fortunate to be surrounded by great friends, and I thank them for their invaluable friendship.

I would like to thank Donald Metzler for his mentorship in the first stages of my research. I have benefited from the breadth of his knowledge in the field of information retrieval. I also sincerely thank Don for his advice and help during my job search—I look forward to future collaborations.

Finally, thanks are due to Hal Daume III, Amol Deshpande, Alan Sussman, and Carol Espy-Wilson for serving on my dissertation committee.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The web is expanding and the volume of information it hosts is growing every second. As the amount of information grows, so does the need for search, so much that web search—an important application of Information Retrieval (IR)—has become an indispensable tool for many Internet users: users issue queries to a search service and, in return, expect a ranked list of documents that contain relevant information.

As the web evolves, so do the roles and demands of its users. For example, in recent years we have witnessed the growing popularity of social media among Internet users. The integration of social media in a variety of online services, together with the spread of smart phones, has changed the way we gossip, advertise, engage in political debates, and track public health. Social media and the online presence of news organizations have similarly revolutionized how news is created, distributed, and accessed. These changes, in turn, have brought about increasing engagement and contribution of users on the web, thereby making the web more dynamic.

A more dynamic web presents new challenges for search. A stream of new documents in various forms constantly flows into the web at a very high rate, adding to the old content. In many cases, documents quickly lose their relevance. In these time-sensitive environments, finding relevant content in response to user queries requires a real-time search service. This means that the search service must guarantee immediate availability of new content for search. In addition, the search service must offer a fast—yet effective—relevance assessment and ranking of documents, which requires an optimized search architecture. These aspects of today's web are at odds with how academic IR researchers have traditionally viewed the web, as a collection of static documents. Moreover, search architectures have received little attention in the IR literature. As a result, academic IR research, for the most part, does not provide a mechanism to efficiently handle a high-velocity stream of documents, nor does it facilitate real-time ranking.

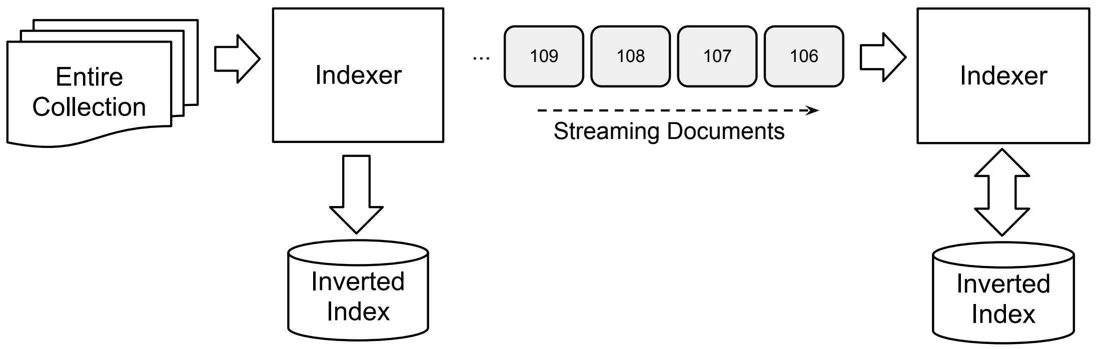
In order to resolve these shortcomings, we need to adapt existing search architectures to the new characteristics of the web. To this end, this dissertation addresses the issues of real-time indexing and explores various search architectures for document ranking. We present an efficient mechanism to index a stream of documents in real-time, thereby enabling immediate availability of content for search. Additionally, we introduce a novel multi-stage search architecture that enables efficient retrieval and offers more design flexibility. As we lay out our multi-stage architecture, we present optimizations in each stage that reduce latency. In the end, we explore efficiency-effectiveness tradeoffs in various architectures and empirically show the improvements that our architecture brings about. The experiments in this dissertation focus on search on microblog documents. However, in cases where our techniques are general and can be applied to web search, we also perform experiments on web collections. Also, in cases where existing microblog collections do not allow thorough experimentation, we experiment on standard collections other than microblog documents. The techniques and algorithms in this dissertation make up our contributions to the field of information retrieval.

1.2 Overview

1.2.1 Indexing Streaming Documents

To rank documents efficiently in response to a query, search services take advantage of a data structure called an inverted index—a structure that can quickly tell us which documents contain a particular term. In a search service, the task of constructing and maintaining an inverted index is performed by an indexer. The input to an indexer is a set of documents and the output is an inverted index.

In cases where documents remain unmodified for long periods of time, or where delays of hours in reflecting the changes made to documents is acceptable, the indexer can construct an inverted index using batch systems such as MapReduce [1]. With this approach, the system is



(a) Static collection of documents (b) Stream of documents

Figure 1.1: Different indexing scenarios.

unable to start indexing until it has obtained the entire collection. Figure 1.1(a) illustrates an indexer that reads documents from a collection as a batch and constructs an inverted index.

In many situations, however, it is more desirable to perform indexing *while* documents are being collected. For example, we can form a pipeline by connecting the output of a web crawler to the indexer, in order to index the crawled documents as other documents are being pulled from the web by the crawler. This behavior is essential in systems where documents are pushed to the search service (e.g., blog posts and tweets) and are expected to be indexed with little or no delay; in other words, an indexer cannot stall waiting for a complete batch of documents to begin indexing. In this scenario, as depicted in Figure 1.1(b), an indexer can model the documents it receives as an unbounded stream. We can then formulate indexing as an "update" problem: indexing a stream of documents is equivalent to updating an existing inverted index given one document at a time. This formulation is known as "incremental indexing."

Many incremental indexing algorithms have been proposed in the past [2, 3, 4, 5]. A majority of these techniques assume that the inverted index does not fit in the main memory and should be organized on the disk instead. Reading an inverted index from the disk, however, involves slow disk seeks. Thus, serving the inverted index from the disk can adversely impact document ranking latency [6], which operates on the inverted index. A low-latency ranking—which, as argued earlier, is a requirement of the web today—is easier to achieve by serving indexes from the main memory. Furthermore, in situations where indexing and query evaluation on the same index happen simultaneously, building an inverted index on disk makes low-latency ranking *during* indexing even more challenging. As such, existing incremental indexing algorithms do not conform to the new characteristics of today's web.

To address these challenges, we introduce an efficient indexing mechanism that operates entirely in main memory, thereby avoiding the disk altogether. Our indexer is capable of incrementally indexing thousands of web pages per second, which makes it suitable for real-time indexing of a high-velocity stream of documents.

1.2.2 Multi-Stage Ranking Architectures

The second component in search is a document ranking algorithm, which operates on the inverted index and computes the relevance of documents in a collection with respect to a user query. These algorithms range from simple information retrieval metrics (such as tf-idf, BM25 [7], and Dirichlet scores from language modeling), to complex machine-learned models (such as gradient boosted regression trees [8, 9], additive ensembles [10], and cascades of rankers [11, 12]), known as learning-to-rank, that generally lead to higher search quality. However, learning-to-rank models often require the computation of more sophisticated features such as phrase features (i.e., co-occurrence of query terms as a phrase). Because these features generally require term positions and therefore are costly to compute, it is not feasible to compute them for every document and apply a ranking algorithm to a document collection in its entirety.

Most learning-to-rank work leave aside these cost issues and focus exclusively on effectiveness. In practice, however, to deal with cost issues discussed above, query evaluation is divided into two stages [13, 11, 14]. The first stage, which we refer to as "candidate generation," uses a fast, simple algorithm to generate a subset of documents (henceforth, referred to as "candidates") that exhibit stronger relevance signals, which are measured by a relevance scoring function. The relevance scoring function used in the first stage should be computationally light but also effective at filtering out non-relevant documents; there are various query-dependent functions (e.g., BM25

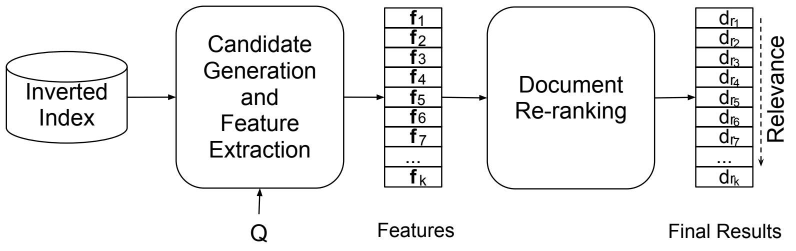


Figure 1.2: A monolithic architecture where candidate generation and feature extraction occur at the same time.

and tf-idf) as well as query-independent scores (e.g., PageRank [15], HITS [16], SALSA [17], and page quality score) that can be used for these purposes. The second stage, which we refer to as "document re-ranking," then re-ranks the candidates generated by the first stage using a machine-learned model; the model computes a relevance score for every candidate, and subsequently sorts the candidates in order—based on the computed scores—in order to form the final ranked list.

A machine-learned model needs vectors of features as input, where each vector describes a candidate document. So before candidates from the first stage can be re-ranked using the machine-learned model, we need to compute feature values for them. Feature value computation, however, is an under-explored problem. The learning-to-rank literature focuses on developing machine learning models for higher effectiveness and assumes an architecture that performs candidate generation and feature extraction in one stage, as illustrated in Figure 1.2. As such, the learning-to-rank literature is not clear on how, when, and using what data structures we can compute features more efficiently.

We address this gap by treating feature extraction as a distinct task and delegating this task to a third stage, resulting in our novel three-stage architecture as depicted in Figure 1.3. As such, the "feature extraction" stage receives a list of candidates from the first stage, candidate generation, and computes feature values for the candidates with respect to the input query. The feature extraction stage then emits a list of feature vectors—one per candidate document—to the last stage, document re-ranking. These three stages make up what we refer to as a learning-to-rank pipeline.

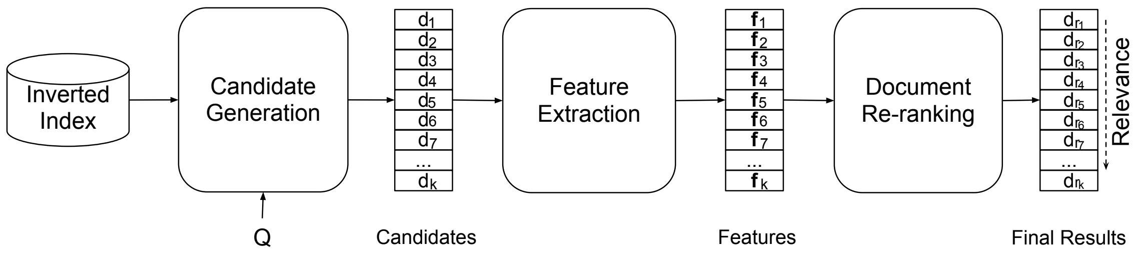


Figure 1.3: Stages of a learning-to-rank pipeline. The candidate generation stage receives a query and returns a set of candidates that are likely more relevant to the input query. The second stage computes features for the given candidate list with respect to a query. Finally, the third stage applies a machine-learned algorithm to rank the candidate documents. The output is a ranked list of documents.

In this work, we study the two main architectural variants of a learning-to-rank pipeline: A monolithic architecture, as illustrated in Figure 1.2, represents the traditional approach; and our three-stage architecture as shown in Figure 1.3. We discuss the design flexibilities of each architecture in detail. In our three-stage architecture, we examine each stage in isolation and present optimizations that enable faster end-to-end query evaluation. We also explore the tradeoffs between search quality (i.e., effectiveness), speed (i.e., query latency), and space (i.e., memory requirements). Our experiments emphasize on microblog documents (tweets), however, we also evaluate our contributions on web collections where appropriate.

1.3 Contributions and Outline

In this work, we study the problem of indexing for streaming documents and propose a fast indexing approach that addresses the concerns we raised earlier about current incremental indexing algorithms. We also explore the implications of our multi-stage architecture on search quality, speed, and memory usage. Additionally, we break down and examine the stages of a learning-to-rank pipeline and present techniques to improve the efficiency of each stage in isolation. We show that, when the improved stages are put back together, this approach ultimately leads to a more efficient and more flexible end-to-end learning-to-rank pipeline. Our contributions are as follows:

- **Online indexing:** We propose a novel inverted index structure as well as an indexing algorithm that operates entirely in memory, and is capable of indexing thousands of documents per second. Our indexer also provides a mechanism to control postings list contiguity (i.e., how contiguously postings of a term are stored in the final inverted index). In order to demonstrate the tradeoffs between memory usage, contiguity, and query evaluation speed, we present experimental evaluation on large-scale web collections, and discuss the results in Chapter 3.

- **Bloom filter chains:** We introduce a scalable variant of Bloom filters [18] that can automatically expand with minimal memory and computational cost, in order to accommodate an unbounded stream of sorted integers. This data structure guarantees a theoretical upper-bound on the false positive rate and offers fast membership tests. We describe Bloom filter chains in more detail in Chapter 4. We have originally proposed this data structure in Asadi and Lin [19].

- **Approximate top k retrieval:** We present a novel top k retrieval algorithm, designed specifically for microblog search, which we use in the candidate generation stage. To enable fast retrieval, we utilize Bloom filter chains, a probabilistic data structure to *approximate* the list of top k documents for an input query. More specifically, in lieu of *searching* standard postings lists to identify common document ids, we perform *approximate* set membership tests on Bloom filter chains—which are alternative representations of postings. Evaluation on microblog data shows that our approach is much faster and that the probabilistic nature of it does not degrade the quality of the candidate set and further does not compromise end-to-end search quality. We empirically characterize the tradeoff space between effectiveness (result quality), time (retrieval speed), and space (index size). We discuss these results in Chapter 4. This chapter has originally appeared in Asadi and Lin [19].

- **Document vectors for feature extraction:** Within the feature extraction stage, we experimentally evaluate the efficiency of different data structures in terms of speed (latency) and memory usage, as detailed in Chapter 5. To our knowledge, this is the first quantitative comparison of alternative data structures intended for fast feature extraction. Our results show that a lightweight inverted index together with an efficiently-organized document vector index, compares favorably to a monolithic positional inverted index. It also offers additional flexibility in terms of systems engineering. Our findings in this chapter have originally appeared in Asadi and Lin [20].

- **Architecture-conscious implementations of tree ensembles:** Among learning-to-rank algorithms, tree ensemble models (such as gradient boosted regression trees) have proven to be superior in terms of effectiveness. However, when it comes to efficiency tree models are expensive to execute since tree implementations contain many conditional jumps (i.e., branches), which consequently result in an under-utilization of processor resources. We propose novel implementations of tree-based models that are highly-tuned to modern processor architectures, as they take advantage of cache hierarchies and superscalar processors. We illustrate our techniques for implementing tree-based models in a standard learning-to-rank task. In Chapter 6, we show that our implementations perform significantly better than standard implementations. This chapter will appear in Asadi et al. [21].

- **Training efficient tree ensembles:** As we discuss in Chapter 6, the performance of our architecture-conscious implementation of tree ensembles is directly correlated with the depth of each individual tree. Therefore, to enable faster evaluation, we present an approach to penalize deeper regression trees during tree ensemble construction. Experimental results in Chapter 7 on a standard learning-to-rank dataset demonstrate significant gains in speed. The techniques used in this chapter have originally appeared in Asadi and Lin [22].

- **End-to-end efficiency-effectiveness tradeoffs:** Chapter 8 evaluates the overall impact of our contributions on end-to-end effectiveness and efficiency, and explores the tradeoffs. Our experiments on microblog documents suggest that our techniques can improve end-to-end query latency without detriment to search quality. We show that our techniques increase the overall memory footprint, but we justify this in the context of modern server configurations and within a broader search service. This chapter has partially appeared in Asadi and Lin [23].

In the chapters that follow, we discuss each of the above contributions in more detail. Chapter 9 concludes this work by a discussion on the limitations of this work. However, before we proceed to further discuss these contributions, in Chapter 2 we present a literature review on the topics related to this work.

Chapter 2

Related Work

We begin with an overview of the academic literature on the components of a search pipeline. This chapter covers existing incremental indexing algorithms, candidate generation algorithms, feature extraction, an introduction to learning-to-rank algorithms and tree ensemble models, and finally, modern processor architectures.

2.1 Inverted Index and Incremental Indexing

At the heart of most retrieval systems lies the inverted index. The inverted index is a mapping from vocabulary terms to postings lists. A postings list contains a number of postings, one for every document that contains the corresponding vocabulary term. The length of a postings list is referred to as a term's document frequency (*df*). Each posting, in turn, holds a document id and a payload. In general, the payload contains information such as term frequency. In positional indexes, the payload also contains the positional information of a term within a document. Alternatively, the payload can hold pre-computed scores known as impacts [24, 25]. Figure 2.1 illustrates the structure of a positional inverted index for a sample collection of three documents.

Postings in a postings list are organized in two ways: Either postings are sorted by document ids or by the payload (e.g., term frequency or impact scores). There are a few tradeoffs in this design space. Each design leads to a different space requirement, demands a certain organization of postings in order to effectively utilize compression, requires its own mechanism for updates and maintenance, and supports different types of query evaluation algorithms. We refer the reader to Zobel and Moffat [26] for an overview of approaches to organizing postings lists. In a search service, an indexer is responsible for construction and maintenance of inverted indexes.

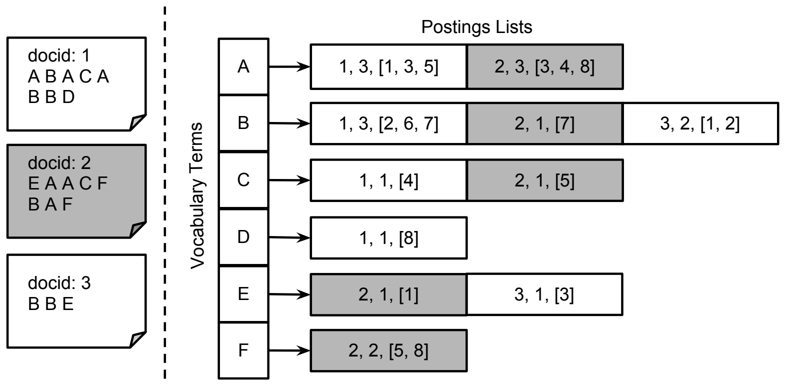


Figure 2.1: Structure of a sample positional inverted index for three documents. Each posting contains a document id (the first integer), term frequency (the second integer), and a list of term positions (in brackets).

2.1.1 Incremental Indexing

As explained in Chapter 1, the problem of indexing streaming documents is equivalent to updating an existing inverted index with one document at a time—a problem known as "incremental" indexing. Solving this problem boils down to designing inverted index update algorithms. It is clear that different index organizations require different update algorithms. Since global statistics for streaming documents constantly change, updating an impact-sorted index with a new document could lead to substantial modifications to postings lists. Therefore impact-sorted indexes are not appropriate for streaming documents. As such, we only consider document id-sorted indexes throughout this work.

The problem of incremental indexing has received considerable attention in the literature, and decades of research have led to many inverted index update algorithms. However, most of these algorithms assume that the inverted index does not fit in the main memory and must ultimately be organized on disk. As such, the fundamental challenge that update algorithms attempt to address lies within the tradeoff between index update speed and query evaluation speed.

To understand this tradeoff, consider an inverted index in which each postings list is stored contiguously on disk.[1] Such storage offers a high query evaluation speed, because one disk seek is enough to find and read a term's postings. However, maintaining a postings list's contiguity after

[1]Note that, storage–hence, contiguity–of an on-disk index is delegated to the file system.

an update complicates the update algorithm and leads to slower updates. At the other extreme, consider an inverted index where each postings list is a discontiguous linked list of postings. Such index is very fast to update; to insert a posting into a postings list, an indexer can simply append the new posting to the current end of the index file, and link it to the last posting. However, query evaluation becomes very slow, since reading a postings list involves many random disk seeks. Between these two extremes fall a vast literature of update algorithms that attempt to achieve a middle ground.

The majority of update algorithms accumulate postings of new documents in the main memory, until the size of in-memory postings exceeds a threshold or when the memory is exhausted. When this happens, the in-memory postings are merged with the on-disk index. As such, an update problem becomes a merge problem. Different merge strategies present a different update-query evaluation speed tradeoff. These strategies can be categorized into three general classes of algorithms: re-build, merge, and in-place updates.

Re-build: A simple, but costly approach is to *re-build* the entire index whenever a merge is triggered. The high cost of a re-build operation is amortized because we accumulate postings over time and perform the merge in batches. While a merged index is being constructed, queries continue to be served from the old index until the new index is complete. Only then, the new index replaces the old one. This approach is very slow in practice [27], however, in situations where there is no urgency to update the index (e.g., a school's intranet), it is a favorable solution due to its simplicity.

Several algorithms have been proposed to ensure contiguity in the re-built index. One such approach is the so-called "two-pass" solution [28, 29, 30]. In the first pass, the algorithm collects some statistics from the collection (e.g., document frequency). Using these statistics, the algorithm determines how much on-disk space to allocate for every postings list. The second pass subsequently reads the collection one more time and inserts postings into the allocated space to construct a contiguous inverted index. This approach is straightforward to implement, however, reading the collection twice can become very expensive for large collections.

To address this issue, Moffat and Bell [3] propose a single-pass, sort-based inversion algorithm. In their method, when a merge is triggered, in-memory postings are flushed onto the disk as individual batches. In the end, a post-processing step sorts these individual batches and combines them together to build the final inverted index. As an extension to this work, Heinz and Zobel in [4] introduce optimizations by compressing in-memory postings on-the-fly, and by holding only a small portion of the vocabulary in memory. Other attempts to re-build the index in a single-pass (such as the work of Harman and Candela [31]) do not lead to significant improvements.

Merge: An alternative to the re-build strategy is to *merge* the in-memory postings with the on-disk index on the fly. That is, when a merge is triggered, the algorithm reads on-disk postings lists one at a time, appends in-memory postings when appropriate, and finally writes the updated list to a different location on disk.

This recipe is referred to as "immediate merge," which merges the entire index with in-memory postings every time a merge is triggered. Lester et al. [5, 27] study this approach in detail and show that, with large-enough memory, immediate merge is indeed very fast. However, its performance is severely deteriorated if the available memory is small. This is because, in such cases, the merge operation happens more frequently, which translates into more disk interactions—a high-latency operation.

Researchers have introduced smarter merge strategies. One thread of work [32, 33, 34] focuses on strategies that merge in-memory postings with only a *fraction* of the on-disk index, as opposed to the *entire* index. The general idea is to partition the on-disk index into multiple segments, and merge in-memory postings in a hierarchical way. A different thread of work [35, 36] presents solutions that merge only a subset of in-memory postings with the on-disk index, as opposed to merging all in-memory postings.

In-place updates: Another general approach to updating the inverted index is what is referred to as *in-place* updates. The idea is to directly append a term's in-memory postings to the current end of its on-disk postings. If the current on-disk list does not have enough space to accommodate new postings, the algorithm moves the list to another location on disk and expands it such that new postings can safely be inserted into the allocated space. When expanding the list, the algorithm can over-allocate space to allow future postings to be inserted in place. Determining the appropriate amount of space to over-allocate is a challenging question and requires an effective over-allocation policy [2, 37].

The in-place merge strategy needs to deal with several practical problems. First, a cut-expand-and-paste mechanism along with an over-allocation policy leaves free space behind. The algorithm should keep track of this free space and manage it efficiently. Second, the presence of free space eventually leads to fragmentation in the inverted index which, if not carefully dealt with, can result in a very large inverted index file. Finally, unlike the strictly-sequential disk access pattern in the re-build and most merge strategies, an in-place strategy requires random disk seeks. These issues together are the reason why an in-place update can be even slower than a re-build update [27, 5].

Brown et al. [38] proposed one of the first in-place update strategies. In their approach, if there is enough space at the current end of an on-disk list, the in-memory postings are appended in-place. Otherwise, a larger amount of space is allocated and the content of the old block is moved to the new location, with new postings appended to its end. Space is allocated in powers of two ($2^4, 2^5, ..., 2^{13}$). That is, a list which is 2^i bytes long will be expanded to 2^{i+1} bytes. If a list requires more than 2^{13} bytes, the old block is left untouched and a new 2^{13} byte-long block is allocated and linked to the last block—thereby, creating a linked list of blocks.

Tomasic et al. [2] present an in-place approach which distinguishes between short and long postings lists. In their work, they store postings into fixed-size buckets, where each bucket accumulates postings for a disjoint subset of the vocabulary. The algorithm inserts a new posting to its corresponding bucket. Once a bucket overflows, the longest list in the bucket is declared a *long* list and is moved out to a separate location on disk. If a term's postings list is *long*, the algorithm merges its new postings with the long list using a merge policy. The authors present various policies in [2]. An "update efficient" policy optimizes the update time by simply appending postings to the end of the inverted index file. A "read efficient" policy combines long lists with new postings, and writes the combined list contiguously at a different location on disk. As a compromise between the two extremes, another policy uses predictive over-allocation methods and allows some degree of discontiguity for long lists.

Büttcher et al. [39] propose a hybrid approach where "short" lists are updated by a merge strategy while "long" lists are updated in place. In this design, there are two independent on-disk indexes: an inverted index that contains short postings lists (short index), and another that contains long lists (long index). The algorithm views a term's postings list as a concatenation of a long list and a short list. New in-memory postings are always merged with the current on-disk short index. When a list exceeds a predefined number of postings, its postings are removed from the short index and are appended to the long index. This update algorithm also allows for discontiguity in the long index, however, the degree of discontiguity can be managed through different parameter settings.

2.1.2 In-Memory Indexes

There are only a few recent studies [35, 40] that explore storage other than the disk. Li et al. [35] study incremental updates to inverted indexes on Solid-State Drives (SSDs). Luk and Lam [40] propose an in-memory scheme for small inverted indexes, and use a linked list data structure to represent postings—the scalability of their approach remains an unexplored question. However, the decreasing trend in the cost of main memory over the past decade opens doors to exciting opportunities, allowing us to weigh pros and cons of large-scale *in-memory* indexing.

In-memory indexes have a number of attractive properties, which we believe justify this work's exclusive focus on in-memory indexes. First, keeping the entire inverted index in main memory enables faster query evaluation [6]. Second, as our experiments highlight, serving indexes from the main memory leads to a behavior different from what we observe with on-disk indexes. In particular, we show that full postings list contiguity is not a requirement for having an ideal query evaluation speed; through relaxing constraints on contiguity, we observe no statistically distinguishable increase in query latency. Third, we can take advantage of block compression techniques such as PFOR [41, 42, 43] or VSEncoding [44], which offer fast decompression speeds, whereas utilizing block compression on-the-fly further complicates the update process for on-disk indexes. Finally, since operations on an in-memory index avoids high-latency interactions with the disk, in-memory indexes are more appropriate for the task of indexing a high-velocity stream of documents.

An example of an in-memory index structure designed for streaming documents is the work of Busch et al. [45]. They introduce Earlybird—Twitter's real-time search service—in which indexes are served from and maintained in the main memory. They describe a memory layout for postings lists which facilitates efficient *in-place* updates. However, their index structure is designed specifically for tweets with limited length, whereas in this work we propose a generic framework which can index documents of arbitrary length. In addition, EarlyBird leaves postings uncompressed during indexing and postpones compression to a post-processing phase, while our proposed indexing mechanism performs on-the-fly compression.

2.1.3 Dictionary

Finally, a note on the dictionary structure: Just as documents are assigned unique document ids, each vocabulary term is assigned a unique integer (known as a term id). The rationale is that we can save space by storing integers in lieu of strings of characters, thereby enabling further compression of postings using integer compression algorithms. As such, we need a dictionary data structure that maps every vocabulary term to its term id. Because the dictionary is probed at a high frequency during indexing and query evaluation, the choice of the dictionary data structure directly impacts indexing and query evaluation speeds. Primarily, we are interested in an in-memory dictionary.

Zobel et al. [46] compare various dictionary data structures (binary search trees, splay trees, tries, hash tables, etc.) among which, optimized hash tables are concluded to be significantly more efficient. In particular, they use a bit-wise hash function [47], and reorder in-chain nodes using the move-to-front technique [48]. There has been other studies since that confirm their findings [49, 50]. However, in situations where the dictionary structure does not fit in memory and must be organized on disk, hash tables are not an appropriate choice. This is because hash tables distribute keys randomly and therefore do not preserve the lexicographical order of terms, which subsequently results in an increase in the number of random disk seeks when accessing a dictionary record. Therefore, a slower data structure that ensures a lexicographical order (such as a B-tree) is favorable. However, since we focus on in-memory indexing and assume the dictionary is also held in memory, such sorted order is not a requirement. As such, we use an optimized hash table throughout this work.

2.2 Candidate Generation

As mentioned in Chapter 1, the first stage in a learning-to-rank pipeline is the candidate generation stage. Given a query, the candidate generation stage retrieves the top k documents (known as candidates) that exhibit stronger relevance signals. Relevance is measured by a scoring function. By definition, this stage comprises two main components: a scoring function and a retrieval algorithm.

A scoring function should be simple and computationally cheap, so that it does not dominate the cost of candidate generation. At the same time, it should be effective at distinguishing non-relevant documents from relevant documents. A range of query-dependent scoring functions (e.g., BM25 and tf-idf) as well as query-independent scores also known as static priors (e.g., content quality scores) can be used for these purposes. In this work, we use a combination of a static prior as well as a query-dependent scoring function.

Retrieval algorithms are divided into two general categories: conjunctive and disjunctive query processing. In *conjunctive* query processing, only documents that contain *all* query terms are considered as candidates. Previous work on web search has shown that this approach leads to high early precision [13]. Conjunctive query processing requires solving the problem of postings lists intersection. This problem has been studied in detail and various algorithms have been presented in the literature for impact-sorted indexes [24, 25, 51], as well as document id-sorted indexes [52, 53, 54, 55, 14]. We do not consider methods that operate on impact-sorted indexes further, since impact-sorted indexes are not appropriate for our problem setup. Algorithms designed for document id-sorted indexes typically pick a target element, called an "eliminator," from one of the lists. The intersection algorithm then performs random-access lookups in the remaining lists for that element. Among these methods, the SvS algorithm [55] offers a strong baseline.

The SvS algorithm for postings list intersection first sorts the postings lists in increasing length. It begins by intersecting the two smallest lists. Then, at each step, the algorithm intersects the current result set with the next postings list, until all lists have been consumed. In our implementation, we use galloping search to intersect postings lists. The SvS algorithm has no mechanism for early termination, and thus must exhaustively compute the entire intersection set. As a result, in order to retrieve the top k documents using the SvS algorithm, we must first retrieve the entire intersection set and compute relevance scores for every document in the set. Then, we can rank all the documents by their relevance scores, and return the top k documents from the sorted list.

The alternative to conjunctive query processing is disjunctive query processing. In this approach, documents that contain *any* of the query terms are considered in the retrieval algorithm. Because of this generous consideration, disjunctive query processing is significantly slower than conjunctive query processing. To improve efficiency, researchers have designed clever algorithms that (I) can skip over large portions of postings lists [56, 57, 13, 58, 59], or (II) consider more promising documents early in the process [24, 25, 51]. The two state-of-the-art disjunctive query processing algorithms are Block-Max WAND [59] (an extension of the WAND algorithm [13]) and the work of Strohman and Croft [51]. The latter requires impact-sorted indexes, which again are inappropriate for our problem formulation.

The WAND algorithm for top k retrieval uses a *pivot*-based pointer-movement strategy which enables the algorithm to skip over postings of documents that cannot possibly be in the top k results. In this approach, each postings list has a "current" pointer that moves forward as the algorithm proceeds. Postings to the left of the pointer have already been examined, while postings to the right have yet to be considered. The algorithm keeps postings lists sorted in increasing order

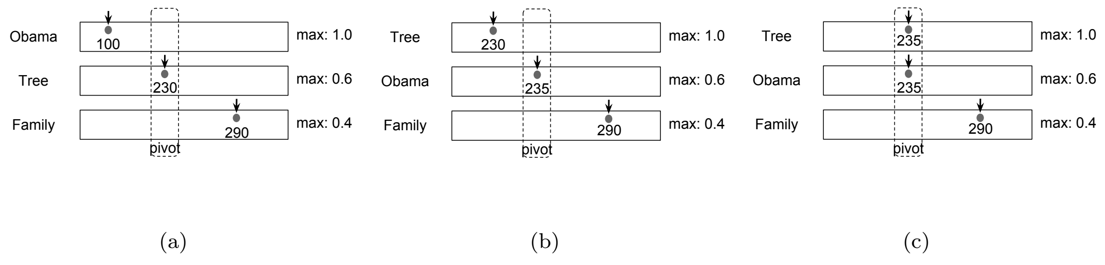


Figure 2.2: Illustration of the WAND algorithm on a three-term query in which the current pointers point to documents 100, 230, and 290. With a score threshold of 1.4, WAND selects "tree" as the pivot term (a). Since the current pointer for "Obama" points to a document id that is less than 230, the algorithm moves its current pointer to the posting whose document id is at least 230, and re-orders the lists (b). Current pointers now point to 230, 235, and 290. Document 235 becomes the pivot. Due to the same reason as in (a), the first term's (now, "tree") current pointer is moved to the posting whose document id is at least 235, resulting in (c). Finally, since the pivot term's document id is equal to that of its preceding term, the algorithm computes the score for document 235, and adds it to the heap if its actual score is greater than the threshold.

of the docid of the "current" posting. In addition, each postings list is associated with a score upper bound, indicating the maximum score contribution of the corresponding term. At first, the algorithm creates an empty heap of size k. At each step, the algorithm finds a pivot term: it does so by adding up the score upper bounds of each term (in sorted order). The pivot term is the first term where the sum exceeds a given threshold. The threshold is typically set to the lowest score in the heap—that is, a document's score must be higher than this value for the document to appear in the top k results. If no such term exists, then the algorithm terminates. Otherwise, there are two cases: If all the preceding terms' current docids are equal to that of the pivot term, the candidate document pointed to by the pivot term is scored using the scoring function, and is then added to the heap if the actual score exceeds the threshold. Otherwise, the algorithm moves the current pointer for one of the preceding terms to the posting whose document id is greater than or equal to that of the pivot term, and the algorithm repeats.

As an example, assume that we are given the query "Obama family tree," where the score upper bounds for "Obama," "family," and "tree" are 1.0, 0.4, and 0.6 respectively. Further assume that, as illustrated in Figure 2.2(a), the current pointers point to documents 100, 290, and 230 for the query terms. WAND sorts the postings lists such that the current document ids are in ascending order. Suppose the current threshold is set to 1.4 (the current lowest score in the heap). Therefore, "tree" is selected as the pivot term ($1.0+0.6 > 1.4$). The algorithm can now confidently claim that the smallest document id that can make it into the top k is 230. Because the current document id for "Obama" is not equal to 230, the algorithm moves its pointer to the posting whose document id is greater than or equal to 230, and re-orders the lists, resulting in Figure 2.2(b). Current pointers now point to documents 230 for "tree," 235 for "Obama," and 290 for "family." Note that document 230 is never fully scored because it does not contain "Obama." The term "Obama" is selected as the pivot next, and similar to the case in Figure 2.2(a), the pointer for "tree" is moved to the posting whose document id is at least 235, resulting in Figure 2.2(c). Since the current document id for the pivot term is equal to that of its preceding term, the algorithm computes the score for document 235, and if its actual score is greater than the threshold 1.4, it adds the document to the heap.

Recently, Ding and Suel [59] introduced an optimization on top of WAND, called Block-Max WAND that substantially increases query evaluation speed. The idea is that instead of using the global maximum score of each term to compute the pivots, the algorithm uses a piece-wise upper-bound approximation of the scores for each postings list. However, this optimization becomes difficult to apply when dealing with streaming documents since the global statistics constantly change—the indexing algorithm would need to adjust all piece-wise upper-bounds every time a new document is added to the index.

In this work, we propose a fast top k retrieval algorithm which is a variant of WAND. Our solution is to replace postings list lookups with membership tests in Bloom filters. We provide an overview of Bloom filters in Section 2.3, and address the challenges we face in using Bloom filters for streaming documents in Chapter 4.

We should also mention that, while index pruning [60, 61] and early termination [57, 51, 10] are two families of techniques that improve the efficiency of intersection algorithms, they are incompatible with the index structures and query evaluation algorithms that we present in our work.

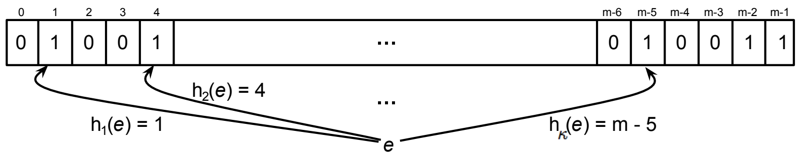


Figure 2.3: Insertion of element e into a Bloom filter of length m using κ hash functions h_1 to h_κ.

2.3 Overview of Bloom Filters

A Bloom filter [18] is a space-efficient data structure that offers fast *approximate* set membership tests. It is comprised of a bit array of size m, and κ hash functions that generate values in the range $[0, m-1]$. Bloom filters provide two operations: insertion, and membership test. An insertion adds a new element to the set. A membership test determines if an element has previously been inserted into the data structure. Bloom filters support very efficient $O(1)$ membership tests.

In order to insert an element e into a Bloom filter, as illustrated in Figure 2.3, we compute the values of the given κ hash functions on e, and then set the corresponding bit positions in the underlying bit array to one. If a bit has already been set to one, it is left untouched; that is, hash collisions are ignored.

Similar to an insertion, to query a Bloom filter for an element e, we compute κ hash values on e, using the same hash functions as in the insertion process. We then probe the corresponding bit positions. The membership test passes if all the bit positions are set, and it fails if any of the bit positions is unset. A common optimization for probing is to compute the hash values one at a time, and terminate as soon as a probed bit position is unset. This short-circuiting technique helps avoid unnecessary hash value computations.

Membership tests on Bloom filters are approximate. When a membership test on a test element fails, it is guaranteed that the test element has never been inserted into the Bloom filter, making the occurrence of false negatives impossible. A false positive, on the other hand, is possible. That is, a membership test may assert true on a test element, while in fact the element was never inserted into the Bloom filter.

False positives occur when there are hash collisions, and become more likely as more elements are inserted into a Bloom filter. Fortunately, the false positive rate (also known as the false drop rate) can be theoretically predicted and controlled given the values m and κ. The probability of a false positive is equal to the probability of a successful membership test, after n elements are inserted into a Bloom filter of length m, using κ hash functions.

Assuming hash functions select a bit position in the array of m bits with equal probability,[2] the probability that a bit is not set by one hash function during an insertion is equal to $(1-\frac{1}{m})^{\kappa}$. After n insertions the probability that a bit is set to one becomes:

$$p = 1 - \left(1 - \frac{1}{m}\right)^{\kappa n}.$$

As mentioned earlier, a membsership test passes if all κ bit positions (as computed by the hash functions) are set to one. A strict lower-bound estimate of the probability that a membership test passes after n insertions, is as follows [62] for $\kappa \geq 2$:

$$p^{\kappa} = \left(1 - \left(1 - \frac{1}{m}\right)^{\kappa n}\right)^{\kappa} \approx \left(1 - e^{\frac{-\kappa n}{m}}\right)^{\kappa}, \tag{2.1}$$

while the upper-bound [62] is:

$$p^{\kappa} \times \left(1 + \frac{\frac{\kappa}{p}\sqrt{\frac{\ln m - 2\kappa \ln p}{m}}}{1 - \frac{\kappa}{p}\sqrt{\frac{\ln m - 2\kappa \ln p}{m}}} + \frac{2}{\sqrt{m}}\right) \tag{2.2}$$

For sufficiently large values of m and sufficienctly small values of κ, the difference between p^{κ} and Equation 2.2 is negligible. This means, Equation 2.1 provides a close estimate of the actual false positive rate. Since in our work we set m and κ to large and small values respectively, we use Equation 2.1 to predict and control the false positive rate.

From Equation 2.1, it is clear that we can reach a desirable false positive rate through adjusting two parameters: the number of hash functions, κ, and the ratio of the allocated number of bits to the number of elements, $r = \frac{m}{n}$. Figure 2.4 illustrates the impact of these two factors on the false positive rate as predicted by Equation 2.1. In this figure, it is assumed that the number of elements (n) is fixed.

This figure demonstrates the tradeoffs encoded in a Bloom filter through parameters r and κ. Assuming a fixed number of hash functions, allocating more space (i.e., increasing r) improves

[2]In practice, hash functions do not behave ideally. As a result, the probability of false positives is slightly higher than what theory predicts.

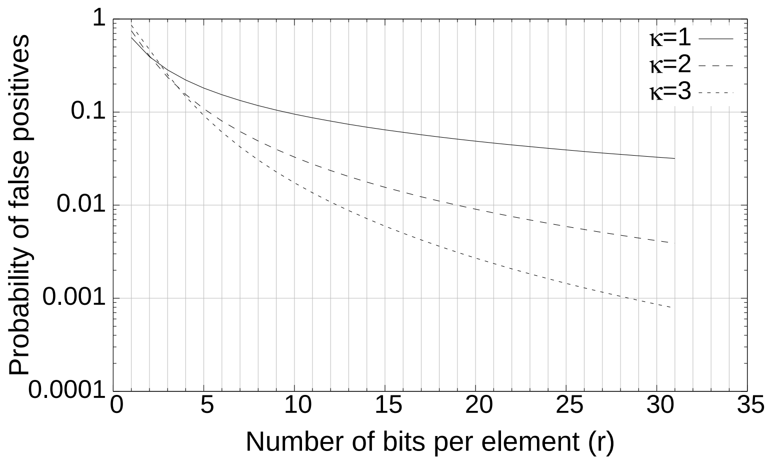


Figure 2.4: Probability of a false positive (lower-bound) for a Bloom filter for different values of r and κ.

the false positive rate. However, this comes at the cost of larger memory requirements. On the other hand, using more hash functions while fixing r, reduces the false positive rate. This gain too is not free to achieve as it requires more hash value computations, thereby affecting speed. These factors form a tradeoff space between accuracy (i.e., false positive rate), time (i.e., speed of membership tests), and memory footprint (i.e., space requirements). We will explore these tradeoffs in our experiments.

Parameters κ and r from Equation 2.1 can only guarantee a certain false positive rate if the number of elements (n) is known beforehand. However, if the number of inserted elements exceeds a Bloom filter's capacity (i.e., n), the underlying bit vector becomes saturated with 1's. As a result, the rate of false positives drastically increases. This disqualifies standard Bloom filters for use in situations where the cardinality of a set is not known *a priori*; that is, when the set is dynamic.

To deal with the scalability issues, Almeida et al. [63] propose an extension to Bloom filters. To construct a scalable filter, the authors use a series of Bloom filters of geometrically-increasing sizes. When a Bloom filter reaches its capacity, a new Bloom filter is constructed and added to the list. New elements are then inserted into this new filter. To perform a membership test, all Bloom filters are probed. A membership test passes if at least one of the Bloom filters claims to contain the element in question. There are a few drawbacks to this approach. First, membership tests become slower as more and more Bloom filters are added to the list. Second, to fix the false positive rate, this approach increases the size of Bloom filters progressively, which subsequently adds to the memory cost. In this work, we address these issues by introducing constraints on the elements that are to be inserted into a Bloom filter. More specifically, we require that the elements be a stream of sorted integers. Chapter 4 details our approach.

There have been many studies that utilize Bloom filters for information retrieval applications [64, 65, 66]. For example, Li et al. [66] have used Bloom filters to perform keyword search in Peer-to-Peer (P2P) systems. In particular, they present a mechanism to perform (distributed) postings list intersection which reduces the amount of data transferred between nodes in the P2P system; to perform an intersection, a node shares a Bloom filter representation of postings from its inverted index with other nodes instead of the actual postings list, while the nodes receiving the Bloom filters perform intersection between their postings lists and the Bloom filters. In contrast, in our work, we focus on a single machine and utilize Bloom filters to speed up candidate generation in a multi-stage search architecture.

2.4 Feature Extraction

Document ranking algorithms in a learning-to-rank retrieval pipeline often consider features that are costly to compute. As discussed earlier, in order to reduce the cost of computing these expensive features, we apply the document ranking algorithm to only a small set of candidate documents. Feature extraction on this relatively small set is affordable and practical.

Architecturally, we can view feature extraction as a distinct stage between the candidate generation and document ranking stages. The task of a feature extraction stage is to compute features that are used in the document re-ranking stage, which are potentially more sophisticated.

An example of more complex features is the class of term proximity features. Term proximity features can be divided into two types: query terms occurring in order within a fixed span (e.g., as a phrase), and query terms co-occurring within a fixed span in any order. Computing these features requires storing term position information which substantially increases the size of the inverted index. Additionally, it requires decoding and comparing term positions to determine whether terms co-occur within a specified span. For these reasons, computing term proximity features is relatively expensive. One recent set of microbenchmarks has shown that computing bigram features is 20 times slower than computing unigram features [12]. However, studies have shown that term proximity features lead to statistically significant improvements over less complicated features such as term-based (i.e., unigram) features [67, 68, 69, 70, 71, 72].

Delaying the computation of expensive features to a second stage has its own advantages. We argue that, decoupling feature extraction and candidate generation stages allows us to use a different data structure in each stage. In particular, we do not need to store positional information in the inverted index used in the candidate generation stage—thereby reducing memory footprint in the first stage. As we discuss later in Chapter 5, this is the route we pursue.

Note that, real-world search engines take advantage of caching [73]. In principle, one can cache expensive features in frequently-evaluated documents to avoid re-computation. However, feature caching is a non-trivial engineering problem. This is because the space of cache values is the query space crossed with the document space. This challenge aside, we argue that caching is an orthogonal issue to our work: caching strategies can also be applied on top of our proposed techniques to further increase speed.

2.5 Learning to Rank Algorithms

Another area of relevant previous work is the vast literature on learning-to-rank [74]: application of machine learning techniques to the document ranking problem. Among machine learning algorithms used in learning-to-rank, studies have shown that tree-based models combined with ensemble techniques are highly effective. An example of this class of learning algorithms is the family of Gradient Boosted Regression Trees (GBRTs) [75, 8, 76, 9]. A state-of-the-art GBRT is the LambdaMART [8] algorithm, which is the combination of LambdaRank [77] and MART [78]—a class of boosting algorithms that perform gradient descent using regression trees. LambdaRank, in turn, is based on the RankNet [79] learning algorithm.

RankNet is a pair-wise learning-to-rank approach; that is, it optimizes the relative ordering of documents in a ranked list, as opposed to a list-wise approach which directly optimizes a retrieval effectiveness metric. The training data consists of pairs of documents, together with a probability that one document is ranked higher than the other. Given the training data, the algorithm minimizes a probabilistic cost function defined on the relative ordering of documents. RankNet constructs a neural network for training, where the weights in the network are adjusted using gradient descent.

One particular drawback to using gradient descent is that the cost function must be smooth in order to compute its gradient. Unfortunately, information retrieval measures, such as NDCG [80] and ERR [81], are either discontinuous or flat. This means that, gradient of a retrieval metric is either zero or undefined at any given point. Due to this discontinuity, RankNet cannot be used to directly optimize an information retrieval metric. To address this, Burges et al. [77] developed LambdaRank which bypasses these difficulties. The idea behind LambdaRank is that, rather than deriving gradients from a cost function, we can define desired gradients and use them directly. We refer the reader to [77] for more details. Given these gradients, one can apply gradient descent to optimize a retrieval metric.

LambdaMART uses the LambdaRank algorithm at its core and performs gradient boosting to learn an ensemble of regression trees. The algorithm builds an ensemble by sequentially constructing a new tree that best accounts for the remaining regression error (i.e., the residuals) of the training samples. More specifically, LambdaMART learns a linear predictor $H_{\boldsymbol{\beta}}(x) = \boldsymbol{\beta}^{\intercal}\mathbf{h}(x)$ that minimizes a given loss function $\ell(H_{\boldsymbol{\beta}})$, where the base learners are limited-depth regression trees [82]: $\mathbf{h}(x) = [h_1(x), ..., h_T(x)]$, where $h_t \in \mathcal{H}$, and $\mathcal{H}$ is the set of all possible regression trees.

Assuming the ensemble has $t-1$ regression trees, LambdaMART adds the t_{th} tree that greedily minimizes the loss function, given the current residuals. To construct an individual regression tree, LambdaMART uses the CART [82] algorithm. CART works by recursively splitting the training data. At each step, the algorithm computes the best split (a feature and a threshold) for all the current terminal nodes, and then applies the split that yields the most gain, thereby growing the tree one node at a time. The algorithm continues until there are J terminal nodes in the tree.

In LambdaMART, CART chooses a split that minimizes a least squares error function as follows:

$$C(N, \langle f, \theta \rangle_N) = \sum_{x_i \in L} (y_i - \bar{y}_L)^2 + \sum_{x_i \in R} (y_i - \bar{y}_R)^2, \tag{2.3}$$

where N denotes a node in the tree; $\langle f, \theta \rangle_N$ is a split, consisting of a feature and a threshold; L and R are the left and right sets containing the instances that would fall into the left and right of node N if the split was applied; x_i and y_i denote a training instance and its associated residual; and finally, $\bar{y}_L$ and $\bar{y}_R$ are the average y of instances that would fall to the left or the right branch of node N if the split was applied. Minimizing Equation (2.3) is equivalent to maximizing the difference in $C(\cdot)$ before and after a split is applied to node N. This difference can be computed as follows:

$$G(N, \langle f, \theta \rangle_N) = \sum_{x_i \in N} (y_i - \bar{y}_N)^2 - C(N, \langle f, \theta \rangle_N), \tag{2.4}$$

where $x_i \in N$ denotes the set of instances that are present in node N.

The final LambdaMART model has a low bias, while it is prone to overfitting the training data (i.e., model has a high variance). In order to reduce the variance of an ensemble model, bagging [83] and randomization can be utilized during training. Friedman [84] introduces the following randomization techniques:

- A weak learner is fit on a sub-sample of the training set drawn at random without replacement. This results in substantial improvements in accuracy.
- Similar to Random Forests [85], to determine the best tree split, the algorithm picks the best feature from a random subset of all features (as opposed to choosing the best overall feature).

Ganjisaffar et al. [9] take this one step further and construct multiple ensembles, each built using a random bootstrap of the training data (i.e., *bagging* multiple boosted ensembles). The bagged model achieves the best of both worlds: it lowers the variance of the final model through bagging, and maintains the low bias of regression trees. In this work, we do not explore bagging, primarily because it is embarrassingly-parallel from the runtime execution perspective and thus is not particularly interesting.

Note that, the focus of most learning-to-rank research is on learning effective models, without concern to efficiency ([12, 86] being exceptions). In contrast, we focus exclusively on runtime ranking performance.

2.6 Optimizing Tree Ensembles

Tree ensemble models often consist of a large number of trees, which become expensive to evaluate at test time (i.e., to classify or score a test instance). There are several methods in the literature that address this issue both at training/construction time, and at test time. We here focus on techniques that can be applied during the construction of an ensemble, and leave test-time optimizations aside. Construction-time optimizations can be categorized into four classes.

- Adjusting the Learning Rate: By using a larger shrinkage parameter η we can force the algorithm to converge faster, thereby creating an ensemble with fewer trees. However, larger η often leads to lower effectiveness. In contrast, a smaller η prevents a high model variance, but results in larger ensembles.

- Early Termination: Assuming a small η is utilized, Ganjisaffar [87] proposes several techniques to reduce the runtime execution cost by discarding trees that contribute the least to the final document scores. In what is referred to as *prefix* compression, only the first n trees in an ensemble are preserved, while the rest are removed from the ensemble. The value of n is determined based on a time budget. This is similar to the early termination of training algorithms described by Margineantu and Dietterich [88], where the focus is on minimizing storage (as opposed to speed *per se*).

- Pruning Trees: A different approach is to re-weight all trees, *after* the entire ensemble is built. In this approach, trees whose absense from the ensemble do not significantly change effectiveness, are assigned a weight of zero and are subsequently discarded. Ganjisaffar [87] uses the Lasso [89] method to compute weights for every tree. However, experiments show that if the learning rate is small, this approach does not lead to more compact ensembles.

  Similarly in [88], the authors filter weak learners from an AdaBoost classifier using a "diversity" and classification-accuracy criterion. Results suggest that 60–80 percent of the weak learners can be discarded without a significant loss in accuracy. For an overview of ensemble pruning techniuqes see [90, 91, 92, 93] and the references therein.

- Pruning Nodes: At a different level, pruning can be applied to individual nodes rather than entire trees. Ganjisaffar in [87] also presents a node pruning approach that trims leaves in a tree and collapses them into a single terminal node *after* an ensemble is built. Leaves are trimmed only if the impact of such trimming on effectiveness is not significant. However, despite the costly computation required to perform such pruning, this too fails to make ensembles more compact.

Our work differs from previous work in that we directly model and optimize the evaluation speed *while* constructing the ensemble. The idea behind our work is similar to that of Xu et al. [86]. The authors consider a scenario where features are labeled with their computational cost, and aim at minimizing the overall cost of evaluation by integrating feature costs into the loss function used during training. However, in their model, the authors assume that the cost of evaluating a regression tree at test-time is constant, regardless of its topology. In this work, however, we argue that tree evaluation cost is in fact a function of tree structure (in particular, tree depth).

We consider previous work on post-pruning to be orthogonal to our work, since post-pruning can be applied to an ensemble regardless of how the ensemble was constructed. Early exit strategies [94] at test-time and cascade ranking models [95] are also independent threads of work. Previous work on scaling the *training* of tree-based models to massive datasets [96, 97] is also an orthogonal problem to the issues we tackle in this work.

2.7 Modern Processor Architectures

In recent years, we have witnessed what is known as the multi-core revolution [98]. Experts widely agree that an increased transistor density on a chip does not necessarily translate into significant improvements to instruction-level parallelism in hardware. Instead, adding more cores appears to be a better use of these resources.

An increase in processor speed alone, however, is not sufficient to achieve a higher throughput. There are other key contributing factors that impact throughput in nontrivial ways. One such factor is the memory latency, which dominates runtime because of the inevitable interactions between the processor and the main memory. Unfortunately, the main memory is becoming relatively slow as compared to processor speed. That is, improvements to processor speed has outstripped achievements in reducing memory latency. As a result of higher latencies, interactions with the main memory become a bottleneck and form the so-called "memory wall" [99].

In order to attenuate the impact of this problem, computer architects have introduced hierarchical cache memories. In modern architecutres, there are three levels of cache memories referred to as L1, L2, and L3 in order of proximity to the processor. Cache memories serve as a short-term memory between the processor and the main memory, and "remember" and "forget" data according to a caching strategy.

Cache memories are designed based on the assumption that the processor repeatedly accesses a relatively small region in memory. That is known as "reference locality." When this assumption holds, a fraction of memory accesses can be fulfilled directly from the cache. This fraction is called the *cache hit rate*. On the other hand, accessing data that are not present in the cache causes a *cache miss*. Cache misses propagate through the cache hierarchy until the requested data are found. That is, if a datum is not found in L1 cache, the processor probes L2 cache, and if that fails too then looks for it in L3, and finally in the main memory if L3 cache does not contain the data. A cache miss becomes more expensive as it propagates down this hierarchy.

Managing cache content is a complex problem. However, software developers can take advantage of caches by taking two properties of cache memories into consideration when designing new data structures. First, caches are organized into cache lines. A cache line is the smallest unit of transfer between two caches and the main memory. That is, when a program accesses a datum at a particular memory location, an entire cache line surrounding that location is fetched and placed into the cache. In this way, subsequent accesses to memory locations within that cache line are very fast. Second, if a program accesses memory in a predictable sequential way, the processor will prefetch and move data into cache before the program explicitly requests the data. We refer the reader to [100] for more details on cache architectures.

The use of cache-conscious data structures has been extensively explored in the database community [101, 102, 103, 104, 99]. On the other hand, information retrieval and data mining communities have not taken advantage of the advances in modern processor architectures. In this work, we propose novel implementations of (regression) tree data structures that take advantage of cache properties.

An independent thread of work on processor architecture that improves througput is the introduction of pipelining. In a pipelined execution model, instruction execution is split into several stages (e.g., fetch, decode, execute, and write-back). At each clock cycle, all instructions that are currently in the pipeline advance one stage in the pipeline. New instructions enter the pipeline, while instructions that leave the pipeline are retired. Therefore, a pipelined execution model allows for execution of multiple instructions at the same time, but each at a different stage.

In addition to pipelining, modern *superscalar* processors have the ability to dispatch more than one instruction per clock cycle. That is, more than one instruction can simultaneously be at the same stage in the pipeline. Moreover, superscalar processors allow for out-of-order execution of instructions, provided that the instructions are independent.

Pipelining is susceptible to two dangers known as hazards: data and control hazards. *Data hazards* occur when there is data dependency between consecutive instructions—when one instruction requires the output of another. Such hazards happen frequently when dereferencing pointers, where requesting data requires computing a memory location first—instructions that follow cannot proceed until the memory location is known. In these scenarios, the processor has to simply wait

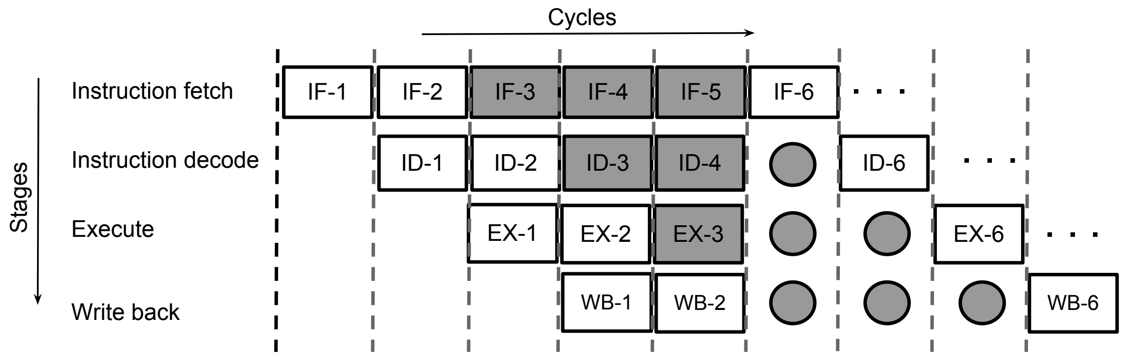


Figure 2.5: Pipelined execution with four stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), and Write Back (WB). A branch misprediction occurs at the fifth cycle (when executing the third instruction, EX-3), which causes the pipeline to be flushed (gray area in the figure: IF-$\{3,4,5\}$, ID-$\{3,4\}$, and EX-3 are flushed). After the pipeline is flushed, stages go into an idle mode (pipeline bubbles), and the processor rolls back and starts executing the correct branch.

for the result. In the meantime, superscalar processors can execute other independent instructions.

Control hazards are instruction dependencies introduced by conditional jumps (e.g., if-else clauses). To prevent control hazards, modern processors use *branch prediction techniques*. In short, the processor tries to predict which code path is more likely to be taken, and goes on to prefetch and execute instructions from the more likely branch. However, if the guess is incorrect, the processor must undo the instructions that were executed after the branching point. In other words, the processor needs to flush the pipeline. Figure 2.5 illustrates a pipelined execution model with four stages, and shows what happens when a control hazard occurs.

The impact of data and control hazards can be substantial. An influential paper in 1999 concluded that in commercial relational database management systems at the time, almost half of the execution time is spent on stalls [101]. It is possible to reduce the impact of control hazards with a technique called predication [105, 106, 107], which converts control dependencies into data dependencies. We will explore this technique in our work in order to reduce runtime execution of tree ensembles.

Another optimization that we adopt, called vectorization, was pioneered by database researchers [108, 104]. The basic idea is that, instead of processing a tuple at a time, a relational query engine should process a vector (i.e., batch) of tuples at a time to take advantage of pipelining. Since instructions to evaluate multiple feature vectors are independent, using this technique, we can take advantage of the out-of-order execution model in a superscalar processor.

Chapter 3

In-Memory Indexing for Streaming Documents

This chapter focuses on the inverted index as the first component of our learning-to-rank pipeline, as depicted in Figure 3.1. An inverted index, as detailed in Section 2.1, is a mapping from vocabulary terms to postings lists. Postings in a postings list contain information about every document that contains the term. The amount of information kept in postings directly impacts the memory requirements of an inverted index.

Information retrieval researchers in academia have long assumed that memory requirements of inverted indexes surpass the capacity of available memory resources. For this reason, inverted indexes have been traditionally organized on disk. However, modern server configurations, along with advances in distributed search architectures, weaken that assumption and provide the opportunity to free indexing algorithms from their dependence on disk.

The main disadvantage of on-disk indexes is that, serving an index from disk leads to slow query evaluation [6]. Low-latency ranking—which, as argued earlier, is a requirement of the web today—can only be achieved by serving indexes from the main memory. Furthermore, building an inverted index on disk makes it difficult to rank with low latency *during* indexing, in situations where indexing and query evaluation on the same index happen concurrently.

We address these shortcomings in this chapter. First, we present a novel index structure for streaming documents which resides entirely in memory, along with an efficient index update algo-

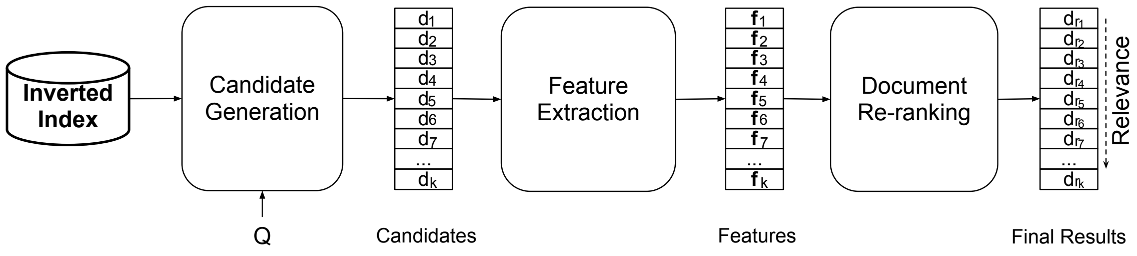

Inverted Index
Candidate Generation
Q
d_1 d_2 d_3 d_4 d_5 d_6 d_7 ... d_k
Candidates
Feature Extraction
f_1 f_2 f_3 f_4 f_5 f_6 f_7 ... f_k
Features
Document Re-ranking
d_{r_1} d_{r_2} d_{r_3} d_{r_4} d_{r_5} d_{r_6} d_{r_7} ... d_{r_k}
Relevance
Final Results

Figure 3.1: Learning-to-rank pipeline.

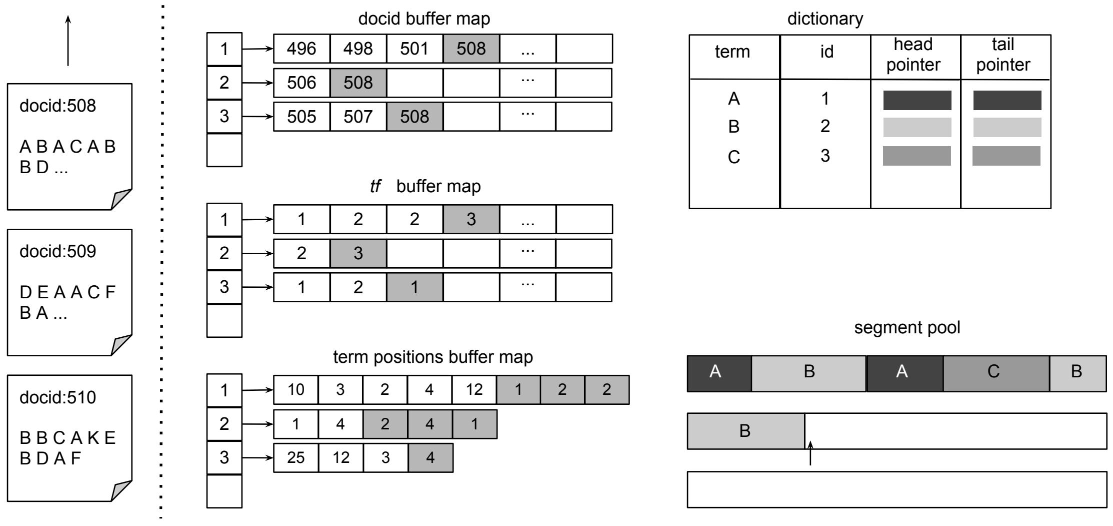


Figure 3.2: A snapshot of our indexing algorithm. In the middle we have buffer maps for storing docids, *tf*s, and term positions: the gray areas show elements inserted for document 508, the current one to be indexed. Once the buffer for a term fills up, an inverted list segment is assembled and added to the end of the segment pool and linked to the previous segment via addressing pointers. The dictionary maps from terms to term ids and holds pointers to the head and tail of the inverted list segments in the segment pool.

rithm. Our index structure scales to modern web collections and takes advantage of state-of-the-art compression schemes. Second, we propose a mechanism to manage postings lists contiguity through co-locating small groups of index segments, and explore the impact of postings list contiguity on both index update speed and query evaluation latency. We find that contiguity does indeed lead to a decrease in query evaluation latency, but beyond a certain degree of contiguity we observe no statistically significant gap in query evaluation performance compared to a fully-contiguous index.

3.1 Approach

3.1.1 Basic Incremental Indexing Algorithm

There are three main components in our index structure, as depicted in Figure 3.2: a dictionary, a set of buffer maps, and a segment pool. Our approach, in summary, is to accumulate uncompressed postings in the buffer maps until the buffer maps are full, and then to apply compression and store the compressed postings in the segment pool.

The dictionary in our implementation is a hash table with a bit-wise hash function [47] and the move-to-front optimization technique [48], which maps terms to integer term ids. For each term, the dictionary additionally holds its document frequency (*df*) as well as a head and a tail pointer that determine the location of the term's postings in the segment pool (more details below). In our implementation, term ids are assigned sequentially as we encounter new terms. Note that there is nothing novel about our implementation of dictionary.

A *buffer map* is a mapping from term ids to arrays of integers (the buffers). In order to construct a positional inverted index, we use three buffer maps: the document id (docid) buffer map, the term frequency (*tf*) buffer map, and a term positions buffer map. The docid map collects document ids of incoming documents, the *tf* map accumulates term frequencies, and the term positions map holds positions of a term within a document. Every document has exactly one entry in the docid and *tf* maps, while there can be more than one entry in the term positions map for a document; the number of entries in a term's positions buffer for a document is equal to the term's frequency in that document. In order to construct a non-positional or a boolean index, we can simply disable the term positions and/or term frequency buffer maps.

Since term ids increase monotonically, we can implement a buffer map using an array of pointers. In this array, each index position corresponds to a term id and the pointer points to the memory location of the associated buffer. We dynamically expand the array to accomodate more terms as needed.

When a new document arrives, we assign a unique document id (docid) to the document—docids increase monotonically. The indexing algorithm parses and groups the document's terms to collect term frequencies and term positions (relative to the current document). The algorithm inserts these data into the appropriate buffer maps. When encountering a new term, the algorithm allocates a new buffer per buffer map and places a pointer to its memory location into the buffer map. Initially, the size of a buffer is set to b, which is the block size used by the compression algorithm. Following best practice today, we use PFOR [41, 43], with the recommended block size of $b = 128$. To accomodate more positions, we expand buffers in the term positions map one block at a time. When the docid buffer for a term fills up, we compress all docids, term frequencies, and term positions into what we call a segment, before we flush the resulting segment to the current end of the segment pool.

Each segment begins with a compressed list of document ids; call this D. Note that, since document ids increase monotonically as new documents arrive, document ids in the docid buffer map are sorted in ascending order. This allows us to use gap compression techniques such as PFORDelta. Next comes a compressed list of term frequencies; call this F. Unlike document ids, term frequencies in the *tf* buffer map are not sorted. Therefore, we use PFOR to compress term frequency values in their raw form. By design, docids and term frequencies occupy exactly one PFOR block each. A segment ends with a compressed list of term positions. For every document, term positions are gap-encoded relative to the first position. For example, in Figure 3.2 term A was found at positions $[1, 3, 5]$ in document 508 and at positions $[3, 4, 8]$ in document 509. When compressing term positions, we encode the positions in these two documents as $[1, 2, 2, 3, 1, 4]$. We can unambiguously reconstruct term positions using term frequency values. Term position buffers are likely longer than the block size, b. Therefore, we split the encoded term positions into multiple blocks and compress each block separately; call the blocks of term positions $P_1 \dots P_m$. Finally, we pack D, F, and P_i's together into a contiguous block of memory to form a segment. The structure of a segment, therefore, is as follows:

$$[\ |D|,\ D,\ |F|,\ F,\ \{|P_i|,\ P_i\}_{0 \le i < m}]$$

where the $|\cdot|$ operator returns the length of its argument. Since all the data are packed into one single array, we need to explicitly store the length of each compressed block to properly decode the data during retrieval.

Each segment is stored at the current end of the segment pool, which contains the compressed inverted index. Conceptually, the segment pool is an infinitely large array, but in practice we allocate the pool in large blocks and dynamically expand it as needed. A pointer keeps track of the current end of the segment pool. To be able to read a term's postings list, we need to record the position within the segment pool where the term's first segment is stored. We refer to this position as the "head" of a postings list, and add its address to the dictionary as a "head" pointer. Since segments of a postings list are written to the segment pool in a discontiguous way, we also need to link them together in order to traverse a term's postings list (e.g., during query evaluation). As such, we prepend each segment with the address of the next segment in the chain. This means that every time we insert a new segment for a term, we have to go back and correct the "next pointer" for the last segment. We leave the next pointer blank for a newly-inserted segment to mark the end of the postings list for a term; this location is stored as the "tail pointer" in the dictionary.

This indexing design has several practical implications. On the positive side, by packing all data contiguously into segments and managing memory allocation manually (i.e., the segment pool), we can utilize memory efficiently. This approach guarantees that the indexing algorithm will not lead to heap fragmentation. In addition, as opposed to flushing the entire in-memory postings to the inverted index as in previous work, our indexer only flushes data associated with the term id whose buffer has reached capacity. On the negative side, postings lists are stored discontiguously. Therefore, traversing postings boils down to chasing pointers accross the heap with an unpredictable access pattern. As a result, in the process of traversing postings, the processor is likely to encounter a cache miss whenever it reaches the end of a segment. However, since PFOR is a block-based compression scheme and decompresses data in blocks regardless of the storage scheme, the extent of this impact is not immediately clear. We will return to discuss this in Section 3.1.2 and measure the impact of contiguity on retrieval speed in our experiments.

Finally, we note that a large portion of the vocabulary consists of rare terms (i.e., the vocabulary follows a Zipfian distribution). Therefore, allocating a buffer of length b for every new term likely leads to a large waste of memory resources. To deal with this problem, when encountering a new term we first allocate a relatively small buffer. We continue to buffer postings for new documents until the buffer fills up. When that happens, we re-allocate a buffer of length b for that term. The length of the initial buffer is set to what we refer to as a document frequency threshold. Terms with a document frequency (*df*) less than the *df* threshold are discarded in the end. In our implementation, we set the *df* threshold to 10. This two-step process reduces memory usage substantially.

3.1.2 Segment Contiguity

It is clear that our baseline indexing algorithm generates discontiguous inverted list segments. In order to create contiguous inverted lists, we would need an algorithm to re-arrange the segments once they are written to the segment pool. One approach to accomplish this is to follow the merge update techniques of disk-based indexes (see Section 2.1 for details). That is, we would need to merge multiple discontiguous segments that belong to a term's postings list and transfer them to another location in memory. Alternatively, we can use an over-allocation policy as in many in-place update techniques. These approaches have been thoroughly studied in the literature, however, when it comes to in-memory indexes we argue that the issues become more complex. This is in part because we do not have an intermediate abstraction of the inverted file. Therefore, the algorithm must explicitly manage memory addresses—which requires a custom implementation of memory management mechanisms similar to `malloc` and `free`.

Devising and implementing a memory management system is a non-trivial task. Therefore, before we further complicate our indexing approach, we first study the impact of postings lists contiguity on indexing and retrieval speed. The gold-standard in such a study is a fully-contiguous inverted index. To construct a contiguous index, let us assume that we have an oracle that tells us exactly how long each inverted list is going to be. Using this oracle, we can lay out an inverted list's segments end-to-end with no discontiguity. In practice, we first build the inverted index as normal and then post-process the inverted index to lay out segments of a postings list contiguously. We acknowledge that this solution for building a fully-contiguous index is not practical in maintaining an index for streaming documents, however, this simple experiment allows us to measure the ideal performance. The query evaluation speed using a fully-contiguous index provides an upper bound, while our indexing algorithm in its original form (Section 3.1.1) provides the lower bound on query evaluation speed.

In order to achieve a middle ground, we developed a simple approach to approximate contiguous postings lists. Instead of moving compressed segments around after they have been added to the segment pool, we change the memory allocation policy for the buffer maps. As an extreme, it is easy to see how we could build a fully-contiguous inverted index if buffers in the buffer maps had an infinite capacity—in which case, we would allocate all postings in an uncompressed form and flush to the segment pool when there is no document left to index. However, it is not necessary to consider such extremes.

In our strategy, whenever the docid buffer for a term fills up (and is subsequently written to the segment pool), we expand that term's docid and *tf* buffers by a factor of two. This means that after the first segment of a term is flushed, new docid and *tf* buffers of length $2b$ replace the old ones; after the second flush, the buffer size increases to $4b$, and then $8b$, and so on. When a buffer of size $2^m b$ fills up, the buffer is broken down to 2^m segments, each segment is compressed as described earlier, and all 2^m segments are written at the end of the segment pool contiguously. This strategy allows long postings to become increasingly contiguous.

To prevent buffers from growing indefinitely, we set a cap on the length of docid and *tf* buffers. That is, if the cap is set to $2^m b$, then when the buffer size for a term reaches that limit, it no longer is expanded. This means that the maximum number of contiguous segments allowed in the segment pool is 2^m. We experimentally show that for relatively small values of m, around 6 or 7, we achieve query evaluation speeds that are statistically indistinguishable from having an index with fully-contiguous inverted lists. However, there is a tradeoff between contiguity and memory usage as increasing m translates into larger memory requirements to hold buffer maps. We experimentally measure the additional memory requirements. In Section 3.4, we consider alternative designs and discuss why we chose this approach.

3.2 Experimental Setup

We performed our experiments on two standard web collections: Gov2 and ClueWeb09. The Gov2 collection is a crawl of .gov sites from early 2004, containing about 25 million pages, totaling 81GB compressed. ClueWeb09 is a best-first web crawl from early 2009. The collection contains one billion pages in ten languages totaling around 25 TB. Of those, about 500 million pages are in English, divided into ten roughly equally-sized segments. Our experiments used only the first English segment, which has 50 million documents (totaling 1.53 TB uncompressed, 247 GB compressed).

For evaluation purposes, we used two sets of queries: the TREC 2005 terabyte track "efficiency" queries, which consists of 50,000 queries total;[1] and a set of 100,000 queries sampled randomly from the AOL query log [109], which contains approximately 10 million queries. Our sample preserves the original query length distribution. Figure 3.3 shows the query length distributions for these query sets.

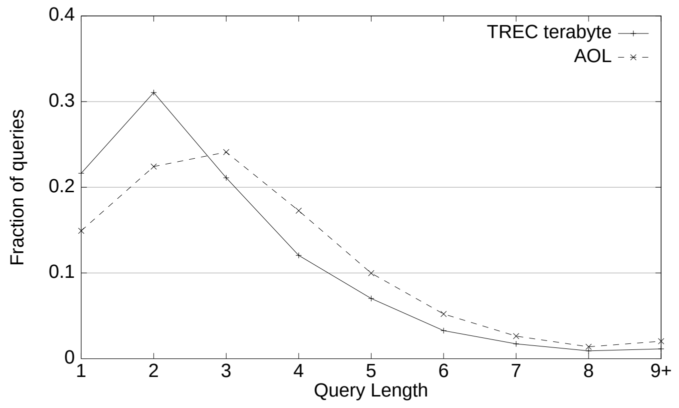


Figure 3.3: Query length distribution for the AOL and the TREC terabyte queries.

We implemented our indexer in C, which is available as an open-source package, Zambezi.[2] It is currently single-threaded. Since this chapter is focused on indexing, we wish to separate document parsing from the actual indexing. Therefore, we assume that input test collections are already parsed, stemmed, with stopwords removed before they are provided to the indexer. As such, our reports of indexing speed does not include document pre-processing time.

We performed our experiments 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. For experimentation purposes, we had exclusive access to the machine with no competing processes.

We examined three aspects of performance: memory usage, indexing speed, and query evaluation latency. The first two are straightforward, but we elaborate on the third. For each indexing

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

[2]`http://zambezi.cc`

configuration, we measured query evaluation speed in terms of query latency for two retrieval strategies: conjunctive retrieval using the SvS algorithm, demonstrated by Culpepper and Moffat [55] to be the best approach to postings intersection, and disjunctive query processing using the WAND algorithm [13], which represents a strong baseline for top k retrieval (with BM25). These baseline algorithms were detailed in Section 2.2. Note that for both cases we first indexed the collection, and then performed query evaluation at the end—the interleaving of indexing and retrieval operations is beyond the scope of this work, since it involves tackling concurrency challenges. Further note that, to compute per-query latency, we measured the elapsed time to process all queries and divided by the number of queries. We conducted multiple trials following this procedure to compute confidence intervals on mean query latency. All timing results in the remainder of this dissertation were performed in the same manner.

As a reference point, we compared our approach against two open-source retrieval engines: Zettair[3] (v0.9.3), which implements the geometric partitioning approach of Lester et al. [110] and Indri[4] (v5.1). To ensure a fair comparison with other systems, we disable their document parsing phase and used the already parsed documents as input. As with our algorithms, reports of indexing speed do not include time spent on document pre-processing. In this work, we do not intend to present a query latency-wise comparison of disk-based (e.g., Zettair) and in-memory indexes, since previous studies have shown improvements the latter can bring to the table [6].

3.3 Results

3.3.1 Query Latency

Table 3.1 summarizes query latency for conjunctive query processing (postings intersection with SvS). The average latency per query is reported in milliseconds across five trials along with 95% confidence intervals. Each column shows different indexing conditions: 1b is the baseline algorithm presented in Section 3.1.1 (linked list of inverted list segments). Each of $\{2, 4, 8 \ldots 128\}b$ represents

[3]`http://www.seg.rmit.edu.au/zettair/`

[4]`http://www.lemurproject.org/indri/`

Table 3.1: Average query latency (in milliseconds) for postings intersection using SvS with different buffer length settings. Results are averaged across 5 trials, reported with 95% confidence intervals.

	Query	$1b$	$2b$	$4b$	$8b$	$16b$	$32b$	$64b$	$128b$	Contiguous
Gov2	Terabyte	14.4 (0.2)	14.2 (0.1)	13.9 (0.1)	13.6 (0.1)	13.3 (0.1)	13.2 (0.1)	13.1 (0.1)	13.1 (0.1)	13.1 (0.1)
	AOL	20.2 (0.4)	19.7 (0.1)	19.3 (0.2)	19.0 (0.3)	18.8 (0.3)	18.7 (0.5)	18.4 (0.2)	18.3 (0.1)	18.2 (0.2)
Clue	Terabyte	49.7 (0.2)	47.1 (0.1)	45.9 (0.4)	44.4 (0.5)	42.9 (0.4)	42.0 (0.3)	41.6 (0.1)	41.6 (0.4)	41.3 (0.1)
	AOL	87.5 (1.6)	83.2 (0.5)	80.7 (0.3)	75.5 (0.5)	75.7 (0.8)	75.8 (0.3)	75.2 (0.2)	75.0 (0.6)	75.3 (1.2)

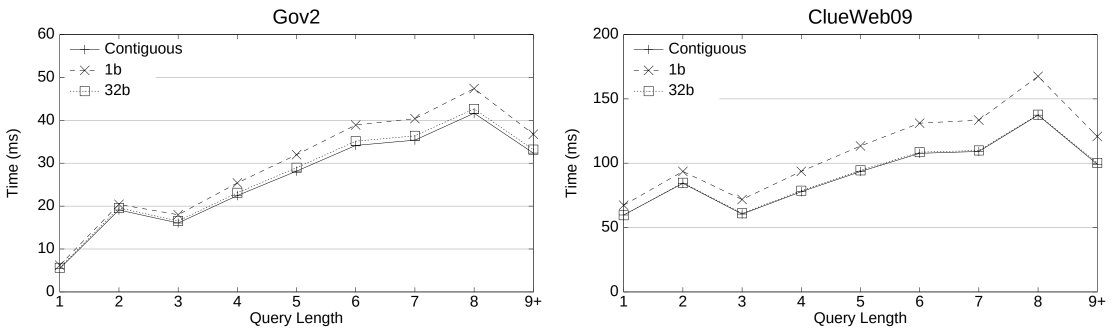


Figure 3.4: Query latency using SvS for the AOL query set, broken down by query length for different buffer length settings.

a different upper bound in the buffer map growing strategy described in Section 3.1.2. The final column marked "contiguous" denotes the oracle condition in which all postings are contiguous; this represents the ideal performance.

From these results, we see that, as expected, discontiguous postings lists ($1b$) yield slower query evaluation: on Gov2, queries are approximately 10% slower, while for ClueWeb09, the performance decreases by 16% to 20%. At $32b$, query evaluation performance is statistically indistinguishable from the performance upper bound (i.e., their confidence intervals overlap). That is, we only need to arrange inverted list segments in relatively small groups of 32 to achieve ideal performance. Later, we discuss the memory requirements of allocating larger buffer maps.

Figure 3.4 illustrates query latency by query length, for the AOL query set on Gov2 and ClueWeb09, using different conditions. Not surprisingly, the latency gap between contiguous and the $1b$ condition widens for longer queries. On the other hand, the difference between a contiguous index and the $32b$ condition is indistinguishable across all query lengths.

For disjunctive query processing, we used the WAND algorithm to retrieve the top 1000 documents using BM25. Table 3.2 summarizes these experiments on different collections and

Table 3.2: Average query latency (in milliseconds) to retrieve top 1000 hits in terms of BM25 using WAND (5 trials, with 95% confidence intervals).

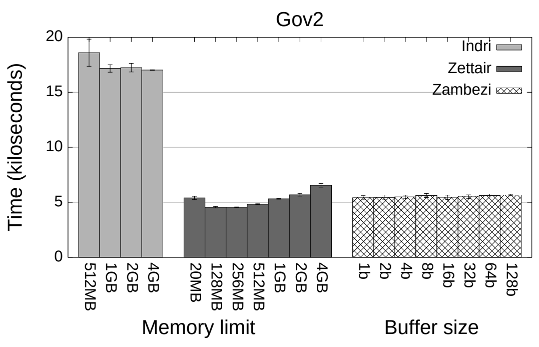
	Query	$1b$	$32b$	Contiguous
Gov2	Terabyte	65.0 (±0.4)	62.5 (±0.8)	62.0 (±0.4)
	AOL	103.5 (±0.5)	100.3 (±0.1)	100.2 (±0.4)
Clue	Terabyte	150.0 (±0.5)	141.1 (±0.6)	141.1 (±0.2)
	AOL	455.7 (±5.1)	434.3 (±5.8)	432.6 (±4.9)

queries. We only report results for select buffer length configurations. These results are consistent with the conjunctive processing case. A maximum buffer size of $32b$ yields query latencies that are statistically indistinguishable from a contiguous index. Note that the performance difference between fully-contiguous postings lists and $1b$ discontiguous postings lists is less than 7%. In other words, there is much less performance degradation than in the SvS case. We will explain this behavior in Section 3.4.

3.3.2 Indexing Speed

Figure 3.5 illustrates indexing times for our indexer, Zettair, and Indri. We can adjust the amount of memory provided to Zettair and Indri through different parameter settings. Due to its implementation, Zettair is unable to consume more than 4GB of memory—the amount of memory to be used is represented by a 32-bit integer. For our indexer, we report results with different postings lists contiguity settings. Error bars show 95% confidence intervals across three trials.

Increasing the available memory size appears to have little or no impact on Indri's indexing speed. Overall, Indri is slower than both Zettair and our indexer. On the other hand, we observe a significant change in Zettair's performance as we tune the amount of available memory. However, providing more memory to Zettair counter-intuitively slows down the indexing algorithm. The reason behind this, we believe, is that smaller in-memory postings lists are more cache-friendly. For exmaple, since our system has a 12MB L3 cache, in the 20MB condition, more than half of the in-memory postings will reside in cache. The downside of smaller segments, however, is that indexing will require more merge operations. Based on our experiments, the first factor (i.e., cache-friendliness of smaller segments) seems to lead to a gain larger than the loss in speed from



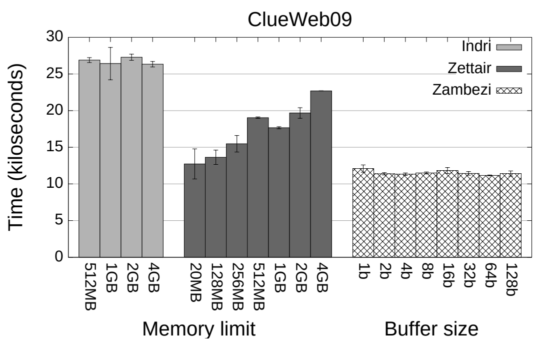


Figure 3.5: Indexing speed for Indri and Zettair with different memory limits, and our indexer with different contiguity conditions on Gov2 and ClueWeb09. Error bars show 95% confidence intervals across 3 trials.

the second factor (i.e., more merges). For ClueWeb09, indexing is fastest with 20MB buffers. For Gov2, increased cache performance is not sufficient to compensate for additional time spent on merging, but the optimal balance occurs with 128MB buffers, which is still relatively small.

Finally, Figure 3.5 shows that our in-memory indexing algorithm is not substantially faster than an on-disk algorithm. In fact, for some conditions on Gov2 our indexer is actually slower than Zettair. This is not surprising since the state-of-the-art on-disk indexing techniques are a product of decades of research; the implementations have been highly tuned to the characteristics of disks. In addition, as the Zettair results suggest, an increase in memory footprint leads to less cache locality—as more data compete over cache. This in turn slows down the indexing algorithm. In contrast, merge operations for on-disk indexes access data in a sequential pattern, which creates an opportunity for the processor to pre-fetch data and mask memory latencies. To test this hypothesis, we ran Indri with a memory size of 120GB on Gov2, and the indexer took 38k seconds to complete, which is roughly double the times reported in Figure 3.5. This result appears to support our analysis.

Finally, we note that there are other components in an indexer that independently impact end-to-end performance. The choice of compression schemes—we use PFOR whereas Zettair does not—and dictionary—we use a hash table whereas Zettair uses a B+ tree—are two such factors. However, the research question we are trying to answer is the impact of postings lists contiguity on query evaluation and indexing speed. Zettair and Indri merely contextualize our findings.

3.3.3 Memory Usage

Incremental indexing algorithms all require some amount of memory to hold intermediate data. Previous work on on-disk indexing have assumed that the available memory is relatively small. In our work, on the other hand, we set no limit on the amount of available memory. However, the space we allocate to hold intermediate data takes away space that can be used to hold the final compressed inverted index. This "waste" in memory determines the maximum size of the collection that we can index given a fixed server configuration.

By default, the buffer maps in our indexer hold the most recent b document ids, term frequencies, and term positions. In this work we set b to 128 as guided by previous work on PFOR compression. In order to increase the contiguity of the inverted list segments, we increase the length of the buffers, as described in Section 3.1.2. This, in turn, increases the space requirements of the buffer maps.

Figure 3.6 shows the maximum size of the buffer maps for different contiguity configurations, broken down by space devoted to docids, term frequencies, and term positions. The reported values were computed analytically from term statistics, making the assumption that all terms reach their maximum buffer size at the same time, which makes these upper bounds on memory usage. To make comparison across the two collections easier, we normalize the values with respect to the 1b condition. For the 1b condition, the total buffer map sizes are 12.6GB for Gov2 and 22.1GB for ClueWeb09. At 128b, where we allow the buffer to grow to 128 blocks of 128 32-bit integers, the algorithm requires 71% more space for Gov2 and 95% more space for ClueWeb09 (compared to

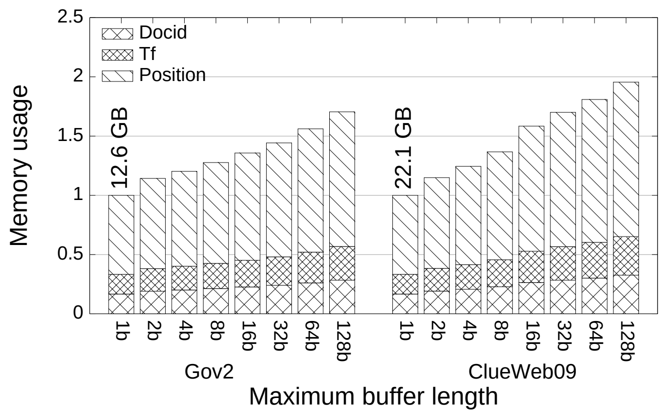


Figure 3.6: Memory required to hold all buffer maps for different buffer length settings, normalized to the 1b setting, on Gov2 ClueWeb09.

the $1b$ condition). At $32b$, which from our previous results achieves query evaluation performance that is statistically indistinguishable from contiguous postings lists, we require 44% and 70% more memory for Gov2 and ClueWeb09, respectively.

As reference, the total size of the segment pool (i.e., size of the final index) is 31GB for Gov2 and 62GB for ClueWeb09. This means that on the Gov2 collection, setting the maximum buffer length to $1b$, $32b$ and $128b$ results in a buffer map that is approximately 41%, 59%, and 69% of the overall size of the segment pool, respectively. Similarly, for ClueWeb09 the buffer map sizes are approximately 32%, 54%, and 63% of the size of the segment pool, respectively. These statistics quantify the overhead of our in-memory indexing algorithms.

Note that most of the working memory is taken up by term positions; in comparison, the requirements for buffering docids and term frequencies are relatively modest. In all cases the present implementation uses 32-bit integers, even for term positions. We could easily cut the memory requirements for those in half by switching to 16-bit integers, although this would require us to either discard or arbitrarily truncate long documents. Ultimately, we decided not to sacrifice the ability to index long documents.

The total number of unique terms is 31M in Gov2 and 79M in ClueWeb09. Since these collections consist of web pages, most of the terms are unique and correspond to JavaScript fragments that our parser inadvertently included and other HTML idiosyncrasies; such issues are prevalent in web search and HTML cleanup is beyond the scope of this work. Our indexer discards terms that occur fewer than 10 times, which results in a vocabulary size of 2.9M for Gov2 and

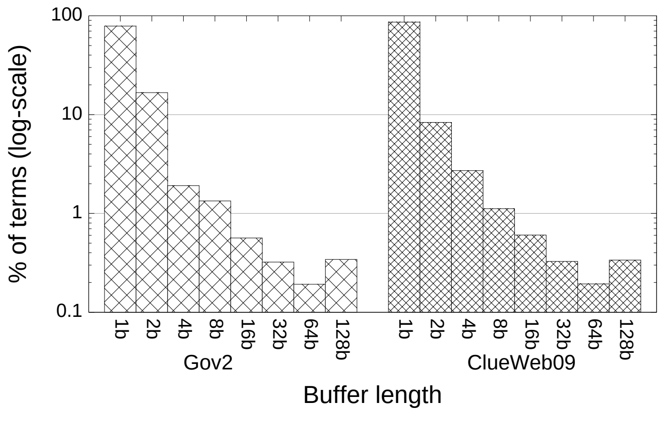


Figure 3.7: Percentage of terms for which a buffer of length $m \times b$ is required, for different values of m, and block size $b = 128$.

6.9M for ClueWeb09. Of these, Figure 3.7 shows the percentage of terms that require a maximum buffer length of $m \times b$, for different values of m in our contiguity settings. For example, the $1b$ bar represents terms whose document frequencies are ≥ 10 but < 128. The $2b$ bar represents terms whose document frequencies are ≥ 128 but less than $1b + 2b = 384$, and so on. The $128b$ bar represents terms whose document frequencies exceed the maximum buffer length of 128 blocks. From this we can see why significantly increasing the b value only yields a modest increase in memory requirements.

3.4 Discussion

We can summarize our findings as follows: we observe that a linked list of inverted list segments leads to query latencies that are significantly slower than what a fully-contiguous inverted index (i.e., an ideal condition) would yield. However, the performance gap is not significant if we organize segments in groups of 32 through increasing buffer sizes. Thus, contiguity is an important factor but only to a point.

There are two major factors that contribute to this phenomenon. First, PFOR decompression masks memory latency associated with pointer chasing in traversing postings lists. Since we read and decompress postings in blocks—as opposed to reading individual postings one by one—the processor can hide cache misses by dispatching memory requests for the next segment while decompressing the current block of postings. These instructions can be executed out of order due to their independence from one another. Second, query evaluation is more complex than a simple linear scan of postings lists; the SvS algorithm performs a binary (galloping) search and WAND uses pivoting to skip over unpromising postings lists. This behavior results in unpredictability in memory access patterns which reduces the opportunity to pre-fetch data. To illustrate this, consider the difference between ideal performance and the $1b$ baseline condition: the performance gap is much smaller for WAND than for SvS. This makes sense, since at each stage, SvS is intersecting the current postings list with the working set: this implies more predictable access patterns, hence greater cache locality, so we obtain a bigger performance boost with contiguous postings list. On the other hand, WAND pivots from term to term and, at each step, may advance the current pointer by an unpredictable amount; thus, even if the postings list are contiguous, the processor may still encounter cache misses. Thus, it makes less of a difference if the postings lists are discontiguous to begin with.

The downside of our approach is that our algorithm needs to set aside working memory to buffer intermediate data. Increasing the buffer sizes means we have less space to hold the final compressed index. This tradeoff limits the size of the collection that we can handle. However, in practice, this is not an issue. Search engines partition the entire document collection into small subsets and assign each partition to individual servers [111]. Many different factors determine the size of these partitions. These factors include processor speed and memory latencies which directly impact query evaluation speed. However, the maximum possible partition size is dictated by the amount of memory available. Since memory capacities are growing at a much faster pace than the speed of processors and improvements in memory latencies, the overhead of our indexing approach will become less of a concern over time.

As an alternative to increasing the size of the buffer maps to increase postings list contiguity, we could use over-allocation policies similar to in-place update techniques. We had considered this approach, but decided not to pursue it for several reasons. First, because of our choice of a block compression (PFOR), there is greater potential for waste with over-allocation. For example, if we only reserved 800 bytes but the next inverted list segment occupies 801 bytes, then the entire reserved memory is left unused—this leads to under-utilization of memory resources. Second, over-allocation policies inevitably result in fragmentation in the segment pool unless we re-organize data by moving them around—which requires an implementation of a garbage collector. In contrast, our indexing approach does not result in “holes” in the segment pool. At the same time, the space used for buffering postings is transient. That is, it is freed after the indexing process is complete. For these reasons, we opted to not utilize over-allocation and instead explore a buffering approach.

Chapter 4

Approximate Top k Retrieval for Candidate Generation

As discussed earlier, we study the stages of a learning-to-rank retrieval pipeline in isolation, while introducing techniques to make each stage more efficient. The first stage is candidate generation, which, as illustrated in Figure 4.1, takes a query as input and returns a list of documents that are potentially most relevant to the given query. We refer to these documents as candidates. The candidates are ranked using a “cheap” ranker (e.g., timestamp) and are forwarded to the next stages. This chapter focuses on the candidate generation stage, assuming an inverted index that has already been created using our indexer from Chapter 3.

We first introduce a novel variant of Bloom filters [18] that can automatically expand in an efficient manner in order to accommodate an unbounded stream of sorted integers. Furthermore, this data structure guarantees a theoretical upper-bound on the false positive rate and offers fast membership tests. These properties make our scalable variant of Bloom filters suitable for storing document ids in a streaming setting.

Using this data structure, we design a novel approximate top k retrieval algorithm that, given a query, efficiently retrieves the k documents that are potentially most relevant. We designed this algorithm primarily for search on microblog documents (such as tweets). Our algorithm takes advantage of two unique characteristics of microblog documents; namely, that every document has a limited length and bears a timestamp. As such, the scoring function and early-termination

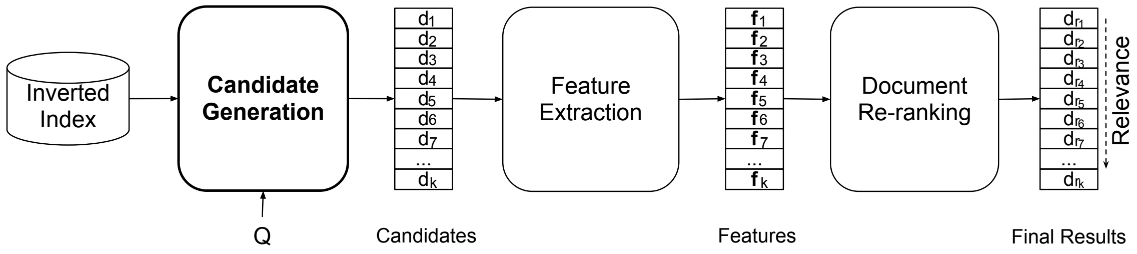


Figure 4.1: Learning-to-rank pipeline.

strategies in our algorithm are specific to microblog documents and do not generalize to web documents. Our experiments demonstrate that our top k algorithm outperforms strong baselines in terms of speed, with a small loss in effectiveness.

4.1 Bloom Filter Chains

A Bloom filter, as introduced in Section 2.3, supports two operations: insert and membership test. We discussed that membership tests are not exact and, in fact, produce false positives. That is, a membership test may return true even though the Bloom filter does not actually contain the test element. However, we can guarantee a theoretical upper-bound on the false positive rate of a Bloom filter by appropriately adjusting the filter's parameters: κ (number of hash functions) and r (number of bits allocated per element). Once a Bloom filter is constructed and space is allocated, we are no longer able to modify the filter's parameters. Therefore, to guarantee a target upper-bound on the false-positive rate, we need to know the exact number of elements that will be inserted into the filter in advance.

Due to this requirement, standard Bloom filters are not suitable in situations where the cardinality of the set of elements is not known in advance, and the number of elements constantly increases; we have no way of determining the right amount of space to guarantee a false-positive upper-bound.

To resolve this problem, we introduce Bloom filter chains which are extensions of standard Bloom filters and are designed to dynamically expand to represent any *sorted* list of integers with a guaranteed false positive rate, without requiring a prior knowledge of the list's length. This data structure addresses the shortcomings of previous work on scalable Bloom filters. With the assumption that elements are inserted into the filter in a sorted order, Bloom filter chains require minimal memory overhead (i.e., the amount of space not directly used to store elements), and provide efficient membership tests.

The general idea behind Bloom filter chains is similar to the work of Almeida et al. [63]. We initially create a Bloom filter of a default length. Given a target false positive rate (i.e., a pair of

r and κ, the number of bits per element and the number of hash functions, respectively), we can calculate the maximum number of elements that the Bloom filter can safely accommodate. Once a Bloom filter reaches its capacity, we construct another Bloom filter and link it to the last filter. In the end, we will have a linked list of Bloom filters, each containing disjoint partitions of the sorted list.

Despite some similarities, our work is different from the work of Almeida et al. in two ways. First, in our approach, the sizes of Bloom filters in the chain are arbitrary while in their work [63], the sizes of Bloom filters follow a geometric progression which is necessary to guarantee a target false positive rate. Second, in [63] a probe operation requires a membership test on *all* Bloom filters in the series. However, in our case, because we assume elements are inserted into the filter in sorted order, each Bloom filter in the chain contains a non-overlapping range of elements. That is, if the n_{th} Bloom filter contains elements in the range $[\alpha_n, \beta_n]$, where α_n and β_n indicate the first and last elements inserted into the Bloom filter respectively, then $\beta_n < \alpha_{n+1}$. This means that we can compare a test element with the range associated with each Bloom filter to determine whether the queried element might have been inserted into that filter; these range checks tell us the right Bloom filter to probe, and thus we can avoid unnecessary membership tests on other Bloom filters in the chain. To enable this comparison, we must keep track of α_n and β_n for every Bloom filter in the chain. However, since $\alpha_n < \beta_n$ and because $\beta_n < \alpha_{n+1}$, storing α_n's alone suffices—we need to explicitly store the first element that is inserted into each Bloom filter in the chain.

Like standard Bloom filters, Bloom filter chains support insertion and membership tests. Let us formalize these operations as follows:

- INSERT(d) inserts an integer d into the Bloom filter chain,
- PROBE(d) performs an approximate membership test to determine if the Bloom filter chain contains element d.

Insertion of elements into a Bloom filter requires computing κ hash values. The hash computations must be fast and the hash values must distribute the keys as evenly as possible.

Algorithm 1: PROBE(d)

```
Input: $d$ – test element
Output: true if the Bloom filter chain contains the test element, false otherwise

  $n \leftarrow$ TAILPOINTER
  while $d < \alpha_n$ AND HASPREVIOUS($n$) do
    $n \leftarrow$ PREVIOUS($n$)
  end while
  return $B_n$.CONTAINS($d$)
```

In our implementation, we construct an arbitrary number of hash functions using Jenkin's integer hash, of the form $h(x, S) \mod L$, where S is a seed and L is the length of the Bloom filter. For $\kappa = 1$, we simply use a large prime number as the seed. For $\kappa > 1$, we compute the n_{th} hash value by setting the seed to $h^{n-1}(x, S)$.

Next, we describe the implementation of the PROBE operation (see pseudo-code in Algorithm 1). Let us assume that we have a tail pointer stored along with the data structure which contains a pointer to the last Bloom filter in a Bloom filter chain; call this n. Testing for membership begins by comparing the test element d with α_n—the first document id that was inserted into the Bloom filter pointed to by n. There are three possible outcomes:

1. $d = \alpha_n$: the membership test passes.

2. $d > \alpha_n$: this means that d must have been inserted into B_n—the Bloom filter pointed to by pointer n. In this case, we probe B_n to determine if d is a member.

3. $d < \alpha_n$: it is clear that if d had been inserted at all, it must be contained in an earlier Bloom filter in the chain. In this case, we follow the link to the previous filter and repeat this comparison. When we reach the first Bloom filter, we probe the filter regardless of d.

In theory, PROBE is an $O(n)$ operation, where n is the length of the Bloom filter chain. However, in practice only a few range checks are necessary to find the correct Bloom filter to probe, which is an $O(1)$ operation. We will return to discuss this in more detail in Section 4.2.

We use Bloom filter chains to complement traditional inverted lists; every term has a standard postings list as well as a Bloom filter chain associated with it. In the Bloom filter chain associated with a term, we store document ids (docid) of documents in which that term appears. In order to integrate Bloom filter chains in our index structure from Chapter 3, we do the following: Once a docid buffer fills up, we proceed to pack the data into an inverted list segment as explained previously. However, while compressing a term's docids and placing them into a segment, we construct a Bloom filter using parameters r and κ, and subsequently insert all the docids into the filter. Before writing the segment in the segment pool, we append the segment with the Bloom filter; call this B. This way, each segment will contain a Bloom filter in addition to docids and other data. To support efficient membership tests, we explained that we would have to keep track of the first element that is inserted into a Bloom filter. Therefore, we explicitly include the first docid in inverted list segments; call this α. As a result, our new inverted list segments will have the following structure:

$$[\ |D|,\ D,\ |F|,\ F,\ (\{|P_i|,\ P_i\}_{0\leq i<m}),\ B,\ \alpha]$$

where the $|\cdot|$ operator is the length of its argument; D, F, and P_i's are compressed docid, *tf*, and blocks of term positions respectively (see Chapter 3 for details). Since the length of B can be easily calculated given $|D|$ and Bloom filter parameter r, we do not need to explicitly store $|B|$.

4.2 Efficient Candidate Generation with Bloom Filters

In this section, we introduce our novel top k retrieval algorithm that takes advantage of Bloom filters chains. Using this data structure to complement traditional inverted lists, we describe a novel variant of the WAND algorithm (see Chapter 2 for a description), called BWAND (short for Bloom WAND), that is many times faster than the WAND algorithm and, like WAND, is capable of performing conjunctive and disjunctive query processing.

Before we explain the details of BWAND, recall that a top k retrieval algorithm requires a scoring function that assesses the relevance of a document with respect to an input query. As mentioned before, the scoring function should be fast but also effective at distilling the document collection to find more relevant documents. In this work, we use a mixture of time and a simplified BM25 function (as a query-dependent scoring function).

In the original BM25 formulation, the score of a document D with respect to a query is defined as follows:

$$\textsc{Score}_{\text{BM25}}(Q, D) = \sum_{q \in Q} \frac{(k_1 + 1) \cdot \text{tf}(q, D)}{K + \text{tf}(q, D)} \log \left[\frac{N - \text{df}(q) + 0.5}{\text{df}(q) + 0.5} \right] \tag{4.1}$$

where $\text{tf}(q, D)$ is the term frequency of query term q in document D, $\text{df}(q)$ is the document frequency of q, and N is the number of documents in the collection. K is defined as $k_1[(1-b) - b \cdot (|D|/|D|')]$, where $|D|$ is the length of document D and $|D|'$ is the average document length. Finally, k_1 and b are free parameters.

There are three components that contribute to a BM25 score: a term frequency factor (the first term in the summation above), an inverse document frequency (Idf) factor (the second term in the summation), and a length normalization factor (K). However, since we primarily focus on collections of microblog documents with limited length (tweets), length factorization should not have much of an impact. Additionally, as we will empirically verify, term frequency is almost always one—thereby, this factor too should not matter. Therefore, a BM25 function can be reduced to its Idf factor. In other words, if a query term is contained in a document, we simply assume that its term frequency is one and ignore document length normalization. More specifically, we define the Idf scoring function as follows:

$$\textsc{Score}_{\textsc{Idf}}(Q, D) = \sum_{q \in Q \cap D} \log \left[\frac{N - \text{df}(q) + 0.5}{\text{df}(q) + 0.5} \right] \equiv \sum_{q \in Q \cap D} \textsc{Idf}(q) \tag{4.2}$$

These simplifications allow us to substantially increase the speed of candidate generation. We examine the impact of these assumptions on search quality later in this work.

There is a hidden component in our scoring function: time. Recall that, our indexing algorithm indexes documents in order of arrival (see Chapter 3) and assigns document ids that increase monotonically. In this way, a larger document id indicates a more recent document. As a result, documents in a postings list are not only sorted by their document ids, but they are also implicitly sorted in chronological order. As such, walking down a postings list from its tail translates into a reverse-chronological traversal of documents—a desirable ordering in real-time search.

Algorithm 2: BWAND(Q, k, ω)

```
**Input:** $Q$ – query consisting of $|Q|$ terms
**Input:** $k$ – number of documents to retrieve
**Input:** $\omega$ – conjunctive/disjunctive tuning parameter
**Output:** top $k$ documents ranked by IDF scoring model
  $H \leftarrow$ new Heap$(k)$
  $\theta \leftarrow \omega \times \sum_{q_i \in Q} \text{IDF}(q_i)$
  $q_b \leftarrow \text{argmin}_i[\text{DICTIONARY.DF}(q_i), q_i \in Q]$
  $P_b \leftarrow \text{GETPOSTINGS}(q_b)$
  while $P_b$.HASNEXT() AND $\theta \neq \sum_{q_i \in Q} \text{IDF}(q_i)$ do
    $d \leftarrow P_b$.GETDOCID()
    $s \leftarrow \text{IDF}(q_b)$
    for $q_i \in Q - q_b$ do
      if GETBLOOMFILTERCHAIN$(q_i)$.PROBE$(d)$ = TRUE then
        $s \leftarrow s + \text{IDF}(q_i)$
      end if
    end for
    if $s > \theta$ then
      $H$.INSERT$(\langle d, s \rangle)$
      if $H$.ISFULL() then
        $\theta \leftarrow H$.MINSCORE()
      end if
    end if
  end while
  return $H$
```

Now that we defined our scoring function, let us formally describe our BWAND algorithm, the pseudo-code of which is shown in Algorithm 2. The input to our algorithm is a query Q, the number of hits k, and a parameter ω that controls whether the algorithm operates in conjunctive or in disjunctive mode.

The algorithm begins by creating an empty heap of size k, and initializing a threshold θ to ω times the sum of the IDF scores of all query terms. The algorithm then identifies the rarest query term and fetches its postings list—we refer to this as the *base* postings list. Query evaluation proceeds by traversing this base postings list backwards from its tail, one document at a time. Initially, the score of a document is set to the IDF score of the rarest term. For every document in the base postings list, the algorithm then probes the Bloom filter chains associated with other query terms in an attempt to determine whether that document contains other query terms. Whenever a membership test passes, the algorithm adds the IDF score of the corresponding term to the document score. If the final score is strictly greater than θ, the document is placed in the heap. When the heap is full, every time a document is added to the heap, θ is adjusted to the minimum score in the heap (i.e., the score of the k_{th} document). It is clear that, if θ is equal to the sum of all query terms' IDF scores, then no other document can make it into the heap. This provides an opportunity to early-terminate the algorithm (see the condition of the **while** loop in Algorithm 2). The strict comparison of scores with θ guarantees that our algorithm breaks ties in favor of more recent documents, given the reverse-chronological ordering of our postings traversal.

Similar to WAND, initializing ω to 0 yields disjunctive query processing. However, our algorithm still requires the presence of the rarest term—we experimentally explore the impact of this heuristic on search quality. Whereas setting ω to $1-\epsilon$ (for a negligible ϵ) reduces the algorithm to conjunctive query processing. Like with WAND, values between these two extremes determine whether query evaluation is conjunctive-like or disjunctive-like.

In conjunctive mode, it is easy to see that the IDF scoring function has no impact on the final ranking of documents. That is, since candidate documents contain all query terms, Equation 4.2 yeilds the same score for all documents that can make it to the heap. Since the algorithm breaks ties in favor of more recent documents and because we traverse postings in reverse-chronological order, conjunctive query processing, in practice, returns the first k documents that contain all query terms. Therefore, after the algorithm accumulates k documents, no other document can find its way to the candidate set. This enables early-termination in conjunctive mode: the algorithm can terminate as soon as the heap is full.

Another optimization in conjunctive mode is to short circuit the membership tests once a PROBE fails and proceed to the next document in the base postings list. This way we can avoid unnecessary membership tests.

Additionally, PROBE can take advantage of the fact that test document ids are monotonically decreasing, because we traverse the base postings list in reverse chronological order. Therefore, assuming we probed the Bloom filter chain at position n to perform a membership test for d_i, we can begin Algorithm (1) from the same location to perform the membership test for d_{i+1}. In other words, instead of starting from the last filter in a term's Bloom filter chain each time, we can begin with the last filter probed. This optimization improves performance by reducing pointer chasing to find the right filter, and makes the PROBE operation $O(1)$ in the context of the candidate generation algorithm.

4.3 Experimental Setup

We performed our experiments on the same hardware as in the previous chapter (again, with exclusive access to the machine). In the following sections, we describe the evaluation data, implementational details of the candidate generation algorithms, and the metrics we used to evaluate our approach.

4.3.1 Test Collections and Evaluation Methodology

Our experiments used the Tweets2011 collection, created for the TREC 2011 and TREC 2012 microblog tracks [112, 113, 114, 115].[1] The collection includes approximately 16 million tweets over a period of two weeks (24th January 2011 until 8th February, inclusive), which covers both the time period of the Egyptian revolution and the US Superbowl. Different types of tweets are present, including replies and retweets, in English as well as other languages. Following standard practice, terms in tweets were stemmed and stopwords were removed prior to indexing.

Analysis of the collection reveals that term frequency is one for 96% of all term instances and is two for 3.5% of the instances. Most of the terms that appear more than once in a tweet do not appear to be important from a retrieval perspective. Examples include “rt”, “http”, “haha”, “like”, “da”, “youtub”, “video”, and top level domains (e.g., “com”, “edu”). Often, spam tweets have terms with high term frequencies due to attempts at keyword stuffing. However, cursory examination does reveal that, occasionally, terms of interest (such as “world”, “TSA”) appear multiple times in tweets.

For evaluation, we used three different sets of queries. For efficiency experiments, we used the TREC 2005 terabyte track “efficiency” queries as well as the down-sampled AOL queries, which

[1] `http://trec.nist.gov/data/tweets/`

were introduced in Chapter 3. For effectiveness experiments, we used the TREC microblog track topics from 2011 and 2012, which contain 110 queries in total. These queries comprise a complete test collection in that we have relevance judgments, but there are too few queries for meaningful efficiency experiments.

In order to evaluate the performance of our proposed candidate generation algorithm, we implemented two baselines. For conjunctive retrieval (i.e., postings list intersection) we implemented the SvS algorithm [52, 55] (see Section 2.2 for details). For disjunctive retrieval, we used WAND [13] with the IDF scoring function (see Section 4.2). We call this WAND_{IDF}. We had to adopt these algorithms such that they work on linked lists of discontiguous inverted list segments. In all cases, the candidate generation stage returned 1000 hits. Recall that the SvS algorithm must first compute the entire intersection set before returning the top k documents, ranked by time. However, for queries that contain one or two terms, we can early-terminate the SvS algorithm as soon as the algorithm accumulates k documents.

To ensure a fair comparison, we have put in a best faith effort to optimize our implementation of SvS and WAND. 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.

4.3.2 Evaluation Metrics

There are three important considerations in the design of search engines: the quality of the results, query evaluation speed (time), and memory footprint (space). Each of the algorithms explored in this paper represents a tradeoff point in this design space, which we seek to better understand. For the BWAND algorithm, these three considerations are influenced by the parameters used in the Bloom filters: r (number of bits per element) and κ (number of hash functions). Based on preliminary calculations from theoretical bounds (see Section 2.3), we restricted our consideration to $r = \{8, 16, 24\}$, and $\kappa = \{1, 2, 3\}$. Note that these values are fixed for all Bloom filters. Index size can be easily measured and will be reported as in Section 3.3.3.

Speed is measured in terms of query latency: the amount of time it takes to perform candidate generation and return a list of top $k = 1000$ documents. In conjunctive retrieval, the returned documents contain all query terms and are sorted in reverse chronological order, whereas in disjunctive retrieval, documents are sorted with respect to the scoring function. Elapsed time is reported in microseconds. We compute average latency across five trials on each query set, and also break down the results by query length. Since our current implementation is single-threaded, throughput (i.e., queries per second) is simply the inverse of latency.

Finally, it is necessary to measure both component-level and end-to-end effectiveness, since we introduce approximations that trade effectiveness for speed. At the component level, we measure effectiveness in terms of relative recall with respect to the exact baseline: SvS in the case of conjunctive query processing and WAND with full BM25 scores (WAND$_{\text{BM25}}$) in the case of disjunctive query processing. That is, of all documents retrieved by the baseline, what fraction is retrieved by our candidate generation algorithm? In the conjunctive case, this metric captures the false positive errors introduced by the Bloom filters, and in the disjunctive case, this metric additionally captures a number of simplifications and approximations in the scoring function compared to BM25. Relative recall is computed via macro-averaging (i.e., computed per topic, then averaged across topics), which has the effect of disproportionately weighting topics that have fewer matching documents. Note that since we are concerned with generating a candidate list that will be passed to a re-ranker, precision or any metric that incorporates precision such as average precision or NDCG is not an appropriate component-level measure.

The other aspect of quality is end-to-end effectiveness. As we will show in Chapter 8, approximations and simplifications introduced in our algorithm do not degrade end-to-end effectiveness.

4.4 Results

4.4.1 Component-Level Effectiveness

Our first set of experiments examines the impact of Bloom filter parameter settings on component-level effectiveness. Table 4.1 shows the relative recall of BWAND with different Bloom filter con-

Table 4.1: Relative recall for different Bloom filter chain settings, with respect to SvS for conjunctive query processing and $\textsc{Wand}_{\text{BM25}}$ for disjunctive query processing.

(a) Conjunctive

$r \backslash \kappa$	1	2	3
8	0.981	0.993	0.997
16	0.991	0.998	0.999
24	0.994	0.998	0.999

(b) Disjunctive

$r \backslash \kappa$	1	2	3
8	0.354	0.365	0.368
16	0.364	0.369	0.370
24	0.367	0.370	0.370

Table 4.2: Relative recall for different Bloom filter chain settings, considering only relevant documents, with respect to SvS for conjunctive query processing and $\textsc{Wand}_{\text{BM25}}$ for disjunctive query processing.

(a) Conjunctive

$r \backslash \kappa$	1	2	3
8	0.972	0.992	0.997
16	0.989	0.999	1.0
24	0.994	1.0	1.0

(b) Disjunctive

$r \backslash \kappa$	1	2	3
8	0.684	0.707	0.711
16	0.704	0.714	0.715
24	0.708	0.714	0.714

figuration parameters, r (bits per element) and κ (number of hash functions), for both conjunctive and disjunctive query processing. That is, of the documents retrieved by the baselines (SvS and $\textsc{Wand}_{\text{BM25}}$), we compute the fraction that is retrieved by BWand. Table 4.2 shows relative recall figures, but only with respect to relevant documents.

In the conjunctive query processing case, BWand achieves nearly perfect recall in all cases. This means that nearly all documents retrieved by the exact algorithm are also retrieved by our approximate algorithm and are subsequently available to the learning-to-rank model. BWand erroneously includes some documents that do not contain all query terms in the candidate set. Since we only retrieve the top 1000 document, false positives displace relevant candidates and do not let them in the final candidate set. If we increased k to infinity (i.e., returned the full intersection set), the Bloom filter errors would not degrade relative recall. Focusing on only the relevant documents, relative recall is still well above 97%.

In the disjunctive query processing case, relative recall with respect to $\textsc{Wand}_{\text{BM25}}$ is low, as expected. However, when we focus only on relevant documents, relative recall increases substantially. That is, although with BWand we "lose" documents that would have been retrieved under $\textsc{Wand}_{\text{BM25}}$, the lost documents have a greater tendency to be non-relevant.

Table 4.3: Average query latency (in microseconds) for the TREC 2005 terabyte track queries and the AOL queries to retrieve 1000 candidate documents, averaged across five trials, with 95% confidence intervals.

(a) Conjunctive Query Processing

	SvS	$r\backslash\kappa$	1	2	3
TREC05	172.8 (±0.2)	8	52.4 (±3.8)	61.1 (±1.6)	76.2 (±7.2)
		16	58.0 (±7.1)	58.4 (±1.4)	64.0 (±3.3)
		24	63.1 (±13.6)	67.3 (±7.6)	65.2 (±4.4)
AOL	407.1 (±1.2)	8	61.8 (±4.9)	68.0 (±0.1)	76.6 (±0.1)
		16	60.9 (±0.1)	63.8 (±0.1)	67.9 (±0.1)
		24	62.9 (±0.1)	63.5 (±0.1)	66.3 (±0.1)

(b) Disjunctive Query Processing

	WAND$_{\text{IDF}}$	$r\backslash\kappa$	1	2	3
TREC05	958.8 (±38.6)	8	94.2 (±1.6)	100.5 (±2.0)	119.1 (±1.2)
		16	103.5 (±2.3)	102.6 (±1.4)	113.5 (±2.6)
		24	127.5 (±3.3)	110.4 (±2.0)	114.4 (±1.9)
AOL	2,048.6 (±10.2)	8	128.6 (±2.2)	133.5 (±6.3)	149.4 (±1.2)
		16	144.7 (±2.9)	145.5 (±2.6)	146.9 (±1.1)
		24	157.7 (±3.6)	154.0 (±3.6)	157.3 (±3.1)

Tables 4.1 and 4.2 together show that relative recall can be controlled by appropriately tuning Bloom filter parameters. The tradeoffs are exactly what we would expect: Increasing the number of hash functions, κ, reduces the false positive rate and therefore improves relative recall. Similarly, increasing the number of bits per element, r, reduces hash collisions—thereby reducing the false positive rate and increasing relative recall. However, as we will discuss in Sections 4.4.2 and 4.4.3, larger values of κ reduce speed and larger values of r increase memory requirements.

4.4.2 Query Evaluation Speed

Thus far, we have shown that our BWAND candidate generation algorithm leads to high relative recall against the baseline algorithms. We now turn our attention to query evaluation speed and measure what we gain by our approximations and simplifications.

Table 4.3 shows the average query latency of BWAND for different values of r (bits per element) and κ (number of hash functions). For these efficiency experiments, there are not enough queries from the microblog track to produce meaningful results, and so we used queries from the TREC 2005 terabyte track and the AOL query log. The table also includes query latencies of the baseline algorithms for reference. Reported values represent the average across five trials for each parameter setting and include 95% confidence intervals.

For conjunctive query processing, the fastest setting of BWand ($r = 8, \kappa = 1$) achieves a threefold increase in speed over SvS on the TREC queries, and a sixfold increase in speed on the AOL queries. For disjunctive query processing, we observe that BWand with $r = 8, \kappa = 1$ is more than ten times faster than Wand_{IDF} on the TREC queries and is about fifteen times faster on the AOL queries. Other settings of BWand lead to query latencies that are slightly higher than the (8,1) condition, but are still multiple times smaller than that of the baselines.

The observed differences between different settings of the BWand algorithm are due to two main reasons. First, increasing the number of hash functions, κ, forces the algorithm to compute more hash values and probe additional bit positions for membership tests. The effect of this is clear from Table 4.3: query evaluation becomes slower with increasing values of κ. However, larger values of κ have the advantage of reducing Bloom filter false positives, thereby yielding higher component-level recall, per the results in Tables 4.1 and 4.2.

Second, increasing the number of bits per element, r, gives rise to two counteracting forces. On the one hand, increasing r leads to larger Bloom filters. This translates into less locality and longer memory latencies. On the other hand, a larger Bloom filter—with a fixed number of elements—improves the false positive rate. Therefore, fewer hash computations are needed to reject a non-existent document id. The impact of these factors is clear in Table 4.3. In some cases, $r = 16$ is faster than $r = 8$ (meaning the drop in false positive errors overcomes the decreased locality), while we observe the opposite for $r = 24$ versus $r = 16$. From Tables 4.1 and 4.2, we see the benefit of increasing r in terms of higher relative recall.

Figures 4.2 and 4.3 show a break-down of average query latency for the TREC terabyte queries and the AOL queries by query length for conjunctive and disjunctive query processing using $r = 8$. For $r = 16$ and $r = 24$, the plots appear similar, so they are not included for brevity. Due to different query characteristics, results differ slightly from one query set to another, but the trends are consistent. For conjunctive query processing, longer queries are not necessarily slower: for example, seven-term queries are actually faster than three-term queries. This is possible because longer queries tend to contain rarer query terms. On the other hand, disjunctive query processing becomes slightly slower for longer queries.

As a reference, Figure 4.4 shows query latencies for the baseline algorithms, broken down by query length. Interestingly, SvS actually becomes faster for longer queries. Once again, this can happen because longer queries tend to contain rarer query terms. For disjunctive query processing, WAND latency grows with the number of query terms at a much faster rate than our BWAND algorithm; note that the y axis of Figure 4.4(b) is in log scale.

4.4.3 Index Size

In previous sections, we have examined our BWAND algorithm in terms of effectiveness and query evaluation speed. We have established that our algorithm yields high relative recall against exact baselines, while increasing query evaluation speed at least ten times. However, this gain in speed comes at the cost of an increase in memory footprint. Recall that, our BWAND algorithm requires Bloom filter chains *in addition* to a non-positional inverted index (i.e., an inverted index that only contains document ids). On the other hand, the WAND_{IDF} baseline only requires a non-positional index. We here evaluate the overhead of BWAND in terms of index size.

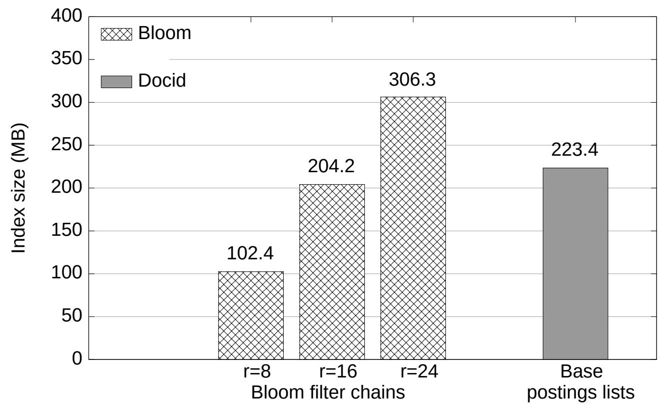
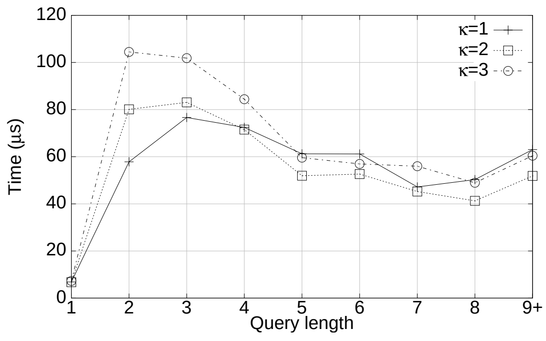
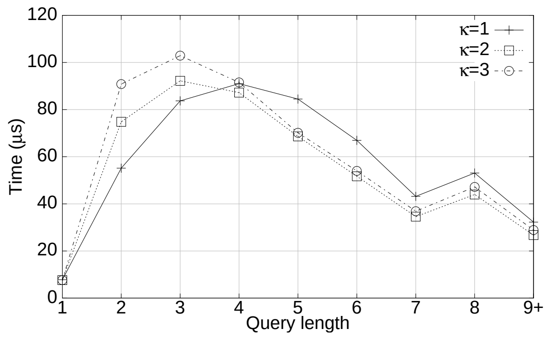


Figure 4.5: Effect of r (bits per element) on index size for the Tweets2011 collection, along with the size of a baseline index that only contains Docids.

Figure 4.5 illustrates index size as a function of r, number of bits per element. With $r = 8$, we observe a 45% increase in the overall index size. This is the cost of our approach: We are trading space for time by building auxiliary data structures that increase the speed of candidate

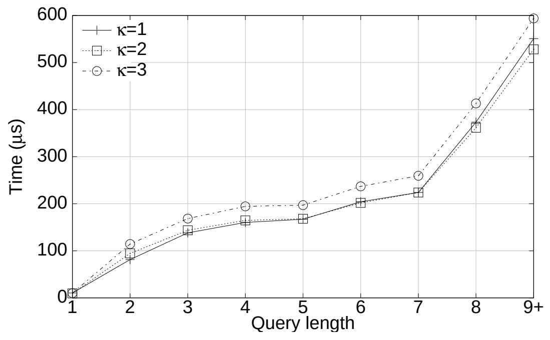


(a) TREC terabyte

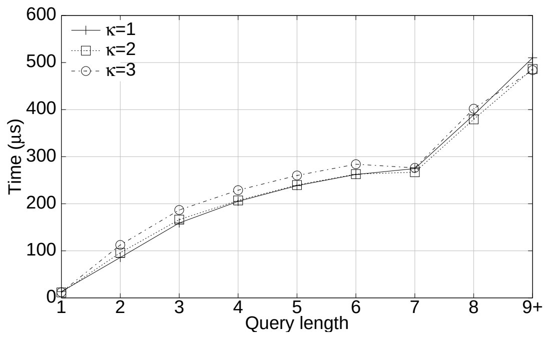


(b) AOL

Figure 4.2: Effect of query length on latency (μs) for BWand performing conjunctive retrieval ($r = 8$).

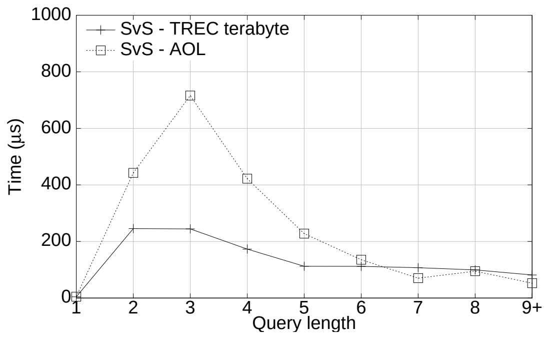


(a) TREC terabyte

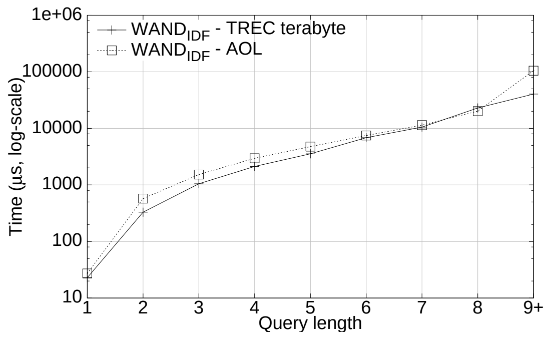


(b) AOL

Figure 4.3: Effect of query length on latency (μs) for BWand performing disjunctive retrieval ($r = 8$).



(a) Conjunctive



(b) Disjunctive

Figure 4.4: Effect of query length on latency (μs) for baseline algorithms.

generation. However, we believe this is a justifiable tradeoff for modern retrieval environments where query latency has a direct impact on site usage [116].

Our experiments have examined the tradeoffs between quality, speed, and memory usage that result from the approximations and simplifications in our BWand candidate generation algorithm. It is possible to achieve higher relative recall, but at the cost of longer query latencies or a larger memory footprint. If we wish to make the Bloom filter chains more compact, we must either sacrifice speed or quality, and so on. It is up to a developer to select the parameter setting that is appropriate for a particular server configuration and application context.

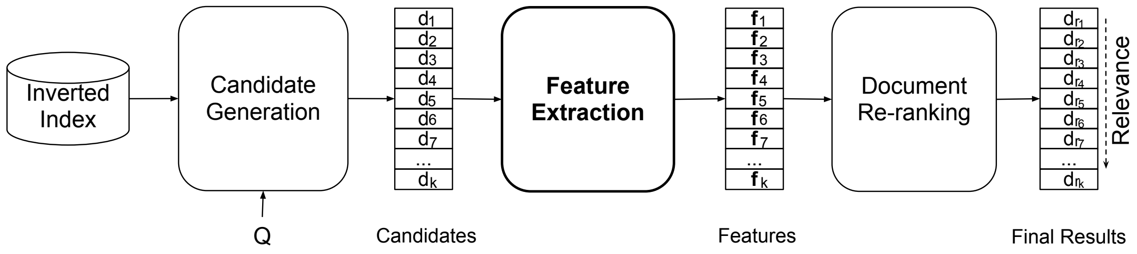
Chapter 5

Document Vector Organizations for Feature Extraction

Having explored the first stage in a learning-to-rank pipeline, we now turn our attention to the second stage: feature extraction. The feature extraction stage takes as input a set of candidate documents and returns as output vectors of features (one vector per candidate document). This is illustrated in Figure 5.1.

In a recent study [20], we have shown that decoupling candidate generation and feature extraction to form a two-stage retrieval architecture has advantages over a monolithic architecture in many ways. Most importantly, it allows us to use a different data struture in each stage. In the two-stage architecture, we have explored the use of document vectors (i.e., representing every document as a vector of terms) in lieu of a positional inverted index. We discussed the implications of each design and empirically evaluated the impact of using document vectors on feature extraction. Results suggest that we can achieve query latencies that are on par with a monolithic baseline with positional inverted indexes, but with an overall smaller memory footprint.

However, in that study we focused on relatively static document collections and made a few assumptions that do not hold for streaming documents. The key change is that, we assigned term ids based on the term frequency distribution of a collection in a way that more frequent terms have a smaller term id. However, this is not possible for streaming documents since we assign monotonically increasing term ids as we observe new vocabulary terms, regardless of term frequencies.



Inverted Index
Candidate Generation
Q
d_1 d_2 d_3 d_4 d_5 d_6 d_7 ... d_k
Candidates
Feature Extraction
f_1 f_2 f_3 f_4 f_5 f_6 f_7 ... f_k
Features
Document Re-ranking
d_{r_1} d_{r_2} d_{r_3} d_{r_4} d_{r_5} d_{r_6} d_{r_7} ... d_{r_k}
Relevance
Final Results

Figure 5.1: Learning-to-rank pipeline.

The change from static collections to streaming documents has implications for memory usage and speed in the feature extraction stage. For example, arbitrary term id assignment may degrade compression ratio and consequently increase the overall memory footprint. An increase in memory footprint may subsequently affect latency.

In this chapter, we summarize the highlights from our previous study [20], and explore these new tradeoffs. We focus on only one of the various document vector representations from our previous work [20], and compare it with a monolithic baseline in terms of latency, memory usage, and design flexibility. We assume a standard candidate generation algorithm like WAND and turn our attention to the feature extraction stage. Our experiments are performed on tweets as well as web collections since the techniques introduced in this chapter are applicable to arbitrary documents. Results suggest that, with slightly larger memory requirements, feature extraction with document vectors can be faster than a baseline approach and offer more flexibility.

5.1 Features

In the feature extraction stage we leave aside query-independent features which can be computed at indexing time and be stored in a hash table—reading these pre-computed scores require a constant access time. We instead consider query-dependent features, which by definition cannot be computed until a query is issued. We consider a standard set of query-dependent features defined in Metzler and Croft [117], as listed in Figure 5.2.

We use two families of scoring functions: BM25 and Dirichlet score from language modeling. Each family consists of a unigram feature f_T , a bigram proximity feature f_O that takes term order into account and is parameterized with a window $S \in \{1, 2, 4, 8, 16\}$ (i.e., co-occurrence of terms in order within a window), and a bigram feature score for unordered terms f_U which is parameterized with a window $S' \in \{2, 4, 8, 16, 32\}$. In total, there are 22 features. Among these features, term proximity features are the most interesting: their use has been shown to improve search quality, but they are expensive both in terms of computational cost and memory requirements for storing and accessing term positional information.

$$
\begin{aligned}
f_{T,Dir}(q,D) &= \log\left[\frac{\mathrm{tf}(q,D)+\frac{\mu}{|C|}\mathrm{cf}(q)}{|D|+\mu}\right]\\
f_{T,BM25}(q,D) &= \frac{(k_1+1)\cdot \mathrm{tf}(q,D)}{K+\mathrm{tf}(q,D)}\log\left[\frac{N-\mathrm{df}(q)+0.5}{\mathrm{df}(q)+0.5}\right]\\
f_{O,Dir,S}(q_j,q_{j+1},D) &= \log\left[\frac{\mathrm{tf}(\textsc{od}(S,q_j,q_{j+1}),D)+\frac{\mu}{|C|}\mathrm{cf}(\textsc{od}(S,q_j,q_{j+1}))}{|D|+\mu}\right]\\
f_{O,BM25,S}(q_j,q_{j+1},D) &= \frac{(k_1+1)\cdot \mathrm{tf}(\textsc{od}(S,q_j,q_{j+1}),D)}{K+\mathrm{tf}(\textsc{od}(S,q_j,q_{j+1}),D)}\log\left[\frac{N-\mathrm{df}(\textsc{od}(S,q_j,q_{j+1}))+0.5}{\mathrm{df}(\textsc{od}(S,q_j,q_{j+1}))+0.5}\right]\\
f_{U,Dir,S'}(q_j,q_{j+1},D) &= \log\left[\frac{\mathrm{tf}(\textsc{uw}(S',q_j,q_{j+1}),D)+\frac{\mu}{|C|}\mathrm{cf}(\textsc{uw}(S',q_j,q_{j+1}))}{|D|+\mu}\right]\\
f_{U,BM25,S'}(q_j,q_{j+1},D) &= \frac{(k_1+1)\cdot \mathrm{tf}(\textsc{uw}(S',q_j,q_{j+1}),D)}{K+\mathrm{tf}(\textsc{uw}(S',q_j,q_{j+1}),D)}\log\left[\frac{N-\mathrm{df}(\textsc{uw}(S',q_j,q_{j+1}))+0.5}{\mathrm{df}(\textsc{uw}(S',q_j,q_{j+1}))+0.5}\right]
\end{aligned}
$$

Figure 5.2: Definition of features used in our model. $\mathrm{tf}(e,D)$ is the count of concept e in D, $\mathrm{df}(e)$ is the document frequency of concept e, $\mathrm{cf}(e)$ is the collection frequency of concept e, where e is defined as follows: q is a query term; $\textsc{od}(S,q_j,q_{j+1})$ is an ordered phrase, span of S ($S \in \{1,2,4,8,16\}$); $\textsc{uw}(S',q_j,q_{j+1})$ is an unordered phrase, span of S' ($S' \in \{2,4,8,16,32\}$). N is the number of documents in the collection; $|D|$ is the length of document D; $|D|'$ is the average document length in the collection; $|C|$ is the total length of the collection; for Dirichlet features, μ is a smoothing parameter; for BM25, $K = k_1[(1-b)-b\cdot(|D|/|D|')]$, and k_1, b are free parameters.

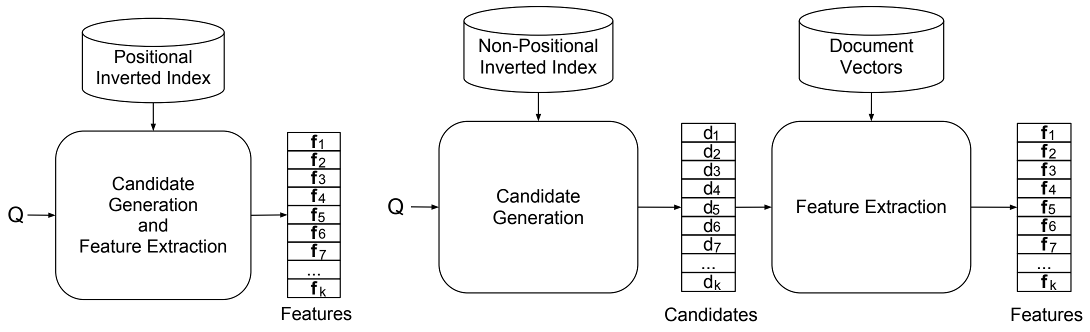
Real-world search engines take advantage of many more features. However, our simplified feature set retains the important characteristics of the problem we wish to tackle. We believe our findings can be generalized to larger and/or more sophisticated feature sets.

5.2 Index Organizations

As a baseline, we consider the standard positional inverted index, using which an algorithm can simultaneously perform candidate generation and feature extraction (input: query, output: list of feature vectors). Figure 5.3(a) illustrates a monolithic architecture. We use the approach in Chapter 3 to construct a positional inverted index (using the $32b$ contiguity setting with $b = 128$).

During postings traversal over a positional inverted index, we decode and collect query terms' positional information for every candidate. Once all top k candidates are generated, we can compute features using term positions. Therefore, a single stage suffices to generate candidate documents and to extract features.

We compare this monolithic approach with an architecture that has distinct candidate generation and feature extraction stages, each using separate data structures. In the candidate


Positional Inverted Index
Q
Candidate Generation and Feature Extraction
f1 f2 f3 f4 f5 f6 f7 ... fk
Features
Non-Positional Inverted Index
Q
Candidate Generation
d1 d2 d3 d4 d5 d6 d7 ... dk
Candidates
Document Vectors
Feature Extraction
f1 f2 f3 f4 f5 f6 f7 ... fk
Features

(a) Monolithic architecture (b) Two-stage architecture

Figure 5.3: Two architectures for candidate generation and feature extraction: a monolithic approach with a positional inverted index and a two-stage approach using a non-positional index and a document vector index.

generation stage, it is possible to use non-positional indexes, which are far more compact; when constructing a non-positional index, we disable term positions buffer maps in our indexing algorithm from Chapter 3, thereby reducing overall memory requirements. In the feature extraction stage, we can rely on a document vector index: a mapping from document ids to document vectors. This architecture is shown in Figure 5.3(b).

A document vector index provides random access to documents vectors, which is the main data structure used in our feature generation algorithm. We explored different approaches for representing document vectors in our previous work [20]. Among them, a flat array representation showed superior performance in many aspects. Therefore, we limit this study to flat arrays.

In the flat array document vector representation, a document D is represented as a list of terms $\{t_{p_1}, t_{p_2}, t_{p_3} \ldots\}$ such that t_{p_i} is the term at position p_i where $1 \leq p_i \leq |D|$. In lieu of actual (string) terms, we use integer term ids. A flat array representation can be compressed using standard integer compression techniques. Note that gap-compression is not appropriate in this case since there is no guarantee that term ids are in ascending order. In Asadi and Lin [20], we experimented with three different integer coding techniques. We used variable-length integers (VByte) and PFOR, and also proposed a document-adaptive coding scheme. Experimental evaluation of these schemes have shown that VByte is relatively slow. Therefore, we exclude VByte in our experiments in this chapter. On the other hand, our coding scheme was designed based on a set of assumptions that do not hold for streaming documents—we do not include our compression scheme in this chapter. This leaves us with one data structure: PFOR compressed document vectors.

We can easily augment our index structure from Chapter 3 with document vectors: When a new document arrives, we index the document as usual to construct a non-positional inverted index (with contiguity setting $32b$), but store a copy of the document as a compressed flat array of term ids in a separate table. Since document ids increase monotonically, we can implement this table just as buffer maps in our indexing algorithm, as detailed in Section 3.1.1—only here the array index corresponds to document ids (not term ids), and individual buffers have an arbitrary length. We index term frequencies whenever necessary (see section 5.4).

5.3 Feature Extraction

As previously discussed, we ignore query-independent features in our experiments since they can be modeled as constant factors. Here, we describe how query-dependent features (e.g., those in Figure 5.2) are computed. We assume that global term statistics (e.g., document length and collection frequency) are stored in the dictionary and are available in the memory.

Given a query and a document vector, we compute query-dependent features using the fastest algorithm detailed in [20]. In this approach, we first extract a list of term positions for every query term from the document vector "on the fly." Computation of proximity features can then be treated as a set intersection problem, for which we use a variant of the SvS algorithm. More specifically, feature extractors are provided with pairs that consist of a query term q_i and a list of positions P_i at which that term appears in the document. In order to compute the number of co-occurrences of terms q_j and q_{j+1} for a proximity feature with window size W, we traverse the list associated with the first query term (i.e., list P_j). For every position p in P_j, we count the number of positions in P_{j+1} that is within a window of length W from p. For example, if the current position from P_j is p_k, then we find positions $p'_k \in P_{j+1}$ where:

- $p'_k - p_k \leq S,\ p'_k > p_k$, to extract ordered window proximity features of the form $\textsc{od}(S, q_j, q_{j+1})$,

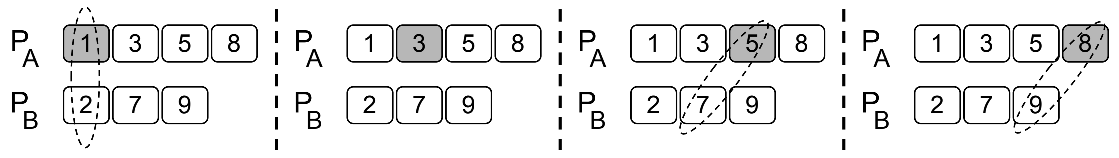


Figure 5.4: Computing the number of occurrences of OD$(2, \mathrm{A}, \mathrm{B})$ for query "A B" given document vector ⟨A B A D A C B A B⟩. Gray area indicates the current position in P_{A} under consideration. The dashed oval shows an occurrence of query terms (in order) within a window of length 2.

based on the definition in [117],

- $\begin{cases} p'_k - p_k + 1 \le S', & p'_k > p_k \\ p_k - p'_k + 1 \le S', & p_{k-1} < p'_k < p_k \end{cases}$, to extract unordered window proximity features in the form of UW(S', q_j, q_{j+1}) [117].

To illustrate the above, let us assume that we have a a query "A B," and a document vector with terms ⟨A B A D A C B A B⟩. From the vector, we can extract the positions for terms "A" (P_A) and "B" (P_B): $P_{\mathrm{A}} = [1, 3, 5, 8]$ and $P_{\mathrm{B}} = [2, 7, 9]$. Figure 5.4 shows how the number of occurrences of OD$(2, \mathrm{A}, \mathrm{B})$ (i.e., "A" and "B" can be separated by one term, but should appear in order) can be computed using the above algorithm. At each stage, we pick one position from P_{A} in order, and count the number positions in P_{B} that is within a window of length 2 from the position in P_{A}.

This approach can easily be generalized to phrases that consist of more than two query terms. Note that, since the candidate generation algorithm in the monolithic architecture extracts term positions for every candidate, we can easily use this class of feature extractors in the monolithic architecture as well.

5.4 Experimental setup

We performed our experiments on the first English segment of the ClueWeb09 collection, the Gov2 collection, as well as the Tweets2011 collection, as described in previous chapters. There are about 50 million documents in the ClueWeb09 collection (6.9 million unique indexed terms), 25 million

Table 5.1: Average time per query in milliseconds to generate 1000 candidate documents and extract features for the TREC 2005 terabyte track queries (across 5 trials, with 95% confidence intervals).

	Tweets2011	ClueWeb09	Gov2
Wand	0.9 (±0.1)	141.1 (±0.6)	62.5 (±0.8)
PII	4.2 (±0.1)	169.3 (±1.8)	100.9 (±8.3)
DV	3.7 (±0.1)	162.6 (±1.7)	137.7 (±8.3)

documents in the Gov2 collection (2.9 million unique indexed terms), and 16 million tweets in the Tweets2011 collection (400K unique indexed terms). For evaluation, we used the "efficiency topics" of the TREC 2005 terabyte track, same as in Chapters 3 and 4.

For candidate generation, we retrieved the top 1000 hits using the Wand algorithm with a standard BM25 scoring function for experiments on the ClueWeb09 and Gov2 collections, and Wand with the Idf scoring function (as explained in Chapter 4) for experiments on the Tweets2011 collection. Therefore, in the two-stage architecture, we construct a non-positional index with term frequencies for the ClueWeb09 and Gov2 collections, and without term frequencies for the Tweets2011 collection. Along with the non-positional inverted index, we store compressed document vectors.

For our baseline comparison using positional inverted indexes, we use the Wand algorithm (with BM25 for ClueWeb09 and Gov2, and Idf for Tweets2011). To enable feature extraction, we need to accumulate term positions. As such, we modified the heap structure in Wand to keep track of term positions for every candidate. Every time a document passes the score-threshold test and is inserted into the heap, the Wand algorithm decodes term positions for every query term present in that document and stores them in the heap along with the document id. In the end, all candidates have a list of positions for all query terms, which can be used in the feature extractors to compute feature values.

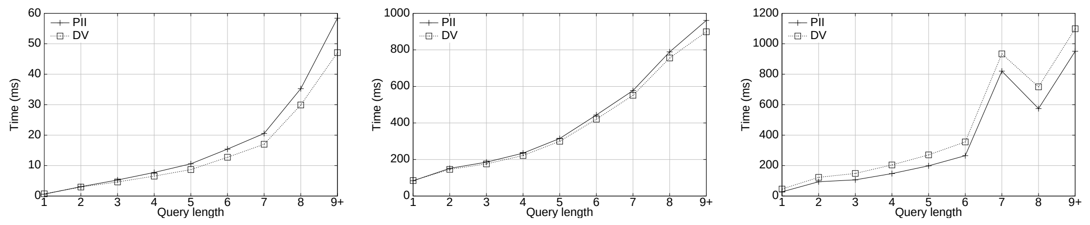
5.5 Results

5.5.1 Speed

Table 5.1 shows the average query latency for candidate generation and feature extraction, reported in milliseconds (averaged across 5 trials, with 95% confidence intervals), for different document collections. To be precise, query latency is defined as the elapsed time between the arrival of a query until all 22 features described earlier are extracted for the top 1000 candidate documents. Note that, for the positional inverted index (PII), both candidate generation and feature extraction are performed together, while with document vectors (DV) in a two-stage architecture, the two stages are distinct. The table also includes the latency of candidate generation (i.e., no feature extraction is performed) for reference.

On the ClueWeb09 and Tweets2011 collections, a two-stage architecture is slightly, nevertheless statistically significantly faster than PII. However, we observe that on the Gov2 collection, PII achieves a smaller end-to-end query latency. We explain this observation by the average document length of the collections under study. The average document lengths for Tweets2011, ClueWeb09, and Gov2 are 8 (standard deviation of 5), 535 (standard deviation of 732), and 627 terms (standard deviation of 1,877). When working with document vectors in a two-stage architecture, feature extractors must first decompress the entire document in order to collect position information for every query term. A longer document spans more blocks and therefore requires more memory fetches and decompression operations. Additionally, traversing a longer document to extract term positions adds to the computational cost of feature extractors. On the other hand, in a positional inverted index, we only decompress the term position block from the index segment that contains a term's positional information. Therefore, feature extraction in a monolithic architecture requires less "work" when it comes to a relatively long document.

Figure 5.5 shows the average end-to-end query latency broken by query length. To no surprise, query latency increases for longer queries. PII is consistently faster than a two-stage approach on the Gov2 collection. This trend is reversed for Tweets2011 and ClueWeb09. However,



(a) Tweets2011 (b) ClueWeb09 (c) Gov2

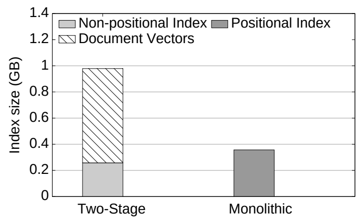
Figure 5.5: Average time to generate 1000 candidate documents and extract features for the terabyte track queries, broken down by query length.

the performance gap between the two approaches widens for longer queries. We explain this finding as follows: for longer queries, the PII approach requires more probes to random memory locations in order to decompress term positions associated with query terms. On the other hand, a document vector must be completely decompressed regardless of the number of terms in a query; the only extra cost to pay for longer queries is the additional comparisons of query terms with terms in the document.

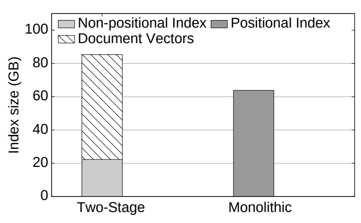
5.5.2 Index Size

We now turn our attention to index size, which corresponds to memory footprint, since we hold all index structures in memory. These results are shown in Figure 5.6 for different collections. In contrast to our findings in [20], document vectors together with a non-positional index have a larger memory footprint than a positional inverted index. The overall index organization in the two-stage architecture is 1.5 times larger than a positional index for Tweets2011, and is 0.3 times larger for the ClueWeb09 and Gov2 collections.

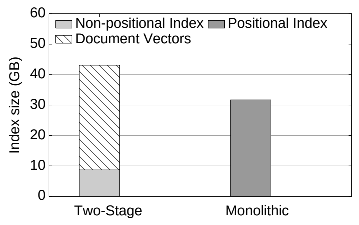
These results make sense since term ids are assigned arbitrarily, which in turn degrades compression ratio. This phenomenon is more pronounced for the Tweets2011 collection: The compression algorithm is more effective for a positional inverted index since term positions and term frequencies are relatively small, while compression becomes less effective for document vectors with relatively large term ids. Additionally, because term positions for tweets are generally small, a non-positional index is not much smaller than a positional index.



(a) Tweets2011



(b) ClueWeb09



(c) Gov2

Figure 5.6: Index size for different architectures. The two-stage architecture requires a non-positional index in the candidate generation stage and document vectors in the feature extraction stage. The monolithic architecture only requires a positional inverted index.

5.6 Discussion

Our two-stage architecture leads to faster feature extraction on collections of tweets, as well as web collections like ClueWeb09, relative to a monolithic approach using a positional inverted index. However, the index organization has a larger memory footprint. But we argue that there are other advantages to a two-stage architecture that are not directly measurable; a two-stage architecture provides more flexibility from an engineering perspective.

First, in our two-stage architecture, feature engineering does not need to involve the candidate generation stage, which reduces system-level changes needed for experimentation. Incorporating richer features, for example, named-entity tags, markup attributes, multiple text fields, etc. is easier to accomplish in the document vector index than in the inverted index.

Second, in an architecture where candidate generation and feature extraction are decoupled, there is in principle no reason why both stages need to run in the same process space, or even on the same machine, for that matter. This fits well with modern web architectures, where each rendering of a page involves dozens or even hundreds of individual services. From a management point of view, fine-grained service decomposition makes monitoring, performance profiling, and load balancing easier.

Finally, we note that in production systems, there must be some service for snippet generation (i.e., the keyword-in-context summary search results that are displayed to the user). We can easily augment our document vector indexes to hold such information without any architectural changes (e.g., as another field). In fact, without stemming, the flat array document vector representation can be directly used for snippet generation. With a positional inverted index, in contrast, some other document-lookup service is required to generate snippets, which further increases the overall memory footprint of the system.

Chapter 6

Architecture-Conscious Implementations of Tree Ensembles

In previous chapters, we have studied the problem of adapting indexing and the first two stages (candidate generation and feature extraction) of a learning-to-rank search architecture to streaming documents. In this chapter and the next, we explore the final stage: document re-ranking using a machine-learned model. As illustrated in Figure 6.1, the input to this stage is a set of feature vectors, one per candidate document, and the output is a ranked list of candidate documents sorted by relevance scores. Relevance scores are computed using a machine-learned model.

Among machine learning algorithms used in the last stage, Gradient Boosted Regression Trees (GBRTs) have become popular due to their superior effectiveness. A GBRT model is essentially an ensemble of (weighted) regression trees. To evaluate a feature vector, the algorithm traverses every tree in the ensemble independently until it reaches a terminal node. Within each tree, internal nodes direct the evaluation algorithm to the right or left subtree by conditioning on a feature value, while terminal nodes return scores. The (weighted) sum of scores from individual trees form the relevance score for an input feature vector.

Traversing a tree structure is an extremely straightforward task. However, a naïve implementation of trees is susceptible to data and control hazards, since it involves branches and leads to an unpredictable memory access pattern (see Section 2.7 for details). These hazards consequently degrade the efficiency of traversal of an individual tree. Their impact becomes more severe when

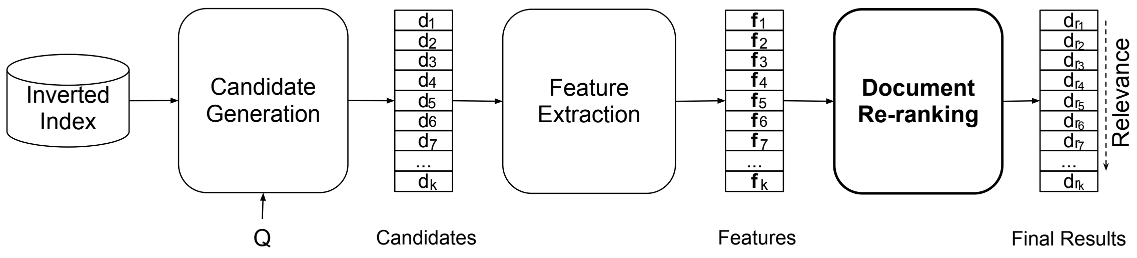


Figure 6.1: Learning-to-rank pipeline.

there are many more trees to traverse. This is an important issue in a search architecture because GBRT ensembles often comprise a large number of trees (hundreds, even thousands).

To alleviate this problem, this chapter describes novel implementations of tree-based models that are highly tuned to modern processor architectures. Our implementations take advantage of cache hierarchies and superscalar processors. In order to evaluate our implementations, we isolate the document re-ranking stage and—since our tweet collection is not sufficient for thorough experimentation due to its small size and our small number of features—we perform experiments on three separate learning-to-rank datasets for web search. Our experiments show significant performance improvements over standard implementations. In Chapter 8, we demonstrate the impact of these implementations on the end-to-end retrieval pipeline on a microblog collection, however, with fewer number of features.

6.1 Tree Implementations

We describe various implementations of an individual tree in this section, starting from two baselines and incrementally introducing architecture-conscious optimizations. Our focus is on binary trees, where every internal node branches out into left and right subtrees[1]. An internal node contains a predicate check that involves comparing a feature value with a threshold: if the feature value is less than the threshold, we take the left branch; otherwise, we move to the right branch.

CodeGen: As a baseline, we consider statically-generated if-else blocks. A code generator takes a tree model and directly generates C code, which is then compiled and used to make predictions. We refer to this as the CodeGen implementation.

We expect this approach to be fast. The entire model is statically specified; machine instructions are relatively compact and will fit into the processor's instruction cache, thereby exhibiting good reference locality. Furthermore, this implementation takes advantage of decades of compiler optimizations that have been built into `gcc`. Note that this implementation eliminates data dependencies completely by converting them all into control dependencies. The downside, however,

[1]Trees with a greater branching factor can be converted into an equivalent binary tree.

is that this approach is inflexible; integrating a new model requires generating static code followed by compiling and linking with the rest of the system.

Object: As another baseline, we consider an implementation of trees with nodes and associated left and right pointers in C++. In this implementation, we represent each tree node by an object which contains a feature id as well as a decision threshold. We refer to this as the Object implementation.

This implementation is both simple and flexible. However, the downside is that we do not have control over the layout of these objects in memory. Consequently, this implementation does not guarantee good reference locality. Because of this, traversing a tree requires frequent dereferencing and chasing pointers which potentially causes a large number of cache misses and can negatively impact speed.

Struct: The Object implementation also suffers from inefficient memory utilization due to object overhead in C++. The solution is to avoid objects and implement each node as a `struct` (comprising feature id, threshold, left and right pointers). We construct a tree by allocating memory for each node (`malloc`) and assigning the pointers appropriately. Prediction with this implementation remains an exercise in pointer chasing, but now across much more memory-efficient data structures. We refer to this as the Struct implementation.

Struct$^+$: An improvement over Struct is to manually manage the memory layout. Instead of allocating memory for each node separately, we allocate memory for all the nodes at once (i.e., an array of `struct`s) and linearize the order of the nodes via a breadth-first traversal of the tree, with the root at index 0.

This implementation occupies the same amount of memory as Struct, except that the memory is contiguous. The goal is to achieve better reference locality, thereby speeding up memory interactions. We call this the Struct$^+$ implementation. This is similar to the idea behind CSS-Trees [102] used in the database community. There are a few differences between the work of Rao and Ross [102] and this work: First, their focus is on search trees for sorted integers, while we primarily focus on traversing regression trees. Second, our work is in the context of learning-to-

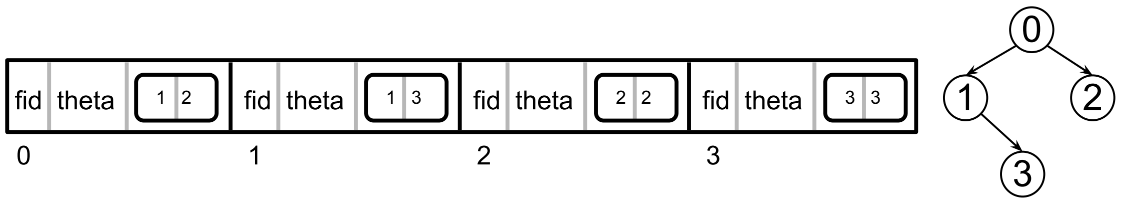


Figure 6.2: Memory layout of the PRED implementation for a sample tree.

rank, while their work has applications in databases. Finally, they consider mutable trees while trees in our work are static and do not change once a model is trained—which provides more opportunity for optimizations.

PRED: The STRUCT$^+$ implementation addresses the problem of reference locality that exists with OBJECT and STRUCT, but it fails to resolve another important drawback of all previous implementations: the presence of branches and hence inevitable branch mispredicts which can cause pipeline stalls and wasted cycles (as explained in Section 2.7). While modern processors deal with branches with complex branch prediction methods, removing branches from the execution cycle may increase performance. A well-known technique in the compiler community for overcoming these issues is known as predication [106, 107]. The idea is to convert control dependencies into data dependencies, thus avoiding jumps in the underlying assembly code.

We adopt the idea of predication for tree structures as follows: We encode the tree as an array of `struct`s in C, `nd`, where `nd[i].fid` is the feature id to examine, and `nd[i].theta` is the threshold. The nodes are laid out via breadth-first traversal of the tree, similar to the STRUCT$^+$ implementation. Each node, regardless of being intermediate or terminal, additionally holds an array of two integers. The first integer field holds the *index* of the left child in the array representation, and the second integer holds that of the right child. We use self loops to connect every terminal node to itself (more below). Figure 6.2 illustrates a sample memory structure of nodes in the proposed implementation.

To make the prediction, we probe the array d times, where d is the depth of the tree, in the following manner:

```
i = nd[i].index[(v[nd[i].fid] >= nd[i].theta)];
i = nd[i].index[(v[nd[i].fid] >= nd[i].theta)];
  ...
```

That is, if the condition holds, we visit the right node (index 1 of the index array); otherwise, we visit the left node (index 0 of the index array). In case we reach a terminal node before the d_{th} statement, we would be sent back to the same node regardless of the condition by the self loops.

We completely unroll the tree traversal loop, so the above statement is repeated d times for a tree of depth d. In the end, i contains the index of the leaf node corresponding to the prediction. Self loops allow us to represent unbalanced trees without introducing additional complexity into the tree traversal logic. More importantly, tree traversal does not need a condition to check whether we have reached a terminal node. Therefore, we can safely unroll the loop without undermining the correctness of the traversal algorithm. A final implementation detail: we hard code a prediction function for every possible tree depth, and then dispatch calls dynamically using function pointers.

There are a few minor differences between our implementation and previous work, which nevertheless may have a substantial impact on performance. For example, the GPU implementation of tree models by Sharp [118] similarly takes advantage of predication and loop unrolling, but in order to handle unbalanced trees each node must store a flag indicating whether or not it is a terminal node. Tree traversal requires checking this flag at each step, which translates into an additional conditional branch at each level in the tree. We avoid this additional comparison by the "self loops" trick: note that these self loops have minimal impact on performance, since they are accessing data structures that have already been placed into cache. In another example, the work of Essen et al. [119] also takes advantage of predication, but their implementation also requires this additional terminal node-test.

VPred: Predication eliminates branches by converting them into data dependencies. Each statement in Pred is dependent on the result of its preceding statement: an index to the tree array. Therefore, statements must execute in order, one by one, each followed by a memory fetch. While referencing the memory, the processor will simply idle. Because main memory access latencies are relatively very slow, Pred's data dependency becomes a bottleneck in tree traversal.

A common technique adopted in the database literature to mask these memory latencies is called *vectorization* [108, 104]. The idea is to process multiple instances (in our case, feature vectors) at once, in an interleaved way. Provided there are no dependencies between processes, the processor can dispatch multiple instructions in parallel and take advantage of pipelining. That is, while the processor is waiting for the memory access from the predication step on the first instance, it can start working on the second instance. In fact, we can work on v instances in parallel. For $v = 4$, this looks like the following, processing instances `v0`, `v1`, `v2`, `v3` in parallel:

```
i0 = nd[i0].index[(v0[nd[i0].fid] >= nd[i0].theta)];
i1 = nd[i1].index[(v1[nd[i1].fid] >= nd[i1].theta)];
i2 = nd[i2].index[(v2[nd[i2].fid] >= nd[i2].theta)];
i3 = nd[i3].index[(v3[nd[i3].fid] >= nd[i3].theta)];

i0 = nd[i0].index[(v0[nd[i0].fid] >= nd[i0].theta)];
i1 = nd[i1].index[(v1[nd[i1].fid] >= nd[i1].theta)];
i2 = nd[i2].index[(v2[nd[i2].fid] >= nd[i2].theta)];
i3 = nd[i3].index[(v3[nd[i3].fid] >= nd[i3].theta)];
  ...
```

In other words, we traverse one layer in the tree for four instances at once. While we are waiting for `v0[nd[i0].fid]` to resolve, we dispatch instructions for accessing `v1[nd[i1].fid]`, and so on. The hope is that by the time the final memory access has been dispatched, the contents of the first memory access are available, and we can continue without processor stalls.

Again, we completely unroll the tree traversal loop, so each block of statements is repeated d times for a tree of depth d. In the end, the i's contain indexes of the terminal nodes corresponding to the prediction for v instances. Setting v to 1 reduces this implementation to Pred (i.e., no vectorization). The optimal value of v is dependent on the relationship between the amount of computation performed and memory latencies, which we can determine empirically. We refer to the vectorized version of the predication technique as VPred.

6.2 Experimental Setup

The focus of this chapter is efficiency. As such, our primary evaluation metric is prediction speed. We define this as the elapsed time between the moment a feature vector (i.e., a test instance) is presented to the tree-based model, to the moment that a prediction is made for the instance (in our case, a regression value). To increase the reliability of our results, we conduct multiple trials and report the average as well as some characterization of the variance.

We conduct two sets of experiments. First, we quantify the performance of isolated individual trees using synthetically-generated data. Second, we verify our findings on standard learning-to-rank datasets using full ensembles.

All experiments were run on the same hardware used in previous chapter with no other competing processes. Code was compiled with GCC (version 4.1.2) using optimization flag `-O3`. In our main experiments, all code ran single-threaded, but we report on multi-threaded experiments in Section 6.4.

6.2.1 Synthetic Data

The synthetic data consist of randomly generated trees and randomly generated feature vectors. Each intermediate node in a tree has two fields: a feature id and a threshold on which the decision is made. Each leaf is associated with a regression value. Construction of a random tree of depth d begins with the root node. We pick a feature id at random and generate a random threshold to split the tree into left and right subtrees. By performing this process recursively, we can build each subtree until the desired tree depth is reached. When we reach a terminal node, we generate a regression value at random. Note that our randomly-generated trees are fully-balanced, i.e., a tree of depth d has 2^d terminal nodes.

The next step is to generate random feature vectors. Each random feature vector is simply a floating-point array of length f (= number of features), where each index position corresponds to a feature value. We assume that all paths in the decision tree are equally likely; the feature vectors are generated in a way that guarantees an equal likelihood of visiting each leaf. To accomplish this, we take one leaf at a time and follow its parents back to the root. At each node, we take the node's feature id and produce a feature value based on the position of the child node. That is, if the child node we have just visited is on the left subtree we generate a feature value that is smaller than the threshold stored at the current parent node and vice versa. We randomize the order of instances once we have generated all the feature vectors. To avoid any cache effects, we generate a large number of instances (512k).

Given a random tree and a set of random feature vectors, we run experiments to assess the various implementations of tree-based models described in Section 6.1. To capture the variance, we repeat each experiment 5 times; in each trial we construct a new random binary tree and a different randomly-generated set of feature vectors. To explore the design space, we conduct experiments with varying tree depths $d \in \{3, 5, 7, 9, 11\}$ and varying feature sizes $f \in \{32, 128, 512\}$—these values follow our learning-to-rank datasets (see below).

6.2.2 Learning-to-Rank Experiments

In addition to randomly-generated trees, we conduct a set of learning-to-rank tasks using standard datasets, containing training, validation and test sets. Using the training and validation sets we learn a complete tree ensemble. We then evaluate test instances using our tree implementations to determine the end-to-end performance. These end-to-end experiments give us an insight on how different implementations compare in a real-world application.

We used three standard learning-to-rank datasets: LETOR-MQ2007,[2] MSLR-WEB10K,[3] and the Yahoo! Webscope Learning-to-Rank Challenge [120] dataset (C14 Set 1).[4] All three datasets are pre-folded, providing training, validation, and test instances. Table 6.1 shows the dataset size and the number of features. To measure variance, we repeated experiments on all folds. Note that MQ2007 is much smaller and contains a more impoverished feature set, considered by many in the community to be outdated. Further note that the C14 dataset only contains a single fold.

These learning-to-rank datasets guide the values of f (number of features) in our synthetic experiments. We selected feature sizes that are multiples of 16 so that the feature vectors are integer multiples of cache line sizes (64 bytes): $f = 32$ roughly corresponds to LETOR features

[2]http://research.microsoft.com/en-us/um/beijing/projects/letor/letor4dataset.aspx

[3]http://research.microsoft.com/en-us/projects/mslr/

[4]http://learningtorankchallenge.yahoo.com/datasets.php

Table 6.1: Average number of training, validation, and test instances in our learning-to-rank datasets, along with the number of features.

Dataset	Train	Validate	Test	Features
C14	473K	71K	165K	519
MSLR-WEB10K	720K	240K	240K	136
LETOR-MQ2007	42K	14K	14K	46

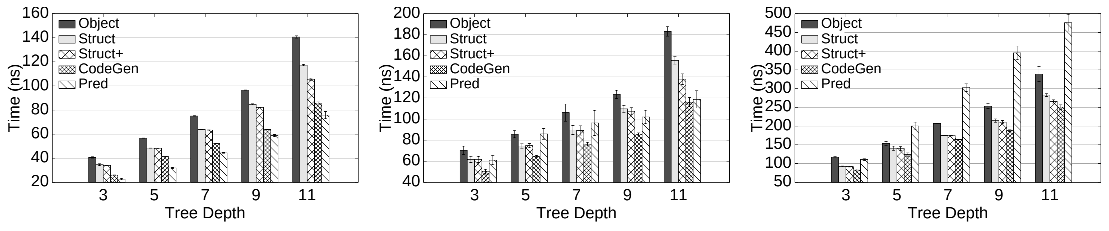
and is representative of a small feature space; $f = 128$ corresponds to MSLR and is representative of a medium-sized feature space. Finally, the third condition $f = 512$ corresponds to the C14 dataset and captures a large feature space condition.

We use the open-source jforests implementation[5] of LambdaMART to optimize NDCG [80]. Although there is no way to precisely control the depth of each tree, we can adjust the size distribution of the trees by setting a cap on the number of leaves. To train an ensemble, we initialize the randomized LambdaMART algorithm with a random seed S. In order to capture the variance, we repeat this experiment $E = 100$ times for the LETOR and MSLR-WEB10K datasets and $E = 20$ times for the C14 dataset. We repeat this procedure independently for each cross-validation fold. For LETOR and MSLR, we use the parameters of LambdaMART suggested by Ganjisaffar et al. [9]: feature and data sub-sampling parameters (0.3), minimum observations per leaf (0.5), and the learning rate (0.05). We vary the max leaves parameter as part of our experimental runs. Note that Ganjisaffar et al. did not experiment with the C14 dataset, but we retain the same settings for all parameters except for the maximum number of leaves.

6.3 Results

In this section we present experimental results, first on synthetic data and then on learning-to-rank datasets.

[5]http://code.google.com/p/jforests/



(a) $f = 32$ (b) $f = 128$ (c) $f = 512$

Figure 6.3: Prediction time per instance (in nanoseconds) on synthetic data using various implementations. Error bars denote 95% confidence intervals.

6.3.1 Synthetic Data: Base Results

Figure 6.3 illustrates the average prediction time (in nanoseconds) per randomly-generated test instance. Error bars indicate 95% confidence intervals. It is clear that prediction speed decreases for trees with a larger depth. This is not surprising since deeper trees require more feature accesses and predicate checks, more pointer chasing, and more branching (depending on the implementation).

First, let us consider the Object and CodeGen baselines. As expected, the Object implementation is the slowest in most cases. Comparing this implementation with Struct reveals that object overhead is the reason for its slowness. On the other hand, the CodeGen implementation is very fast: with the exception of $f = 32$, hard-coded if-else blocks outperform all other implementations regardless of tree depth.

Struct$^+$ does not achieve significant improvements for shallow trees over the Strcut implementation, but it does lead to a significant speedup for deeper trees. Recall that Struct$^+$ manages memory layouts manually and enforces contiguity of tree structures in memory. These results show that reference locality is an important factor when it comes to deeper trees.

Finally, we turn our attention to the Pred condition. We observe a very interesting behavior: For small feature vectors $f = 32$, the technique is actually faster than CodeGen. This shows that at least for small feature sizes, predication helps to overcome branch mispredicts. That is, converting control dependencies into data dependencies increases performance. For $f = 128$, the performance of Pred is worse than that of CodeGen (except for trees of depth 11). With the exception of the deepest trees, Pred is approximately the same speed as Struct and Struct$^+$.

Table 6.2: Prediction time per instance (in nanoseconds) for the VPred implementation with optimal vectorization parameter, compared to Pred and CodeGen, along with relative improvements.

(a) $f = 32, v = 8$

d	VPred	Pred	Δ	CodeGen	Δ
3	17.4	22.6	23%	26.0	33%
5	21.3	31.9	33%	41.3	48%
7	25.1	44.4	44%	52.4	52%
9	28.9	58.9	51%	63.9	55%
11	39.2	75.8	57%	85.8	54%

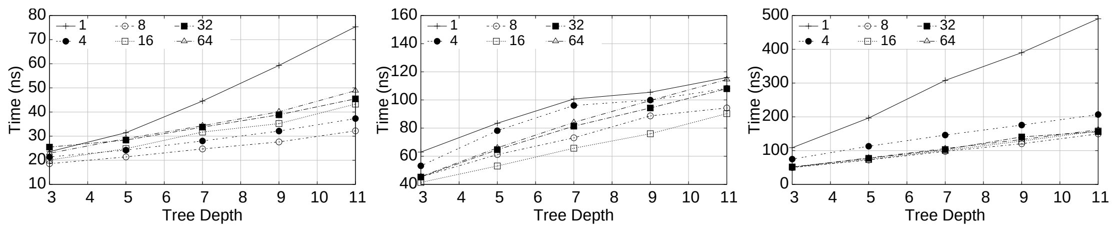
(b) $f = 128, v = 16$

d	VPred	Pred	Δ	CodeGen	Δ
3	42.2	61.0	31%	50.0	16%
5	55.8	85.9	35%	64.6	14%
7	69.3	96.3	28%	76.2	9%
9	77.9	102.0	24%	85.8	9%
11	89.6	118.7	25%	116.0	23%

(c) $f = 512, v = 16$

d	VPred	Pred	Δ	CodeGen	Δ
3	49.7	110.6	55%	82.8	40%
5	72.3	200.3	64%	123.9	42%
7	97.8	302.5	68%	164.2	40%
9	120.4	395.5	70%	187.8	36%
11	149.5	476.1	69%	250.7	40%

However, for larger feature vectors ($f = 512$), the performance of Pred is very poor, even worse than the Object implementation. To explain this behavior, recall that Pred's performance is entirely dependent on memory access latency. When traversing a tree, the processor idles until the memory references are resolved. With small feature vectors, we get excellent locality: 32 features take up two 64-byte cache lines, which means that evaluation results in at most two cache misses. Since memory is fetched by cache lines, once a feature is accessed, accesses to all other features on the same cache line are essentially free. On the other hand, larger feature vectors span more cache lines and therefore have poorer locality. As a result, cache misses and subsequently memory latencies dominate Pred's prediction time.



(a) $f = 32$ (b) $f = 128$ (c) $f = 512$

Figure 6.4: Prediction time per instance (in nanoseconds) on synthetic data using vectorized predication, for varying values of the batch size v. $v = 1$ represents the Pred implementation.

6.3.2 Tuning Vectorization Parameter

As explained in Section 6.1, instead of simply stalling while we wait for memory references to resolve, we can try to perform other useful computation—this is exactly what vectorization attempts to accomplish. The idea is to process v feature vectors at the same time, so that while the processor is waiting for memory access for one instance, it can process other instances and perform useful computation. This takes advantage of pipelining and multiple dispatch in modern superscalar processors.

The effectiveness of vectorization depends on two factors: the amount of time spent on computation, and memory latencies. For example, if memory fetches take only one clock cycle, then vectorization cannot possibly help. The longer the memory latencies, the more we would expect vectorization to help. However, beyond a certain point, we would expect larger values of v to have little impact. In fact, values that are too large start to bottleneck on memory bandwidth and cache size.

Figure 6.4 shows the impact of various batch sizes, $v \in \{1, 4, 8, 16, 32, 64\}$, for different feature sizes. When v is set to 1, we evaluate one instance at a time, reducing the implementation to Pred. We measure speed in nanoseconds and report *per-instance* prediction time. For $f = 32$, $v = 8$ yields the best performance; for $f = 128$, $v = 16$ yields the best performance; for $f = 512$, $v = \{16, 32, 64\}$ provide approximately the same level of performance. These results are exactly what we would expect: since memory latencies increase with larger feature sizes, a larger batch size is necessary to mask the latencies.

The combination of vectorization and predication, VPred is the fastest of all our implementations on synthetic data. Comparing Figures 6.3 and 6.4, we see that VPred (with optimal vectorization parameter) is even faster than CodeGen. Table 6.2 summarizes this comparison. Vectorization is up to 70% faster than the non-vectorized implementation; VPred can be twice as fast as CodeGen. In other words, we retain the best of both worlds: speed and flexibility, since the VPred implementation does not require code re-compilation for each new tree model.

6.3.3 Learning-to-Rank Results

Thus far, we have demonstrated the superiority of VPred over other implementations using synthetic data. We now turn our attention to learning-to-rank datasets using tree ensembles. As described previously, we train tree ensembles using LambdaMART and evaluate the trained model on test data to measure prediction speed. We compute mean and variance of prediction time across multiple runs (see Section 6.2.2).

One important difference between the synthetic and learning-to-rank experiments is that LambdaMART produces an *ensemble* of trees, whereas the synthetic experiments focused on a single tree. To handle this, we simply add an outer loop to the algorithm that iterates over all the individual trees in an ensemble.

From our experiments on synthetic data we know that shallower trees are desirable when it comes to performance. However, there is a relationship between tree depth and effectiveness which we examine first. Even though we cannot control tree depth with our particular training algorithm, we can indirectly influence it by constraining the maximum number of leaves for each individual tree. Table 8.1 summarizes the average NDCG values at different ranks measured across all folds on the LETOR and MSLR datasets, and on the C14 dataset. The reported tree depths in the table represent micro-averages. That is, we first compute the average tree depth for every ensemble and then average across the folds. We determine statistical significance using the Wilcoxon test (p-value$<$0.05); none of the differences on the LETOR dataset is significant.

Results show that on LETOR, the tree depth makes no significant difference, whereas deeper

Table 6.3: Average NDCG values and average tree depth (95% confidence intervals in parentheses) measured across cross-validation folds using various settings for max number of leaves. For MSLR, $^+$ and * show statistically significant improvements over models obtained by setting max leaves to 10 and 30 respectively. For C14, $^+$ and * show improvements over 70 and 110 respectively.

(a) LETOR-MQ2007

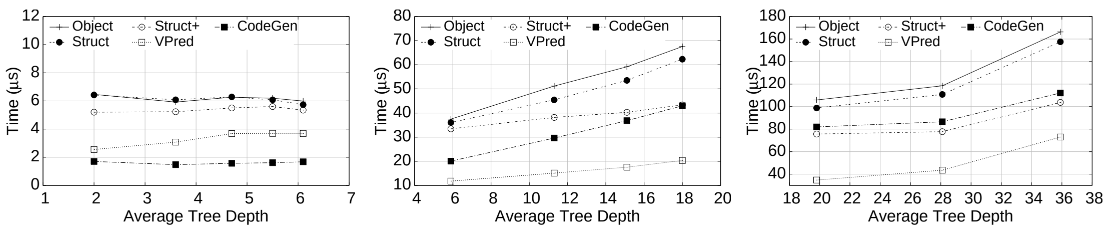
Max Leaves	Avg. Depth	@1	@3	@5	@10	@20	@∞
3	2.0 (0.0)	0.475	0.478	0.487	0.522	0.591	0.701
5	3.6 (0.1)	0.477	0.479	0.487	0.521	0.591	0.701
7	4.7 (0.1)	0.477	0.479	0.486	0.520	0.591	0.700
9	5.5 (0.1)	0.476	0.478	0.486	0.519	0.590	0.699
11	6.1 (0.2)	0.476	0.477	0.485	0.519	0.590	0.699

(b) MSLR-WEB10K

Max Leaves	Avg. Depth	@1	@3	@5	@10	@20	@∞
10	5.9 (0.1)	0.467	0.452	0.456	0.473	0.504	0.719
30	11.3 (0.1)	0.471^+	0.456^+	0.461^+	0.479^+	0.510^+	0.721^+
50	15.1 (0.1)	0.471^+	0.457^+	0.461^+	0.480^{+*}	0.511^{+*}	0.722^{+*}
70	18.0 (0.1)	0.471^+	0.456^+	0.461^+	0.480^{+*}	0.511^{+*}	0.722^{+*}

(c) C14

Max Leaves	Avg. Depth	@1	@3	@5	@10	@20	@∞
70	19.8 (0.2)	0.713	0.715	0.735	0.778	0.820	0.863
110	28.1 (0.3)	0.712	0.716	0.735	0.778	0.820	0.863
150	35.9 (0.5)	0.712	0.716	0.736^{+*}	0.779^{+*}	0.821^{+*}	0.863



(a) LETOR-MQ2007 (b) MSLR-WEB10K (c) C14

Figure 6.5: Per-instance prediction times (in microseconds), averaged across cross-validation folds (100 ensembles per fold for LETOR and MSLR, and 20 ensembles for C14) using LambdaMART on different datasets.

trees yield better quality results on MSLR and C14; however, there appears to be little difference between 50 and 70 max leaves on MSLR. The results make sense: we need deeper trees to capture the necessary relationships between features in a large vector.

Turning to efficiency results, Figure 6.5 illustrates per-instance prediction speed for various implementations on the learning-to-rank datasets. Note that this is on the entire ensemble, with latencies now in microseconds instead of nanoseconds. The *x*-axis plots the tree depths from Table 8.1. In this set of experiments, we use the VPRED approach with the vectorization parameter set to 8 for LETOR and 16 for MSLR and C14.

Our findings from synthetic datasets mostly hold on learning-to-rank datasets. OBJECT is the slowest implementation and STRUCT is slightly faster. On the LETOR dataset, STRUCT is only slightly slower than STRUCT$^+$, but on MSLR and C14, STRUCT$^+$ is much faster than STRUCT in most cases. While on LETOR it is clear that there is a large performance gap between CODEGEN and the other approaches, the relative advantage of CODEGEN decreases with deeper trees and larger feature vectors (which is consistent with the synthetic results). VPRED outperforms all other techniques, including CODEGEN on MSLR and C14 but is slower than CODEGEN on LETOR.

6.4 Discussion

We have shown that by adopting predication and vectorization, we can develop architecture-conscious implementations of tree-based models which have superior test-time performance over

Table 6.4: Average percentage of examined features (variance in parentheses) across cross-validation folds using various max leaves settings.

(a) LETOR-MQ2007

	3	5	7	9	11
Percentage of features	76.7 (5.0)	72.2 (8.8)	80.2 (5.6)	77.6 (7.6)	84.8 (1.9)

(b) MSLR-WEB10K

	10	30	50	70
Percentage of features	92.7 (1.7)	96.5 (1.1)	96.3 (1.9)	95.6 (1.6)

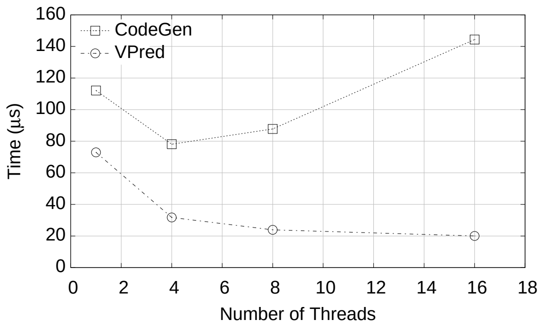
(c) C14

	70	110	150
Percentage of features	88.2 (1.0)	88.6 (1.1)	90.7 (1.6)

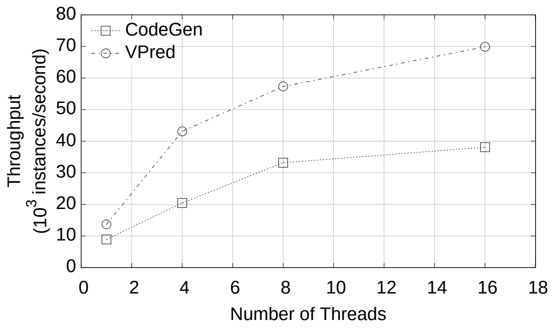
standard implementations. However, throughout this work we have assumed that features are fully computed prior to the last stage (i.e., the document ranking stage). There is an alternative to this strategy: a lazy feature extractor. That is, we compute feature values only when the predicate at a tree node needs to access that particular feature. This on-demand feature extraction is difficult to study, since results will be highly dependent on various factors ranging from the implementation of feature extractors to the cost of individual features. However, if a sufficiently large percentage of features are accessed during the evaluation of a test instance, then it may makes sense to pre-compute the entire feature vector and avoid complexities that accompany a lazy approach.

Table 6.4 shows the average fraction of features accessed in the final ensembles for all three learning-to-rank datasets, with different max leaves configurations. It is clear that for all datasets most of the features are accessed during the course of making a prediction. Therefore, it makes sense to separate feature extraction from prediction. In fact, there are other reasons to do so: a distinct feature extraction stage can benefit from better reference locality (when it comes to document vectors, postings, or whatever underlying data structures are necessary for computing features).

Thus far, all of our experiments have been performed on a single thread, despite the fact that multi-core processors are ubiquitous today. We leave careful consideration of this issue for



(a)



(b)

Figure 6.6: Impact of number of threads on latency and throughput, using C14 with max leaves of 150. In (a), each thread evaluates instances using a subset of trees in the ensemble, thus exploiting intra-query parallelism to reduce latency. In (b), each thread operates independently on instances, thus exploiting inter-query parallelism to increase throughput.

future work, but present some preliminary results here. There are two primary ways we can take advantage of multi-threaded execution: the first is to reduce latency by exploiting intra-query parallelism, while the second is to increase throughput by exploiting inter-query parallelism. We consider each in turn.

Since each tree model is an ensemble of trees, parallelism at an ensemble level can be obtained by assigning each thread to work on disjoint subsets of trees. This has the effect of reducing prediction latency. Figure 6.6(a) shows the impact of this approach on prediction time using the C14 dataset with max leaves set to 150. By using 16 threads, latency decreases by 70% compared with a single-threaded implementation for VPred. Note that since our machine only has eight physical cores, the extra boost obtained in going from 8 to 16 threads comes from hyper-threading. The small performance difference between the two different conditions suggests that our VPred implementation is effectively utilizing available processor resources (therefore, the gains obtained from hyper-threading are limited). For CodeGen, we observe a 30% improvement in latency by going from 1 to 4 threads. However, adding more threads actually increases prediction time.

The alternative multi-threaded design is to use each thread to independently evaluate instances using the entire ensemble. This represents inter-query parallelism and increases throughput. More precisely, each thread evaluates one test instance using CodeGen or v test instances (the batch size) in the case of VPred. Figure 6.6(b) shows the gain in throughput (number of test instances per second) for different number of threads on the C14 dataset with max leaves set to 150. Comparing implementations with a single thread and 16 threads, throughput increases by 320% (from 9K to 38K instances/second) for CodeGen and 400% (from 14K to 70K instances/second) for VPred. This shows that VPred benefits more from multi-core processors.

Chapter 7

Training Efficient Tree Ensembles for Web Learning-to-Rank

We have demonstrated in the previous chapter that the VPred implementation achieves a lower latency among the various implementations studied. As detailed in Section 6.1, VPred evaluates a tree of depth d using d statements per test instance. Therefore, the performance of VPred is directly correlated with the tree depth; deeper trees are slower to evaluate. However, we concluded in Section 6.3.3 that deeper trees lead to higher effectiveness in many cases. These observations suggest that tree depth is a factor that trades off efficiency for effectiveness and vice versa. The question that arises here is that, can we construct tree ensembles that are shallow (hence, faster to evalute) but also highly effective? We here show that the answer is yes.

We can accomplish the best of both worlds by encouraging the LambdaMART algorithm to construct shallower trees. We explore two techniques to make this happen. First, we directly modify the splitting criterion during induction of each individual tree such that the algorithm would penalize splits that increase the tree depth. Second, when generating each tree in the ensemble, we can proceed as normal, and then prune back the tree to a shallower tree before proceeding to the next boosting stage. We explore both approaches in detail in Sections 7.1 and 7.2. Similar to Chapter 6, we evaluate our techniques using standard learning-to-rank datasets, since our tweet collection does not allow thorough experimentation.

Before we proceed further, recall from Section 2.5 that LambdaMART learns a ranking model by sequentially adding a new tree to an ensemble that best accounts for the remaining regression error (i.e., the residuals) of the training samples.

Assuming we have constructed $t - 1$ regression trees, LambdaMART adds the t_{th} tree that greedily minimizes the loss function, given the current regression residuals. To construct an individual regression tree with J terminal nodes, LambdaMART uses the CART [82] algorithm. CART works by recursively splitting the training data. At each step, the algorithm computes the best split (a feature and a threshold) for all the current terminal nodes, and then applies the split that yields the most gain, thereby growing the tree one node at a time. CART maximizes the following gain function:

$$G(N, \langle f, \theta \rangle_N) = \sum_{x_i \in N} (y_i - \bar{y}_N)^2 - C(N, \langle f, \theta \rangle_N), \tag{7.1}$$

where $x_i \in N$ denotes the set of instances that are present in node N, and y_i is the pseudo-response associated with x_i; $\langle f, \theta \rangle_N$ is a split, consisting of a feature and a threshold; and $C(\cdot)$ is defined in Equation 2.3. Equation 7.1 measures the difference in regression error between the current state of the tree and after a split is applied.

7.1 Cost-Sensitive Tree Induction

The cost of evaluating a feature vector using an ensemble of trees is bounded by the following: $\boldsymbol{c}(H_{\boldsymbol{\beta}}) = \sum_{t=1}^{T} c \cdot d_t$, where c is the cost of evaluating one VPRED statement, and d_t is the depth of the t_{th} tree, $h_t \in H_{\boldsymbol{\beta}}$. In other words, c can be associated with the cost of evaluating a tree, when the tree becomes 1 level deeper. Thus, we want to jointly minimize both the loss, $\ell(H_{\boldsymbol{\beta}})$ as well as the cost $\boldsymbol{c}(H_{\boldsymbol{\beta}})$.

In order to solve the above multi-objective problem, we take a greedy approach in which we modify the CART splitting criterion. To do so, the algorithm should not only maximize the gain $G(\cdot)$, but it should also penalize the splits that result in an increase in the depth of the tree under construction, h_t. That is, to select the next node to split, we want to maximize Equation 7.1 in addition to minimizing $\boldsymbol{c}([h_t]) = c \cdot d_t$; we can safely ignore c since it is a constant.

We approach this problem by *a priori* articulating the preferences, where we determine the importance of each objective function and prioritize accordingly. We then solve the problem using the lexicographic approach [121, 122]. This method iterates through the functions sorted in increasing order of importance. At step i, it finds a solution for function f_i. A solution X', must fulfill f_i's constraints, and must also satisfy $f_j(X') < \alpha f_j(X) \; \forall_{j<i}$, where X is the solution found so far, and α is the relaxation parameter. In this way, among all terminal nodes, we find the node

and a split that maximizes Equation 7.1:

$$\underset{N, \langle f, \theta \rangle_N}{\arg\max}\, G(N, \langle f, \theta \rangle_N). \tag{7.2}$$

We denote the solution to Equation (7.2) as N^* and its split $\langle f^*, \theta^* \rangle_{N^*}$. Next, among all terminal nodes, we choose the node that minimizes the depth of the tree d_t, by relaxing a constraint on $G(\cdot)$. That is, our second optimization problem becomes:

$$\begin{aligned} &\underset{N, \langle f, \theta \rangle_N}{\arg\min}\, D(N, \langle f, \theta \rangle_N), \\ &\qquad \text{such that } G(N, \langle f, \theta \rangle_N) > (1 - \rho)\, G(N^*, \langle f^*, \theta^* \rangle_{N^*}), \end{aligned} \tag{7.3}$$

where $\rho \in [0, 1]$. Function $D(\cdot)$ is defined as follows:

$$D(N, \langle f, \theta \rangle_N) = \begin{cases} d_t, & d_N + 1 \leq d_t \\ d_t + 1, & \text{otherwise} \end{cases}, \tag{7.4}$$

where d_t is the current depth of the tree, and d_N is the depth of node N. In the second optimization problem, in order to obtain a single solution, we break ties in favor of a higher gain. Setting ρ to 0 reduces this model to an unmodified LambdaMART model.

We can interpret this optimization problem as follows: If the split that results in maximum gain does not increase the maximum depth of the tree, then continue with the split. Otherwise, find a node closer to the root which, if split, would result in a gain larger than the discounted maximum gain.

7.2 Pruning While Boosting

In the pruning approach, we construct the t_{th} tree using the original CART algorithm, but before proceeding to add the tree to the ensemble, we prune the tree with a focus on depth. Our algorithm starts discarding the two deepest terminal nodes in the tree and turning their parent into a new terminal node (and restoring the original regression value at the leaf pre-split) until a stopping criterion is met (discussed below). However, unlike the method proposed by Ganjisaffar [87], we do not include effectiveness in our criterion and exclusively focus on the depth and density of the tree. The intuition is that, since we are performing the pruning *while* boosting, additional boosting stages can compensate the loss in effectiveness while at the same time reducing the average depth of trees in the ensemble.

In this work, we explore a simple criterion: we continue collapsing terminal nodes until the total number of nodes in the tree is equal or greater than a fraction of the maximum possible number of nodes in the tree given its depth (i.e., in a perfectly balanced tree), as follows:

$$|h_t| \geq \alpha \left(2^{d_t+1} - 1\right), \tag{7.5}$$

where $|h_t|$ is the number of nodes in tree h_t, d_t is the current depth of the tree, and $\alpha \in [0, 1]$ is a parameter. Intuitively, α controls to what extent we want our trees to "look like" perfectly-balanced binary trees. Setting α to 0 reduces this model to the original LambdaMART algorithm. Increasing α translates into more aggressive pruning, and with $\alpha = 1.0$, we prune back the tree until we obtain a perfectly-balanced binary tree. Obviously, this pruning method does not preserve the target number of leaves.

7.3 Experimental Setup

All experiments were run on the same hardware as in previous chapters Similar to the previous chapter, we compiled the code with GCC (version 4.1.2) using optimization flags `-O3`. All code ran single-threaded. To evalute test instances, we used the VPred implementation, introduced in Chapter 6.

We implemented our proposed training algorithms on top of jforests, and limited our experiments to the MSLR-WEB10K dataset. We ported the code to Hadoop, which allows us to run multiple experiments in parallel in order to capture the variance introduced by randomization—we ran $E = 100$ trials per fold as in the previous chapter. Following standard practice, we optimize NDCG [80]. Based on the previous chapter, we set the number of leaves to 70, feature and data sub-sampling parameters to 0.3, minimum observations per leaf to 0.5, and the learning rate to 0.05.

We evaluate the resulting ensembles using the following metrics:

- **Depth (d_{avg})**: the average *maximum* depth of all trees in the ensembles. Specifically, we first compute the average maximum depth of each tree in the ensemble, then compute the mean across all E ensembles.
- **Ensemble size (T_{avg})**: the average number of trees in an ensemble is important since shallower trees might require larger ensembles (i.e., with more stages) to converge.
- **Average total number of nodes (n_{avg})**: the total number of nodes in the entire ensemble, averaged across trials.
- **Latency**, our metric for efficiency (performance): the latency of evaluating a single instance using an ensemble of trees. That is, we measure the time between when a feature vector enters the system to when a relevance score is computed for the input instance. We report the mean latency across all folds (each fold containing $E = 100$ ensembles) in microseconds.
- **NDCG**, our metric for effectiveness: average (across different trials) NDCG at different cutoffs for each fold. We use a Wilcoxon test (p-value< 0.05) to determine statistical significance.

7.4 Results

Table 7.1 summarizes the results for all our experiments, averaged across all trials and all five folds, for both the modified splitting criterion in Table 7.1(a) and the pruning technique in Table 7.1(b). The first column shows the ρ and α settings; the next columns characterize the tree topology: average maximum depth (d_{avg}), the average ensemble size (T_{avg}), and average number of total nodes in the ensemble (n_{avg}). The next set of columns show NDCG values at various cutoffs. The final two columns show the prediction latency (with 95% confidence intervals in parentheses) and relative improvement over the baseline. The first entry in Table 7.1(a) and Table 7.1(b) shows the unmodified LambdaMART baseline.

For the approach involving modifications to the CART splitting criterion: results suggest that increasing ρ (i.e., the relaxation parameter) yields shallower trees—more precisely, trees with smaller maximum average depths. In some cases, the sizes of the ensembles increase slightly. Note,

Table 7.1: Tree topology, effectiveness, and efficiency results on the MSLR-WEB10K dataset, averaged across 5 folds. The columns show topological properties of the tree, effectiveness (NDCG at various cutoffs), and latency (with 95% confidence intervals). For NDCG, * denotes statistical significance with respect to the baseline (p-value < 0.05).

(a) Modified Splitting Criterion

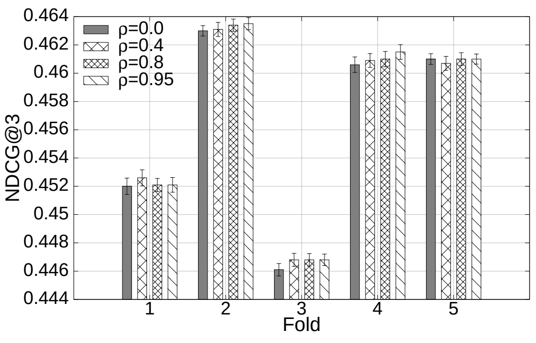
				NDCG							
ρ	d_{avg}	T_{avg}	n_{avg}	@1	@3	@5	@10	@20	@∞	Latency	Δ
0.0	17.8	321	44115	0.471	0.456	0.461	0.480	0.511	0.722	19.4 ($\pm$0.6)	
0.4	15.7	327	45462	0.471	0.457	0.461	0.480	0.511	0.722	16.6 ($\pm$0.5)	−14.4%
0.8	12.3	317	44173	0.471	0.457	0.461	0.480	0.511	0.722	12.9 ($\pm$0.4)	−33.5%
0.95	11.7	325	44421	0.472	0.457	0.461	0.480	0.511	0.722	12.4 ($\pm$0.3)	−36.1%

(b) Pruning

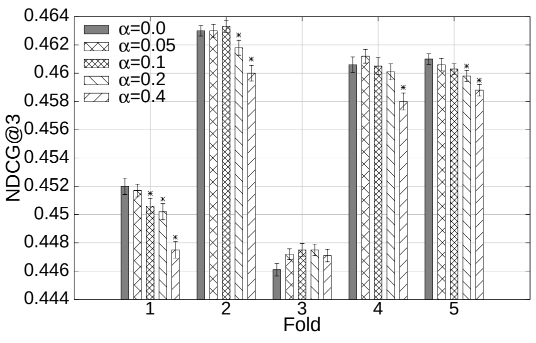
				NDCG							
α	d_{avg}	T_{avg}	n_{avg}	@1	@3	@5	@10	@20	@∞	Latency	Δ
0.0	17.8	321	44115	0.471	0.456	0.461	0.480	0.511	0.722	19.4 ($\pm$0.6)	
0.05	9.1	395	30554	0.471	0.457	0.461	0.479	0.510*	0.721*	10.7 ($\pm$0.3)	−44.8%
0.1	7.8	432	26228	0.472	0.456	0.460	0.479	0.510*	0.721*	9.5 ($\pm$0.3)	−51.0%
0.2	6.4	474	22372	0.471	0.456	0.460	0.478*	0.509*	0.721*	8.5 ($\pm$0.2)	−56.2%
0.4	4.8	567	17524	0.469*	0.454*	0.458*	0.476*	0.507*	0.720*	7.6 ($\pm$0.2)	−60.8%
0.6	3.6	675	13660	0.467*	0.452*	0.456*	0.474*	0.505*	0.719*	7.0 ($\pm$0.2)	−63.9%

however, that the total number of nodes in the ensemble remains about the same—we get shallower and more balanced trees. While we observe no difference in effectiveness as a result of changing ρ, latency improves significantly (i.e., confidence intervals do not overlap). Figure 7.1(a) and 7.2(a), which show NDCG@3 and query latency per fold for different values of ρ, suggest that the findings are consistent across all cross-validation sets. For both figures, we show 95% confidence intervals across the trials in each condition. Setting ρ to a value greater than 0.95 yields results that are indistinguishable from $\rho = 0.95$, and for the sake of brevity, we do not include them in the result set.

For the pruning approach: increasing the parameter α also improves latency significantly. There is a considerable increase in the number of trees in the ensemble as we increase α, but the individual trees are much shallower and more balanced. Furthermore, the average total number of nodes in the ensembles decreases significantly (unlike with the modified splitting criterion). With pruning, we obtain shallow trees, though not necessarily balanced by the design of the α parameter. However, aggressive pruning comes at the cost of lower effectiveness. As we increase α, we get significantly lower NDCG values, first at higher cutoffs, then at lower cutoffs. However, these

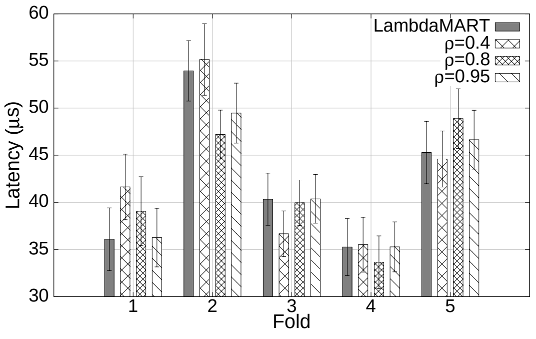


(a) Modified Splitting Criterion

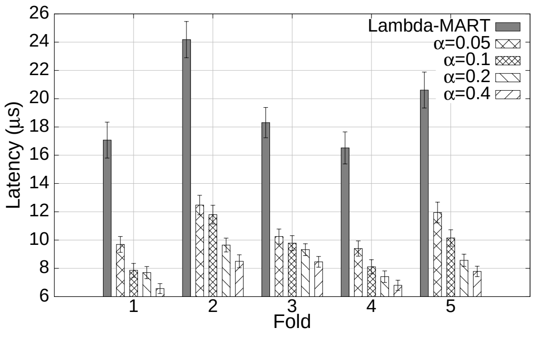


(b) Pruning

Figure 7.1: Mean NDCG@3 across all trials per fold, with 95% confidence intervals. * shows statistical significance.



(a) Modified Splitting Criterion



(b) Pruning

Figure 7.2: Mean average latency (in microseconds) across all trials per fold, with 95% confidence intervals.

reductions are very small. At $\alpha = 0.6$, we obtain significantly lower NDCG at all cutoff values, and thus we did not further explore larger values of α. Figure 7.1(b) and 7.2(b) show per-fold NDCG@3 and query latency with confidence intervals for different values of α. Overall, $\alpha = 0.1$ appears to be a good setting, yielding approximately 51% decrease in latency while decreasing NDCG@20 by only a small amount and leaving NDCG at the other cutoffs unaffected.

We have experimented with a select set of parameters on the C14 dataset using 150 as the maximum number of leaves, and measured the evaluation metrics across $E = 20$ trials. As Table 7.2 shows, results carry over from the MSLR dataset: Setting ρ to 0.95 in the approach with modified splitting criterion leads to shallower trees and slightly larger ensembles. However, latency

Table 7.2: Tree topology, effectiveness, and efficiency results on the C14 dataset, averaged across 5 folds. Columns as in Table 7.1.

				NDCG							
	d_{avg}	T_{avg}	n_{avg}	@1	@3	@5	@10	@20	@∞	Latency	Δ
$\rho, \alpha = 0$	35.9	527	158748	0.712	0.716	0.736	0.779	0.821	0.863	72.9 ($\pm$1.2)	
$\rho = 0.95$	24.3	538	161093	0.712	0.716	0.736	0.779	0.821	0.863	49.2 ($\pm$1.4)	-32.5%
$\alpha = 0.1$	17.3	643	91629	0.712	0.716	0.736	0.778^*	0.819^*	0.862^*	38.8 ($\pm$0.9)	-46.8%

improves by about 32% over the original LambdaMART algorithm, without loss in effectiveness. On the other hand, setting α to 0.1 in the tree pruning approach leads to even shallower trees, but significantly larger number of trees in the ensemble. However, there are fewer nodes in the final ensemble. While query latency improves by about 46%, NDCG at cutoffs $\{10, 20\}$ as well as mean NDCG degrade significantly.

Chapter 8

End-to-End Efficiency-Effectiveness Tradeoffs

Previous chapters have demonstrated our contributions to each stage of a learning-to-rank pipeline in isolation. These include an approximate top k retrieval algorithm for microblog documents, the use of document vectors for feature extraction, architecture-conscious implementations of tree ensembles, and training mechanisms to learn tree ensmbles that have a faster runtime execution.

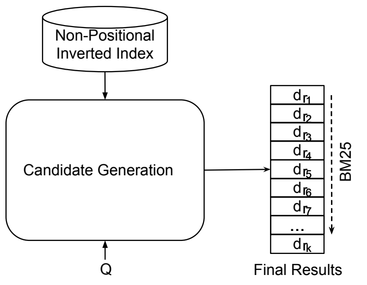
While we have discussed the impact of our contributions on each individual stage in previous chapters, we have so far not explored how the end-to-end learning-to-rank pipeline benefits from our contributions. We here present end-to-end experiments that measure the overall impact of our contributions on retrieval. Results show that the techniques we introduce in this dissertation significantly improve query latency with no detriment to search quality.

8.1 Setup

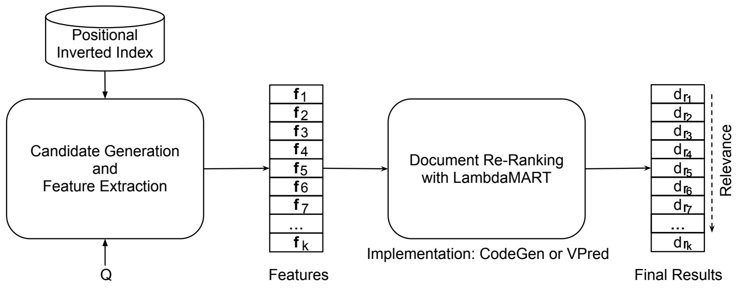
We explore the impact of our contributions on end-to-end efficiency and effectiveness. To that end, we start with a monolithic learning-to-rank architecture, and progressively decouple the stages and apply the best of our techniques from each chapter, until we obtain the multi-stage architecture as illustrated in Figure 1.3. As such, the end-to-end configurations that we consider are as follows:

- BM25: As a baseline, we generate candidates using WAND with full BM25 scoring function and rank candidates according to their scores—in this scenario we do not use a learning-to-rank model. This configuration is illustrated in Figure 8.1(a).

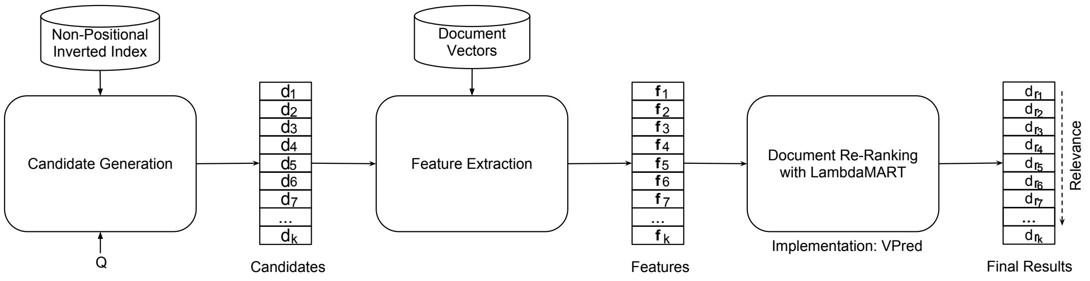
- $\text{Monolithic}_{\text{CodeGen}}$: This architecture uses a positional inverted index to generate candidates *and* extract features, before passing the feature vectors to a LambdaMART ensemble for evaluation. To generate candidates, we use the WAND algorithm with the IDF scoring function from Chapter 4 (denoted WAND_{IDF}). The tree model is implemented as hard-coded if-else



(a) BM25


Positional Inverted Index
Candidate Generation and Feature Extraction
Q
f1
f2
f3
f4
f5
f6
f7
...
fk
Features
Document Re-Ranking with LambdaMART
Implementation: CodeGen or VPred
dr1
dr2
dr3
dr4
dr5
dr6
dr7
...
drk
Relevance
Final Results

(b) Monolithic. Monolithic$_{\text{CodeGen}}$ uses CodeGen in the final stage, while Monolithic$_{\text{VPred}}$ uses VPred. Candidate Generation uses Wand$_{\text{IDF}}$.



(c) Multi-Stage. Multi-Stage$_{\text{Wand}}$ uses Wand$_{\text{IDF}}$ in Candidate Generation, while Multi-Stage$_{\text{BWand}}$ uses BWand with $r = 8$, and $\kappa = 1$.

Figure 8.1: Architectures used in end-to-end experiments.

blocks. Figure 8.1(b) depicts this configuration.

- Monolithic$_{\text{VPred}}$: The setup is similar to Monolithic$_{\text{CodeGen}}$, however, this setting evaluates the LambdaMART ensemble using the VPred implementation from Chapter 6. This variant measures the impact of our architecture-conscious tree implementation on the end-to-end speed.

- Multi-Stage$_{\text{Wand}}$: We break the candidate generation and feature extration tasks into two distinct stages. In candidate generation, similar to previous configurations, we use the Wand$_{\text{IDF}}$

algorithm to retrieve top k documents, but now using a non-positional inverted index. The feature extraction stage takes advantage of document vectors, and extracts features using the algorithm presented in Chapter 5. Finally, the last stage re-ranks documents using a LambdaMART model with the VPred implementation. This system evaluates the advantages of decoupling candidate generation and feature extraction against the previous system. This configuration is depicted in Figure 8.1(c).

- Multi-Stage$_{\text{BWand}}$: Similar to Multi-Stage$_{\text{Wand}}$, but it uses the BWand algorithm from Chapter 4 to generate top k candidates using a non-positional inverted index. This system measures the impact of our contributions to the candidate generation stage.

We also evaluate Monolithic and Multi-Stage configurations using a LambdaMART model that is trained using the techniques detailed in Chapter 7. We use $\rho = 0.95$ and $\alpha = 0.1$ in our experiments. Additionally, all Monolithic and Multi-Stage configurations can be run in either conjunctive or disjunctive mode.

We perform our experiments on the Tweets2011 collection, introduced in previous chapters, and perform stemming and remove stopwords. We again use three classes of queries for evaluation purposes: the TREC 2011-2012 microblog track queries (consisting of 110 queries) for effectiveness experiments, and AOL and TREC 2005 terabyte track queries for efficiency experiments. We set k, the size of the candidate set, to 1000. Note that, to measure per-query latency, we measure the elapsed time to process all queries, and divide by the number of queries.

For effectiveness experiments, we randomly split the microblog queries into three equally-sized subsets and perform cross-validation. We tune the LambdaMART training parameters using the validation sets: minimum observations per leaf is set to 0.75, the learning rate is set to 0.05, feature and data sub-sampling are set to 0.3, and maximum number of leaves is set to 15. Since the LambdaMART algorithm is randomized, we construct $E = 100$ ensembles per cross-validation fold and report average effectiveness across all ensembles.

To train a LambdaMART model, we use the same feature set as in Chapter 5 in our experiments. In addition, we also include tweet-specific features such as: presence of the "@" character, presence of links, number of unique characters in the document (intended to penalize terms like "haaaaa"), ratio of at-metions to terms, and number of unique terms. Since these features are query-independent, feature extraction boils down to a table lookup which has a constant cost. There are 27 features in total, which is approximately the size of feature vectors in the LETOR dataset.

We post-process the final ranked list to discard retweets as well as tweets that were created after the query time. This makes our evaluations consistent with the TREC microblog track guidelines.

Given the emphasis on early precision in the web context, we measure precision and NDCG at various cutoffs [80]. The ubiquity of Twitter mobile usage also suggests that these metrics are appropriate—the limited screen size of mobile devices means that we should emphasize early precision. We use a paired t-test to measure statistical signifiance.

For efficiency experiments, we evaluate test instances using all $E = 100$ ensembles per fold (i.e., 300 experiments total) and report the average latency along with 95% confidence intervals. We use the same hardware as in previous chapters to run these experiments. We had exclusive access to the machine with no competing processes.

8.2 Effectiveness

This section examines end-to-end retrieval scenarios in terms of effectiveness. As discussed in previous chapters, there are two components that trade off effectiveness for efficiency and index size. First, simplifications and approximations that led to WAND_{IDF} and BWAND in the candidate generation stage enable faster query evaluation, but degrade relative recall against an exact baseline. Second, modifications to the LambdaMART training algorithm in Chapter 7 can also improve query latency, but those approaches too lead to loss in effectiveness in some cases.

To understand these factors, we first study the impact on effectiveness of our techniques in the candidate generation stage. Next, we measure effectiveness resulting from modified LambdaMART models and compare with effectiveness obtained by the original LambdaMART model.

8.2.1 Candidate Generation

Recall from Chapter 4 that a number of simplifications to the BM25 scoring function (namely, ignoring term frequencies and length normalization) has led to the WAND_{IDF} algorithm. Further, by incorporating Bloom filters and requiring the presence of the rarest term, we designed the BWAND algorithm, which has approximate aspects. We showed that the BWAND algorithm yields high relative recall against an exact baseline, as a component-level metric. However, we did not evaluate these algorithms in terms of end-to-end effectiveness.

In order to contextualize the effectiveness obtained by the WAND_{IDF} and BWAND algorithms, we need exact baselines. As such, we use SvS for conjunctive experiments and $\text{WAND}_{\text{BM25}}$ (i.e., WAND with full BM25 scoring function) for disjunctive experiments. To make a fair comparison between all retrieval scenarios, we re-rank all candidates using a LambdaMART model. Therefore, the end-to-end systems of interest are as follows:

- Conjunctive query processing
  - SvS followed by LambdaMART. $\text{Multi-Stage}_{\text{WAND}}$ and the two Monolithic end-to-end configurations yield effectiveness equal to that of the SvS condition.
  - $\text{Multi-Stage}_{\text{BWAND}}$: Comparing this with the SvS condition highlights the impact of Bloom filter false positives on effectiveness.
- Disjunctive query processing
  - $\text{WAND}_{\text{BM25}}$ followed by LambdaMART
  - $\text{Multi-Stage}_{\text{WAND}}$: Comparing this condition with the previous condition assesses the impact of simplifications to the scoring function. Note that, the two Monolithic configurations yield effectiveness equal to that of $\text{Multi-Stage}_{\text{WAND}}$.
  - $\text{Multi-Stage}_{\text{BWAND}}$: This condition introduces additional constraints by requiring the presence of the rarest query term and by allowing false positives due to its use of Bloom filters. Comparing this condition with previous conditions further measures the impact of these changes on effectiveness.

Table 8.1: End-to-end effectiveness in terms of precision and NDCG at various cutoffs for different candidate generation algorithms. * indicates statistical significance with respect to the SvS condition.

(a) Conjunctive Query Processing

	Precision			NDCG			
	@5	@10	@30	@1	@3	@5	@10
SvS	0.29	0.24	0.14	0.26	0.24	0.23	0.21
Multi-Stage$_{\text{BWAND}}$ (8,1)	0.32*	0.28*	0.18*	0.26	0.25	0.25*	0.24*
Multi-Stage$_{\text{BWAND}}$ (24,3)	0.29	0.24	0.14	0.26	0.24	0.23	0.21

(b) Disjunctive Query Processing

	Precision			NDCG			
	@5	@10	@30	@1	@3	@5	@10
BM25	0.42	0.39	0.30	0.34	0.34	0.33	0.32
WAND$_{\text{BM25}}$	0.47	0.46	0.36	0.39	0.38	0.39	0.37
Multi-Stage$_{\text{WAND}}$	0.47	0.45	0.36	0.39	0.38	0.39	0.37
Multi-Stage$_{\text{BWAND}}$ (8,1)	0.47	0.45	0.35	0.39	0.38	0.38	0.37
Multi-Stage$_{\text{BWAND}}$ (24,3)	0.47	0.46	0.36	0.39	0.38	0.38	0.37

Table 8.1 shows precision and NDCG at various cutoffs, averaged across the cross-validation sets. For Multi-Stage$_{\text{BWAND}}$, the relevant Bloom filter parameter settings are r (number of bits per element) and κ (number of hash functions): in these experiments, we report results for $r = 8, \kappa = 1$, denoted (8,1) in the tables, and $r = 24, \kappa = 3$, denoted (24,3) in the tables; these represent the extremes of the parameter value ranges we explore. Table 8.2 provides statistics for retrieved results at rank ten to offer more context: it shows the number of topics for which the algorithm retrieves at least one document, the number of unjudged tweets, and the number of retrieved tweets that are non-relevant (0), relevant (1), and highly-relevant (2).

We first examine the conjunctive results. From Table 8.2, the SvS condition returned results for only 81 out of 110 topics, whereas the Multi-Stage$_{\text{BWAND}}$ (8,1) condition returned results for 101 topics; the Multi-Stage$_{\text{BWAND}}$ (24,3) condition returned results for 82 topics. Note that in all cases we compute the effectiveness metrics in Table 8.1 over all 110 topics. Some examples of queries for which SvS did not return any results include "Pakistan diplomat arrest murder" and "Dog Whisperer Cesar Millans techniques." For the most part, they are long queries that appear to over-specify the information need. On the other hand, the Multi-Stage$_{\text{BWAND}}$ (8,1) condition retrieves results for some of these long topics. This is happening because of the false positive errors

Table 8.2: Statistics for retrieved results at rank ten cutoff for different candidate generation algorithms. Last three columns indicate relevance grades: non-relevant (0), relevant (1), and highly-relevant (2).

(a) Conjunctive Query Processing

	#topics	unjudged	0	1	2
SvS	81	40	198	176	159
Multi-Stage$_{\text{BWAND}}$ (8,1)	101	103	327	224	181
Multi-Stage$_{\text{BWAND}}$ (24,3)	82	46	198	183	156

(b) Disjunctive Query Processing

	#topics	unjudged	0	1	2
BM25	110	89	526	278	207
WAND$_{\text{BM25}}$	110	87	489	297	227
Multi-Stage$_{\text{WAND}}$	110	81	497	299	223
Multi-Stage$_{\text{BWAND}}$ (8,1)	110	86	512	281	221
Multi-Stage$_{\text{BWAND}}$ (24,3)	110	84	509	285	222

associated with Bloom filters; the retrieved results do not actually contain all terms. In cases where exact postings list intersection (SvS) returns an empty set, the false positives become advantageous since something is always better than nothing. At the same time, in cases where exact postings list intersection returns a non-empty set, the false positives introduce noise. However, the noise does not appear to have a negative impact on the final rasults ranked by the LambdaMART model. Overall, Multi-Stage$_{\text{BWAND}}$ (8,1) achieves significantly higher precision at all cutoffs and NDCG at cutoffs five and ten. For Multi-Stage$_{\text{BWAND}}$ (24,3), we observe no significant differences compared to SvS.

We now turn our attention to the disjunctive query processing results. For reference, we included effectiveness results for the BM25 condition—recall that no learning-to-rank model is used in this condition. All conditions in disjuncitve query processing mode significantly outperform the BM25 condition. The results in Table 8.1 show no significant differences between WAND$_{\text{BM25}}$ and Multi-Stage$_{\text{WAND}}$ in disjunctive query processing mode. There are also no significant differences between WAND$_{\text{BM25}}$ and Multi-Stage$_{\text{BWAND}}$ in disjunctive query processing mode, with either Bloom filter parameter setting. Looking at Table 8.2, we can confirm that these results do not appear to be the product of unjudged documents or idiosyncrasies in the way that judgment pools were constructed during the TREC evaluations.

Table 8.3: End-to-end effectiveness in terms of precision and NDCG at various cutoffs for different configurations (W for Multi-Stage$_{\text{WAND}}$ and B for Multi-Stage$_{\text{BWAND}}$). Models include a LambdaMART model along with the modified LambdaMART model with $\rho = 0.95$ and $\alpha = 0.1$. * shows statistical significance of models with $\rho = 0.95$ and $\alpha = 0.1$ against LambdaMART.

		Precision			NDCG			
		@5	@10	@30	@1	@3	@5	@10
W	λ-MART	0.47	0.45	0.36	0.39	0.38	0.39	0.37
W	$\rho = 0.95$	0.47	0.45	0.35	0.40	0.38	0.38	0.38
W	$\alpha = 0.1$	0.47	0.44	0.34*	0.39	0.38	0.36*	0.35*
B	λ-MART	0.47	0.45	0.35	0.39	0.38	0.38	0.37
B	$\rho = 0.95$	0.47	0.45	0.35	0.39	0.37	0.37	0.37
B	$\alpha = 0.1$	0.46	0.44	0.33*	0.38	0.37	0.35*	0.34*

Overall, we see that disjunctive query processing is more effective than conjunctive query processing. We believe the small size of the document collection is a major cause; for approximately one quarter of the topics in our test collection, SvS returned zero results. Note here that our tweet collection is only a small sample of the entire tweet stream—it is perhaps possible that we would obtain different results using a larger collection. Regardless, our experiments confirm that our BWAND candidate generation algorithm with Bloom filter chains yields end-to-end effectiveness that is statistically indistinguishable from exact algorithms in both conjunctive query processing mode and disjunctive query processing mode.

Due to the superiority of disjunctive query processing over conjunctive query processing on our tweet collection, we limit the rest of this chapter to disjunctive query processing. As such, the following sections run all end-to-end experiments in disjunctive mode. Throughout the remaining sections, we also fix the BWAND parameters r and κ to 8 and 1 respectively.

8.2.2 Tuning LambdaMART Model

We have established that the use of WAND$_{\text{IDF}}$ or BWAND to generate candidates results in no significant loss in effectiveness against an exact baseline; that is, the Monolithic and Multi-Stage configurations are on par with an exact baseline in terms of effectiveness. We now explore the impact of modifications to LambdaMART presented in Chapter 7.

Table 8.3 shows the end-to-end effectiveness in terms of precision and NDCG for different configurations. The two Monolithic conditions result in effectiveness that is equal to Multi-

Table 8.4: Query latency (ms) for the TREC 2005 terabyte queries and the AOL queries, using various configurations. Models include a LambdaMART model along with the modified LambdaMART model with $\rho = 0.95$ and $\alpha = 0.1$.

	TREC 2005			AOL		
Model	λ-MART	$\rho = 0.95$	$\alpha = 0.1$	λ-MART	$\rho = 0.95$	$\alpha = 0.1$
Monolithic$_{\text{CodeGen}}$	4.9 ($\pm$0.1)	4.9 ($\pm$0.1)	4.6 ($\pm$0.1)	6.7 ($\pm$0.1)	6.6 ($\pm$0.1)	6.4 ($\pm$0.1)
Monolithic$_{\text{VPred}}$	4.8 ($\pm$0.1)	4.8 ($\pm$0.1)	4.5 ($\pm$0.1)	6.6 ($\pm$0.1)	6.5 ($\pm$0.1)	6.4 ($\pm$0.1)
Multi-Stage$_{\text{WAND}}$	4.3 ($\pm$0.1)	4.3 ($\pm$0.1)	4.1 ($\pm$0.1)	5.8 ($\pm$0.1)	5.7 ($\pm$0.1)	5.5 ($\pm$0.1)
Multi-Stage$_{\text{BWAND}}$	2.3 ($\pm$0.1)	2.2 ($\pm$0.1)	2.0 ($\pm$0.1)	2.4 ($\pm$0.1)	2.3 ($\pm$0.1)	2.1 ($\pm$0.1)
BM25	1.7			3.9		

Table 8.5: Query latency (ms) for the TREC 2005 terabyte queries and the AOL queries, using various configurations, broken down by stages. For clarity in presentation, we only include results for an unmodified LambdaMART model.

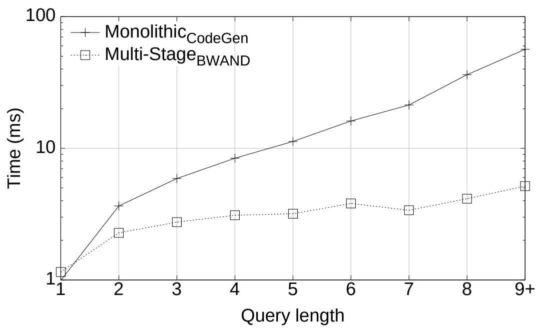
	TREC 2005				AOL			
Stages	Total	Candidates	Features	Re-ranking	Total	Candidates	Features	Re-ranking
Monolithic$_{\text{CodeGen}}$	4.9	4.2		0.7	6.7	6.1		0.6
Monolithic$_{\text{VPred}}$	4.8	4.2		0.6	6.6	6.1		0.5
Multi-Stage$_{\text{WAND}}$	4.3	0.9	2.8	0.6	5.8	2.0	3.3	0.5
Multi-Stage$_{\text{BWAND}}$	2.3	0.1	1.8	0.4	2.4	0.1	1.9	0.4

Stage$_{\text{WAND}}$, therefore they are excluded from this table. For both conditions, setting ρ to 0.95 leads to precision and NDCG that is indistinguishable from a LambdaMART model. On the other hand, setting α to 0.1 for pruning degrades NDCG at cutoffs $\{5, 10\}$ as well as precision @30 significantly, as anticipated. These results are consistent with the results in Chapter 7.

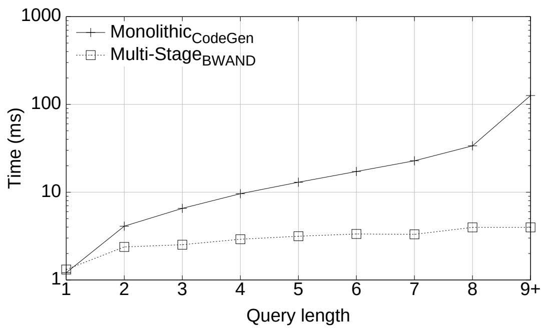
8.3 Query Latency

Next, we turn our attention to query latency. Table 8.4 shows per-query latency values in milliseconds for all retrieval systems, to retrieve and rank top 1000 documents. Reported values are the average over multiple trials and are accompanied with 95% confidence intervals. Note that the BM25 condition uses the full BM25 scoring function to retrieve and rank documents. For reference, Table 8.5 shows a break-down of the results for the unmodified LambdaMART model using different configurations.

Let us first consider the LambdaMART model. Moving from the Monolithic$_{\text{CodeGen}}$ configuration to Monolithic$_{\text{VPred}}$ on both query sets highlights the improvements gained by our VPred implementation over hard-coded if-else blocks. However, the improvements do not seem to be



(a) Terabyte



(b) AOL

Figure 8.2: End-to-end per-query latency.

statistically significant (i.e., confidence intervals overlap). This is primarily because our ensembles have a small average depth (about 8.6); from Chapter 6, we know that the performance of the VPred implementation is on par with the CodeGen approach for shallow trees.

A comparison of Monolithic$_{\text{VPred}}$ and Multi-Stage$_{\text{Wand}}$ shows that separating candidate generation and feature extraction—and subsequently using different data structures in each stage—improves query latency by about 10% on both query sets. Note again that the tree ensembles are evaluated using the VPred algorithm in both systems.

Finally, changing Wand to BWand further lowers end-to-end query latency by about 46% for the terabyte queries and about 58% for the AOL queries. There are two factors that contribute to this gain. First, the superior efficiency of the BWand algorithm over the Wand algorithm accounts for about 40% of the speedup. Second, because BWand requires the presence of the rarest term, the size of the candidate set is for many queries smaller than what is obtained by Wand. As a result, per-query feature extraction becomes less expensive as well.

Figure 8.2 breaks down the end-to-end query latency for Monolithic$_{\text{CodeGen}}$ and Multi-Stage$_{\text{BWand}}$. From these figures, we observe that Multi-Stage$_{\text{BWand}}$ is relatively insensitive to query length, while the latency of Monolithic$_{\text{CodeGen}}$ increases exponentially for longer queries. This phenomenon is not suprising, given the behavior of the Wand algorithm as explained in Chapter 4.

Overall, by comparing Monolithic$_{\text{CodeGen}}$ and Multi-Stage$_{\text{BWand}}$ we observe a 53% decrease

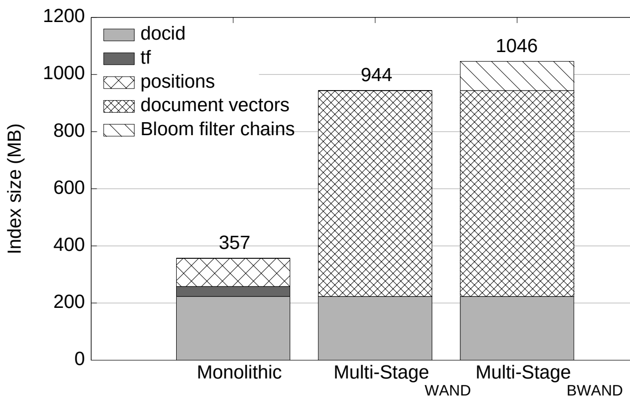


Figure 8.3: Index size required for each system.

in end-to-end latency for the terabyte queries and a 64% decrease for the AOL queries. We should note that, these gains are highly dependent on the topology of the ensemble, size of the collection, and complexity and number of features.

Additionally, we trained modified LambdaMART models using $\rho = 0.95$ and $\alpha = 0.1$ from Chapter 7, and evaluated all systems on the new models. Results show no significant change to query latency as a result of setting ρ to 0.95. This is not surprising since the trees are generally shallow to begin with, therefore increasing ρ does not translate into much shallower trees—new average tree depth is 7.5, as opposed to LambdaMART's 8.6. However, pruning with $\alpha = 0.1$ does result in a small but statistically significant gain over the original LambdaMART model—average depth is now 5.7. However, as the effectiveness results suggest, pruning the tree results in significant loss in P@30 and NDCG at rank cutoffs 5 and 10.

8.4 Memory Usage

Finally, we turn to memory usage. The Monolithic configurations require a positional inverted index for candidate generation and feature extraction. On the other hand, the Multi-Stage configurations operate on a non-positional inverted index in the candidate generation stage, but additionally need document vectors in the feature extraction stage. Multi-Stage$_{\text{BWAND}}$ also requires Bloom filter chains in addition to the inverted index in the candidate generation stage.

Figure 8.3 shows the memory requirements for each system. As noted in Chapter 5, term positions and term frequency values are generally small because tweets are of limited length. However, term ids could be relatively very large, which is why document vectors alone are far larger than a positional inverted index. The large size of document vectors makes the overall memory footprint of Multi-Stage$_{\text{WAND}}$ 2.6 times that of Monolithic. However, in practice, the search service needs to show the content of tweets to the user, which ultimately would require raw documents. Therefore, the need for document vectors should not be considered an overhead of our approach. Bloom filter chains with $r = 8$ additionally increase the overall memory footprint. The overall memory footprint used in Multi-Stage$_{\text{BWAND}}$ is about 2.9 times that of Monolithic.

Chapter 9

Conclusion

In this dissertation, we first introduced an efficient indexing mechanism that operates entirely in the main memory. By using additional working memory, our indexer is capable of indexing streaming documents in real-time, and provides a mechanism to control inverted list contiguity. We have also laid out a novel three-stage architecture for document ranking. We have shown that our three-stage architecture offers more flexibility and allows for more efficient ranking.

Within this three-stage architecture, we explored different techniques to improve the latency of each stage in isolation. As such, we introduced a novel approximate top k retrieval algorithm that is many times faster than a strong baseline; we examined the use of document vectors in features extraction and discussed their impact on speed and space; we presented a novel implementation of trees which enables faster document re-ranking using tree-based learning-to-rank models; and finally, we designed a new algorithm to train tree-based learning-to-rank models that are faster to evaluate. By putting all these pieces together to reconstruct our three-stage architecture, we have shown that our techniques can improve end-to-end query latency by 50–60%, without degrading effectiveness. We showed that our architecture leads to larger memory requirements, but explained that this increase in memory footprint can be justified within a broader search service and in the context of modern server configurations.

Despite the overall improvements that the techniques introduced in this work bring about, there are a number of limitations that need to be addressed. We leave these issues as future work. These limitations are as follows:

- Indexing and memory requirements: Our indexing approach as described in Chapter 3 requires a considerable amount of transient memory to guarantee some degree of postings list contiguity. To reduce memory requirements, we should devise alternative approaches that allow us to control postings list contiguity.

- BWand and web documents: Our BWand algorithm was designed specifically for tweets and does not generalize to documents of arbitrary length; we showed that since tweets are of limited length term frequencies are not important, while in contrast, web documents have arbitrary sizes and term frequencies are important in determining relevance. We can resolve this issue by using counting Bloom filters [123] which allow us to store counts associated with keys. By paying a higher memory cost, we could store term frequency information in these data structures and perform query evaluation in domains where term frequency is important.

- Multi-threading: We have conducted preliminary experiments on the impact of multi-threading on different tree implementations in Chapter 6. However, a more thorough study is necessary to understand the impact of multi-threading on cache behavior in all stages.

- Efficient tree ensembles: In Chapter 7 we presented two general techniques to train LambdaMART models that are more efficient to traverse. The impact of each technique depends on the heuristics with which we solved the underlying problems. As a result, it is important to explore other heuristics and other approaches to solve the formulated optimization problems. For example, it is important to study the impact of other stopping criteria on pruning. We leave this as future work.

- Concurrent query evaluation: In this work, we did not consider interleaving of indexing and query evaluation. Concurrent query evaluation and indexing requires tackling a host of concurrency issues which should be comprehensively studied as future work.

- Scaling out: Scaling out a learning-to-rank architecture in a distributed manner is a question that we did not explore in this dissertation. In particular, the impact of distribution of a multi-stage architecture on memory usage and cache behavior, as well as the introduction of network costs to the equation should be studied.

This dissertation has highlighted the importance of efficiency in web search in the wake of drastic changes to the web, including the rise of social media. The techniques introduced in this work represent one solution to balance search efficiency and effectiveness in practice. As web becomes even more dynamic and information continues to be disseminated in real-time, adapting search services to conform to real-time settings will become ever more important.

Bibliography

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 137–150, San Francisco, California, USA, 2004.

[2] Anthony Tomasic, Héctor García-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD 1994)*, pages 289–300, Minneapolis, Minnesota, USA, 1994.

[3] Alistair Moffat and Timothy Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, 1995.

[4] Steffen Heinz and Justin Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science*, 54(8):713–729, 2003.

[5] Nicholas Lester, Justin Zobel, and Hugh Williams. Efficient online index maintenance for contiguous inverted lists. *IP&M*, 42(4):916–933, 2006.

[6] Trevor Strohman and W. Bruce Croft. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2007)*, pages 175–182, Amsterdam, The Netherlands, 2007.

[7] Stephen E. Robertson, Steve Walker, Micheline Hancock-Beaulieu, Mike Gatford, and A. Payne. Okapi at TREC-4. In *Proceedings of the Fourth Text REtrieval Conference (TREC-4)*, pages 73–96, Gaithersburg, Maryland, USA, 1995.

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

[9] Yasser Ganjisaffar, Rich Caruana, and Cristina Lopes. Bagging gradient-boosted trees for high precision, low variance ranking models. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2011)*, pages 85–94, Beijing, China, 2011.

[10] B. Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, Jiang Chen, Ciya Liao, Zhaohui Zheng, and Jon Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM 2010)*, pages 411–420, New York, New York, USA, 2010.

[11] Irina Matveeva, Chris Burges, Timo Burkard, Andy Laucius, and Leon Wong. High accuracy retrieval with multiple nested ranker. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2006)*, pages 437–444, Seattle, Washington, USA, 2006.

[12] Lidan Wang, Jimmy Lin, and Donald Metzler. A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2011)*, pages 105–114, Beijing, China, 2011.

[13] Andrei Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 12th International Conference on Information and Knowledge Management (CIKM 2003)*, pages 426–434, New Orleans, Louisiana, USA, 2003.

[14] Shirish Tatikonda, B. Barla Cambazoglu, and Flavio Junqueira. Posting list intersection on multicore architectures. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2011)*, pages 963–972, Beijing, China, 2011.

[15] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the Web. Stanford Digital Library Working Paper SIDL-WP-1999-0120, Stanford University, 1999.

[16] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[17] R. Lempel and S. Moran. The stochastic approach for link-structure analysis (SALSA) and the TKC effect. *Computer Networks*, 33(1-6):387–401, 2000.

[18] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[19] Nima Asadi and Jimmy Lin. Fast candidate generation for real-time tweet search with bloom filter chains. *ACM Transactions on Information Systems, in press*, 2013.

[20] Nima Asadi and Jimmy Lin. Document vector representations for feature extraction in multi-stage document ranking. *Information Retrieval, in press*, 2012.

[21] Nima Asadi, Jimmy Lin, and Arjen P. de Vries. Runtime optimizations for tree-based machine learning models. *IEEE Transactions on Knowledge and Data Engineering, in press*, 2013.

[22] Nima Asadi and Jimmy Lin. Training efficient tree-based models for document ranking. In *Proceedings the 35th European Conference on Information Retrieval (ECIR 2013)*, pages 146–157, Moscow, Russia, 2013.

[23] Nima Asadi and Jimmy Lin. Effectiveness-efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In *To Appear: Proceedings of the 36th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2013)*, Dublin, Ireland, 2013.

[24] Vo Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2001)*, pages 35–42, New Orleans, Louisiana, USA, 2001.

[25] Vo Anh and Alistair Moffat. Simplified similarity scoring using term ranks. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2005)*, pages 226–233, Salvador, Brazil, 2005.

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

[27] Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *Proceedings of the 27th Australasian Conference on Computer Science - Volume 26 (ACSC 2004)*, pages 15–23, Dunedin, New Zealand, 2004.

[28] Edward A. Fox and Whay C. Lee. Fast-inv: A fast algorithm for building large inverted files. Technical report, Blacksburg, Virginia, USA, 1991.

[29] William Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms.* Prentice-Hall, 1992.

[30] Ian Witten, Timothy Bell, and Alistair Moffat. *Managing Gigabytes.* John Wiley, 1994.

[31] Donna Harman and Gerald Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41:581–589, 1990.

[32] Stefan Büttcher and Charles L. A. Clarke. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In *Proceedings of the 14th ACM Conference on Information and Knowledge Management (CIKM 2005)*, pages 317–318, Bremen, Germany, 2005.

[33] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast on-line index construction by geometric partitioning. In *Proceedings of the 14th ACM Conference on Information and Knowledge Management (CIKM 2005)*, pages 776–783, Bremen, Germany, 2005.

[34] Nicholas Lester, Alistair Moffat, and Justin Zobel. Efficient online index construction for text databases. *ACM Transactions on Database Systems*, 33(3):19:1–19:33, 2008.

[35] Ruixuan Li, Xuefan Chen, Chengzhou Li, Xiwu Gu, and Kunmei Wen. Efficient online index maintenance for SSD-based information retrieval systems. In *High Performance Computing and Communication*, pages 262–269, 2012.

[36] Giorgos Margaritis and Stergios V. Anastasiadis. Low-cost management of inverted files for online full-text search. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM 2009)*, pages 455–464, Hong Kong, China, 2009.

[37] Wann-Yun Shieh and Chung-Ping Chung. A statistics-based approach to incrementally update inverted files. *IP&M*, 41(2):275–288, 2005.

[38] Eric Brown, James Callan, and W. Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB 1994)*, pages 192–202, Santiago de Chile, Chile, 1994.

[39] Stefan Büttcher, Charles Clarke, and Brad Lushman. Hybrid index maintenance for growing text collections. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2006)*, pages 356–363, Seattle, Washington, USA, 2006.

[40] Robert Luk and Wai Lam. Efficient in-memory extensible inverted file. *Information Systems*, 32(5):733–754, 2007.

[41] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar RAM-CPU cache compression. page 59, Atlanta, Georgia, USA, 2006.

[42] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International Conference on World Wide Web (WWW 2008)*, pages 387–396, Beijing, China, 2008.

[43] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web (WWW 2009)*, pages 401–410, Madrid, Spain, 2009.

[44] Fabrizio Silvestri and Rossano Venturini. Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM 2010)*, pages 1219–1228, Toronto, ON, Canada, 2010.

[45] Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin. Earlybird: Real-time search at Twitter. In *Proceedings of the 28th International Conference on Data Engineering (ICDE 2012)*, pages 1360–1369, Washington, D.C., USA, 2012.

[46] Justin Zobel, Steffen Heinz, and Hugh Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277, 2001.

[47] M. Ramakrishna and Justin Zobel. Performance in practice of string hashing functions. In *Proceedings of the 5th International Conference on Database Systems for Advanced Applications (DASFAA 1997)*, pages 215–224, Melbourne, Australia, 1997.

[48] Justin Williams, Hughand Zobel and Steffen Heinz. Self-adjusting trees in practice for large text collections. *Software–Practice & Experience*, 31(10):925–939, 2001.

[49] Steffen Heinz, Justin Zobel, and Hugh Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.

[50] Nikolas Askitis and Justin Zobel. Redesigning the string hash table, burst trie, and bst to exploit cache. *Journal of Experimental Algorithmics*, 15:1.7:1.1–1.7:1.61, 2011.

[51] Trevor Strohman and W. Croft. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2007)*, pages 175–182, Amsterdam, The Netherlands, 2007.

[52] Erik Demaine, Alejandro López-Ortiz, and J. Munro. Experiments on adaptive set intersections for text retrieval systems. In *Revised Papers from the 3rd International Workshop on Algorithm Engineering and Experimentation (ALENEX 2001)*, pages 91–104, Washington, D.C., USA, 2001.

[53] Jérémy Barbay, Alejandro López-Ortiz, and Tyler Lu. Faster adaptive set intersections for text searching. In *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA 2006)*, pages 146–157, Berlin, Heidelberg, 2006.

[54] Dimitris Tsirogiannis, Sudipto Guha, and Nick Koudas. Improving the performance of list intersection. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB 2009)*, pages 838–849, Lyon, France, 2009.

[55] J. Culpepper and Alistair Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems*, 29(1):Article 1, 2010.

[56] Eric W. Brown. Fast evaluation of structured queries for information retrieval. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1995)*, pages 30–38, Seattle, Washington, USA, 1995.

[57] Howard Turtle and James Flood. Query evaluation: Strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[58] Trevor Strohman, Howard Turtle, and W. Bruce Croft. Optimization strategies for complex queries. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2005)*, pages 219–225, Salvador, Brazil, 2005.

[59] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2011)*, pages 993–1002, Beijing, China, 2011.

[60] David Carmel, Doron Cohen, Ronald Fagin, Eitan Farchi, Michael Herscovici, Yoelle Maarek, and Aya Soffer. Static index pruning for information retrieval systems. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2001)*, pages 43–50, New Orleans, Louisiana, USA, 2001.

[61] Alexandros Ntoulas and Junghoo Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2007)*, pages 191–198, Amsterdam, The Netherlands, 2007.

[62] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. On the false-positive rate of bloom filters. *Information Processing Letters*, 108:210–213, 2008.

[63] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

[64] Kamran Tirdad, Pedram Ghodsnia, J. Ian Munro, and Alejandro López-Ortiz. COCA filters: Co-occurrence aware Bloom filters. In *Proceedings of the 18th International Symposium on String Processing and Information Retrieval (SPIRE 2011)*, pages 313–325, Pisa, Italy, 2011.

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

[66] Jinyang Li, Boon Thau Loo, Joseph M. Hellerstein, M. Frans Kaashoek, David R. Karger, and Robert Morris. On the feasibility of peer-to-peer web indexing and search. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS 2003)*, pages 207–215, Berkeley, California, USA, 2003.

[67] Joel Fagan. Experiments in automatic phrase indexing for document retrieval: A comparison of syntactic and non-syntactic methods. Technical Report TR87-868, Cornell, 1987.

[68] W. Bruce Croft, Howard R. Turtle, and David D. Lewis. The use of phrases and structured queries in information retrieval. In *Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1991)*, pages 32–45, Chicago, Illinois, USA, 1991.

[69] David Hawking and Paul Thistlewaite. Proximity operators - so near and yet so far. In *Proceedings of 4the th Text REtrieval Conference (TREC 1995)*, pages 131–144, Gaithersburg, Maryland, USA, 1995.

[70] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. Shortest substring ranking (MultiText experiments for TREC-4). In *Proceedings of 4the th Text REtrieval Conference (TREC 1995)*, pages 295–304, Gaithersburg, Maryland, USA, 1995.

[71] S. Büttcher, C. Clarke, and B. Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2006)*, pages 621–622, Seattle, Washington, USA, 2006.

[72] Tao Tao and ChengXiang Zhai. An exploration of proximity measures in information retrieval. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2007)*, pages 295–302, Amsterdam, The Netherlands, 2007.

[73] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2007)*, pages 183–190, Amsterdam, The Netherlands, 2007.

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

[75] Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. Stochastic gradient boosted distributed decision trees. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM 2009)*, pages 2061–2064, Hong Kong, China, 2009.

[76] Stephen Tyree, Kilian Q. Weinberger, and Kunal Agrawal. Parallel boosted regression trees for web search ranking. In *Proceedings of the 20th International Conference on World Wide Web (WWW 2011)*, pages 387–396, Hyderabad, India, 2011.

[77] Christopher J.C. Burges, Robert Ragno, and Quoc Viet Le. Learning to rank with nonsmooth cost functions. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems*, pages 193–200. MIT Press, Cambridge, Massachusetts, USA, 2007.

[78] J.H. Friedman. Greedy function approximation: A gradient boosting machine. In *Technical Report, IMS Reitz Lecture, Stanford*, 1999.

[79] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd International Conference on Machine Learning (ICML 2005)*, pages 89–96, Bonn, Germany, 2005.

[80] Kalervo Järvelin and Jaana Kekäläinen. Cumulative gain-based evaluation of IR techniques. *ACM Transactions on Information Systems*, 20(4):422–446, 2002.

[81] Olivier Chapelle, Donald Metzler, Ya Zhang, and Pierre Grinspan. Expected reciprocal rank for graded relevance. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM 2009)*, pages 621–630, Hong Kong, China, 2009.

[82] L. Breiman, J. Friedman, C. Stone, and R. Olshen. *Classification and Regression Trees*. Chapman and Hall, 1984.

[83] Leo Breiman. Bagging predictors. In *Machine Learning*, pages 123–140, 1996.

[84] Jerome Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

[85] Leo Breiman. Random forests. In *Machine Learning*, pages 5–32, 2001.

[86] Zhixiang Xu, Kilian Weinberger, and Olivier Chapelle. The greedy miser: Learning under test-time budgets. In *Proceedings of the 29th International Conference on Machine Learning (ICML 2012)*, Edinburgh, Scotland, 2012.

[87] Yasser Ganjisaffar. *Tree ensembles for learning to rank*. PhD thesis, UC Irvine, 2011.

[88] Dragos Margineantu and Thomas Dietterich. Pruning adaptive boosting. In *Proceedings of the 14th International Conference on Machine Learning (ICML 1997)*, pages 211–218, Nashville, Tennessee, USA, 1997.

[89] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

[90] Gonzalo Martínez-Muñoz, Daniel Hernández-Lobato, and Alberto Suárez. An analysis of ensemble pruning techniques based on ordered aggregation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(2):245–259, 2009.

[91] Robert E. Banfield, Lawrence O. Hall, Kevin W. Bowyer, and W. Philip Kegelmeyer. Ensemble diversity measures and their application to thinning. *Information Fusion*, 6, 2004.

[92] Christino Tamon and Jie Xiang. On the boosting pruning problem. In *Proceedings of the 11th European Conference on Machine Learning (ECML 2000)*, pages 404–412, Barcelona, Spain, 2000.

[93] Cigdem Demir and Ethem Alpaydin. Cost-conscious classifier ensembles. *Pattern Recognition Letters*, 26(14):2206–2214, 2005.

[94] B. Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, Jiang Chen, Ciya Liao, Zhaohui Zheng, and Jon Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM 2010)*, pages 411–420, New York, New York, USA, 2010.

[95] Lidan Wang, Jimmy Lin, and Donald Metzler. A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2011)*, pages 105–114, Beijing, China, 2011.

[96] Biswanath Panda, Joshua Herbach, Sugato Basu, and Roberto Bayardo. PLANET: Massively parallel learning of tree ensembles with MapReduce. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB 2009)*, pages 1426–1437, Lyon, France, 2009.

[97] Krysta Svore and Christopher Burges. Large-scale learning to rank using boosted decision trees. In *Scaling Up Machine Learning*. Cambridge University Press, 2011.

[98] Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, 3(7):27–34, 2005.

[99] Peter Boncz, Martin Kersten, and Stefan Manegold. Breaking the memory wall in MonetDB. *Communications of the ACM*, 51(12):77–85, 2008.

[100] Bruce Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It.* Morgan & Claypool Publishers, 2009.

[101] Anastassia Ailamaki, David DeWitt, Mark Hill, and David Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB 1999)*, pages 266–277, Edinburgh, Scotland, 1999.

[102] Jun Rao and Kenneth Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB 1999)*, pages 78–89, Edinburgh, Scotland, 1999.

[103] Kenneth A. Ross, John Cieslewicz, Jun Rao, and Jingren Zhou. Architecture sensitive database design: Examples from the Columbia group. *Bulletin of the Technical Committee on Data Engineering*, 28(2):5–10, 2005.

[104] Marcin Zukowski, Peter Boncz, Niels Nes, and Sándor Héman. MonetDB/X100—a DBMS in the CPU cache. *Bulletin of the Technical Committee on Data Engineering*, 28(2):17–22, 2005.

[105] Kenneth A. Ross. Conjunctive selection conditions in main memory. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2002)*, pages 109–120, Madison, Wisconsin, USA, 2002.

[106] David August, Wen mei Hwu, and Scott Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 1997)*, pages 92–103, Research Triangle Park, North Carolina, USA, 1997.

[107] Hyesoon Kim, Onur Mutlu, Yale N. Patt, and Jared Stark. Wish branches: Enabling adaptive and aggressive predicated execution. *IEEE Micro*, 26(1):48–58, 2006.

[108] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, California, USA, 2005.

[109] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *Proceedings of the 1st International Conference on Scalable Information Systems (InfoScale 2006)*, Hong Kong, 2006.

[110] Nicholas Lester, Alistair Moffat, and Justin Zobel. Efficient online index construction for text databases. *ACM Transactions on Database Systems*, 33(3):19, 2008.

[111] Florian Leibert, Jake Mannix, Jimmy Lin, and Babak Hamadani. Automatic management of partitioned, replicated search services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC 2011)*, pages 27:1–27:8, Cascais, Portugal, 2011.

[112] Iadh Ounis, Craig Macdonald, Jimmy Lin, and Ian Soboroff. Overview of the TREC-2011 Microblog Track. In *Proceedings of the Twentieth Text REtrieval Conference (TREC 2011)*, Gaithersburg, Maryland, USA, 2011.

[113] Ian Soboroff, Dean McCullough, Jimmy Lin, Craig Macdonald, Iadh Ounis, and Richard McCreadie. Evaluating real-time search over tweets. In *Proceedings of the 6th International AAAI Conference on Weblogs and Social Media (ICWSM 2012)*, pages 579–582, Dublin, Ireland, 2012.

[114] Richard McCreadie, Ian Soboroff, Jimmy Lin, Craig Macdonald, Iadh Ounis, and Dean McCullough. On building a reusable Twitter corpus. In *Proceedings of the 35th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2012)*, pages 1113–1114, Portland, Oregon, 2012.

[115] Ian Soboroff, Iadh Ounis, Craig Macdonald, and Jimmy Lin. Overview of the TREC 2012 Microblog Track. In *Proceedings of the Twenty-First Text REtrieval Conference (TREC 2012)*, Gaithersburg, Maryland, USA, 2012.

[116] Jake Brutlag. Speed matters for Google web search. Technical report, Google, 2009.

[117] Donald Metzler and W. Bruce Croft. Combining the language model and inference network approaches to retrieval. *Information Processing and Management*, 40(5):735–750, 2004.

[118] Toby Sharp. Implementing decision trees and forests on a GPU. In *Proceedings of the 10th European Conference on Computer Vision (ECCV 2008)*, pages 595–608, Marseille, France, 2008.

[119] Brian Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA? In *Proceedings of the 2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2012)*, pages 232–239, Toronto, Ontario, Canada, 2012.

[120] Olivier Chapelle and Yi Chang. Yahoo! learning to rank challenge overview. *Journal of Machine Learning Research - Proceedings Track*, 14:1–24, 2011.

[121] W. Stadler. *Multicriteria Optimization in Engineering and in the Sciences.* Mathematical Concepts and Methods in Science and Engineering. Springer, 1988.

[122] A. Osyczka. *Multicriterion Optimization in Engineering with FORTRAN Programs.* E. Horwood, 1984.

[123] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.