Managing Tail Latency in Large Scale Information Retrieval Systems

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy.

Joel M. Mackenzie
Bachelor of Computer Science, Honours – RMIT University

Supervised by

Associate Professor J. Shane Culpepper
Associate Professor Falk Scholer

School of Science
College of Science, Engineering, and Health
RMIT University
Melbourne, Australia

October, 2019

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed. I acknowledge the support I have received for my research through the provision of an Australian Government Research Training Program Scholarship.

Joel M. Mackenzie
School of Science
College of Science, Engineering, and Health
RMIT University, Melbourne, Australia
October 15, 2019

Abstract

As both the availability of internet access and the prominence of smart devices continue to increase, data is being generated at a rate faster than ever before. This massive increase in data production comes with many challenges, including efficiency concerns for the storage and retrieval of such large-scale data. However, users have grown to expect the sub-second response times that are common in most modern search engines, creating a problem — how can such large amounts of data continue to be served efficiently enough to satisfy end users?

This dissertation investigates several issues regarding *tail latency* in large-scale information retrieval systems. Tail latency corresponds to the high percentile latency that is observed from a system — in the case of search, this latency typically corresponds to how long it takes for a query to be processed. In particular, keeping tail latency as low as possible translates to a good experience for all users, as tail latency is directly related to the worst-case latency and hence, the worst possible user experience. The key idea in targeting tail latency is to move from questions such as *"what is the median latency of our search engine?"* to questions which more accurately capture user experience such as *"how many queries take more than* 200*ms to return answers?"* or *"what is the worst case latency that a user may be subject to, and how often might it occur?"*

While various strategies exist for efficiently processing queries over large textual corpora, prior research has focused almost entirely on improvements to the *average* processing time or cost of search systems. As a first contribution, we examine some state-of-the-art retrieval algorithms for two popular index organizations, and discuss the trade-offs between them, paying special attention to the notion of tail latency. This research uncovers a number of observations that are subsequently leveraged for improved search efficiency and effectiveness.

We then propose and solve a new problem, which involves processing a number of related queries together, known as *multi-queries*, to yield higher quality search results. We experiment with a number of algorithmic approaches to efficiently process these multi-queries, and report on the cost, efficiency, and effectiveness trade-offs present with each. Ultimately, we find that some solutions yield a low tail latency, and are hence suitable for use in real-time search environments.

Finally, we examine how *predictive models* can be used to improve the tail latency and end-to-end cost of a commonly used *multi-stage* retrieval architecture *without* impacting result effectiveness. By combining ideas from numerous areas of information retrieval, we propose a prediction framework which can be used for training and evaluating several efficiency/effectiveness trade-off parameters, resulting in improved trade-offs between cost, result quality, and tail latency.

Not everything that can be counted counts, and not everything that counts can be counted.

W. B. Cameron.

Acknowledgements

During my candidature, I have had the pleasure of working with and among a number of incredible individuals who have helped shape my academic and personal growth.

Firstly, I would like to thank my advisors, Shane and Falk. I first met Shane when I was an undergraduate student — he was lecturing both the *Operating Systems Principles* and *Algorithms and Analysis* courses. This was when I became fascinated with the elegance of clever algorithms, system design, and efficiency. Falk introduced me to the fundamentals of search during the *Information Retrieval* course, while I was undertaking my honours project on efficient location-aware web search. Since then, Shane and Falk have provided me with numerous opportunities that have developed my skills as a computer scientist, an academic, and as a person. I will be forever grateful to them.

Luke Gallagher has been an immense support over the last few years, assisting me in both professional and personal capacities — he even proofread this thesis! I am extremely grateful to him. I am also thankful to Rodger Benham, Sheng Wang, Matt Crane, and Matthias Petri, who always had time for useful discussions, ideas, and repartee, either over coffee, beer, and occasionally, email. There are many other colleagues, friends, and mentors from RMIT, the University of Melbourne, and other institutions around the world who have supported and encouraged me to become a better academic. I am indebted to you all.

Milad Shokouhi and Ahmed Hassan Awadallah allowed me to spend three months at Microsoft Research during 2018. It was a pleasure working on interesting problems outside of the efficiency space, and I am extremely grateful to them for taking me on as an intern.

Craig Macdonald (University of Glasgow) and Simon Puglisi (University of Helsinki) allowed me to visit and present my research to their groups. Each was a wonderful experience, and I look forward to visiting them again in the future.

To all of my friends — thank you for your support during my candidature. The many coffees, beers, barbecues, bike adventures, disc golf outings, camping trips, weightlifting sessions, and hikes helped reinvigorate me and kept me motivated.

Finally, I dedicate this thesis to my parents, Sandra and Jock, my sister, Chloe, and my partner, Mikaela. Without your endless love, support, patience, and encouragement, none of this would have been possible. I will look back on the years of my Ph.D as some of the best years of my life.

List of Publications

Publications included in this dissertation

R. Benham, J. Mackenzie, A. Moffat, and J. S. Culpepper: **Boosting Search Performance Using Query Variations.** In *ACM Transactions on Information Systems (TOIS)*, October, 2019.

J. Mackenzie, J. S. Culpepper, R. Blanco, M. Crane, C. L. A. Clarke, and J. Lin: **Query Driven Algorithm Selection in Early Stage Retrieval.** In *Proceedings of the 11th International ACM Conference on Web Search and Data Mining (WSDM 2018)*, February, 2018.

J. Mackenzie, F. Scholer, and J. S. Culpepper: **Early Termination Heuristics for Score-at-a-Time Index Traversal.** In *Proceedings of the 22nd Annual Australasian Document Computing Symposium (ADCS 2017)*, December, 2017.

M. Crane, J. S. Culpepper, J. Lin, J. Mackenzie, and A. Trotman: **A Comparison of Document-at-a-Time and Score-at-a-Time Query Evaluation.** In *Proceedings of the 10th International ACM Conference on Web Search and Data Mining (WSDM 2017)*, February, 2017.

Additional Publications

A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel: **PISA: Performant Indexes and Search for Academia.** In *Proceedings of the Open-Source IR Replicability Challenge, co-located with SIGIR (OSIRRC 2019)*, July, 2019.

M. Petri, A. Moffat, J. Mackenzie, J. S. Culpepper, and D. Beck: **Accelerated Query Processing Via Similarity Score Prediction.** In *Proceedings of the 42nd International ACM Conference on Research and Development in Information Retrieval (SIGIR 2019)*, July, 2019.

J. Mackenzie, K. Gupta, F. Qiao, A. H. Awadallah, and M. Shokouhi: **Exploring User Behavior in Email Re-Finding Tasks.** In *Proceedings of the 2019 World Wide Web Conference (WWW 2019)*, May, 2019.

J. Mackenzie, A. Mallia, M. Petri, J. S. Culpepper, and T. Suel: **Compressing Inverted Indexes with Recursive Graph Bisection: A Reproducibility Study.** In *Proceedings of the 41st European Conference on Information Retrieval (ECIR 2019)*, April, 2019.

L. Gallagher, J. Mackenzie, and J. S. Culpepper: **Revisiting Spam Filtering in Web Search.** In *Proceedings of the 23rd Annual Australasian Document Computing Symposium (ADCS 2018)*, December, 2018.

R. Benham, J. S. Culpepper, L. Gallagher, X. Lu, and J. Mackenzie : **Towards Efficient and Effective Query Variant Generation.** In *Proceedings of the First Biennial Conference on Design of Experimental Search and Information Retrieval Systems (DESIRES 2018)*, August, 2018.

J. Mackenzie, C. Macdonald, F. Scholer, and J. S. Culpepper: **On the Cost of Negation for Dynamic Pruning.** In *Proceedings of the 40th European Conference on Information Retrieval (ECIR 2018)*, March, 2018.

J. Mackenzie: **Managing Tail Latencies in Large Scale IR Systems (Doctoral Consortium Extended Abstract)** In *Proceedings of the 40th International ACM Conference on Research and Development in Information Retrieval (SIGIR 2017)*, August, 2017.

List of Figures

List of Tables

List of Symbols

Notation	Description
D	A document corpus.
N	The number of documents in an index or corpus.
d	A document identifier.
l_d	The length of document d.
t	A query term.
f_t	The number of documents in the index containing term t.
$f_{d,t}$	The number of times term t appears in document d.
$\mathcal{W}(d,t)$	The score contribution of term t to document d.
$i_{d,t}$	The quantized contribution of term t to document d.
$\mathcal{S}_{d,q}$	The score of document d with respect to query q.
θ	The kth highest observed score thus far.
$\hat{\theta}$	The document score threshold.
q	A bag-of-words query of length n containing terms $\{t_1, t_2, \ldots, t_n\}$.
Q	A multi-query; a set of n unique queries, $\{q_1, q_2, \ldots, q_n\}$.
r	A ranked results list.
R	A set of n ranked lists, $\{r_1, r_2, \ldots, r_n\}$.
k	The depth of a ranked list.
F	Dynamic pruning aggression parameter.
ρ	Early termination parameter for SAAT retrieval.
$\mathcal{L}$	A postings list.
$\mathcal{P}$	A set of postings lists.
P	A query processing thread.
$\mathcal{M}$	An evaluation metric.
A	An accumulator table.
$\vec{x}$	A feature vector.
M_y	A model for predicting variable y.
T	A decision tree.
$\mathcal{T}$	An ensemble of decision trees.
w_x	The weight of a corresponding variable indexed by x.
C	The cost of generating and re-ranking a results list.

Contents

1 Introduction

Finding relevant information from massive collections of data is a fundamental requirement in many modern computing applications. In particular, *web search* is now a multi-billion dollar industry, and is the primary access point to the world wide web. However, the extreme scale of the web poses a number of problems for search practitioners. Firstly, users expect results to be *effective* in satisfying their information needs with respect to often short and potentially under-specified queries. Secondly, users expect these results to be served *efficiently*, with sub-second response times being the norm for most user-facing search systems. Clearly, these goals are in tension — providing high quality search results is a non-trivial task, yet it is expected to be completed almost instantaneously. To make matters worse, data is being generated faster than ever before. For example, Seagate and the International Data Corporation estimated that over 50% of all digital data was generated within the *last two years*, with around 175 *zettabytes* of data predicted to be generated *per year* by 2025 [209].

As a large-scale example, consider the Google search engine, which at the time of writing has around 90% of the global market share in web search. This search engine is estimated to process upwards of 3.5 *billion* queries per day across *trillions* of indexed pages. However, less popular web search engines still handle an enormous amount of traffic and data. DuckDuckGo, which is a privacy conscious search engine, has less than 0.5% of the global market share in web search. Nevertheless, it currently handles over 35 million queries per day.[1] Thus, maintaining efficient, effective, and scalable search will continue to challenge search practitioners into the future.

The delicate balance between providing effective results within a reasonable amount of time has been documented in the past. For example, Dean and Barroso [82] state that:

> "In large IR systems, speed is more than a performance metric; it is a key quality metric, as returning good results quickly is better than returning the best results slowly."

Indeed, the impact of latency on user behavior is far more substantial than may be expected. For example, a set of studies from Microsoft (Bing) and Google showed that added latency of as little as 200ms can have negative impacts on users, with significant reductions observed across

[1] https://duckduckgo.com/traffic

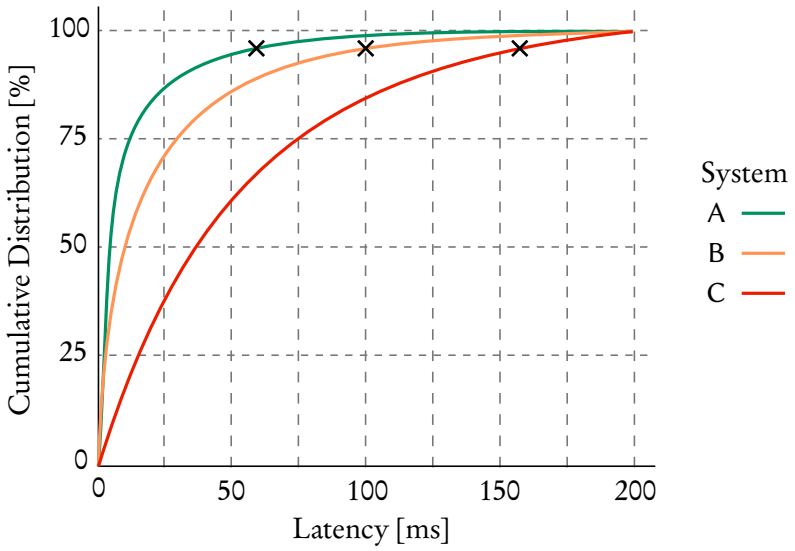


Figure 1.1: An example of how tail latency is measured for three systems. Each cross represents the 95th percentile latency of the given system. System A, B, and C each have a 95th percentile latency of 60ms, 100ms, and 160ms, respectively.

multiple user satisfaction metrics [224]. Even worse, however, was that some users never returned to the search system, presumably because they had switched to a competitor. Another view on the importance of efficiency was provided by Kohavi et al. [134], who claimed that every 100ms improvement in efficiency leads to around a 0.6% increase in revenue, which translates to *tens-of-millions* of dollars per annum for major search providers. They further illustrate this point as follows:

> "An engineer that improves server performance by 10ms (that's 1/30 of the speed that our eyes blink) more than pays for their fully-loaded annual costs."

Thus, latency is an incredibly important and often overlooked aspect of providing a high quality search experience, and maximizing company revenue.

Large-scale search systems are immensely complex, often containing *thousands* or even *millions* of servers dedicated to processing portions of the search index [20]. As such, hundreds or thousands of individual servers may be involved in processing *each* query. Unfortunately, containing the cost of query processing at this scale is difficult, and even a slight delay in any given stage of processing may translate to an unacceptable response latency. Furthermore, it is not sufficient to keep the *average* latency low — the occasional slow response may be enough to deter users from continuing to use the search engine. Hence, it is important to quantify the *tail latency*, which corresponds to the *high percentile* response times of the system.

Tail latency is characterized by the response times exceeding the nth percentile response latency. Typically, n is set to 95 or 99, but other high percentiles can also be considered. Therefore, tail latency gives useful information on the *worst* case response times. For example, consider Figure 1.1, which plots the cumulative distribution of the latency observed for three systems. If only the median time was considered (50th percentile), then both system A and B may seem like attractive options. However, system A exhibits a lower tail latency than system B, making it a more robust choice, as it is less likely to respond slowly. This toy example motivates the case for measuring and minimizing tail latency — it is crucial for ensuring positive user experiences. Unfortunately, the complexity of keeping the tail latency low is much harder than it may seem. Dean and Barroso [82] provide a good example of why this is the case:

> "Consider a system where each server typically responds in 10ms but with a 99th percentile latency of one second. If each user request is handled on just one server, one user request in 100 will be slow (one second). If a user request must collect responses from 100 such servers in parallel, then 63% of user requests will take more than one second."

Recent innovations in machine learning have allowed a new breed of *learning-to-rank* models to be deployed, providing enhanced search effectiveness. To accommodate these models, the once single-stage search process, which involved identifying and ranking documents, has been recast as a *multi-stage* retrieval problem, where a sequence of processing stages aim to identify, rank, and prune the results set [192]. This novel approach has opened up a wide range of questions relating to the best practices for efficient and effective end-to-end retrieval. In particular, little attention has been paid to the cause of tail latency during query processing, leaving many open questions regarding the best practices for implementing tail-tolerant search systems. For example, most prior works that focus on efficient indexing and retrieval opt to report efficiency numbers by reporting basic summary statistics such as the mean or median latency. As demonstrated above, this is not sufficient for ensuring a good user experience. In addition, a variety of solutions currently exist for efficiently processing queries across inverted indexes, but there are few best practices in place for choosing the underlying index organization and traversal algorithm in the context of modern retrieval architectures. Similarly, the notion of balancing efficiency and effectiveness is currently under-explored. The importance of both efficient and effective retrieval implies that a deeper understanding of these inherent trade-offs is required.

1.1 Contributions

We investigate several related threads pertaining to efficient and effective multi-stage retrieval, with a focus on reducing tail latency. In doing so, we aim to increase the understanding of the current trade-offs between the various index traversal approaches in a number of contexts.

1.1.1 Comparing Efficient Index Traversal Strategies

Chapter 4 presents a large empirical study of the state-of-the-art index traversal algorithms, including Document-at-a-Time (DAAT) algorithms such as WAND [36] and BMW [88], and Score-at-a-Time (SAAT) algorithms such as JASS [148]. In particular, we focus on the associated latency of these algorithms with respect to many important parameters, such as the length of the incoming query, the number of results that are required, and the collection size, paying special attention to the variability of latency, and tail latency. Furthermore, we examine some commonly used heuristics which allow retrieval efficiency to be improved with a loss in effectiveness, and whether these heuristics are suitable in all processing situations. Finally, we apply *selective parallelization* [119] to the JASS algorithm to improve latency without negatively impacting search effectiveness. This chapter focuses on the following primary research question:

> **RQ1:** *What are the primary causes of high-percentile latency during top-k retrieval?*

The research presented in this chapter resulted in the following publications:

- Matt Crane, J. Shane Culpepper, Jimmy Lin, **Joel Mackenzie**, and Andrew Trotman: *"A Comparison of Document-at-a-Time and Score-at-a-Time Query Evaluation,"* In Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM), 2017, pages 201–210.
- **Joel Mackenzie**, Falk Scholer, and J. Shane Culpepper: *"Early Termination Heuristics for Score-at-a-Time Index Traversal,"* In Proceedings of the 22nd Australasian Document Computing Symposium (ADCS), 2017, pages 8.1–8.8.

1.1.2 Efficient Online Rank Fusion

Our second investigation, outlined in Chapter 5, poses a novel retrieval problem concerned with *batch processing* a set of *related* queries known as *multi-queries*, and leveraging rank fusion to improve the effectiveness and robustness of retrieval. We formally define the problem, and propose a few approaches for solving it, leveraging the outcomes from Chapter 4 to guide the algorithm design process. In particular, we examine solutions based on parallel processing, dynamic pruning, and early termination to achieve a range of cost/latency trade-offs, using both DAAT and SAAT retrieval mechanisms. We also discuss the implications of our approaches, and whether they are suitable for use in real-time search situations. We answer the following research question:

> **RQ2:** *What is the most efficient way to process multi-queries over inverted indexes?*

The findings from this chapter make up Section 4 of the following journal article:

- Rodger Benham, **Joel Mackenzie**, Alistair Moffat, and J. Shane Culpepper: *"Boosting Search Performance Using Query Variations,"* ACM Transactions on Information Systems (TOIS), 37(4):41.1–41.25, 2019. **Note:** This article is joint work, and the findings in Section 5 of this article are attributed to Rodger Benham (RMIT University) [27].

1.1.3 Reducing Tail Latency via Algorithm Selection

In Chapter 6, we expand our analysis to consider the costs of all stages of the multi-stage retrieval architecture. Building on work from Culpepper et al. [75], we propose a regression scheme that can predict the best parameter settings for balancing the efficiency and effectiveness of the candidate generation stage in a multi-stage retrieval system, while implicitly reducing the downstream cost of the learning-to-rank stage. Using this framework, we propose a hybrid query processing system that can adaptively select which query processing scheme to use. This research also makes important contributions on how predictive models can be trained and evaluated in the absence of manually labeled data, which is currently a major problem in academic settings, where budget restrictions are insufficient to conduct large-scale annotation campaigns. This study answers the following research question:

> **RQ3:** *How can we predict performance sensitive parameters on a query-by-query basis to minimize tail latency during candidate generation?*

The research from this chapter appears in the following manuscript:

- **Joel Mackenzie**, J. Shane Culpepper, Roi Blanco, Matt Crane, Charles L. A. Clarke, and Jimmy Lin: *"Query Driven Algorithm Selection in Early Stage Retrieval,"* In Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining (WSDM), 2018, pages 396–404.

1.2 Organization

The remainder of this thesis is structured as follows: Chapter 2 presents an overview of the field of information retrieval, including some historical information, approaches for improving ranking, tools for evaluating search engine performance, and a sketch of the key problems relating to real large-scale search engines. Chapter 3 introduces prior art in efficient indexing and query processing for large-scale IR systems, providing a view of the current state-of-the-art for efficient search over large collections. Chapter 4 presents an empirical study of tail latency in state-of-the-art search systems, and explores simple methods for improving tail latency, including an analysis of the shortcomings of current retrieval approaches. Chapter 5 describes the new problem of efficiently processing *multi-queries* for real-time rank fusion, some possible solutions, and the trade-offs therein. Chapter 6 examines how prediction frameworks can be leveraged to improve tail latency for early stage retrieval while reducing the cost of downstream retrieval components. Finally, in Chapter 7, the thesis is concluded, and some interesting future directions are outlined.

2 Background

2.1 Information Retrieval

The field of *Information Retrieval* (IR) is concerned with problems related to the storage and retrieval of information such as text, music, images, and videos. Information retrieval has been studied since the 1950's, but the origins of IR begin well before computers were available for the automation of search. In the late 1800's, the well known Dewey Decimal System was proposed, allowing libraries to better organize their books via a category *index*. However, organizing information for efficient access has been reported to exist for over 4,000 years, with evidence of inscribed clay tablets being stored in special areas for principled access [142, 229]. The first *automated* systems for the efficient retrieval of information were developed in the late 1800's and early 1900's, which were mainly mechanical devices that allowed rapid scanning of physical records. Later, during the 1940's and early 1950's, the first computerized information retrieval systems were described, and a large amount of work was conducted throughout the following decades to improve the search process [219, 229]. In 1989, the *World Wide Web* was invented, and by 1992, hosted around ten websites. By 1995, the web had expanded to around 25,000 sites, and had broken the 1 million mark by 1997. Figure 2.1 visualizes this incredible growth across the first decade of the web.[1] Thirty years on, the world wide web has over a *billion* websites, and close to 50% of the worlds population has access to the internet. This rapid expansion meant that finding relevant pages and documents on the web became difficult, prompting a huge amount of research on improving web search. Today, web search is the most popular and wide-reaching IR service, with web search engines collectively processing *billions* of queries per day to as many users.

Although most commonly used for web search, IR systems are extremely important in a number of other paradigms, including social media [39, 236], e-commerce [231, 245], and career matching [176, 259], among many others. With the ever increasing volume of data being generated, search engines will continue to be the primary tool for finding *the needle in the haystack*.

[1] Data collected from https://www.internetlivestats.com/

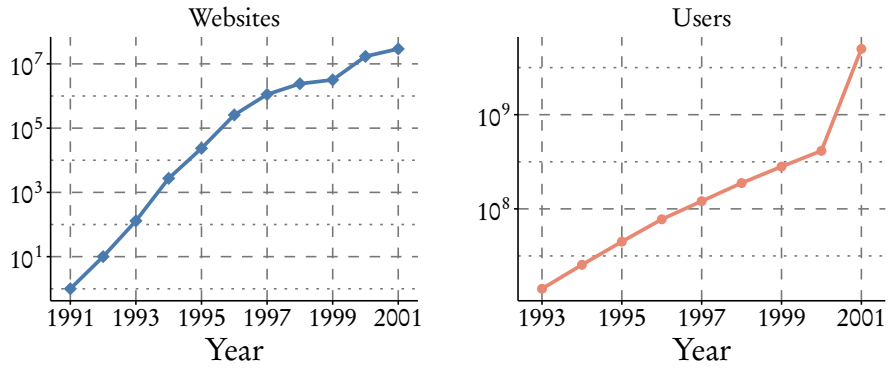


Figure 2.1: Website growth over the first decade of the web, from 1991 to 2001, and the number of internet users, from 1993 to 2001. The rate of growth of websites was no less than 30% between any of these successive years, with a maximum growth > 2000% between 1993 (130 sites) and 1994 (2,738 sites). Note the log scale in both figures.

2.1.1 Information Retrieval Process

The information retrieval process begins with a user, who has an internal sense of requiring some *unknown* information, which we refer to as an *information need*. In order to resolve this issue, the user can generate a *query* that represents the gap in their knowledge, and this query can be issued to a *search engine*. The search engine will find and present a list of *documents* that it deems relevant to the input query. To decrease the cognitive load on users, search engines try to *rank* the returned documents by their estimated relevance to the input query. These documents are then presented on a *search engine results page* (SERP), with each document represented by a link to the document itself, and a small snippet which summarizes the content of each document (Figure 2.2). Then, the user will work through the ranked list, observing each of the snippets, and possibly clicking into some documents, until their information need is met. An unfortunate paradox that arises is that users are often not aware of *what* information is required to answer their information need [63]. Hence, the generated query often does not map to the required information need, resulting in non-relevant results or insufficient information for the user. Consequently, users may enter the second or third page of results, or may issue a reformulated query, in pursuit of relevant information. This entire process may be repeated many times, until the user is satisfied. In some cases, the user may abandon the search engine before they can address their information need.

While users are often unable to generate queries that accurately capture their underlying information need, search engines are still expected to provide accurate rankings. Hence, a lot of prior work has focused on improving the IR process from the *user* perspective, including methods for improving the rankings provided by search engines, and better ways of evaluating the performance of different search engines.

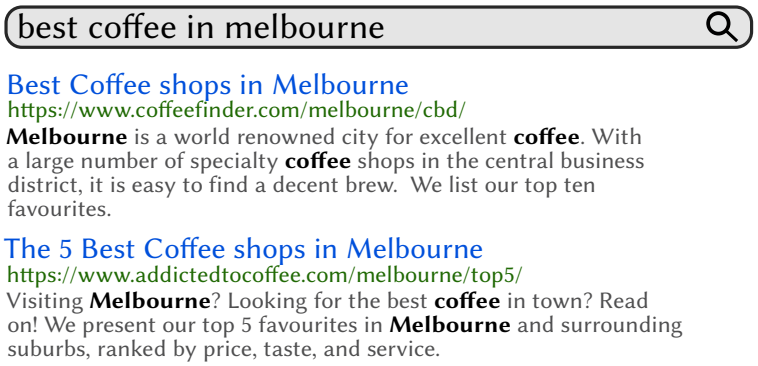


Figure 2.2: An example of a search engine results page (SERP). The top two documents are shown with respect to the query, *"best coffee in melbourne."* Each document is presented with its title, URL, and a small snippet which provides a summary of the document. The snippet may also contain cues as to why the search engine has retrieved it, such as the emboldened search terms.

2.2 Document Ranking

Contemporary search engines aim to maximize their utility while minimizing the cognitive cost for users. Hence, accurately ranking documents such that users can rapidly satisfy their information needs is of the highest importance. In this section, we provide an overview of how search engines typically rank documents, examining some of the key ideas used in modern search engines, including the current state-of-the-art in ranking using machine learning.

2.2.1 Boolean Matching

Early search systems supported only Boolean matching operations without ranking [229]. These types of systems would deal with queries that are structured around Boolean operators such as conjunctions (AND), disjunctions (OR), and negations (NOT). For example, if a user wished to find documents discussing coffee shops in Melbourne, Australia, they may issue the query "best AND coffee AND melbourne". Upon receiving results discussing coffee venues in Melbourne, Florida, they may refine this query to "best AND coffee AND melbourne NOT florida", thereby filtering out results which contain the term *florida*. While Boolean queries are still commonly used in certain applications of IR such as technology assisted reviews [222] and medical search [128], these queries are generally constructed by domain experts, and are not easily understood by the average search user.

A more natural method for non-expert users is to formulate queries using *natural language*. For example, a user may enter the query *"best coffee in melbourne"* and expect to receive a *ranked* list of matching documents. This is the standard for modern search engines in many user-facing domains.

2.2.2 Bag-of-Words Models

A simple, yet effective, method for ranking documents with respect to natural language queries is to consider both the query, q, and each document, d, as a *bag-of-words* (BOW). This means that the *order* or *structure* of the query and document corpus does not have any influence on the underlying ranking model. Instead, these models use simple statistics to build evidence for the likely relevance of a document, and rank documents according to this evidence. Although many bag-of-words models exist, most of them leverage the key ideas of *term frequency*, *inverse document frequency*, and *inverse document length* in their underlying computations. Term frequency corresponds to the number of times that a given term t appears within a document d, and can be leveraged based on the assumption that the more often a term occurs within a document, then the more likely that the document is related to that term, and vice versa. Inverse document frequency (IDF) is another idea based on counting the frequency of terms, and the main assumption of IDF is that the relative importance of a term can be roughly determined by the inverse of the number of documents it appears in. To this end, IDF gives more weight to *rare* terms, as they are more descriptive in identifying relevant documents. The final core idea, inverse document length (IDL), is based on the observation that *longer* documents are more likely to be ranked higher than shorter documents, as they naturally contain more terms. To remedy this issue, longer documents can be penalized, and shorter documents can be rewarded, with the aim of generating rankings that are not biased towards either short or long documents. In this thesis, we consider *additive* BOW models. Suppose that we want to find the score, denoted $\mathcal{S}_{d,q}$, of document d with respect to query q. Under an additive BOW model, $\mathcal{S}$ is computed as the *sum* of the contributions of a ranking function, $\mathcal{W}$, across each term in the query, as

$$\mathcal{S}_{d,q} = \sum_{t \in q} \mathcal{W}(d, t). \tag{2.1}$$

Indeed, various bag-of-words rankers have been proposed that leverage this method, including vector space models, probabilistic models, and language models. We now discuss how these models approach the task of calculating the similarity between a query and document.

Vector Space Models. Many of the earliest rankers projected documents and queries into *vector space* and measured the similarity between the subsequent vectors [216, 217]. In particular, each document can be represented as a V-dimensional vector, where V is the size of the collection vocabulary. When a query is issued, the query is cast into a V-dimensional vector, and then each document is scored by computing the *distance* between the document and query vectors. Terms within each vector can be weighted using the notions of TF and IDF as discussed above, and the distance can be computed by taking the cosine between the query and document vector, for example. One drawback of this approach is that it directly optimizes the similarity between query and document terms, rather than optimizing the ranking with respect to the *perceived relevance* of the document to the query. Nevertheless, the core ideas behind vector space models paved the way for more effective ranking models.

Probabilistic Models. More recently, rankers based on the probabilistic ranking principle have seen a lot of focus, especially the BM25 model [211, 212]. Instead of considering documents and queries as vectors and computing their similarity, these methods aim to estimate the probability of each document being relevant to the query, and then rank the documents by this estimated probability. A commonly used version of BM25 is defined as

$$\mathsf{BM25}(d,q) = \sum_{t \in q} \mathsf{IDF}(t) \cdot \mathsf{TF}(d,t), \tag{2.2}$$

$$\mathsf{IDF}(t) = \log\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right), \tag{2.3}$$

$$\mathsf{TF}(d,t) = \frac{f_{d,t} \cdot (k_1 + 1)}{f_{d,t} + k_1 \cdot \left(1 - b + b \cdot \frac{l_d}{l_{\text{avg}}}\right)}, \tag{2.4}$$

where N is the number of documents in the collection, f_t is the document frequency of term t, $f_{d,t}$ is the frequency of t in d, l_d is the length of document d, and l_{avg} is the average document length [174]. In this particular version of BM25, k_1 and b are parameters that must be set. Trotman et al. [246] recommend using $k_1 = 0.9$ and $b = 0.4$, and showed these settings to be effective on many collections. It is important to note that many alternative functions for both the TF and IDF components have been proposed and examined. However, there is empirically very little difference between these models if they are tuned correctly [247]. For a more exhaustive examination of the many variants of BM25, we refer the reader to the survey from Robertson and Zaragoza [211].

Divergence from Randomness (DFR) models provide an alternative approach for probabilistic ranking [2, 4]. The main idea behind the DFR models is that the more the $f_{d,t}$ value diverges from its frequency within the collection, the more informative term t is at distinguishing document d. One of the most simple DFR models is the DPH model, which is *parameter free*.

$$\mathsf{DPH}(d,q) = \sum_{t \in q} \frac{\left(1 - \left(\frac{f_{d,t}}{l_d}\right)\right)^2}{f_{d,t} + 1} \cdot f_{d,t} \cdot \log_2\left(\frac{f_{d,t} \cdot l_{\text{avg}}}{l_d} \cdot \frac{N}{F_t}\right) + \frac{1}{2}\log_2\left(2\pi \cdot f_{d,t} \cdot \left(1 - \frac{f_{d,t}}{l_d}\right)\right), \tag{2.5}$$

where F_t is the number of times t occurs in the collection, π is the mathematical constant, and the remainder of the variables are the same as described above. DPH was recently shown to perform as well as the BM25 model used in the Terrier search engine [150],[2] making it an attractive ranker for situations where no data exists for tuning.

[2] terrier.org

Language Models. *Language models* provide yet another angle on estimating the relevance between queries and documents, and aim to estimate how likely each document would be to *generate* the given query [204]. Language models use similar statistics to the aforementioned probabilistic rankers, but rely heavily on *smoothing* to correct for errors made in the estimation process. For example, language models typically assign a probability of 0 to an unseen term, which can have negative implications on the relevance estimations. To avoid this issue, language models typically assign very small (yet non-zero) probabilities to unseen or otherwise rare terms. One instantiation of the language model approach which combines relevance values by summing them together is the *Language Model with Dirichlet Smoothing* (LMDS) method, as described by Zobel and Moffat [273].

$$\text{LMDS} = l_q \cdot \log\left(\frac{\mu}{l_d + \mu}\right) + \sum_{t \in q} f_{q,t} \cdot \log\left(\frac{f_{d,t}}{\mu} \cdot \frac{|D|}{F_t} + 1\right), \tag{2.6}$$

where l_q is the length of the input query, $f_{q,t}$ is the number of times that t appears in q, μ is the smoothing parameter (usually set to 2500 [273]), and $|D|$ is the number of terms in the entire corpus. One key difference between the LMDS model and the aforementioned models is that LMDS computes a *per-document* weight separately to the per-term sum of impacts.

2.2.3 Improving Effectiveness

While BOW models are simple and perform relatively well, there are several models that are more effective in practice. However, achieving these effectiveness gains usually comes at a higher cost. We briefly discuss some of these ranking models, arriving at the current state-of-the-art which leverages machine learning to optimize document ranking.

Query Expansion. A common problem that exists for BOW models is the *term mismatch* problem, where documents use different terms than the user (who is generating the query) to describe the same ideas. For example, a document might discuss *"espresso bars"* — a BOW ranker would not consider this document to be relevant to a query such as *"coffee shop"*, even though they are clearly related. Hence, relevant documents might be missed by a simple BOW ranker, because the ranking model only retrieves documents containing the terms present in the query. To alleviate this problem, many modes of *query expansion* have been explored in the literature [48]. Query expansion involves expanding the original query to capture related synonyms or concepts, and issuing the expanded query in lieu of the original query. In some cases, the input query may be *rewritten* entirely [72, 76, 124]. One useful source of data for query rewriting is the input query log, which can be mined to find associated queries [124]. However, log data may not always be available. An alternative approach may leverage *anchor text* as a source of data for generating rewritten queries [72, 76]. Query expansion usually results in more expensive query processing operations, as expanded queries are usually more complex than the often short input query.

Proximity Models. One disadvantage of the BOW models is that they treat each term independently, and do not consider *where* terms appear. However, the proximity of query terms can provide a strong signal of relevance, as there is often a dependency between terms. For example, the occurrence of a set of query terms within the same *sentence* is more likely to indicate relevance than if the query terms appeared in vastly different locations within a document. This is what proximity models aim to capture. Proximity models can capture *exact phrases*, as well as less rigid proximity such as a number of query terms appearing in *any order* within some fixed length window [154, 178]. One of the earliest works that explored term proximity was the seminal work of Clarke et al. [60], which allowed soft-phrase matching using a *followed-by* operator. More recently, the *sequential dependency model* (SDM) and the *full dependency model* (FDM) [178] leverage term proximity to improve the effectiveness of document ranking. SDM uses both *independent* term scores (like a BOW ranker), as well as dependencies between *sequential* terms in the input query, such as observing when sequential terms appear as bigrams, or within an ordered or unordered window of terms. The FDM model extends SDM to compute dependencies between *all* query terms, making it more expensive, yet slightly more effective [178]. In order to compute n-grams and window-based features, proximity models usually require *positional* or *forward* indexes to generate global statistics for ranking, making them much more expensive than BOW models in practice [155]. One reason for this increased cost is that, in order to generate the necessary features, the candidate positional indexes must be iterated across a number of times for multiple sub-queries. Recently, Lu et al. [156] proposed a novel method for integrating proximity into BOW ranking efficiently, which does not require global statistics, thus avoiding expensive feature generation. Their approaches were shown to outperform SDM and FDM with respect to both efficiency, and effectiveness.

Field-Based Models. Another simple extension to BOW (and proximity) models is to consider the underlying *structure* of the documents which are being ranked. For example, the text within a document title is likely to be highly relevant to the subject of the document. Hence, various document *fields* can be used to rank documents more effectively. To support advanced rankers across multiple fields, each field may be represented by field-based postings lists [210]. Each field is typically given a weight, and the document score can be computed as a weighted sum across the given fields. Robertson et al. [213] found that extending the BM25 model to fields (BM25F) was effective in practice [211, 267]. Both proximity models and relevance modeling have also been extended to work over fields, and show good effectiveness improvements over non field-based models [101, 151]. The downside to field-based approaches is that indexing fields requires more complex index formats, resulting in higher space consumption. Further, multiple postings lists may need to be accessed for each term in the query, making field-based approaches less efficient than simple BOW methods on term postings.

Learning to Rank. One downside to the previous models is that they primarily exploit a single or very few features for document ranking. For example, BOW models base their output rankings entirely on the notion of term matching, and proximity models may use both unordered BOW

matches as well as term proximity to form a ranking. However, there are potentially *hundreds* of features that could be useful for ranking documents with respect to a query. In order to capture a much larger number of features, machine learning algorithms can be applied to document ranking tasks in what is known as *learning-to-rank* (LtR) [152]. LtR systems typically learn a ranking model by consuming a number of annotated training examples. Each example is composed of a set of documents, represented by *feature vectors*, and a relevance annotation for each document. The feature vectors encode the many signals for relevance, including query-document features (BOW scores, proximity scores), static document features (PageRank, clickthroughs, spam rankings, inlinks), and query features (query length, IDF of terms) [164, 165]. Once the model is learnt, it can be deployed for ranking, where it will estimate the score for a new document based on its feature vector. To answer a query, a feature vector is generated for every document, and is applied to the LtR model. Then, the documents can be ranked by the scores output from the model. LtR models represent the current state-of-the-art for many ranking scenarios, including web search [270]. While LtR models can handle many features, resulting in highly effective rankings, computing both the features and the score for each document is very expensive. For now, we defer the discussion on how to deploy LtR models efficiently to Section 2.4.

2.3 Evaluating Search Effectiveness

High quality search results help users to satisfy their information needs. Therefore, a critical task for building and maintaining competitive search systems is evaluating the *effectiveness* of the search ranking. Consequently, a major research branch of information retrieval involves establishing models, metrics, and experimental frameworks for evaluating the effectiveness of search systems.

In this section, we focus on offline evaluation using reusable test collections, which is the *de facto* standard for evaluating search systems. In particular, Section 2.3.1 outlines how reusable test collections are built via the Cranfield paradigm [62]. Section 2.3.2 discusses how documents are judged, and the various relevance grades that may be assigned to documents. Next, Section 2.3.3 and 2.3.4 outline a number of commonly used metrics which support the evaluation of search systems. Finally, Section 2.3.5 discusses some best practices for the evaluation of search systems, and Section 2.3.6 outlines the test collections used further in this dissertation.

2.3.1 Cranfield Paradigm

In order to measure performance differences between candidate IR systems, a *standardized* experimental framework is required, where confounding factors can be minimized. In the 1960's, Cleverdon [62] proposed one such framework, which is now widely known as the *Cranfield* paradigm.The Cranfield approach utilizes a fixed document corpus, a set of topics, and a set of relevance judgments that correspond to whether each of the given documents is relevant with respect to each information need. To measure the search quality of a new IR system, the system is deployed on the given corpus across each of the topics to yield a ranked list. Then, the relevance judgments can be used to calculate an *evaluation metric* which reflects the quality of the system.

One issue with the Cranfield paradigm is the *cost* of relevance annotation. For example, modern web-scale collections contain millions of documents, so it is intractable for relevance assessors to annotate *every* document in the collection with respect to *every* topic. Instead, a process known as *pooling* is used. Pooling involves judging the top ranked documents from a set of unique IR systems which each supply their own ranking for each topic. These runs usually come from a number of different IR research groups, who may use different techniques to establish the rankings. Pooling aims to find most relevant documents by assuming that each of the contributing systems are high quality systems, and the quality of the judgments directly depends on how good the provided systems were in surfacing relevant documents.

Although many improvements have been explored and implemented, the Cranfield paradigm is still the primary approach for creating reusable test collections [108], and is used by many evaluation campaigns from the *National Institute of Standards and Technology* (NIST) *Text REtrieval Conference* (TREC), among others.

2.3.2 Relevance Judgments

When document assessors judge a document for a given topic, they are judging the *relevance* of the document to the supposed information need of the topic. To this end, judgments may be *binary*, where a document is either considered as *relevant* or *not relevant*, and each is assigned a weight of 0 and 1 during metric calculations. However, there are cases where more nuanced levels of judgment are desired. For example, *graded* relevance judgments may be provided, where annotators can assign a label of *not relevant* (0), *marginally relevant* (1), *relevant* (2), or *highly relevant* (3). Retrieval metrics may make use of these grades by providing different weightings based on a pre-defined *gain* function, which maps the label to a value in the range $[0, 1]$. While many gain functions exist, the most commonly used are the *binary* and the *exponential* gain functions. Without loss of generality, we denote as g_i the resultant gain derived from calling some gain function on the ith document of the ranked list under evaluation.

Another important aspect of relevance judgments is that they may be *incomplete*, due to the pooling process described above. If a set of judgments is incomplete, then documents that have not been annotated are usually assumed to be non relevant. However, an alternative is to compute a *residual*, which calculates the maximum increase of the effectiveness metric assuming that the non-annotated documents are relevant.

2.3.3 Utility Based Metrics

Utility-based metrics aim to model the utility that is accrued by a user as they scan down a ranked results list. Hence, finding a relevant document at a higher rank generally gives more utility to a user than finding the same document (or possibly an even more relevant document) further down in the ranked list. While there are a vast number of utility-based metrics, we focus on two main instances in our experiments, namely, *rank-biased precision* (RBP), and *expected reciprocal rank* (ERR).

RBP models a user who works down a ranked list with a given persistence, which is modeled using a geometric distribution, governed by a parameter ϕ [183]. The value of ϕ corresponds to the probability that the user will *continue* to the next document down the ranked list. Hence, values of ϕ close to 1 represent an extremely patient user who will continue a long way down the ranked list. On the other hand, smaller values of ϕ represent impatient users who assign a high weight to very few documents appearing in the top of the ranked list. RBP is formulated as follows:

$$\mathsf{RBP} = (1-\phi)\sum_{i=1}^{\infty}\phi^{i-1}\cdot g_i. \tag{2.7}$$

ERR captures a similar notion to RBP. However, ERR assumes that as a user accrues utility, they will be *more likely* to stop viewing further documents [55]. Thus, the stopping criterion is modeled explicitly:

$$\mathsf{ERR} = \sum_{i=1}^{k}\frac{g_i}{i}\cdot\prod_{j=1}^{i-1}(1-g_j). \tag{2.8}$$

2.3.4 Recall Based Metrics

Recall-based metrics normalize their results based on the *best possible outcome* for each topic in question. As such, they assume that the relevance annotations that exist for each topic are complete when creating the ideal ranking for the topic.

One of the oldest and most widely used recall-based metrics is *average precision* (AP). AP computes the precision at each relevant document in the ranked list, and divides by the total number of relevant documents for the topic:

$$\mathsf{AP} = \frac{\sum_{i=1}^{k} g_i \cdot \mathsf{Precision@}i}{|\mathrm{Rel}|}, \tag{2.9}$$

where $\mathsf{Precision@}i$ is the fraction of relevant documents observed up to rank i, and $|\mathrm{Rel}|$ is the total number of relevant documents for the topic. Usually, AP is calculated to extremely deep ranks ($k = 1{,}000$), and is less appropriate for use where judgments are incomplete.

An alternative to AP which is suited for computing early precision across incomplete judgments is the *normalized discounted cumulative gain* (NDCG) metric [118]. NDCG is based on the unnormalized version, *discounted cumulative gain* (DCG), which awards higher ranking documents with more utility:

$$\mathsf{DCG} = \sum_{i=1}^{k}\frac{g_i}{\log_2(i+1)}. \tag{2.10}$$

One problem with DCG is that values cannot be compared across different queries. NDCG aims to remedy this issue, by *normalizing* the gain value across each query:

$$\text{NDCG} = \frac{\text{DCG}}{\text{IDCG}}, \tag{2.11}$$

where IDCG corresponds to the DCG computed on the *perfect* ranked list for the topic in question — if the candidate ranked list contains the documents ordered by highest to lowest relevance, the value of NDCG will be 1.0.

2.3.5 Best Practices for Evaluation

Recently, recall-based metrics have come under scrutiny, as they assume to have complete knowledge of the universe of relevant documents for any given topic, which is unlikely to be true in practice. This makes computing residuals on recall-based metrics very difficult, and may result in confounding results in the presence of uncertainty. Lu et al. [157] recently conducted an analysis on the effect of pooling depth on commonly used evaluation metrics, including commonly used recall-based metrics. They found that these recall-based metrics can become unstable if evaluation beyond the pooling depth is carried out. In addition, they indicate that, if recall-based metrics are being reported, utility-based metrics should also be featured, presumably to show any ill-effects that may be created by evaluating with recall-based metrics inappropriately. Finally, they recommend reporting residuals where possible, especially if the system under evaluation did not contribute to the pool of judged documents. We follow these recommendations in the evaluation campaigns presented in this thesis. For a deeper discussion on IR metrics, pooling, and evaluation, we refer the interested reader to the textbook from Büttcher et al. [42], the survey from Sanderson [218], and the Ph.D dissertation of Lu [153].

2.3.6 Collections and Queries

For the experiments conducted henceforth, we use three different collections. These collections are publicly available, are used in many TREC tracks, and are described as follows:

- Gov2 — A crawl of around 25 million .gov domains from 2004. Each document is truncated to 256KB. This collection was used for the TREC Terabyte track from 2004–2006, and there are 150 corresponding topics with relevance judgments.
- ClueWeb09B — A sample of the web, crawled in January and February, 2009. We use the roughly 50 million document *category B* set. This collection was used for the TREC Web track in 2009–2012, and contains 200 corresponding topics with relevance judgments.
- ClueWeb12B — An updated sample of the web, crawled between February and May, 2012. We use the 52 million documents from the *category B-13* collection, which is a uniform sample of the English portion of the crawl. This collection was used for the TREC Web track in 2013 and 2014, and contains 100 corresponding topics with relevance judgments.

Name	No. Docs	Topics
Gov2	25,205,179	TREC 701–850 (‘04–‘06)
ClueWeb09B	50,220,423	TREC 1–200 (‘09–‘12), MQT
ClueWeb12B	52,343,021	TREC 201–300 (‘13–‘14), UQV100

Table 2.1: Summary of the collections and topics that are used for the experiments in this dissertation.

In addition to these collections, we also employ two other sets of topics. Firstly, the UQV100 topics contain a number of *query variations* corresponding to the same 100 topics listed above for the ClueWeb12B collection [18]. As such, the UQV100 topics are suitable for use on ClueWeb12B. This collection also contains shallow relevance judgments. Secondly, we employ the 2009 *Million Query Track* (MQT) queries [49], which consist of 684 topics with shallow partial judgments (50 overlap with the 2009 Web Track discussed above) over the ClueWeb09B corpus, and a total of 40,000 topics. A summary of this data is shown in Table 2.1.

2.4 Search in the Real World

In the previous sections, we have looked at the search process mainly from the perspective of a *user*, where we are concerned with improving search utility and measuring such improvements. However, there is another extremely important aspect to search which is often ignored, and that is how to *practically* implement efficient and effective search solutions that are capable of handling massive amounts of traffic and data, as is the case for web search. In this section, we discuss the architecture of large-scale search engines, the competing goals of efficiency, effectiveness, revenue, and cost, and some of the techniques used to balance them.

2.4.1 Search Engine Infrastructure

Most large-scale IR systems are backed by thousands of commodity servers which allows the processing workload to be distributed among many peers in a fault-tolerant manner, as evidenced by some recent outlines of large-scale search architectures from companies such as Google, Microsoft, eBay, and Twitter [20, 39, 210, 245]. To achieve rapid response times, large document corpora are divided into a set of smaller partitions, which is known as *index sharding*. Different methods of sharding exist, and result in vastly different processing architectures. For example, documents can be assigned to shards based on their *topic*, which can then be leveraged to search only a *subset* of shards at query time in what is known as *selective search* [131, 137]. However, this adds complexity to the search system, and may result in load-balancing issues. A more common approach is to randomly allocate a document to a shard, such that each shard is kept to a uniform size. For each shard, an index is built, and the index for one or more shards can be stored in an *index server node* (ISN). An ISN is typically a standard commodity server, and ISNs are responsible for the

majority of the processing undertaken to serve query results. For example, researchers from Microsoft noted that at least 90% of the hardware resources in the IR system they were working with were dedicated to ISNs [119]. Clearly, the role of the ISN is crucial to the performance of web search. When an incoming query is received, it is routed to one or more ISNs, where the query is processed, and the results returned. Usually, a *broker* or *aggregator* will wait for results from each ISN, and blend them together to form the final SERP. Since web search engines are typically used for a wide range of query types, there may be a range of separate search pipelines which are each optimized to their own search task, such as ad retrieval, question answering, or ad-hoc web search. Figure 2.3 shows an example web search system, which combines results from six different search pipelines, and the underlying architecture of one of the search pipelines.

2.4.2 Efficiency, Effectiveness, and Revenue

First and foremost, a real search system must be both efficient, and effective. In Section 2.2.3, we discussed some recent innovations in ranking, which enable highly effective retrieval. However, increased effectiveness generally comes at a cost — the more advanced a ranker, the more expensive it is to deploy. Thus, there is a tension between providing high quality results, and providing results within a reasonable time. To make this problem even more difficult, users expect sub-second response times in common search scenarios, and delays can negatively impact user experience [16, 17, 177, 224]. For example, Schurman and Brutlag [224] conducted a large-scale experiment across both the Bing and Google search engines to examine the effects of latency on user behavior, and in turn, revenue. These experiments involved injecting artificial delays into the search process, such that the user perceived latency of the search system was increased. They found that delays as low as 200ms resulted in significant reductions to satisfaction metrics, user interaction, and average revenue per user. Furthermore, the magnitude of the effect was found to be almost linearly associated with the amount of delay added. Another user study conducted by Google showed that users preferred viewing SERPs with 30 results rather than 10 results [177]. However, the added delay in retrieving and serving 20 additional results lead to users searching around 25% *less*.

In a similar line of work, Bai et al. [17] conducted a large analysis of both users (in a lab setting) as well as query logs, to determine the effects of latency on user behavior. Their findings generally aligned with those of Schurman and Brutlag [224], but provide more nuance. For example, they found that users of a fast search system are more likely to notice delays when compared with users of a slow search system. Furthermore, the degree in which added latency impacts users may depend on user attributes. Another interesting finding was that users generally clicked more often on a result page which was served with a lower latency. A follow up study examined the effects of latency on *sponsored search*, where paid advertisements appear within the ranked results list. This study concluded that users tend to becomes less engaged with the search engine if delays are consistently observed, and that revenue is higher when pages are served faster, thus confirming the previous observations from other studies [16] — balancing both efficiency and effectiveness is key in providing a positive user experience.

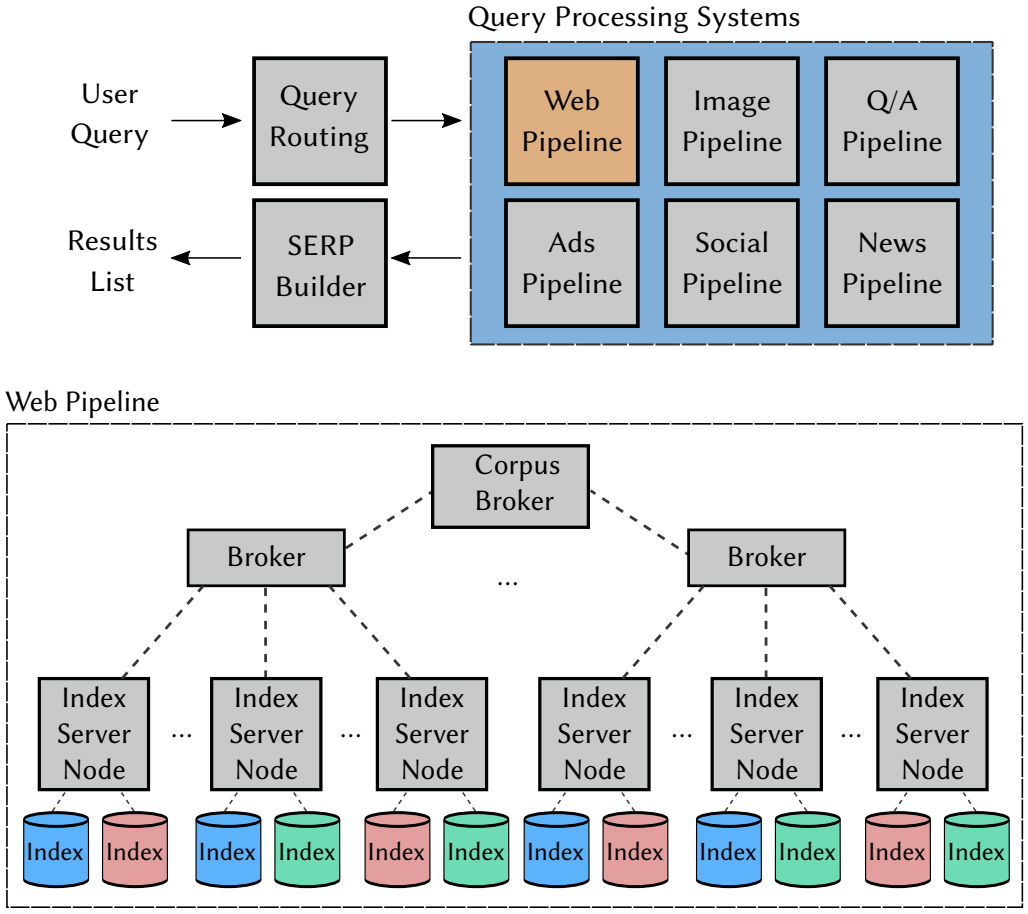


Figure 2.3: An example of some of the complexities of large-scale search. The top figure shows some of the components that may contribute to the final results set. Each component may have its own index and query pipeline. The bottom figure shows an example of the architecture of a single query processing pipeline, based off of the Maguro system [210] which powers the Bing search engine. At the top level, the corpus broker manages the load balancing of query processing across various partitions of the overall index. The second level brokers handle load balancing of systems that share a set of replicas of the given index. Each ISN is responsible for the matching and ranking of that particular index segment, and the results are then passed back up to the top-level broker for merging.

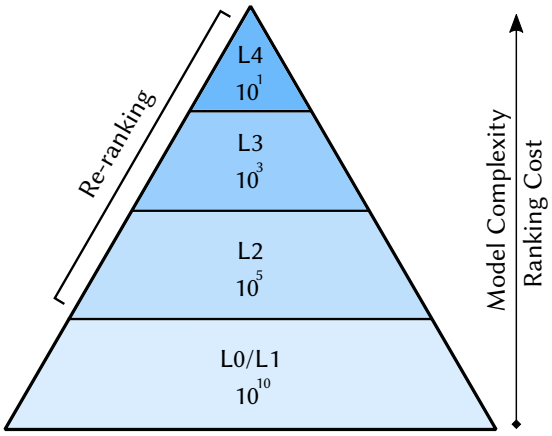


Figure 2.4: An example of a multi-stage retrieval system, as described by Pedersen [192]. This particular system has an initial candidate generation phase, followed by three re-ranking phases. Each subsequent re-ranker uses more expensive features on fewer documents, balancing efficiency and effectiveness.

2.4.3 Multi-Stage Retrieval

Clearly, there is a delicate balance between providing effective results and providing timely responses, and this is a major challenge that must be met by competitive search companies. In Section 2.2.3, we outlined how state-of-the-art learning-to-rank (LtR) models can use hundreds of important features to accurately rank documents. However, deploying these models on production systems is extremely expensive, as the costs of computing or retrieving these feature scores for each document is prohibitive. To improve the practicality of LtR, a sub-field focused on *learning-to-efficiently-rank* has emerged, which focuses directly on efficiency problems in LtR systems [254].

One obvious way to make LtR practical for real-time search is to cast it as a *re-ranking* problem, where a simple ranking algorithm (like a BOW ranker) is used to generate a top-k set of *candidates* which can then be re-ranked using the LtR system [38, 192, 252]. This idea can be taken even further, with any number of arbitrary re-ranking stages, in what is known as *multi-stage* or *cascade* ranking. In order to efficiently deploy the expensive machine-learnt models as described in Section 2.2.3, the ranking process can be divided into a number of *select, rank, and prune* stages [192]. In particular, each stage of ranking will take a list of documents from the previous stage as input, compute some ranking on these documents, and possibly pass only a subset of documents to the following stage. This idea was first discussed in detail by Wang et al. [252], who aimed to achieve effective rankings with reasonable cost. For this reason, each consecutive stage of the multi-stage process usually involves using a *more expensive ranker* on a *smaller set of documents* [57]. Figure 2.4 shows the different stages of a four stage ranking cascade, and the number of matched or re-ranked documents at each stage. Configuring multi-stage retrieval systems to optimize efficiency/effectiveness trade-offs is an ongoing line of research, and we examine this problem further in Chapter 6.

The two-stage retrieval system is the most simple multi-stage retrieval architecture, and the one we focus on in this dissertation. The two primary ranking stages in a two-stage retrieval system are the *candidate generation* phase, and the *re-ranking* phase. The primary goal of candidate generation is to *rapidly* obtain a top-k set of candidate documents. As such, candidate generation algorithms use very simple features, such as Boolean matching or basic BOW rankers, which enable fast retrieval. Next, for each candidate, the *feature vector* is either retrieved (from a data store) or computed, and these vectors are then passed to the LtR model for re-ranking.

2.4.4 Containing Latency

Since large-scale IR systems are highly complex systems with many components, it can be difficult to contain the end-to-end latency. One way in which costs can be contained is by imposing *service level agreements* (SLAs) on the components of the search engine. A service level agreement is an agreed performance budget that must be adhered to [117]. In the context of search engines, SLAs are usually set for each stage of the search process, which ensures that the entire end-to-end latency does not negatively impact users [17]. Typically, SLAs will enforce a guarantee on the tail latency, as this best reflects the worst case observed latency for the end user. For example, consider a three stage retrieval system, where each stage begins processing once it receives input from the prior stage. If each stage had an SLA of 100ms at the 99th percentile, then, in expectation, *at least* 97% ($0.99^3 \approx 0.97$) of queries would finish within 300ms.

2.5 Summary

In this chapter, we introduced some of the salient aspects of information retrieval, including a brief history of information retrieval, methods of ranking documents, how rankings can be evaluated, and an overview of the challenges that must be met by real search providers. Section 2.1 briefly explored some of the history of IR, including the rise of the world wide web, and outlined the lifetime of an information need from the perspective of a user. In Section 2.2, we explained how documents can be matched and ranked with respect to an input query, with a focus on simple bag-of-words models. We also explored how effectiveness can be improved via more expensive models that use additional signals for ranking, arriving at the current state-of-the-art learning-to-rank models. Section 2.3 focused on how retrieval models can be evaluated to ensure their quality. In particular, we explored the notion of evaluation over test collections, and detailed a number of important metrics which capture the effectiveness of rankings. Finally, in Section 2.4, we discussed some of the added complexities of end-to-end retrieval in practice, including efficiency and cost considerations. The next chapter continues to explore this thread of efficient retrieval by examining the core data structures and algorithms that are employed by large-scale IR systems for scalable retrieval over extremely large collections.

3 Efficient Search Systems

Up to this point, we have discussed the search process with a focus on the user, including the search process, effective approaches for ranking documents, and approaches for evaluating the effectiveness of search systems. We also outlined the need to support efficient retrieval, and discussed some considerations for supporting large-scale search. In this chapter, we focus on the underlying data structures and algorithms that are used for scalable query processing in IR systems. In particular, Section 3.1 focuses on how documents are indexed, and various ways in which indexes can be organized. Section 3.2 outlines how queries can be efficiently processed on a variety of index layouts, and the optimizations that make efficient retrieval possible. Finally, Section 3.3 examines the current state-of-the-art for reducing tail latency in large-scale search systems.

3.1 Text Indexing

Before a search engine can provide search functionality over a text corpus, the corpus must be *indexed*. Indexing involves identifying, parsing, and representing the desired text collection appropriately, such that the underlying documents can be retrieved efficiently and effectively. Figure 3.1 highlights some of the key components of a large-scale search system and how they interact. In particular, a crawler traverses the web graph, finding new and updated pages to index, and downloads them to a local document store. Then, the indexer converts these raw documents into a representation which is suitable for efficiently matching and ranking documents. As such, search engine indexes generally store a number of statistics relating to the collection and each document, which are required by the ranking model. Upon receiving a query from the user, the query processing algorithm will access these statistics, and use them to generate a ranked results list, which can be returned to the user.

3.1.1 Document Crawling and Parsing

Crawling. The first step for a search engine is to identify documents which should be indexed. While some document collections are easily controlled, such as those that are mostly static, others evolve rapidly. The world wide web is the largest and most rapidly changing collection of data

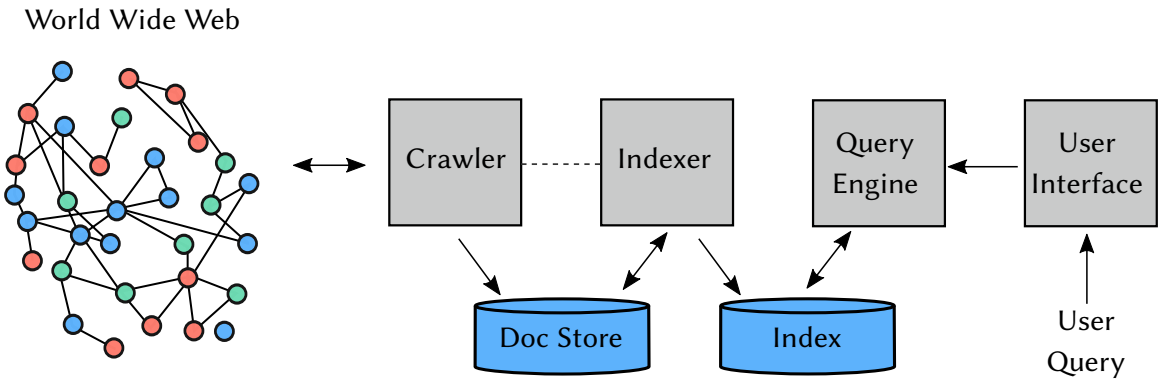


Figure 3.1: The basic organization of a web search engine. The crawler finds and downloads documents from the web, which are loaded into a document store. The indexer is notified, and can then index the document into the relevant index structures. These index structures are then accessed by the query engine to serve incoming user queries which are recieved via the user interface.

in the world, and consequently, web search providers must be able to handle this change. To this end, efficient *crawlers* are constantly enumerating the web graph, traversing across links inside documents to find new or modified documents [50]. The crawler must then download such documents into a local data store, so they can be *indexed*.

Normalization and Indexing. The indexer, which translates the *raw* document text and metadata into a *clean* representation within the index, plays a crucial role in providing high quality search. Since, especially for web search, a wide range of document formats exist, it is up to the indexing module to *parse* the document to extract the natural language, and then *normalize* the text into a uniform format [67]. Document normalization may include multiple operations such as *case folding*, *stopping*, and *stemming*.

Case folding corresponds to normalizing the case of the terms (from upper case to lower case, for example). The motivation for case folding is that the case of the term should not impact its relevance to a user. For example, the term *"Coffee"* would be normalized to *"coffee."*

Stopping involves removing terms which appear extremely frequently, as common terms are not likely to help in identifying relevant documents. Stopping also decreases the space occupancy of the index and the efficiency of traversal, as stopwords are not stored or looked up. However, queries composed entirely from stopwords such as *"take that"* cannot be answered from a stopped index. A more flexible option is to conduct stopping selectively at *query time*, which allows the system to decide whether to keep the stop words or not.

Another important normalization technique is stemming, which involves converting each term to its root form [115]. The idea is similar to case folding, in that variations arising in the tense of a term should not change the relevance of the term to a user. For example, the terms *"listening"*, *"listened"*, and *"listener"* may all be converted to the same term, *"listen."* Though many

stemmers have been proposed, it is generally not clear which stemmer is the best choice. For example, Hull [115] conducted a large analysis of stemming approaches, concluding that *"some form of stemming is almost always beneficial."*

After a document has been parsed and the text normalized, it is ready to be added to the search engine index. Since the index is represented by integer number values, as discussed shortly, each document is associated with a *unique, numerical* identifier, known as the *document identifier* or docID. As discussed above, the indexer may also compute a number of statistics which will be stored within the index to assist with ranking operations. Finally, the documents and associated statistics are propagated into the index structure to allow them to be searched efficiently.

3.1.2 Inverted Indexes

Inverted indexes are the most common type of index used for representing large document collections, as they allow fast and scalable access to a list of *documents* that contain a given *term* [273].

Core Components. Inverted indexes contain a few components. Firstly, the *lexicon* (or dictionary) contains an entry for each unique term, t. Each entry in the lexicon holds two values:

- f_t, the number of documents containing t, and
- A *pointer* to the start of the corresponding *postings list*.

Note that this pointer may refer to a location on a hard disk, or it may refer to a location in memory. The second core component of the inverted index is the *postings file*, sometimes called the *inverted file*. The postings file stores postings lists, where there exists one postings list per unique term, t. Each list consists of:

- A list of docIDs, d, for each document containing t, and
- A list of corresponding frequencies, $f_{d,t}$, which count the number of occurrences of t in d.

Usually, postings lists are conceptually imagined as a list of $\langle d, f_{d,t} \rangle$ pairs, but most underlying implementations store the document identifiers and frequencies as separate lists.

Another important component that is required for inverted indexes is the *document map*, which maps the numerical docID back to metadata about the document. The information stored for each document may vary depending on the search engine, but attributes such as the document URL, the document length, and perhaps a pointer to the document itself may be stored here. We do not discuss this structure further.

Document-Ordered Indexes. The most common inverted index organization is the *document-ordered* inverted index. The document-ordered index organizes its inverted lists in a *monotonically increasing* manner, based on the document identifiers. This layout makes it easy to implement

Doc ID	Parsed Text
1	algorithm data structure important efficient search
2	best data structure depend application data search data
3	efficient search important user experience
4	user search data
5	efficient algorithm depend efficient data structure

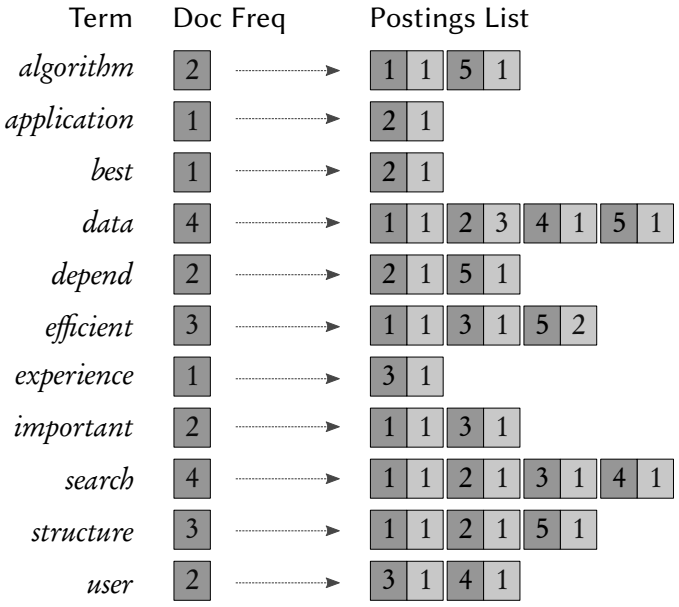


Figure 3.2: An example text corpus and the resultant document-ordered inverted index. In this format, each postings lists stores pairs of document identifier and term frequency ($\langle d, f_{d,t} \rangle$) in ascending order of the value of d.

efficient operators that can quickly *skip* across the postings lists, as the value of the document identifier being examined always increases. We explore these ideas further in Section 3.2. Another advantage of this layout is that it enables efficient index update operations, as new documents will always be appended to the end of the inverted index. Figure 3.2 shows an example document collection, and the resultant document-ordered index.

Frequency-Ordered Indexes. An alternative approach for organizing the postings lists is to order them by the value of the term frequencies. In a *frequency-ordered* index, each postings list is organized such that documents with the *most* occurrences of the given term appear at the *front* of the list [194, 195]. That is, each postings list is sorted in decreasing order of term frequency. The

main motivation of this organization is that documents with higher $f_{d,t}$ values are likely to yield higher scores from the ranking function, and so putting them at the front of the postings list ensures they are scored earlier during query evaluation. Since each postings list is organized around term frequencies, the frequency lists will naturally contain a number of repeated values, where the corresponding documents contain the same number of term occurrences. This redundancy can be exploited to reduce the space occupancy of the inverted index. Instead of storing a list of document identifiers and a list of frequencies, these lists can be combined to generate *postings segments*. A postings segment is simply a monotonically increasing list of document identifiers, and each segment has an associated metadata structure which contains:

- ✦ A single $f_{d,t}$ value for the documents within the segment,
- ✦ The number of documents contained in the segment, and
- ✦ A pointer to the start of the segment.

Thus, each postings lists will contain one segment per unique $f_{d,t}$ value, the $f_{d,t}$ value is only recorded *once*, and the postings list is made up of a number of such segments. Within the postings list itself, the segments are still ordered monotonically decreasing on the frequency, as this allows the more important documents to be scored earlier. Figure 3.3 shows a simplified version of this layout, where each segment is represented as an $f_{d,t}$ value followed by a monotonic list of document identifiers. In this example, the terms *"data"* and *"efficient"* contain two segments, and the remaining terms only contain a single segment.

Impact-Ordered Indexes. The key observation in frequency-ordered indexes is that, by sorting documents in descending order of their term frequencies, the documents that are likely to incur the largest score contributions from the ranking function, $\mathcal{W}$, will be examined earlier in the retrieval process. Anh et al. [9] noted that this idea can be taken even further by *pre-computing* the *impact score*, $i_{d,t} = \mathcal{W}(d,t)$, *for each* posting in the index, and then storing this impact value in lieu of the term frequency. The advantage of this method is two-fold. Firstly, the postings can now be sorted by their true score contributions, not just an estimation based on term frequency. Secondly, at query time, the impact function $\mathcal{W}$ does not need to be computed — instead, the impact is simply retrieved from the index. However, a clear drawback of this method is that most useful instantiations of $\mathcal{W}$ compute *floating point* values which cannot be stored efficiently.

To remedy this problem, *quantization* can be applied to the impact values [9, 184]. Score quantization approximates the value of each impact via a b-bit integer, allowing each impact to be represented by a score in the range $[1, 2^b - 1]$. For example, consider the *uniform* quantization approach, which aims to map the index-wide impacts into fixed buckets. Given some impact i, it will be represented as a bucket by computing

$$\left\lfloor 2^b \cdot \frac{i - i_{\text{Min}}}{i_{\text{Max}} - i_{\text{Min}} + \epsilon} \right\rfloor,$$

Doc ID	Parsed Text
1	algorithm data structure important efficient search
2	best data structure depend application data search data
3	efficient search important user experience
4	user search data
5	efficient algorithm depend efficient data structure

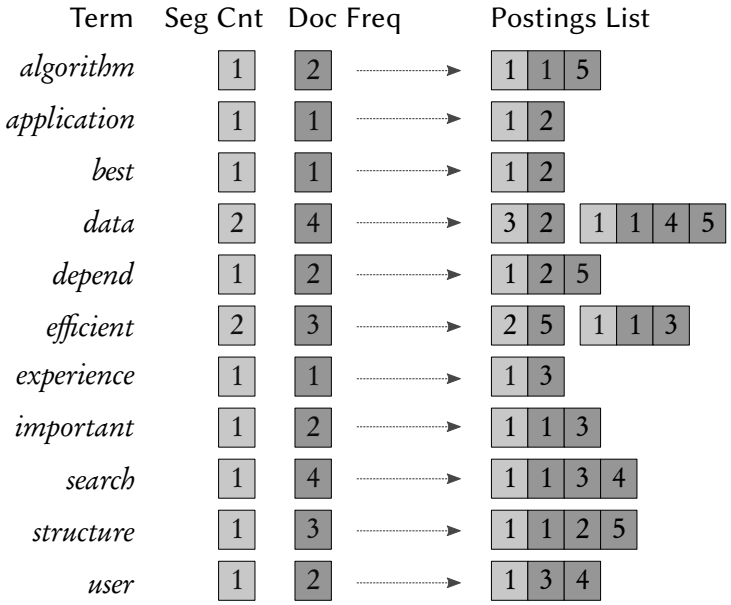


Figure 3.3: An example text corpus and the resultant frequency-ordered inverted index. In this format, each postings lists stores a list of *segments*. Each segment contains a frequency, $f_{d,t}$, and is followed by a list of documents that have the corresponding frequency value. Within each segment, documents are stored monotonically increasing by their identifiers, and segments are stored monotonically decreasing on the $f_{d,t}$ values. Note that an *impact-ordered* index follows the same conventions, but instead of storing $f_{d,t}$ values, quantized impact scores are stored instead.

where i_{Min} and i_{Max} are the smallest and largest observed values of $i_{d,t}$ across the entire index. While other quantization methods have been explored, the most important detail is to set b sufficiently large, which minimizes the precision lost during quantization [66].

Assuming we have scored and quantized each posting in the inverted index, the *impact-ordered* index layout is organized the same way as the frequency-ordered index. That is, each postings list contains a number of segments, and the segments within each list are sorted in descending order of the *quantized scores* (in lieu of the term frequencies).

3.1.3 Alternative Index Organizations

While we have covered the basics of inverted indexes, there are many extensions or alternatives that have been examined in prior research. We briefly outline some of the main alternatives and their trade-offs with traditional inverted indexes. However, we do not consider these indexes further in this dissertation.

Positional Indexes. The *positional* index is an extension to the inverted index described previously. Instead of recording a document identifier and the number of occurrences of a given term in that document, positional indexes also record *where* the terms appeared in the document. This allows partial or exact phrases to be matched by the search engine [260], and is crucial to the proximity-based ranking models described in Section 2.2.3. The clear disadvantage of positional indexes is that they are several times larger than their non-positional counterparts.

Forward Indexes. Where the inverted index stores a list of documents that contain a given term, the forward index opts to store a list of *terms* found within each *document*. These indexes are useful for conducting operations on a list of documents, but are not scalable for document retrieval, as there is no efficient way to find out which documents contain a given term. Like inverted indexes, forward indexes may or may not preserve term position information [12].

Signature Files. Signature files are *probabilistic*, and are radically different to the other types of indexes that have been described thus far. The idea with signature files is to represent each document with an n-bit *signature* [91]. In order to calculate a document signature, each term in the document is hashed (multiple times), which sets a series of bits within the signature. To run a query, the query terms are hashed and a new signature is created. Then, the query signature can be matched against each document signature using basic Boolean operations. While Zobel et al. [274] showed signature files to be larger, slower, and less flexible than inverted indexes, recent improvements have seen a new breed of index files emerge [104, 105].

Geva and de Vries [104] proposed *TopSig*, which generates document signatures from a pre-computed vector-space matrix representation. The main idea was to use dimensionality reduction to effectively represent the content of each document from its vector representation. Since TopSig is derived from a vector-space representation, it can be used for document ranking, unlike traditional signature files, and was shown to perform as well as standard BOW ranking models (like BM25 and LMDS) if signatures of sufficient size were employed.

Goodwin et al. [105] showed how signature files can be used effectively in modern, large-scale IR systems, by devising a range of performance improvements that make them amenable to modern architectures. For example, the number of hash functions used for generating signatures is dynamically set on a term-by-term basis, and documents are sharded by length to improve space consumption. Their proposed system, *BitFunnel*, is used to process the freshly-crawled tier of the Bing web search engine by providing fast and approximate Boolean conjunctions on incoming queries.

Boolean Indexes and Key-Value Stores. Beyond the classic inverted index as desribed above, inverted indexes can be optimized for efficient query processing over Boolean queries. In these indexes, only the document identifiers need to be stored, and postings lists may be stored as bitvectors for rapid Boolean query processing [126]. These types of search systems are suitable for processing large and complex Boolean expressions [93], which may arise in certain scenarios [32]. Once a set of candidate documents are found, the document text and metadata can be accessed via forward indexes or efficient key-value stores.

3.1.4 Compressed Representations

To reduce the storage consumption of the inverted index, compression algorithms are generally applied to the postings lists during indexing. Compression is an extremely well researched area, with many decades of innovation leading to extremely fast and small compression schemes. Here, we provide an overview of how space occupancy can be minimized for text indexes.

Delta Encoding. Given a monotonically increasing sequence of integers, the storage cost can be decreased by storing the *delta gaps* (d-gaps) between the integers rather than the integers themselves. This is because each gap, by definition, must be smaller than the original integer it represented. Since the sequence of document identifiers in a document-ordered postings list are guaranteed to be monotonic, they can be delta encoded prior to compression. The same idea applies to each segment of a frequency- or impact-ordered index. Formally, given a monotonically increasing sequence of integers $\langle x_1, x_2, \ldots, x_n \rangle$, the delta encoded sequence is defined as $\langle x_1, \delta_2, \ldots, \delta_n \rangle$, where $\delta_k = (x_k - x_{k-1})$. To decode element x_j, the prefix sum must be taken up to element x_j as

$$x_j = x_1 + \sum_{i=2}^{j} \delta_i.$$

Since accessing element x_j involves computing the prefix sum, a block of delta encoded integers is typically *batch decoded* and stored in its original form until the block has been processed. Furthermore, recovering the original sequence from the delta encoding can be accelerated through special CPU instructions, known as *single instruction, multiple data* (SIMD), which can perform operations on multiple data points using a single CPU instruction [144].

Compression Methods. A wide range of compression methods exist with many different time-space trade-offs, and researchers continue to invent new schemes that push the frontiers of time and space. While the problem of compression is orthogonal to the work presented in this dissertation, we briefly outline a few of the main compression *families*, the general ideas they use for efficiently representing integer sequences, and an outline of the current state-of-the-art in inverted index compression. Table 3.1 shows a number of compression codecs which are compared in Figure 3.4, organized into families. These families are briefly described now.

Codec	Description
VByte [237]	First byte-aligned code, still commonly used based on its simplicity.
Varint-GB [81]	Extends VByte by organizing data differently to speed up decoding.
Varint-G8IU [233]	Extends Varint-GB to use SIMD instructions for rapid decoding.
Masked-VByte [203]	Extends VByte to use SIMD instructions for rapid decoding.
Stream-VByte [145]	Uses a different organization of data, optimized with SIMD, for rapid decoding.
Opt-VByte [201]	Hybrid encoder selects between using VByte or bitvectors.
Simple16 [269]	Packs a variable number of integers into fixed 32-bit words.
QMX [242, 244]	Packs a variable number of integers into fixed 128-bit words.
Opt-PFOR [263]	Space optimal version of PFOR that uses *exceptions* to store outlier values to avoid redundancy in the representation of the sequence.
BIC [181]	Recursively encodes monotonic sequences, very high compression rate, slow to decode.
PEF [191]	Two-level Elias-Fano encoding that partitions and exploits locality of the underlying sequence.
CPEF [199]	Improved version of PEF based on the clustering of postings lists.
ANS [89, 180]	Entropy coder based on *Asymmetric Numerical Systems*.
DINT [202]	Dictionary coder based on a *Dictionary of INTeger sequences*.

Table 3.1: A summary of some state-of-the-art compression mechanisms for inverted indexes. The table is broadly categorized into different *families* of compression schemes, with byte-aligned codes, word-aligned codes, patched codes, monotonic codes, entropy codes, and dictionary codes represented.

Byte-aligned codes aim to represent each integer by the minimum number of *bytes* required, and these schemes are generally referred to as *variable byte* (VByte) schemes. These approaches are simple and portable, and many variations have been described. The main idea is to encode each byte with a single *continuation* bit, and the remaining 7 bits used to store the *payload*. The continuation bit is set if the next byte makes up part of the current integer value. While the idea is simple, many improvements have been proposed, which mainly focus on improving decoding speed by either re-organizing the storage method of the VByte mechanism, introducing SIMD intrinsics, or both.

An alternative to *byte-aligned* codes is *word-aligned* codes. Word-aligned codes represent a variable-length sequence of integers in a fixed-length block. These methods store information in a *selector* so they know how to handle the subsequent *payload*. The most well known word-aligned

variants are the Simple coders. QMX is another approach which has been extensively optimized for efficient decoding. Word-aligned codes are usually slower, but more space efficient, than byte-aligned codes.

One issue with packing integers into fixed-length blocks is that, if a single outlier value is present among the sequence to be compressed, the encoding scheme will need to use a suitable configuration to represent this outlier value. As such, this leads to suboptimal compression rates, as values other than the outlier will be represented by much larger integers than required. To account for this, *Patched Frame-Of-Reference* (PFOR) coding allows some values in a fixed-length sequence to be represented as *exceptions* using a secondary encoding mechanism. This reduces the impact that large outlier values can have on the effectiveness of packing the sequence, and results in better compression effectiveness.

Monotonic coders operate directly on monotonic sequences, so are not amenable to compressing delta-gaps as described previously. A wide range of these approaches exist, including binary interpolative coding (BIC), which provides extremely good compression but is very slow to decode. BIC recursively encodes a given sequence by splitting it into two parts, and encoding the middle element. Another approach is to use *Elias-Fano* codes to represent inverted lists [249]. The main idea behind Elias-Fano coding is to represent the sequence as a binary sequence, encoding each element with $\lceil \log U \rceil$ bits (where U is the largest integer in the sequence), and then *splitting* this sequence into two parts: the *high* bits, consisting of the first $\lceil \log n \rceil$ most significant bits, and the remaining *low* bits. These two parts are then encoded separately, and concatenated together. For a further description of the Elias-Fano coding mechanism, we refer the interested reader to the preceding references. *Partitioned Elias-Fano* PEF method improves on the Elias-Fano code by partitioning the underlying sequence into variable-length blocks, and using a two-level encoding process. Further improvements to PEF can be achieved by clustering the postings lists prior to partitioning, resulting in the CPEF encoder. Both PEF and CPEF improve on the standard EF encoder with respect to space occupancy, sacrificing some decoding efficiency in the process.

Entropy coders aim to represent frequently occurring patterns with a small number of bits, and rare patterns with a large number of bits. This then reduces the overall storage requirement of the underlying data. *Asymmetric Numerical Systems* are a recent development in entropy coding, and have been shown to give extremely effective compression rates [180]. However, ANS methods are slow to decode.

Dictionary based coders use a separate *dictionary* to store a pattern, and then replace the pattern in the sequence with a *pointer* to the dictionary entry. These have only been explored very recently for encoding postings lists, but seem to be a promising alternative to the other families described above [202].

Selecting a Compression Codec. Figure 3.4 shows the current state-of-the-art in postings list compression, with respect to *conjunctive* query processing efficiency and index space consumption.[1] The experiment shown involves computing the conjunction of each query across the Gov2

[1] We thank Giulio Pibiri for providing the raw data and allowing us to reproduce this figure.

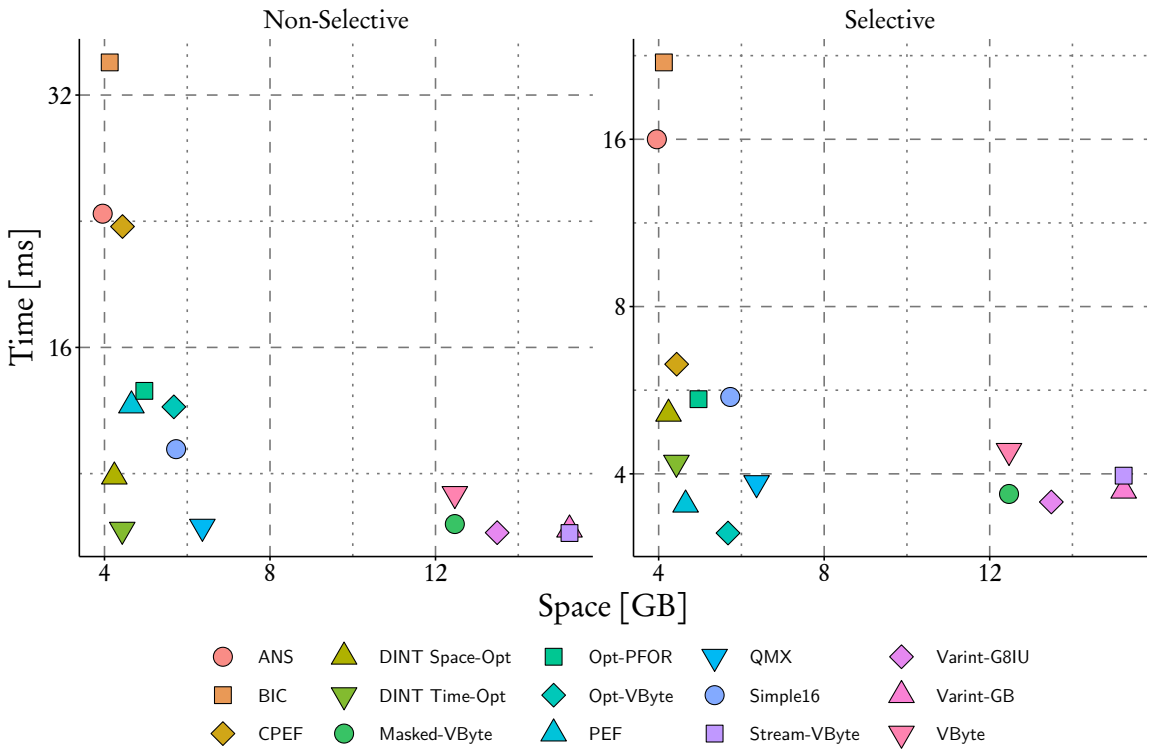


Figure 3.4: The current state-of-the-art in compression codecs, adapted from Pibiri [198]. This figure shows the time and space trade-off for processing conjunctive queries on the Gov2 collection using a random sample of *non-selective* and *selective* TREC 2005 and 2006 efficiency track queries.

collection for both *non-selective* and *selective* queries. Non-selective queries refer to those in which the ratio of matching documents between the Boolean conjunction and disjunction is large (10% in this experiment). On the other hand, selective queries have a low ratio between conjunctive and disjunctive matches (0.5%). As such, processing non-selective queries involves more *sequential access* across the underlying postings, whereas processing selective queries induces more *skipping* across the postings lists. Selecting the best codec therefore depends greatly on the specific type of processing that is required. If decompressing many postings is required, as is the case with non-selective queries, then fast sequential decoding schemes will result in better efficiency, but generally with a higher space occupancy. If the underlying query type will generate a lot of skipping, the time/space trade-off changes considerably. A recent study from Mallia et al. [175] had similar conclusions, though a different set of compression schemes was explored. In any case, it is important to highlight that the best compression codec ultimately depends on the requirements of the given search system. We refer the reader to the Ph.D dissertation of Pibiri [198] and related survey [200] for a deeper analysis of the current state of compression for inverted indexes.

Document Identifier Assignment. Another way of reducing the space consumption of an inverted index is to *re-assign* the underlying document identifiers. Since the document identifiers are represented as d-gaps, a reduction in the overall size of these d-gaps should allow compression codecs to achieve more compact representations. For example, consider the following collection, containing five documents and three postings lists.

$$\mathcal{L}_1 = \langle 1, 3, 4, 5 \rangle$$
$$\mathcal{L}_2 = \langle 1, 5 \rangle$$
$$\mathcal{L}_3 = \langle 2, 3, 5 \rangle.$$

The average d-gap computed across this collection is 3.67. Now, suppose we had a re-ordering algorithm which returned a permutation

$$\pi = (1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 5, 4 \mapsto 3, 5 \mapsto 2),$$

the document identifiers can be re-assigned (and re-ordered) corresponding to π as:

$$\mathcal{L}_1 = \langle 1, 2, 3, 5 \rangle$$
$$\mathcal{L}_2 = \langle 1, 2 \rangle$$
$$\mathcal{L}_3 = \langle 2, 4, 5 \rangle.$$

This newly *re-ordered* collection has an average d-gap of 2.67, hence saving around 1 bit per entry compared to the original ordering. This is the motivation behind document identifier re-assignment, and a large number of methods have been proposed [35, 58, 83, 171, 228]. Although different approaches provide different trade-offs between their computational cost and the resultant savings, they all share a common goal, which is to *cluster* similar documents together — by assigning documents with similar content to similar integer identifiers, the resultant d-gaps in the related postings will be small.

One of the most simple and effective methods for re-assigning document identifiers is to lexicographically sort the document URLs, and assign document identifiers according to this ordering [228]. The reason that URL ordering works well is that it clusters websites with similar URLs together, which is a reasonable proxy for document similarity. One drawback to this approach is that it is only suitable where URLs are available.

An alternative method which uses document *content* to cluster similar documents together is the MinHash ordering [35]. MinHash generates a document *signature* via hashing, and then sorts the documents by their signatures. The hash function which generates each document signature operates on the *terms* present in the document, and hence, sorting the documents based on these signatures is actually *clustering* documents together based on an approximation of their content. Improved versions of MinHash use multiple hash functions, generating better signatures, which improves clustering [58].

Property	Document-ordered	Frequency-ordered	Impact-ordered
Query Types	Boolean, ranked	Ranked	Ranked
Updates	Yes, append	Non trivial	Non trivial
Compression	Block-based	Variable-length	Variable-length
Payload	Term frequencies	Term frequencies	Impact scores
Traversal	DAAT or TAAT	TAAT	SAAT

Table 3.2: A brief summary of some key aspects of various postings list layouts. Document-ordered postings lists are more flexible than either freqency- or impact-ordered postings lists, as they allow efficient Boolean matching and are easy to update.

Recently, Dhulipala et al. [83] proposed an approach called BP for re-ordering graphs using *recursive graph bisection*. Instead of trying to cluster similar documents together using heuristic methods, the BP algorithm *directly* optimizes the global space occupancy of the d-gaps within the collection. BP involves recursively dividing the collection into two halves, and moving documents between each partition if the overall storage cost is reduced. Experimental analysis showed impressive performance gains against a number of competitors across a number of collections, but focused primarily on the compression of large graph structures. Mackenzie et al. [171] revisited the work of Dhulipala et al. [83] with a specific focus on inverted index compression. They showed that BP was able to outperform MinHash and URL ordering across a number of text collections and compression codecs, confirming that BP is the current state-of-the-art for compression via identifier re-assignment. However, BP is much more expensive than both URL and MinHash in practice, in terms of both computational and memory requirements, and should only be considered if a suitable amount of computational resources are available. Otherwise, URL ordering is the next-best choice, as it outperformed MinHash ordering for larger collections.

An interesting side effect of document re-ordering is that it has been shown to improve the efficiency of index traversal [88, 207]. For example, Ramaswamy et al. [207] showed improvements of up to 20% when re-ordering the identifiers in the eBay search system. Ding and Suel [88] showed that document re-ordering can result in latency improvements of over 50% for ranked conjunctions and disjunctions. Thus, re-ordering of document identifiers is an important pre-processing step when building inverted indexes, for both storage consumption and query latency.

3.1.5 Comparing Index Layouts

Table 3.2 summarizes some of the key properties of the different index layouts, and what operations they can efficiently support. Document-ordered indexes are favorable when both Boolean and ranked queries are required, as the ordering of documents within the postings lists facilitates efficient *Document-at-a-Time* (DAAT) traversal, outlined in the following section. In addition, these indexes allow efficient updates by appending new documents to the tail of the relevant postings lists. Document-ordered indexes are better suited to block-based compression, as this

allows different parts of each postings list to be decompressed on-demand, which is important to facilitate efficient traversal of such postings. While document-ordered indexes generally store term frequency data, they can be modified to store impact scores.

On the other hand, frequency- and impact-ordered postings are primarily equipped to support ranked retrieval only, as there is no efficient way of finding a document with a given identifier. As such, these layouts are suited only for *Term-at-a-Time* (TAAT) or *Score-at-a-Time* (SAAT) traversal, respectively. Updating these indexes is not trivial, as an update may require a value to be added to the middle of a postings segment. Since document segments are of arbitrary lengths, the compression schemes used to compress them should be able to handle any variable length partition. The key difference between frequency- and impact-ordered postings is the payload — frequency-ordered indexes store the term frequencies, while impact-ordered indexes store pre-computed impact scores. Hence, impact-ordered indexes cannot change the underlying ranking function (or parameters of the function) without re-computing all impact scores. Thus, from a flexibility standpoint, document-ordered indexes are likely to provide a greater advantage than frequency-ordered indexes, which are in turn more flexible than impact-ordered indexes.

While there is no modern comparison on the resultant sizes of the indexes generated by document-ordered and frequency- or impact-ordered postings, they would be expected to use around the same amount of space, as the statistics stored within either index format are relatively similar. We explore this further in Chapter 4.

3.2 Query Processing

With potentially tens-of-thousands of incoming queries per second, query processing systems must be efficient and scalable. In this section, we elaborate on a number of strategies that can be used for efficiently processing *disjunctive* top-k queries on both the document- and impact-ordered index layouts described previously.

3.2.1 Rank Safe Processing

Rank safety is an important concept when discussing efficient ranked retrieval, as some traversal techniques may sacrifice accuracy for improvements in efficiency. Rank safety refers to the *degree of similarity* of the results generated by some candidate algorithm as compared to the results list achieved under an exhaustive evaluation. The rank safety of a given ranked retrieval process can be broadly categorized into three main outcomes, namely *rank safe*, *set safe*, and *unsafe*. Given *any* arbitrary query and the top-k ranked list that arises from exhaustively processing it, r_A, a candidate top-k algorithm that generates a corresponding ranked list r_B can be characterized as:

- Rank safe: if r_A is identical to r_B,
- Set safe: if the *set* of documents in r_A is identical to the set of documents in r_B, or
- Unsafe: if there exists some document d' in r_A but not r_B, or vice versa.

3.2.2 TERM-AT-A-TIME

Preliminaries and Naïve Algorithm. *Term-at-a-Time* (TAAT) index traversal involves processing the postings list for a single term before considering any other query terms. As such, TAAT algorithms can be deployed on any type of postings list, since they do not require the postings to be in sorted order. Since document scores are not entirely computed at once, an additional data structure known as an *accumulator table* must be used to maintain the *partial* document scores during processing. The accumulator table must efficiently support accessing and updating the score for *any* arbitrary document identifier, and is usually implemented using a hashtable or array-based structure [7, 120]. In addition, a min-heap containing k elements can be used to maintain the set of highest scoring documents at any time during processing. TAAT processing begins by taking a candidate postings list. For each document in the postings list, the impact score is computed, and the value of the corresponding accumulator updated. This process is repeated for all terms in the query.

Optimizations. Improving the efficiency of TAAT traversal usually involves what is known as *early termination*, where a candidate postings list will not be processed in its entirety to speed up query processing. To facilitate effective early termination, most TAAT algorithms opt to process postings in *increasing* order of their document frequencies, meaning that the highest impact (and generally shorter) postings lists are processed first. We now briefly outline a few fundamental optimizations that have been proposed for TAAT search.

In seminal work, Turtle and Flood [248] noted that evaluating *all* postings exhaustively for a given query may result in poor efficiency, and examined a range of optimizations that could be applied to top-k query processing over inverted indexes. In this work, they proposed an early version of the TAAT MaxScore algorithm, which relies on using *termwise upper-bound scores* to avoid scoring documents which are unable to enter the top-k heap. In order to facilitate this operation, a few important values must be pre-computed or tracked during query processing: firstly, the current heap entry threshold, denoted θ, must be maintained during query processing. This value represents the kth highest document score that has been observed, and can be used to govern whether a document will enter the top-k heap or not. Secondly, the largest value of the ranking function, $\mathcal{W}$, must be computed and stored for *each* postings list in the index. We denote this upper-bound score for a given term t as U_t.

The TAAT MaxScore approach works as follows. During processing, the kth highest scoring accumulator is tracked (θ), and at the end of processing each term, this value is compared against the sum of the upper-bounds of the terms that are yet to be scored. That is, assuming the set of terms that are yet to be processed is denoted $\hat{q}$, TAAT MaxScore checks whether

$$\sum_{t \in \hat{q}} U_t < \theta.$$

If this relation is true, then it means that any document with a zero score cannot make it into the top-k results list. Instead of terminating the processing at this time, TAAT MaxScore will continue

to fully evaluate each document with a non-zero accumulator value, hence returning the rank safe top-k list. This optimization was actually first proposed by Smeaton and van Rijsbergen [230] and further refined by Perry and Willett [193], but their approaches terminated processing once this relation held, rather than continuing processing, resulting in unsafe results lists. A secondary optimization is also possible, where accumulators can be pruned. For example, when scoring a document d containing a partial score of $\mathcal{S}'_d$, MaxScore can check whether

$$\mathcal{S}'_d + \sum_{t \in \hat{q}} U_t < \theta.$$

If this relation is true, then document d is unable to make the top-k results, and the corresponding accumulator can be removed from the accumulator table. Recent work from Fontoura et al. [94] indicated that this accumulator pruning optimization incurs a greater cost than simply leaving them in place when considering TAAT MaxScore over memory-resident indexes.

Buckley and Lewit [37] explored a similar idea, where the scores for the top $k+1$ documents were tracked. After a postings list was traversed, the $k+1$th partial document score is summed with the remaining upper-bound scores, and if this sum is lower than the kth document score alone, then no new documents could enter the top-k heap, and the results can be returned. While this optimization is set safe, it does not often result in efficiency improvements, as the kth and $k+1$th document scores are usually quite close together.

Other optimizations for TAAT traversal focused on limiting or pruning the number of available accumulators, pruning entire postings lists, or partially processing *frequency-ordered* postings lists, which results in efficient index traversal with a loss of rank safety. Despite these optimizations, TAAT traversal has fallen out of favor in recent years as collection sizes continue to grow, and so we do not further consider TAAT traversal in this dissertation. For a more detailed overview on efficient TAAT traversal, we refer the interested reader to the survey from Tonellotto et al. [241].

3.2.3 SCORE-AT-A-TIME

Score-at-a-Time (SAAT) traversal, like TAAT traversal, involves accumulating the partial scores of documents in an accumulator table during processing. However, unlike TAAT traversal, which iterates a single *postings list* at a time, SAAT uses an *impact-ordered* index to traverse a single *postings segment* at a time. Recall that in impact-ordered indexes, each postings list is organized around segments of document identifiers which all share the same quantized impact score. The main idea with SAAT traversal is to simply process these segments in *decreasing* order of their respective impacts. By doing so, the algorithm aims to quickly find the highest impact documents, and then gradually refine the document scores until all postings have been processed.

SAAT traversal was first proposed by Anh et al. [9], who showed that it is a logical extension to the idea of early-termination TAAT over frequency-ordered indexes [194, 195], and to the notion of boosting the efficiency of document evaluation by pre-computing term weights [184, 194].

SAAT algorithms can return rank safe results by processing all segments for the given query. However, very good approximations can be found by processing only a subset of high impact segments, and this is what makes SAAT traversal appealing for large-scale search problems. Note that early termination of SAAT traversal is an *unsafe* optimization, as it does not impose any guarantees on the items in the top-k results list.

Anytime Processing. The most obvious advantage of SAAT traversal is that it is an *anytime* algorithm, which means result quality improves as the computation time increases [272]. Recently, Lin and Trotman [148] proposed a new SAAT search system, called JASS, which was shown to provide competitive effectiveness under various time budgets. To adhere to such time budgets, a *cost model* can be built offline, which is then used to determine how long it takes to process a given number of postings. This model can also determine how many postings can be processed in a given time budget. Internally, JASS keeps track of the number of postings it has processed at any given point. A trade-off parameter, ρ, is used to determine how many postings should be processed. JASS will only process the following segment if the sum of postings scored plus the number of postings contained in the following segment is less than ρ. Hence, JASS checks this stopping criteria after processing each segment, which allows it to effectively bound the time of processing.

For the following description of SAAT traversal, it is convenient to view processing a postings segment in terms of an *iterator* or *cursor* over the segment. As such, we now define a few operators that are supported by the cursor.

- cursor.*length*( ) returns the number of documents in the segment.
- cursor.*impact*( ) returns the $i_{d,t}$, the impact score for the documents in the segment.
- cursor.*getdocuments*( ) decompresses and returns the list of document identifiers which are contained in the segment.

Algorithm 3.1 describes the SAAT retrieval process. Firstly, the segments corresponding to each query term are collected, and the cursors stored in an array $\mathcal{P}$. In addition, an accumulator table must be initialized, with each accumulator set to 0. Although different data structures can be used for storing accumulators, we assume there exists a single accumulator for *each* unique document. A min-heap of size k can be built which *points* into the accumulator table to track the current top-k results. Before query processing begins, the SAAT algorithm sorts the array of segments by their impact in decreasing order by calling the *SortByImpact*($\mathcal{P}$) procedure (line 3), which breaks ties by putting *shorter* segments first. The algorithm begins by looping over each segment in decreasing impact order. First, the PostingsProcessed counter is checked to decide whether processing should be terminated (lines 5–7). If the algorithm is not terminated, processing continues on line 8, where the documents in the segment are decompressed and returned in an array, ready for processing. The Impact value of the segment is retrieved on line 9. The loop on line 10–13 iterates over each document in the segment, and adds the impact score to the

Algorithm 3.1: SAAT processing, based on JASS.

Input : An array $\mathcal{P}$ of n posting metadata cursors, an pre-initialized accumulator table, Accumulators, and the number of postings to process, ρ.
Output : The top-k documents.

```
1  Heap ← {}
2  PostingsProcessed ← 0
3  SortByImpact(P)
4  for i ← 0 to n − 1 do
5      if PostingsProcessed + P[i].length() ≥ ρ then
6          break
7      end
8      DocumentList ← P[i].getdocuments()
9      Impact ← P[i].impact()
10     for DocID in DocumentList do
11         Accumulators.addscore(DocID, Impact)
12         Heap.update(Accumulators, DocID)
13     end
14     PostingsProcessed ← PostingsProcessed + P[i].length()
15 end
16 return Heap
```

relevant accumulator, and updates the heap if required. Once the segment has been processed, the PostingsProcessed counter is updated (line 14), and the algorithm continues to the next segment. Once a sufficient number of postings have been processed, the outer loop will be terminated, and the heap containing the top-k results can be returned (line 16). Figure 3.5 shows a conceptual example of SAAT processing over a toy collection, which visualizes these operations.

Additional Optimizations. While simple and efficient, SAAT traversal is a relatively under-explored approach for top-k search. One potential reason is that impact-ordered indexes are quite inflexible, due to the way they structure the documents within. Nevertheless, there has been some interest in SAAT search over the recent years. Lin and Trotman [149] explored the effect that different compression schemes had on SAAT traversal, finding that leaving the segments *uncompressed* resulted in the fastest retrieval efficiency, but is impractical for real situations due to high space consumption. Elbagoury et al. [90] proposed a new retrieval paradigm called *rank-at-a-time* which is based on impact-ordered indexes and SAAT search. They show that score computations for SAAT retrieval can be conducted by computing Boolean intersections across impact segments in descending order. While an interesting idea, the combinatorial nature of the problem makes it impractical for large collections or long queries. Trotman and Crane [243] examined various engineering considerations that must be accounted for when creating SAAT

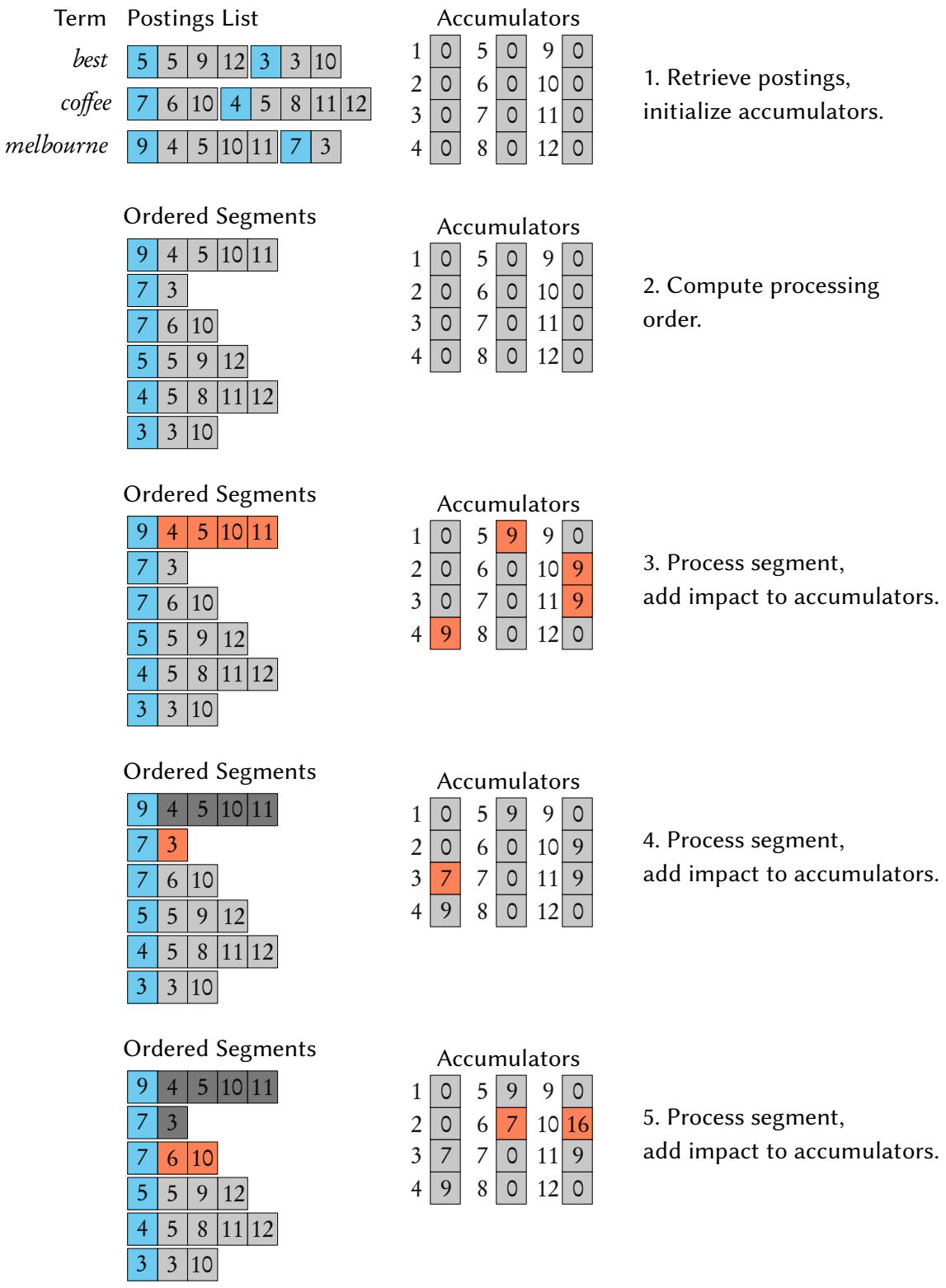


Figure 3.5: An example of Score-at-a-Time processing. This example shows the initialization of the SAAT traversal, and the scoring of three postings segments. Note that a top-k heap that points into the accumulator table may also be maintained, but is not shown for simplicity.

retrieval systems on modern architectures, showing large improvements over the baseline JASS system from careful design choices and improvements.

3.2.4 DOCUMENT-AT-A-TIME

Naïve Algorithm. *Document-at-a-Time* (DAAT) index traversal utilizes a document-ordered index to process a single document at a time before advancing to the next document. Since the underlying postings lists are ordered by their document identifiers, DAAT retrieval also supports efficient Boolean matching. In the following discussion on document-at-a-time processing algorithms, we view a postings list as an *iterator* or *cursor* over the underlying postings list, which can provide a set of efficient operations to facilitate efficient traversal. When a cursor is opened for the first time, it is assumed to point to the first posting in the underlying postings list. We use similar notation to that of Tonellotto et al. [241]. Given a postings list cursor, the operators that are supported are described as follows.

- cursor.*docid*( ) returns the document identifier, d, of the current posting. If the posting list is exhausted, it returns MaxDocID+1, where MaxDocID is the largest document identifier in the index.
- cursor.*score*( ) returns the similarity score for the current document ($\mathcal{W}(d,t)$).
- cursor.*upperbound*( ) returns the highest score observed for the given postings list (U_t). This value is computed during indexing.
- cursor.*next*( ) moves the iterator forward to the next posting in the list.
- cursor.*nextGEQ*(d) moves the iterator forward to the next document that is greater than or equal to d.

To find the top-k documents for a given input query, each postings list is located. Then, a *cursor* is opened on each list. The smallest document identifier currently being pointed to by a cursor is considered the *pivot document*, which will be scored. Then, for each postings list which contains the pivot, the score for the document is computed. Next, the similarity score will be checked against a min heap of size k — if the score of the pivot document exceeds θ, then the lowest scoring document is removed, the pivot document is added, and the heap updated to maintain the min heap property. Otherwise, the heap is not modified. Finally, each cursor that was on the pivot document will be *forwarded* to the next document, and the process repeated until there are no candidate postings remaining.

MaxScore. Previously, we discussed a TAAT version of the MaxScore algorithm, described by Turtle and Flood [248]. In this same work, the authors also described a DAAT implementation of MaxScore, which we discuss now. As was the case in the TAAT version, DAAT MaxScore requires the current heap threshold to be tracked (θ), and relies on pre-computed termwise upper-bounds, denoted U_t. The intuition behind DAAT MaxScore is that, given a sufficiently high heap

threshold, documents appearing only in low impact postings lists will not score highly enough to enter the heap. Thus, the algorithm tries to skip over these documents, resulting in less scoring operations, and faster traversal. Algorithm 3.2 provides the pseudo-code for MaxScore processing, and we now walk through this code.

Before processing begins, the candidate postings cursors are sorted in *ascending* order of their U_t values, and held in array $\mathcal{P}$.[2] The order of the postings lists in $\mathcal{P}$ is fixed for the remainder of processing, such that the lists are always sorted from lowest to highest impact, based on U_t. During processing, $\mathcal{P}$ is logically partitioned into two parts, indexed by a variable called PivotList. All lists on the *left* of this variable represent the *non-essential* lists, and lists on the right (inclusive) represent the *essential* lists. Any document that occurs in only the *non-essential* lists cannot mathematically make it into the top-k heap, so these lists are ignored when finding a candidate document for scoring.

First, the *cumulative upper-bound*, C_t, is computed for each successive term, which involves computing the sum of the U_t values up to the current list, inclusive, and storing it (line 4–7). To process documents, MaxScore will only select documents from the essential lists as pivots, allowing some documents to be skipped. Once a pivot document is found, it is scored for each term in the *essential* lists, at which point a *partial score*, $\mathcal{S}'$, arises (lines 13–21). Next, MaxScore continues by processing the non-essential lists. For each non-essential list, from the highest to lowest impact (line 22), MaxScore checks whether

$$\mathcal{S}' + C_t > \theta,$$

where C_t corresponds to the current lists C_t value (line 23). If this relation does not hold, then the current document is not able to make it into the top-k results, and processing is continued for the next document (line 24). Otherwise, the pivot document is seeked in the current list (line 26), and if found, the score computed (line 27–29). This loop will continue until either the document cannot make the heap, or the document is fully scored. In either case, the document will be added to the heap if its score exceeds θ (line 32). When a new document enters the top-k heap, and the value of θ increases, the essential and non-essential lists are re-computed (line 35–38). To re-compute the essential and non-essential lists, MaxScore finds the first list in which $C_t > \theta$. Processing continues by processing the next candidate, until there are no suitable candidates remaining.

Figure 3.6 shows an example of MaxScore processing, where document 8 is the next document to be scored, as it is the smallest document in the essential lists (*"coffee"* and *"melbourne"*). In the first pane, document 8 is scored across the essential lists, yielding a *partial score* of 2. This corresponds to lines 13–21 in Algorithm 3.2. The second pane shows the cursor being moved forward onto the next document. In the third pane, document 8 is being scored for the non-essential lists. Since the partial score summed with the cumulative upper-bound of the non-essential lists ($2+3=5$) is less than θ, document 8 can not make the top-k heap, and scoring is stopped (line 23 in Algorithm 3.2). The fourth pane shows the state of processing prior to the next iteration of the

[2] Some implementations may sort by the length of the postings list, or the IDF value.

Algorithm 3.2: DAAT MaxScore processing.

Input : An array $\mathcal{P}$ of n postings cursors which are sorted increasing on their upper-bound values, the largest document identifier, MaxDocID, and the number of desired results, k.

Output : The top-k documents.

```
1  Heap ← {}
2  θ ← 0
3  CumulativeBounds ← {}
4  CumulativeBounds[0] ← P[0].upperbound()
5  for i ← 1 to n − 1 do
6  |  CumulativeBounds[i] ← CumulativeBounds[i −1] + P[i].upperbound()
7  end
8  PivotList ← 0
9  PivotID ← MinimumDocID(P)
10 while PivotID ≤ MaxDocID and PivotList < n do
11 |  Score ← 0
12 |  NextCandidate ← MaxDocID
13 |  for i ← PivotList to n − 1 do                    // Score essential lists.
14 |  |  if P[i].docid() = PivotID then
15 |  |  |  Score ← Score + P[i].score()
16 |  |  |  P[i].next()
17 |  |  end
18 |  |  if P[i].docid() < NextCandidate then
19 |  |  |  NextCandidate ← P[i].docid()
20 |  |  end
21 |  end
22 |  for i ← PivotList −1 to 0 do // Complete scoring on non-essential lists.
23 |  |  if Score + CumulativeBounds[i] ≤ θ then
24 |  |  |  break
25 |  |  end
26 |  |  P[i].nextGEQ(PivotID)
27 |  |  if P[i].docid () = PivotID then
28 |  |  |  Score ← Score + P[i].score()
29 |  |  end
30 |  end
31 |  CurrentBound ← θ
32 |  Heap.push(⟨PivotID, Score⟩)
33 |  θ ← Heap.min()
34 |  if CurrentBound < θ then                  // The heap threshold increased.
35 |  |  while PivotList < n and CumulativeBounds[PivotList] ≤ θ do
36 |  |  |  PivotList ← PivotList +1
37 |  |  end
38 |  end
39 |  PivotID ← NextCandidate
40 end
41 return Heap
```

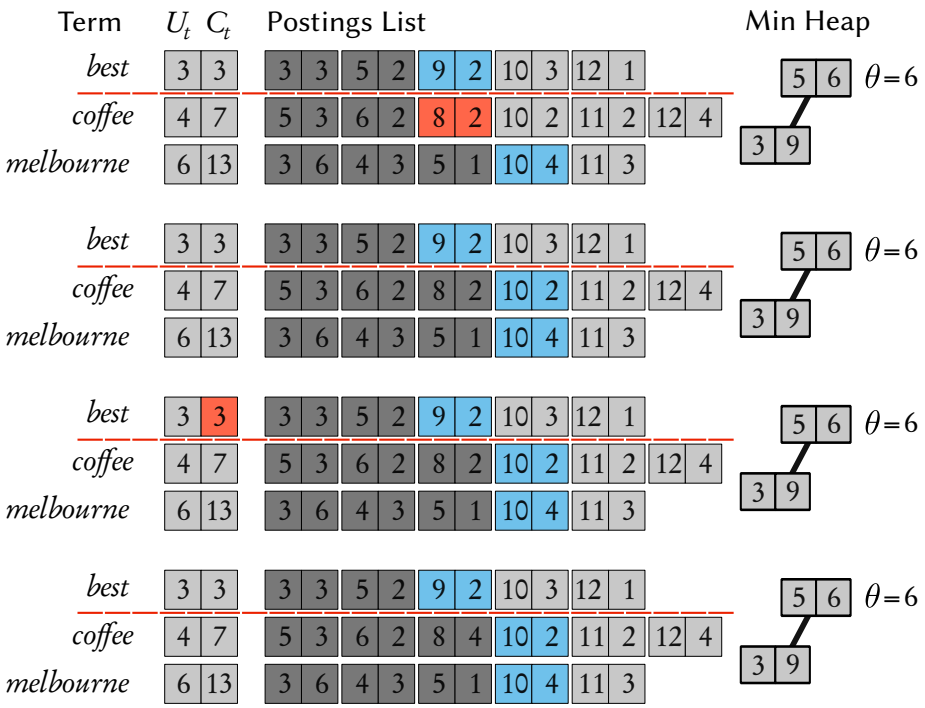


Figure 3.6: An example of MaxScore processing for retrieving the top $k = 2$ documents, where a given pivot document, document 8, is partially scored, but does not make it into the top-k results list. For convenience, each posting is shown as a $\langle d, i_{d,t} \rangle$ pair, and the non-essential lists are separated from the essential lists by red dashed line.

MaxScore algorithm. Figure 3.7 shows the continuation of this example, where document 10 is identified as the new pivot document, as it is the smallest document in the essential lists (*"coffee"* and *"melbourne"*). In the first pane, document 10 is scored across the essential lists, yielding a *partial score* of $4 + 2 = 6$. This corresponds to lines 13–21 in Algorithm 3.2. In the second pane, the corresponding cursors are moved forward. In the third pane, document 10 is being scored for the non-essential lists. Since the partial score summed with the cumulative upper-bound of the non-essential lists ($6 + 3 = 9$) is greater than θ, document 10 must be scored in the non-essential lists (line 22–30). The fourth pane shows document 10 being found and scored in the *"best"* list, resulting in a total score of 9 (line 27–29). The fifth pane shows the heap being updated (line 32), and the sixth pane shows the essential and non-essential lists being re-computed (line 34–38).

While slight variations in the DAAT MaxScore approach have been described in the literature, our description of MaxScore is best aligned with the *in-memory* versions, as discussed by Fontoura et al. [94] and Jonassen and Bratsberg [123]. These particular implementations of MaxScore are tailored for fast in-memory index traversal, whereas the original implementation was focused on the reduction of expensive disk seeks [248].

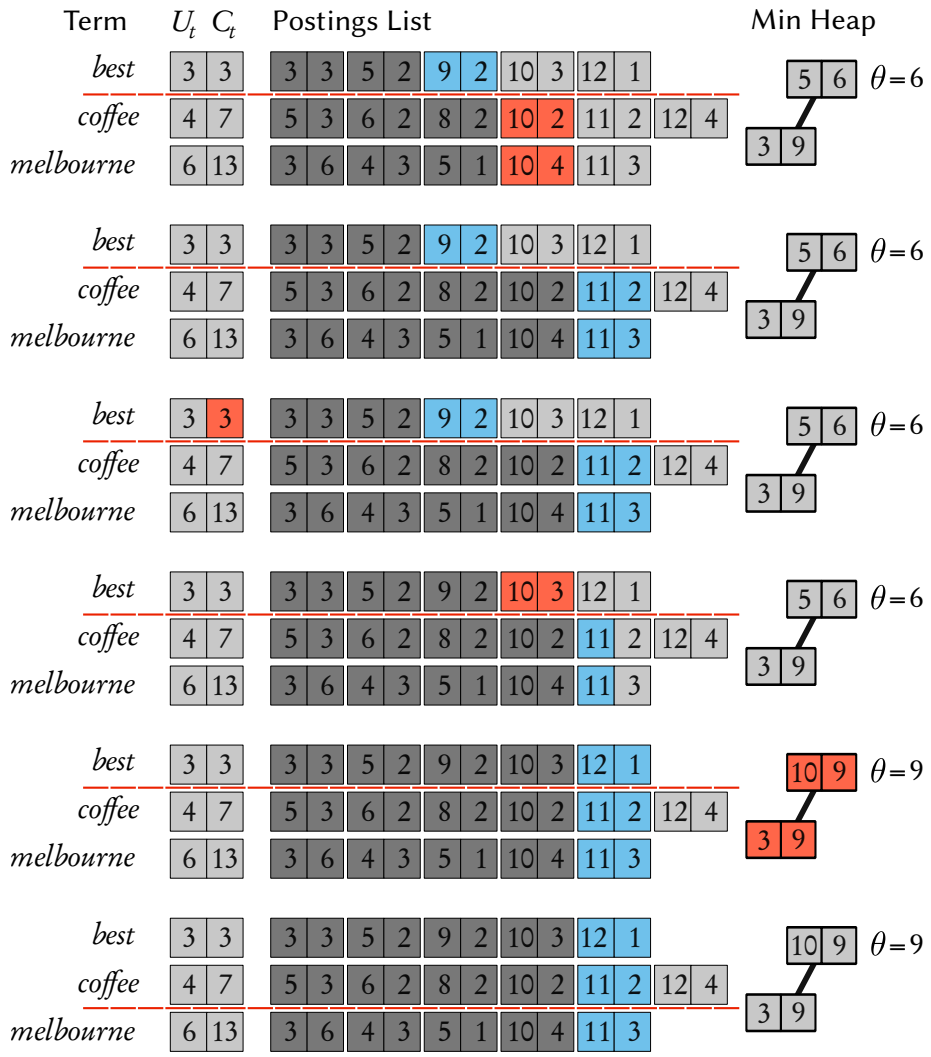


Figure 3.7: A continuation of the example in Figure 3.6, where document 10 is identified as the pivot document. Document 10 is then fully scored, and added to the heap. Finally, the essential lists are re-computed. For convenience, each posting is shown as a $\langle d, i_{d,t} \rangle$ pair, and the non-essential lists are separated from the essential lists by red dashed line.

Weighted AND. An alternative to MaxScore, based on similar principles, was first proposed by Broder et al. [36] and is known as the *weak* or *weighted* AND (WAND) algorithm.

WAND was originally described as an operator which can be applied on a single document at a time as follows. Given a list of Boolean variables $X_1, X_2, \ldots, X_n$ with associated positive weights $w_1, w_2, \ldots, w_n$ and a threshold $\hat{\theta}$:

$$\text{WAND}(X_1, w_1, X_2, w_2, \ldots, X_n, w_n, \hat{\theta}) = \textit{true} \iff \sum_{i=1}^{n} x_i w_i \geq \hat{\theta}, \tag{3.1}$$

where x_i is the indicator variable for X_i such that

$$x_i = \begin{cases} 1, \text{ if } X_i \text{ is } \textit{true} \\ 0, \text{ otherwise.} \end{cases}$$

In practice, the Boolean variables $X_1, X_2, \ldots, X_n$ correspond to the n terms of a given query, and the indicator variables $x_1, x_2, \ldots, x_n$ correspond to whether each term is contained in the given document. This operator provides the basis for the name WAND, as it can be used to implement both conjunction (AND) and disjunction (OR):

$$\text{AND}(X_1, X_2, \ldots, X_n) \equiv \text{WAND}(X_1, 1, X_2, 1, \ldots, X_n, 1, n),$$

$$\text{OR}(X_1, X_2, \ldots, X_n) \equiv \text{WAND}(X_1, 1, X_2, 1, \ldots, X_n, 1, 1).$$

As a further example of the 'weak' nature of the operator, consider the following instance, which matches any document containing at least $n-1$ query terms:

$$\text{WAND}(X_1, 1, X_2, 1, \ldots, X_n, 1, n-1).$$

Generalizing the WAND operator further to *ranked* queries, the weight variables $w_1, w_2, \ldots, w_n$ can be substituted with the pre-computed list upper-bound scores, (U_t), and the threshold $\hat{\theta}$ set to the score of the kth highest scoring document encountered during processing, θ. Now, prior to scoring any arbitrary document d, we first ensure that $\text{WAND}(X_1, U_1, X_2, U_2, \ldots, X_n, U_n, \hat{\theta})$ evaluates to *true*. The intuition behind this check is that only documents that can *potentially* score higher than the heap threshold will be scored, and all other documents will be ignored. To use WAND for DAAT traversal, the U_t value for each list must be pre-computed and stored, and the heap threshold, θ, maintained during query processing. Algorithm 3.3 shows the pseudo-code for WAND processing, as is referred to in the following walk-through.

Unlike MaxScore, which sorts the postings in ascending order of the U_t values (and keeps them sorted in this way for the remainder of processing), WAND sorts the postings such that they are in ascending order of the *current document identifier* under each cursor (line 3). Next, WAND starts working down through the candidate postings lists, computing the sum of the term upper-bound scores until this sum exceeds the value of θ (line 8–18). At this point, the document

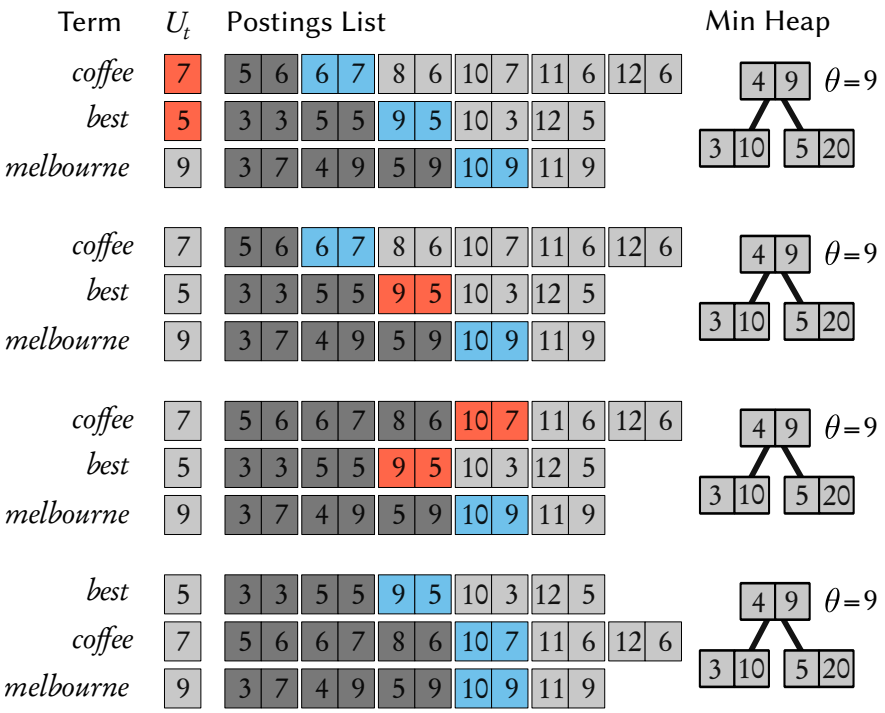


Figure 3.8: An example of WAND processing for retrieving the top $k = 3$ documents, where document 9 is identified as the pivot. Before scoring document 9, WAND tries to seek to it in the first postings list, but finds document 10 instead. For convenience, each posting is shown as a $\langle d, i_{d,t} \rangle$ pair.

identifier under this list is set as the pivot document, and the PivotID recorded (line 11). If no pivot document is found, the algorithm is terminated (line 19–21). Otherwise, WAND checks to ensure that all lists prior to the PivotList point to the PivotID (line 22). If so, the document will be scored, added to the heap, and the heap updated (lines 22–31). Finally, the cursors are sorted by their current identifiers (line 30). If the check at line 22 fails, it means that the current identifier of the first cursor is not pointing to the pivot, so the cursors prior to the PivotList are *forwarded* to the PivotID (line 36), and the order of the cursors is restored by *bubbling* each list downwards to its correct position.

Figure 3.8 shows an example of WAND processing. The top pane shows how the pivot list is found by summing the upper-bounds until θ is exceeded. The second pane shows the pivot document being selected, which is document 9. Since the document in the first list is 6, WAND forwards this list to the pivot using the *nextGEQ*( ) function, as shown in the third pane. In the final pane, the result of the *BubbleDown*( ) operation is shown. Note that document 6 was not evaluated, showing how WAND can skip redundant computation for faster index traversal.

Algorithm 3.3: WAND processing.

Input : An array $\mathcal{P}$ of n postings cursors, the largest document identifier, MaxDocID, and the number of desired results, k.
Output : The top-k documents.

```
1  Heap ← {}
2  θ ← 0
3  SortByDocID(P)
4  while true do
5      UpperBound ← 0
6      PivotList ← 0
7      PivotID ← MaxDocID
8      while PivotList < n do        // Find the pivot list and pivot document.
9          UpperBound ← UpperBound + P[PivotList].upperbound( )
10         if UpperBound > θ then
11             PivotID ← P[PivotList].docid( )
12             while PivotList+1 < n and P[PivotList+1].docid( ) = PivotID do
13                 PivotList ← PivotList + 1
14             end
15             break
16         end
17         PivotList ← PivotList + 1
18     end
19     if UpperBound ≤ θ or PivotID = MaxDocID then          // No pivot. Exit.
20         break
21     end
22     if P[0].docid( ) = PivotID then        // Can evaluate the pivot document.
23         Score ← 0
24         for List ← 0 to PivotList do
25             Score ← Score + P[List].score( )
26             P[List].next( )
27         end
28         Heap.push(⟨PivotID, Score⟩)
29         θ ← Heap.min( )
30         SortByDocID(P)
31     end
32     else                                  // Need to align lists with the pivot.
33         while P[PivotList] = PivotID do
34             PivotList ← PivotList − 1
35         end
36         P[PivotList].nextGEQ(PivotID)
37         BubbleDown(P, PivotList)
38     end
39 end
40 return Heap
```

Block Max Variants. More recently, a line of work has explored how *block-based* postings can be exploited to improve the WAND and MaxScore algorithms. In particular, document-ordered postings lists are generally compressed in fixed-sized blocks, as accessing a given element will then require just the corresponding block to be decompressed, rather than requiring the entire postings list. At roughly the same time, Ding and Suel [88] and Chakrabarti et al. [53] observed that these blocks could be augmented with *blockwise upper-bound* information, which would allow faster top-k processing. In particular, the largest value of the scoring function, $\mathcal{W}$, is stored for each *block* in each postings list, denoted as $U_{b,t}$, as well as the termwise upper-bound, U_t.

Ding and Suel [88] showed how these block-wise upper bounds can be used to improve the efficiency of WAND in what they call *block max* WAND (BMW). BMW processing begins in the same way as WAND — a pivot document is found by summing the listwise upper-bounds, U_t, until the value of θ is exceeded. However, instead of forwarding the cursors of the prior lists up to the pivot document, BMW instead conducts a *shallow* move, which involves forwarding the cursors to the *block* containing the pivot. Before decompressing the relevant blocks, BMW checks that the sum of the blockwise upper-bounds, $U_{b,t}$, also exceeds the value of θ. Since this refined check is much more accurate than the global bound estimate, it results in fewer index decompressions, and fewer scoring operations. A second optimization is possible if the blockwise upper-bound scores are not high enough to exceed θ — this means that *no document* in the current block configuration is able to enter the heap, and so the next pivot document must be outside of this configuration, resulting in larger skips. Chakrabarti et al. [53] used these same ideas to optimize MaxScore traversal, arriving at *block max* MaxScore.

Mallia et al. [173] recently revisited the BMW algorithm, and showed that *variable-sized blocks* can further improve both the time and space trade-offs present in the BMW algorithm, resulting in what is now the state-of-the-art for efficient top-k retrieval, the VBMW algorithm. In the case of fixed-sized blocks, the $U_{b,t}$ score can be inaccurate, especially if only one single document within the entire block has a much higher score than any other document. Using variable-sized blocks allows for more accurate upper-bound information to be stored, as the effect of such outliers can be minimized. However, computing the block partitions involves offline pre-processing, making the variable-block indexes more expensive to build.

The trade-off with block max methods is that additional storage space is required to store each $U_{b,t}$ value, and the subsequent processing algorithms are more complicated. However, these changes have been shown to lead to significant efficiency improvements [53, 88, 173].

Aggressive Traversal. So far, the DAAT dynamic pruning algorithms that have been described all return the rank safe top-k document lists. However, some situations may arise where faster retrieval is required. One way to increase the efficiency of these DAAT dynamic pruning algorithms is to artificially increase the value of θ to induce more pruning. Hence, *aggressive* versions of the prior algorithms can be instantiated by setting $\hat{\theta} = F \cdot \theta$, where $F \geq 1.0$ [36]. However, doing so results in *unsafe* processing, where there are no guarantees on the correctness of the documents in the top-k results. Macdonald et al. [163] examined how the value of F impacts the

efficiency and effectiveness of using WAND as a candidate generation stage in a LtR system, and demonstrated that as F increases, the traversal is biased to returning documents with smaller identifiers, as skipping increases rapidly with the heap threshold.

Threshold Estimation. Another line of work for improving dynamic pruning strategies involves estimating the value of θ *before* processing begins. As demonstrated by Petri et al. [196], many documents are scored in the early stages of DAAT dynamic pruning algorithms, as the low heap threshold invokes full scoring operations on most documents. As the value of θ tends towards its final value, much more skipping is observed. Hence, if an accurate estimation of the final value of θ can be predicted before processing, skipping will be maximized. Deterministic approaches, such as computing and storing the kth highest impact for each term have been explored. In this situation, the *maximum* of these scores across the candidate query terms can be used to safely *prime* the heap threshold [80, 127], reducing latency by up to 17%. Other work explores how predictive models can be used to estimate the value of θ, and the consequences of over estimation [197]. For example, an over-prediction of θ results in unsafe results lists, so a second-stage *patching* phase may be required to find results that were missed in the first pass. This prediction mechanism was shown to give improvements of up to 12% when compared to the best deterministic baseline.

3.2.5 Comparing Index Traversal Approaches

Despite the large range of index organizations and retrieval algorithms that have been enumerated in the previous discussion very few studies have compared them directly. Here, we outline some of the main comparisons that have been reported, and comment on some of the gaps in knowledge that are yet to be filled from these comparisons.

On-Disk Retrieval. Carmel and Amitay [44] conducted a small study between TAAT and DAAT retrieval strategies in their 2006 TREC Terabyte track submission, showing that WAND [36] traversal was more efficient, effective, and scalable than early termination TAAT methods [194, 195] across the Gov2 collection. Lacour et al. [141] reviewed a number of retrieval algorithms with respect to efficiency and effectiveness on inverted indexes, paying close attention to both DAAT and TAAT methods [30, 182, 248]. They found that heuristic retrieval approaches generally performed worse than exhaustive traversal with respect to deep metrics such as AP, but were much faster. However, these studies focused on postings located on-disk. With the reduced price of hardware and increases in RAM size, inverted indexes can now be stored entirely in main memory, resulting in a range of improvements for index traversal algorithms.

In-Memory Retrieval. Büttcher and Clarke [41] presented the first study in which an inverted index was hosted in main memory. They employed *static index pruning* to reduce the size of the index such that it could fit within main memory, and examined the efficiency improvements therein. Motivated by this work, Strohman and Croft [234] presented a deeper analysis of in-memory index traversal. They found that augmenting inverted indexes with skip information

allowed extremely efficient search while also maintaining the property of rank safety. Fontoura et al. [94] conducted a large in-memory comparison of contemporary index traversal methods, examining the naïve DAAT, WAND [36], naïve TAAT, Buckley and Lewit [37], TAAT and DAAT implementations of MaxScore [248], and a range of *hybrid* index traversal approaches, all of which were deployed on document-ordered inverted indexes. Perhaps the most important contribution of this work is how the authors optimize the above implementations for in-memory traversal, showing improvements of up to 60% compared to the original approaches which were all optimized to minimize disk latency.

Ding and Suel [88] were the first to show the benefits possible from the *block max* index, with BMW outperforming WAND and other competitors by a large margin with a space overhead of around 5%. Most recently, Mallia et al. [175] examined state-of-the-art DAAT retrieval methods, including MaxScore [248], WAND [36], BMW [88], and VBMW [173]. In particular, these methods were compared for in-memory retrieval on a number of different collections, compression schemes, and document identifier assignment methods. While the VBMW algorithm proved to be extremely efficient for short queries and small values of k, the MaxScore algorithm proved to be the most efficient processing algorithm for long queries, or where the number of required candidates is large. Thus, the choice of traversal algorithm is not entirely clear cut, as it may depend on the use-case of the search system. Furthermore, there has been no comparison between SAAT systems in modern processing architectures. We examine these ideas further in this thesis.

Selecting a Ranking Model. Previously, we outlined a number of different ways in which documents can be scored in an additive unigram bag-of-words framework. In particular, we focused on *vector space*, *probabilistic*, and *language* models. While each method uses a different formulation to score a document, they all rely on summing per-term score contributions, computed using similar statistics. Thus, a natural question that arises is which of these formulations should be used for ranking documents?

Concerning the effectiveness of these BOW models, some studies have been undertaken. For example, Bennett et al. [28] studied the differences between language models and probabilistic models, indicating that the most effective model depended on the query task. A similar study from Trotman et al. [247] concluded that there is relatively little difference between these models if the parameters are selected appropriately. Recently, Lin [147] examined the use of various rankers as baselines in IR experimentation, finding that BM25 was slightly better than their language model implementation following an exhaustive parameter sweep, further justifying the need for good parameter selection.

Another important consideration is how expensive the ranking function is, and the impact it has on the efficiency of index traversal. We ran a small benchmark which evaluates the efficiency of the three candidate rankers described previously, namely BM25, DPH, and LMDS, to determine if there is any notable cost difference. As a comparison point, we also examined the cost of simply accessing and returning the document term frequency multiplied by the query term frequency, $f_{d,t} \cdot f_{q,t}$, which we denote as Basic. The benchmark was implemented in C++, and ran on a modern server with an Intel Xeon E5-2690 v3 CPU. Data was randomly generated and stored

	Number of Query Terms									
Ranker	1	2	3	4	5	6	7	8	9	10
Basic	14.8	15.6	18.1	18.8	20.8	22.8	24.9	26.6	28.7	31.0
BM25	18.7	18.5	20.8	23.4	25.5	28.7	32.9	35.8	37.9	40.5
LMDS	69.6	83.6	108.4	126.9	144.6	180.5	187.4	206.6	227.5	253.7
DPH	45.8	71.0	98.6	122.7	154.9	182.0	214.3	245.5	266.7	295.6

Table 3.3: The time required, in nanoseconds, for a single document to be evaluated using the given ranker. BM25 is more scalable as query length increases, as it only needs to compute the IDF value *once* for each query term.

in memory, and the average time taken across 1 million scoring operations is presented for each ranker. To test the scalability of the rankers, we conduct the experiment for queries of length 1 to 10. Table 3.3 shows the results. Interestingly, the BM25 ranker can be implemented much more efficiently in practice than either the DPH or LMDS models. The main optimization that is available to BM25 (but not the other models) can be observed in Equation 2.3. In particular, the entire IDF component for BM25 can be computed just *once* and cached for all subsequent scoring operations, reducing the number of expensive log and floating point operations.

Petri et al. [196] explored the impact that the underlying ranking function has on the pruning effectiveness of both WAND and BMW, noting that most prior studies assumed the BM25 ranker. They found that the performance of the WAND algorithm, in terms of the number of computations that were skipped, was greatly reduced when using the LMDS ranker (as opposed to BM25). Interestingly, these findings did not carry over to BMW — using block max scores resulted in a substantial reduction of the number of documents that were required to be scored, thus confirming the utility of the block max index organization. The main reason for this observation appeared to stem from the underlying score distributions generated by each ranker. That is, BM25 generated score distributions which were more favorable for facilitating dynamic pruning.

Hence, for the remainder of this thesis, we use the BM25 model as a representative BOW model, as shown in Equations 2.2—2.4. It has been shown to be effective, cheap to compute, and has good characteristics for dynamic pruning.

3.3 Tail Latency

Bounding tail latency is important for maintaining acceptable response times. Despite the importance of reducing and controlling tail latency in production IR environments, few academic groups have focused on this problem. We now discuss some of the key ideas relating to tail latency, and current literature on reducing tail latency.

3.3.1 Causes of High Percentile Latency

In large and complex IR systems, there can be many sources of latency. For example, network latency, browser latency, and search engine processing latency can all contribute to high response times. Bai et al. [17] showed that at least 40% (but usually closer to 60%) of latency is due to the search engine itself, indicating that search engine query processing is the main factor in user perceived response times. Dean and Barroso [82] point out many contributors of *variability* in processing times for distributed web search systems, including queueing, resource contention, maintenance tasks, load and power management, daemons and other background tasks, among others. These aspects all contribute to increased latency, and may result in high percentile response times. However, to the best of our knowledge, no studies have directly explored the causes of high percentile latency *within* the query processing components of web search engines.

3.3.2 Reducing Tail Latency

While many prior studies focused on reducing the latency of query processing, very few focused directly on tail latency. However, one key idea for reducing tail latency is to enforce a guarantee on high percentile latency by adhering to a *service level agreement* (SLA), as discussed in Section 2.4. Here, we explore some approaches to reducing tail latency, categorizing them by their key ideas.

Predictive Parallelization. One line of work for reducing tail latency in large-scale retrieval systems involves using parallel processing to accelerate computation. Jeon et al. [119] examined this approach in depth, arguing that simply parallelizing all queries would result in wasted resources since most queries have a low latency. On the other hand, the few long running queries that do occur can be rapidly accelerated using multi-core processing. This study shows how the long running queries can be predicted and accelerated effectively using basic features which are available prior to query execution. A follow on study from Kim et al. [129] revisits this problem for *extreme* tail latency (at the 99.99th percentile). The authors propose using *delayed, dynamic, selective* prediction to parallelize the processing of long running queries. In particular, the prediction process is improved by firstly processing the query for a short, fixed period of time. During this time, dynamic features are collected, which help improve the prediction. The advantage of this method is that most of the short running queries will complete their processing during the initial processing period, which improves the precision of the predictor. Finally, predicted long running queries are accelerated using parallel processing.

Reducing Latency by Approximation. Another line of work has involved reducing tail latency by trading off some result effectiveness. These methods therefore exploit different regions of the trade-off space with the aim of providing the best possible experience for the user. Tonellotto et al. [240] explored the use of *selective pruning* to improve latency without significantly reducing search quality. The key idea is to build predictors which can predict inefficient queries, and accelerate them by increasing the aggression of the WAND algorithm. Selective pruning was also

shown to work well in situations where the processing load is high, and can be modified to allow for queueing time as well as processing latency [34]. Lin and Trotman [148] explored a similar idea in the context of SAAT retrieval systems, showing how SAAT search can effectively bound the cost of processing with some loss in effectiveness.

Scheduling and Distributed Processing. One source of added latency in large-scale systems is the accrued idle time when a query is waiting in the processing queue. Queue time is directly proportional to the system load — a system under extreme load will have a longer queue than a mostly idle system. Some early work examined a few different scheduling approaches and how they impact both queueing and processing time over distributed IR systems [162]. However, this work did not examine tail latency directly. Yun et al. [265] examined the problem of selecting an aggregation policy for distributed IR engines. In particular, they assume an input query will be broadcast to a number of ISNs, who will return their candidate results for aggregation. Hence, the processing latency will be bounded by the *slowest* ISN, which can result in a high tail latency. Using production logs from the Bing search engine, they showed that tail latency can be reduced by up to 40% without negatively impacting the utility of the search engine. A similar work focused on selecting *reissue* policies, where an input query is forwarded to a set of replicas to reduce the likelihood of a high percentile response [125].

3.4 Summary and Open Directions

A large amount of work has focused on improving the efficiency and scalability of search engines, and this chapter examined some important contributions to this area of research, including efficient index representations, efficient query processing, and reducing tail latency. In Section 3.1, we introduced the process of indexing a text collection to enable efficient retrieval. A few common approaches for representing documents within inverted indexes were examined in Section 3.1.2, including document-ordered, frequency-ordered, and impact-ordered postings lists. We discussed some of the trade-offs between these index formats in Section 3.1.5, with a focus on flexibility. Document-ordered indexes were shown to offer the most flexibility, as they allow different modes of querying, and are easy to update. Next, we examined different query processing approaches, and their affinities to the various index organizations (Section 3.2). We first introduced TAAT and SAAT traversal, and subsequent *early termination* optimizations. We then discussed DAAT traversal, and showed how *dynamic pruning* can accelerate query processing. Section 3.2.5 outlined a number of prior studies that compare the vast number of index traversal strategies, and comment on the current state-of-the-art for top-k retrieval. Finally, we examined the current knowledge on *tail latency* (Section 3.3) with a focus on different techniques for reducing tail latency.

There remain a number of open directions when considering the efficiency of search systems. Firstly, there has been a large focus on both TAAT and DAAT search algorithms in the past. Once, TAAT algorithms were considered superior, but have since been superseded by DAAT methods. However, very little attention has been paid to SAAT retrieval. In the modern multi-stage retrieval framework, where fast but approximate results are suitable for candidate generation stages of

the end-to-end search pipeline, SAAT retrieval approaches may be extremely competitive. We explore the benefits and problems associated with SAAT retrieval in Chapter 4, and leverage our findings further in Chapters 5 and 6. Secondly, despite the large focus on DAAT methods in prior studies, there is still very little understanding of their performance profiles. This problem stems from most prior efficiency works examining *mean* latency, and reporting results using summary statistics. However, the notion of tail latency is increasingly important for production search systems, and thus, more focus should be placed on understanding and mitigating tail latency. We examine DAAT systems further in Chapter 4, and provide a more nuanced view on their strengths and weaknesses. Thirdly, most prior works examine idealistic cases, where input queries are short, and only top-k BOW retrieval is considered. On the other hand, in real systems, queries may be rewritten into more advanced representations, possibly containing many terms. Chapter 5 proposes one such scenario, where an input query is associated with a number of *related* queries, which are then leveraged to improve the effectiveness of retrieval. We design and evaluate a number of processing algorithms that are suitable for processing this novel query type. A final open direction, which is currently under-explored, is the notion of trade-offs. Indeed, improving efficiency may require shortcuts which reduce the underlying effectiveness of search. As such, it is important to have tools and processes that allow such trade-offs to be quantified. Chapter 6 tackles this problem, aiming to reduce both tail latency and subsequent cost in an end-to-end multi-stage retrieval system, while also providing a framework for evaluating the trade-offs within such systems.

4 Index Organization and Tail Latency

One of the most fundamental problems concerning large-scale search engines is how to efficiently find the top-k documents for a given input query. Much of the prior literature on building efficient information retrieval systems has focused on the various aspects of this problem, including indexing [273], compression [143, 191], and retrieval [36, 88, 248]. In the past, search engines employed simple matching and ranking functionality. For example, some of the first search systems supported only Boolean matching operations without ranking [229]. However, this made it difficult for users to find documents that were relevant to their information need in the increasingly large number of indexed documents. To alleviate this problem, the IR community focused on building systems that could *rank* documents. Many different *bag-of-words* approaches for estimating document-query similarity were proposed, such as the *vector-space model* [216, 217], *probabilistic models* [212], and *language models* [204], which all use the notions of *term frequency*, *inverse document frequency*, and *inverse document length* to rank documents with respect to a given query.

More recently, innovations in machine learning have allowed advanced ranking models to be used within IR systems in what is known as *Learning-to-Rank* (LtR). LtR models use a number of features to accurately rank documents, including static document features, static query features, and more expensive query-document dependent features [164, 165]. The advantage of LtR systems is that they are able to capture relevance signals that simple term-matching approaches cannot. While shown to be much more effective than the simple bag-of-words models discussed previously, these LtR models are not efficient enough to be deployed as a standalone ranker for large-scale search systems, as extracting features for all matched documents is prohibitively expensive. Instead, LtR systems are deployed as a late-stage *re-ranker* in the *multi-stage* retrieval paradigm [57, 252], which involves breaking the end-to-end query processing into a number of distinct stages. First, the *candidate generation* stage retrieves a top-k list of candidate documents that match the input query [13]. Second, the features for each of the returned documents are extracted [12]. Third, the LtR model is applied to these documents, generating a new ranking. After sorting the documents based on this new ranking, the ranked list can be returned to the user.

While document ranking has matured since the days of simple bag-of-words models, the fundamental top-k retrieval problem is still at the core of modern query processing architectures, with efficient candidate generation a key requirement for large-scale IR systems. Indeed, candidate generation is often conducted via top-k bag-of-words query processing, making it as important as ever to ensure efficient top-k search. While a multitude of prior work has focused on efficient top-k retrieval, there are still very few established best practices for search practitioners. For example, Document-at-a-Time (DAAT) and Score-at-a-Time (SAAT) query evaluation represent two fundamentally different approaches to top-k retrieval. These approaches have strong affinities to different inverted index organizations, and thus, have vastly different mechanics. While both methods of processing queries have been studied extensively in the literature, there has been little work comparing the two methods. Thus, it is unclear what the trade-offs between the two approaches are. Furthermore, there is little prior work that focuses on these methods with respect to *tail latency*, which is considered to be as important as mean or median latency in practice [82].

In this chapter, we conduct an empirical comparison between document- and impact-ordered index layouts, and the query processing algorithms therein. We focus primarily on tail latency, but also evaluate the efficiency/effectiveness trade-offs that exist between different query processing algorithms. Furthermore, we examine SAAT retrieval methods more carefully, paying close attention to the trade-offs between the various early termination heuristics that can be applied to SAAT search systems. We also explore how parallel processing can be used within SAAT retrieval to ensure a competitive tail latency without negatively impacting search effectiveness.

Problem Outline. We study the problem of efficient and effective disjunctive top-k retrieval. Given a bag-of-words query, q, we wish to process this query across the given index structure to retrieve the results list r containing the top-k ranked documents with respect to q. The time taken to process q should be minimized while maximizing the effectiveness of r.

Our Contributions. In this chapter, we focus on the following research question:

> **RQ1:** *What are the primary causes of high-percentile latency during top-k retrieval?*

In order to answer this broad research question, we explore the following sub-questions:

> **RQ1.1:** *What are the trade-offs between* DAAT *and* SAAT *retrieval methods?*
>
> **RQ1.2:** *Which early-termination heuristics are suitable for balancing efficiency and effectiveness for* SAAT *retrieval?*
>
> **RQ1.3:** *Can selective parallelization accelerate* SAAT *retrieval without reducing effectiveness?*

To answer these questions, we first conduct an empirical analysis on the top-k candidate generation problem. In particular, we explore both the efficiency and effectiveness trade-offs that are present in generating top-k sets of candidates for two different index organizations: document-ordered, and impact-ordered. We examine the many factors that influence tail latency in candidate generation,

including the size of the candidate set, the size of the collection, the length of the query, the index traversal algorithm, and whether the traversal guarantees rank safety or not (RQ1.1). To make our comparison as fair as possible, we control for a range of external confounders such as the implementation language, the underlying index and how it was normalized, as well as the compression codec. Our major findings from this empirical study can be summarized as follows:

- When considering rank safe query evaluation, DAAT approaches are more efficient than SAAT approaches, as they can leverage rank safe dynamic pruning to speed up query processing. However, the gap in performance is smaller than one might expect, considering that exhaustive SAAT processing examines more than an order of magnitude of *extra* postings than the DAAT methods.

- In both rank safe and approximate query evaluation, SAAT methods exhibit much less variance with respect to latency, and thus have more control over tail latency, than the DAAT methods.

- SAAT processing is much less sensitive to the value of k, since its processing mechanics do not rely on the top-k heap to provide pruning information. The converse is true for the DAAT dynamic pruning methods, which leverage the top-k heap to provide information that allows some query evaluations to be skipped.

- In DAAT systems, the overhead of computing which document evaluations to skip can be slower than simply doing the evaluations, especially when the ranking function is extremely cheap (as is the case for quantized indexes).

- Simply reporting mean running times is not adequate for making recommendations about the efficiency of search systems.

Next, we extend this research to focus on the trade-offs inherent with SAAT index traversal. The empirical study shows that SAAT systems can control tail latency extremely well: SAAT systems achieve this through setting a parameter, ρ, which is used to bound the number of postings processed for each query. We show that this parameter setting can have a negative impact on the effectiveness of the returned results, and explore some alternative heuristics for setting ρ which result in different efficiency/effectiveness trade-offs (RQ1.2). Furthermore, we show how *selective parallelization* [119] can be incorporated within SAAT search engines to reduce tail latency while keeping the system effectiveness high (RQ1.3). Our findings are summarized as follows:

- Using a fixed value of ρ for all queries can result in reduced effectiveness. This problem becomes more pronounced as the quantity of candidate postings increases, as a larger fraction of postings are not processed. This situation can occur for long queries, or queries with low IDF terms.

- We explore how dynamically setting ρ on a per-query basis impacts the efficiency and effectiveness of the search system. In particular, we set ρ on a per-query basis by computing a

fixed *percentage* of postings to process, which can be computed at query-time. We show that the percentage-based settings result in improved effectiveness at the cost of more variance in latency.

- ✦ Intra-query parallelization can be exploited to speed up SAAT query processing. We examine a number of hybrid algorithms that can exploit parallel processing *selectively*, based on the predicted run time of the input query. These methods give a better efficiency/effectiveness trade-off than any of the fixed parameter systems.

We conclude our analysis with a discussion on the findings from the studies presented in this chapter, and future directions in efficient candidate generation.

4.1 Related Work

The majority of the related work on efficient indexing and fast index traversal can be found in Chapter 3. For a detailed description of the mechanics of both DAAT and SAAT retrieval algorithms, we refer the reader to Section 3.2. Section 3.2.5 comments on some empirical comparisons between these algorithms in the literature. Here, we review some recent studies that focus specifically on the problem of *candidate generation* for large-scale search.

4.1.1 Candidate Generation

Overview. The idea of generating a set of candidate documents as input to further re-ranking stages was initially discussed in seminal work on learning-to-rank from Burges et al. [38]. In this work, the authors describe using a real-world data set that, for each query, contains *"up to 1,000 returned documents, namely the top documents returned by another, simple ranker."* While work on improving learning-to-rank frameworks for IR problems continued to gain traction, focus on candidate generation was largely ignored for another five years, until Jan Pedersen's invited talk on *Query Understanding at Bing* at SIGIR, 2010 [192]. During this presentation, Pederson outlined the matching and ranking pipeline used within Bing's production search system, explaining that Bing used *"a sequence of select, rank, and prune steps"* during retrieval.[1] A deeper overview of this architecture was presented in a later paper from Risvik et al. [210], who outline *Maguro*, a core retrieval system used within the Bing search engine. Wang et al. [252] provided the first formal overview of the *cascade ranking model* (otherwise known as multi-stage retrieval), and showed that cascade rankers can achieve a similar effectiveness to monolithic rankers at a fraction of the cost.

Efficient Candidate Generation. More recently, a large amount of work has focused on the efficiency considerations for the candidate generation stage of the cascade ranking pipeline. Asadi and Lin [10] augment postings lists with *Bloom Filters* [29], allowing efficient, approximate

[1] It is likely that other research groups were experimenting with multi-stage retrieval during the same time period, but Bing were the first to publicly outline this approach.

candidate generation via Boolean conjunctions. A similar idea was explored by Goodwin et al. [105], who revisit the use of *signature files* [274] for modern computer architectures, outlining a number of innovations that result in bit-sliced signatures that support extremely efficient Boolean conjunctions. The resulting system, known as *BitFunnel*, is reported to be used in the first-phase matching component in the 'fresh' index of the Bing index, which indexes and serves documents that have been recently crawled.

Asadi and Lin [13] examine efficiency and effectiveness trade-offs for generating candidates, exploring both plain conjunctive, ranked conjunctive, and ranked disjunctive candidate generation stages. They found that plain conjunctive candidate generation was extremely fast, but was less effective for generating candidates. On the other hand, ranked conjunctions were shown to be more efficient than ranked disjunctions while providing similar effectiveness. A more recent analysis from Clarke et al. [61] outlined how trade-offs in multi-stage architectures could be evaluated without requiring explicit relevance judgments, and confirmed the findings of Asadi and Lin [13]. This work also showed that *aggressive* instances of WAND can provide useful efficiency/effectiveness trade-offs for multi-stage retrieval. Other work has explored alternative first-stage rankers such as those that use supervised learning [77], using additional index structures and hybrid processing algorithms to improve the efficiency and effectiveness candidate generation [255], and using candidate generation within question-answering retrieval systems [169].

4.2 Comparing Index Traversal Strategies

While comparisons between DAAT and TAAT systems have been made in prior work, no studies to date have examined SAAT traversal for efficient candidate generation. We now present our empirical comparison between DAAT and SAAT search methods.

4.2.1 Systems

For our comparison, we employ two DAAT search algorithms that represent the state-of-the-art in efficient top-k retrieval across document-ordered indexes. The first algorithm is the WAND algorithm of Broder et al. [36]. The second is the BMW algorithm of Ding and Suel [88]. For SAAT retrieval, we use the recently proposed JASS system from Lin and Trotman [148], which facilitates both exhaustive and early-termination SAAT search.

4.2.2 Common Framework

In order to conduct a fair comparison between the DAAT and SAAT retrieval methods, we build a common framework that factors out many of the issues that are orthogonal to the efficiency characteristics of top-k retrieval. The following section outlines the different issues that were factored out, and justifies why it is important to do so.

Document Parsing. During document indexing, a range of decisions are made to determine how the document text is handled. For example, tokenization, stemming, and stopping all have

an impact on the index, which may have a significant impact on both efficiency and effectiveness. To factor out the differences in the way in which the underlying systems parse their documents, we used the ATIRE system [246] to generate a *master index* for each collection. Next, the DAAT and SAAT systems consume the master index and re-organize the data into the correct format, respectively. Therefore, the term statistics, lexicons, postings lists, and document maps all contain the same information irrespective of the query processing system.

Score Quantization. A key difference between DAAT and SAAT strategies is that the latter is only practical with pre-computed and quantized impact scores. In contrast, DAAT strategies most commonly store term frequency information inside the postings lists, and compute document scores with respect to a given similarity function during query evaluation. Thus, a substantial difference exists within the *cost* of scoring a document between DAAT and SAAT systems: computing a document score within a SAAT system involves only integer additions for summing the impact scores, whereas computing the similarity function (as is done within DAAT systems) may involve executing some floating point operations. In order to factor out this difference, we modify both WAND and BMW to work directly with impact scores. During creation of the aforementioned master index, ATIRE computes the BM25 score for each posting in the index, and then quantizes all of the scores into the range $[1, 2^b - 1]$ such that they can be represented by b bits. Following the recommendation from Crane et al. [66], we let $b = 9$ in our experiments. Then, instead of storing the term frequency in the postings list, the quantized impact score is stored instead. Since the WAND and BMW strategies must also store upper-bound scores to facilitate dynamic pruning, the quantized U_t and $U_{b,t}$ values are stored instead. Now, scoring a term/document pair in the DAAT framework involves just retrieving the impact score from the frequency list.

Document Ordering. Within an inverted index, document identifiers are generally stored as a monotonic sequence of integers. Therefore, they can be easily encoded as *delta-gaps* to make them more compressible. Prior work has shown that reassigning the identifiers of documents within the inverted index can result in substantial gains in both storage cost and retrieval efficiency [83, 88, 171, 175, 228]. To control for this, the master index was built using a single thread, which means the document identifiers are assigned *naturally*, or, according to the order in which they are encountered in the raw document corpus. For our collections, this means that the document identifiers are *crawl ordered*.

Compression. A huge amount of prior work has focused on index compression, resulting in a number of space- and time-efficient compression codecs. Generally speaking, compression methods that result in *smaller* indexes are usually *slower* to encode/decode [143, 175, 202], and so the selection of the index compressor has a huge impact on the latency of traversal and the space occupancy of the index. Hence, we control the compression codec for both DAAT and SAAT systems. In particular, we experimented with two different codecs. Firstly, we employed the *Fast Patched Frame-Of-Reference* (FastPFOR) implementation from Lemire and Boytsov [143] and corresponding optimizations [144]. While many variations of PFOR encoders exist, FastPFOR

aims to balance the space occupancy of the NewPFOR method [263] while maintaining the speed of PFOR [276], and was shown to have a good decoding-time/space trade-off in practice [143]. Secondly, we experimented with the QMX codec [242, 244]. QMX is based on the *simple* family [5, 8] and aims to pack integers into 128-bit words efficiently via SIMD extensions. QMX has been shown to outperform the state-of-the-art SIMD-BP128 method [143] for decoding, and is competitive in terms of space occupancy with other SIMD based encoders (see Section 3.1.4).

For document-ordered indexes, postings are organized around fixed-length blocks of 128 integers, both for compression and for skipping points in the WAND and BMW algorithms. Document identifiers are *delta encoded* prior to compression. For JASS, impact scores are stored uncompressed, since only one impact is stored for each segment. On the other hand, the impacts were compressed in the DAAT index. Note that delta encoding is not possible when compressing the impacts, as they are not guaranteed to be monotonically increasing.

We ran experiments with both the FastPFOR and QMX compression schemes, and found that the compression codec did not influence our findings (other than influencing the overall time/space trade-off). Hence, we report results using only the QMX codec, and leave further analysis on this orthogonal issue for future work.

4.2.3 Experimental Setup

Hardware. Experiments were conducted in memory on a Red Hat Enterprise Linux Server with two Intel Xeon E5-2630 CPUs and 256 GiB of RAM. All algorithms are single threaded and thus executed on one core of an otherwise idle machine. Indexes are stored entirely in main memory, and are warmed up prior to experimentation, which involves accessing each candidate postings list prior to query processing.

Software. We used the ATIRE system [246] for indexing, and then dumped this index into a format readable by both JASS (`https://github.com/lintool/JASS`) and the WAND/BMW system (`https://github.com/JMMackenzie/Quant-BM-WAND`). The codebase used for WAND and BMW was derived from the work of Mackenzie et al. [168] and is heavily inspired by the implementation from Dimopoulos et al. [85]. All implementations are written in C++, and compiled with GCC 6.1.

Document Collections and Indexing. We utilize three collections for our experiments: Gov2, ClueWeb09B, and ClueWeb12B (Table 2.1). All experiments used an identical underlying index generated by the ATIRE system, one for each collection. For simplicity, we kept collection processing to a minimum: for each document, all malformed UTF-8 sequences were converted into spaces, alphabetic characters were separated from numeric characters and s-stripping stemming was applied; no additional document cleaning was performed except for XML comment and tag removal. We did not remove stopwords as these are often used in higher order term proximity feature models [155]. As previously described, we post-processed the ATIRE index into a representation appropriate to each query evaluation strategy, with the same compression techniques to

the extent possible. All document scores were quantized to 9-bit values based on BM25 using the uniform quantization method [9, 66]. Although ATIRE performs multi-threaded indexing by default, we indexed the collection on a single thread in order to preserve the original document order, which can have an impact on latency. As previously discussed, we report results on indexes that are compressed via the QMX codec [242], but the choice of compression does not alter our findings.

Queries and Relevance. We employ standard TREC topics when evaluating the effectiveness of our search systems (see Table 2.1). We also use the UQV100 query variations for measuring efficiency over the ClueWeb12B collection. In terms of effectiveness, we follow best practices as outlined by Lu et al. [157]. In particular, we intentionally use shallow metrics for both of the ClueWeb collections, as computing deep recall-based metrics is not reliable due to the shallow pooling depth [157]. For Gov2, we compute RBP and AP. For ClueWeb09B, we use RBP and NDCG@20. For ClueWeb12B, we use RBP and NDCG@10. For all collections, we use a persistence value of $\phi = 0.8$ when computing RBP.

Measuring Efficiency. We measured query latency in milliseconds, taking the average across 3 runs. To capture query variance, we primarily report results using a 'Tukey' boxplot, in which the box bounds the 25th and 75th percentiles and the bar inside the box denotes the median. The whiskers correspond to data within 1.5× IQR and outliers are plotted as points. This format is used throughout the remainder of this dissertation. Space usage is measured in GB and includes only postings.

4.2.4 Experimental Analysis

Overview: Efficiency, Effectiveness, Space Consumption. For the first experiment, we compare the different systems with respect to mean and median latency, space consumption, and effectiveness. We retrieved the top 1,000 documents using the rank safe instance of each algorithm, and the results are presented in Table 4.1.

Based on just the mean and median latency, we observe that BMW generally outperforms the alternative approaches across all collections, supporting observations from prior literature [88]. We also observe that the DAAT methods have a much lower *median* latency than JASS on the larger ClueWeb collections, while the mean latency is much closer. This indicates that there are some queries with a high latency that bias the mean for the DAAT approaches. With regard to space consumption, it is clear that the indexes are all roughly the same size, with the impact-ordered index being slightly smaller. The BMW index is slightly larger than the WAND index, as the per-block bounds must also be stored. Finally, it is clear that the different approaches result in the same effectiveness irrespective of the collection or evaluation metric used. This validates that the underlying indexing and retrieval approaches are processing the *same* postings lists, supporting the fairness of the given comparison.

Gov2					
System	Mean	Median	Space	AP	RBP
WAND-E	41.6	23.3	17	0.290	0.586 (0.003)
BMW-E	30.8	17.4	17	0.290	0.586 (0.003)
JASS-E	34.9	20.2	15	0.290	0.586 (0.003)
ClueWeb09B					
System	Mean	Median	Space	NDCG	RBP
WAND-E	190.8	73.5	51	0.138	0.259 (0.126)
BMW-E	166.3	61.4	52	0.138	0.259 (0.126)
JASS-E	170.5	117.8	51	0.138	0.259 (0.126)
ClueWeb12B					
System	Mean	Median	Space	NDCG	RBP
WAND-E	210.5	114.3	65	0.124	0.264 (0.355)
BMW-E	210.3	107.9	65	0.124	0.264 (0.355)
JASS-E	221.9	148.7	63	0.124	0.264 (0.355)

Table 4.1: Effectiveness, efficiency, and space consumption for exact query evaluation for TREC topics with $k = 1{,}000$.

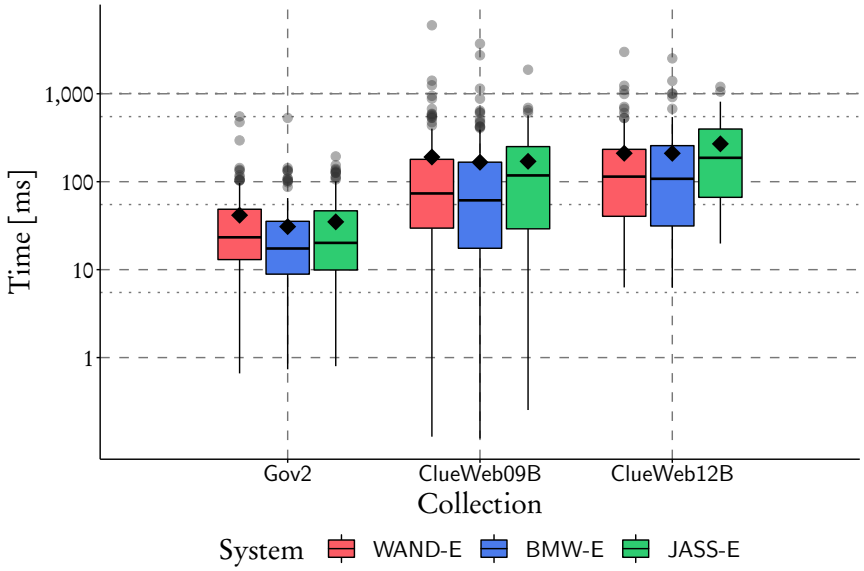


Figure 4.1: Efficiency profile for the rank safe algorithms across each collection where $k = 1{,}000$. The BMW and WAND algorithms outperform the JASS algorithm on average.

Next, we plot the per-query latency in the form of a boxplot. The results are presented in Figure 4.1. As suspected, the DAAT approaches are generally more efficient than JASS, but are susceptible to higher tail latency.

Rank Safe Performance. Next, we explore the variance in latency for the rank safe systems across the ClueWeb12B collection and the UQV100 queries. To gain further insight as to how the different approaches scale with the size of the candidate set, k, and the length of the query, $|q|$, we break our analysis into these facets. Results are shown in Figure 4.2. Firstly, we notice that the BMW algorithm is superior when the length of the input query is short. However, for longer queries, BMW becomes slower than the WAND algorithm. The main cause for this issue is that BMW seems to be more susceptible to high tail latency. In general, the DAAT algorithms become slower as k increases. This makes sense, as the effectiveness of DAAT dynamic pruning approaches depends on the entry threshold to the top-k heap. With a larger value of k, this entry threshold is lower, which induces more scoring operations (and in turn, fewer skipping operations). Similarly, we observe all algorithms becoming slower as the length of the query increases, as the number of candidate postings generally increases with query length.

Turning our attention to JASS, we observe that it is slower on average than both WAND and BMW for all lengths of query and all values of k, except for single-term queries where $k = 1{,}000$. However, it has a much tighter bound on the processing latency, as can be observed from the box-and-whisker plot. This observation teaches an important lesson in the methodology of efficiency experimentation — observing only summary statistics such as mean or median latency is insufficient for understanding the underlying performance characteristics of a system. As such, we recommend researchers to use box-and-whisker plots when visualizing latency data, and report tail latency in tables containing summary statistics.

Approximate Processing. In the prior experiments, we observed the efficiency of query processing systems that returned the *rank safe* results lists, and hence, achieved the maximal effectiveness for the ranking model. However, sometimes it is advantageous to trade off effectiveness for an increase in efficiency. In a multi-stage retrieval system, this idea is not new. In fact, the candidate generation phase only requires a "good enough" set of documents to be identified, such that the LtR model can push the relevant documents to the top of the ranking [13, 163]. This allows for *aggressive* or *approximate* index traversal methods to be employed, which may result in a much lower latency without impacting the precision of the results list. Thus, we are interested in comparing the performance of the different candidate generation approaches when we relax the effectiveness constraints of the candidate generation stage. To facilitate the *aggressive* traversal methods, we deploy the so-called *aggressive* or *approximate* methods of the WAND, BMW, and JASS algorithms, which trade effectiveness for efficiency.

DAAT dynamic pruning algorithms such as WAND and BMW can be accelerated by artificially increasing the value of the score threshold, $\hat{\theta}$. This can be achieved by setting $\hat{\theta} = F \cdot \theta$ where $F > 1.0$, and θ is the current heap threshold. By increasing the score threshold, more *skipping* is

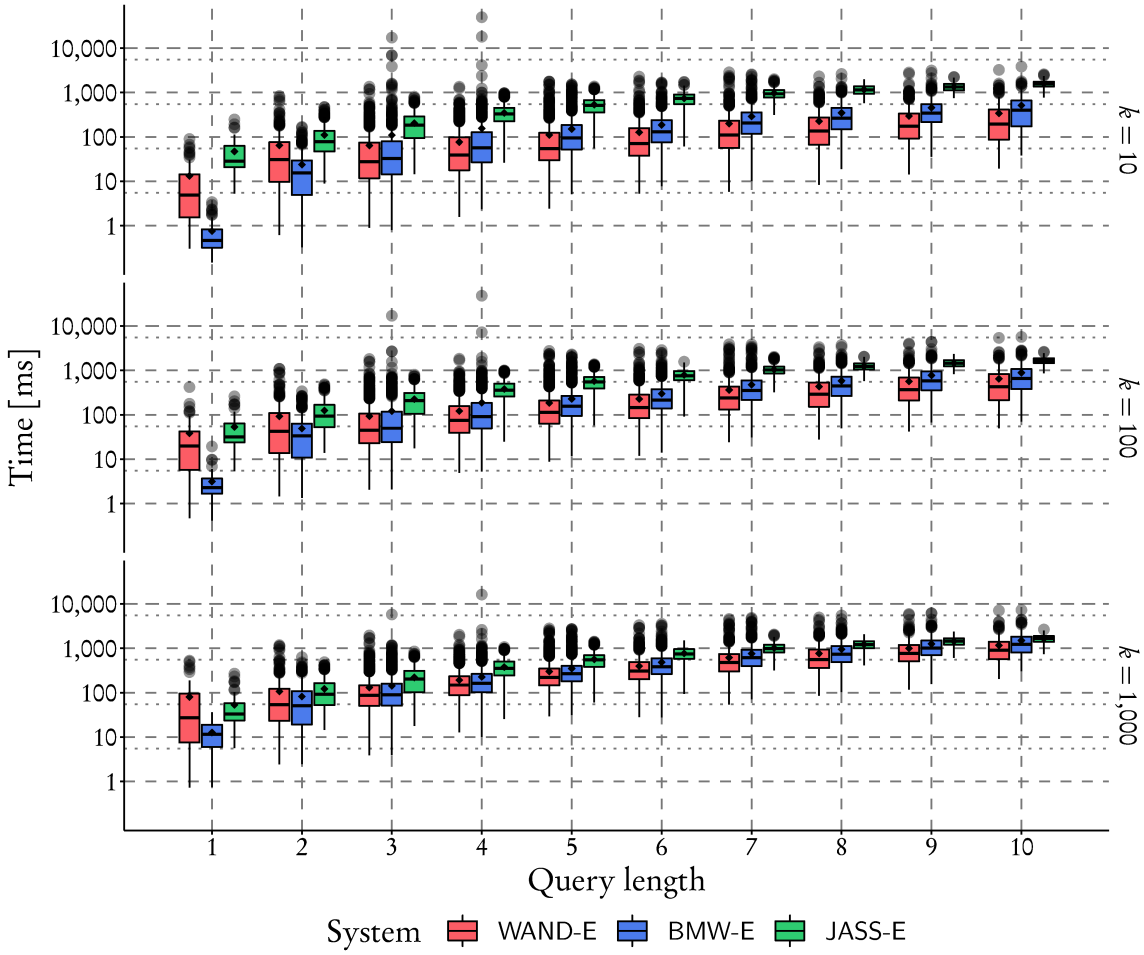


Figure 4.2: Efficiency profile of the WAND-E, BMW-E and JASS-E algorithms across ClueWeb12B and the UQV100 topics as both k and $|q|$ are varied.

induced, which makes query processing faster at the cost of rank safety. SAAT systems such as JASS can be accelerated via early termination by setting a parameter, ρ, which governs the total number of postings to process before terminating the search.

In order to determine the efficiency/effectiveness trade-off for each algorithm, we sweep the aggression parameters (F for the DAAT methods, and ρ for the SAAT method) to build a trade-off curve across the three TREC collections and topics (Table 2.1). Figure 4.3 shows the results, plotting both the median and mean latency when retrieving the top 1,000 documents.

On Gov2 and ClueWeb09B, the latency of the candidate generation stage can be almost halved with respect to the exhaustive instances of each algorithm *without* losing effectiveness. However, the results over ClueWeb12B are less clear, with the aggressive algorithms providing much more

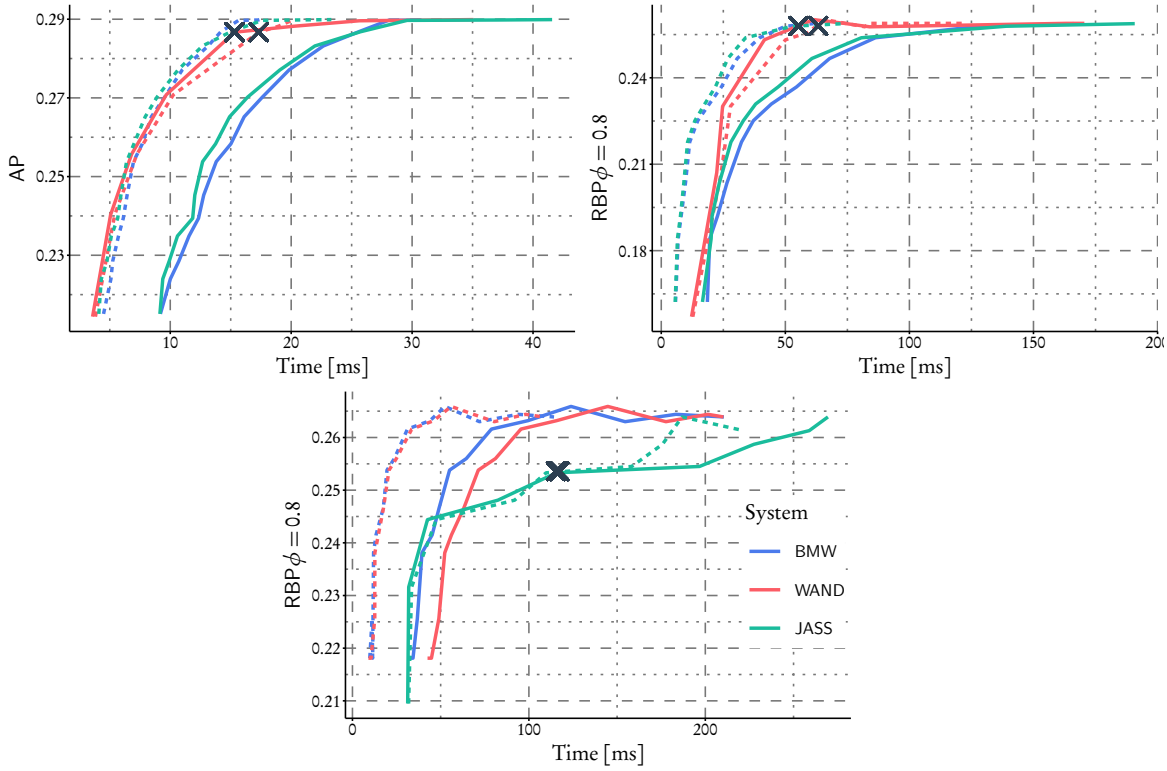


Figure 4.3: Efficiency/effectiveness trade-off for different levels of approximation across Gov2 (top left), ClueWeb09B (top right), and ClueWeb12B (bottom). Note that both mean (solid line) and median (dashed line) trade-offs are shown, and the effectiveness measure is not the same across all collections. Crosses represent the trade-off when setting ρ to 10% of $|D|$.

variance than on the other collections. In any case, we observe that aggressive traversal can reduce latency without reducing effectiveness, making it a favorable strategy for improving the efficiency of candidate generation. Black crosses on the JASS lines represent the efficiency/effectiveness point that corresponds to the recommended heuristic value of ρ [148]. We call this parameterization JASS-A henceforth. To yield a comparable instance of WAND and BMW, we selected settings of F for both algorithms that result in the *same* mean effectiveness as the corresponding JASS-A algorithm, and denote these systems as WAND-A and BMW-A respectively.

To better understand the efficiency profiles of the aggressive traversal techniques, we plot the corresponding boxplots for each collection. We use the same experimental setup as above, retrieving the top 1,000 documents across each collection for the corresponding TREC topics. For convenience, we also include the rank safe query processing latency values from Figure 4.1 in our analysis. The results are shown in Figure 4.4. While the aggressive DAAT methods improve in latency with respect to the exhaustive methods, they are still unable to control tail latency, with some long running queries still present. A failure analysis revealed that the high impact documents

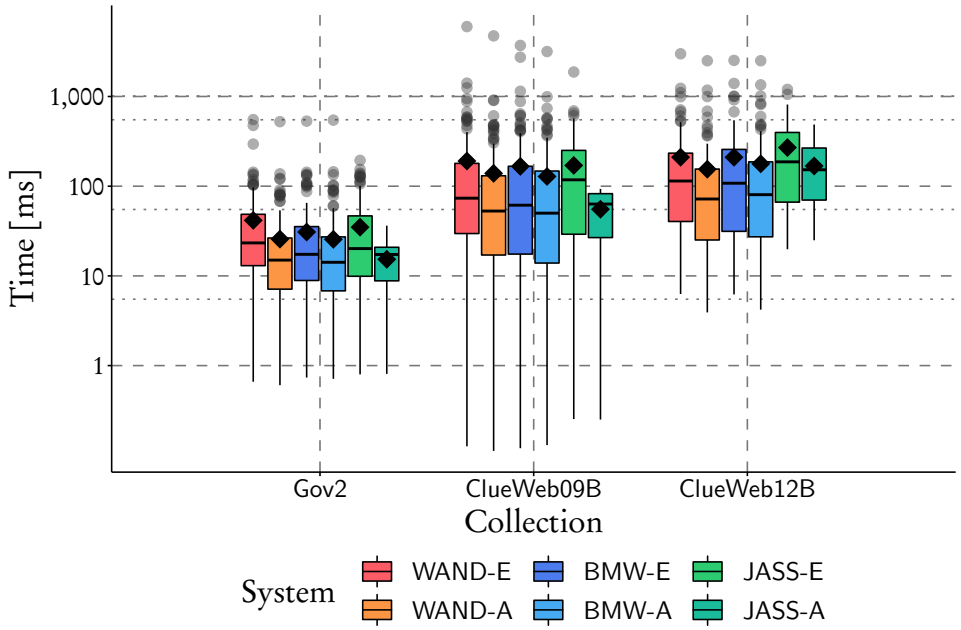


Figure 4.4: Performance of retrieving the top 1,000 documents across three different collections with their respective TREC topics.

for these queries had large document identifiers, meaning that they were stored towards the end of their respective postings lists. Hence, the heap threshold remained low for the majority of the index traversal, leading to less dynamic pruning, more document evaluations, and more block decompressions. In contrast, JASS-A reduces both the mean and median latency, while also reducing the variance of the latency, with respect to JASS-E. This indicates that the ρ setting for JASS can be used to finely control the maximum latency of index traversal, making it very good at controlling tail latency.

Since our analysis only examined the small TREC topic sets, we again turn our attention to the UQV100 topics and the ClueWeb12B collection for a more robust analysis. We run each of the *aggressive* algorithms across three values of k (10, 100, and 1,000), and plot the results by query length. Results are shown in Figure 4.5. Similar to the rank safe algorithms, we observe that longer queries result in a larger processing time. However, this observation only holds for JASS-A while increasing query length from 1 to 4 terms. That is, for any query longer than 4 terms, the efficiency profile of JASS-A remains mostly constant, with no added overhead. Indeed, this is by design; by fixing the value of ρ, the processing latency for JASS is kept under control. In fact, as query length increases, the variance becomes smaller for JASS, as there is a decreasing number of queries with *less than* ρ postings, making the processing time converge to a very tight range. Thus, for longer queries, JASS-A outperforms both of the DAAT algorithms. On the other hand, the DAAT methods are still superior when the input queries are short.

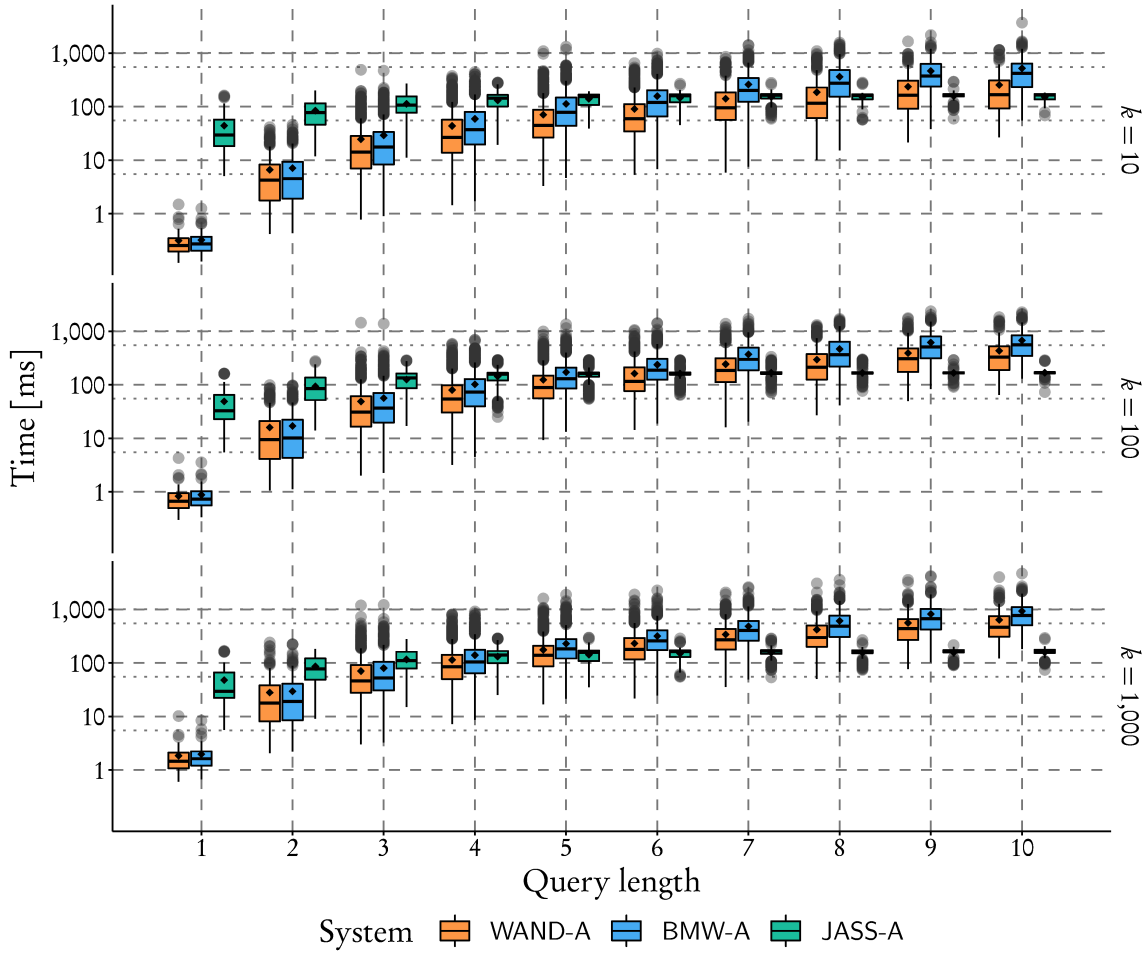


Figure 4.5: Efficiency profile of the WAND-A, BMW-A, and JASS-A algorithms across ClueWeb12B and the UQV100 topics as both k and $|q|$ are varied.

Evaluating Postings. The final experiment in our empirical study explores the number of postings processed to gain intuition on the workload of the different processing methods. Table 4.2 reports the mean and median number of postings scored when retrieving the top 1,000 documents from the UQV100 queries across ClueWeb12B. Clearly, aggressive processing allows for fewer postings to be retrieved on average, with the largest gains seen from the JASS-A approach. Another interesting aspect of this analysis is how BMW processes fewer postings than WAND, but is not more efficient. This is due to BMW spending extra time computing which documents it should skip, whereas WAND simply processes more documents and spends less time computing skips. Most interestingly, JASS processes more than an order of magnitude of additional postings than the equivalent DAAT methods, yet is only moderately less efficient.

Algorithm	Mean	Median
WAND-E	191,269	133,264
BMW-E	126,858	98,260
JASS-E	60.1M	47.4M
WAND-A	108,661	82,515
BMW-A	101,544	78,308
JASS-A	4.3M	4.8M

Table 4.2: The number of postings processed for the UQV100 queries on ClueWeb12B, for exact and approximate query evaluation techniques, $k = 1{,}000$.

4.2.5 Discussion

We now discuss the findings from our empirical evaluation, and put them into context with more recent literature on top-k retrieval.

Modern CPU Architectures. The insight behind the performance optimizations in DAAT dynamic pruning algorithms is that documents that *cannot* make it into the top-k results list can be *skipped*. Skipping the evaluation of a document allows the computation of the underlying ranking model to be bypassed, which saves CPU cycles and results in faster traversal. However, in order to skip documents, a non-trivial amount of computation must be undertaken, especially in the block-based algorithms such as BMW. This skipping logic uses branches to control the flow of the algorithm, which can result in branch mispredictions creating pipeline stalls. Furthermore, skipping document evaluations results in non-uniform memory access patterns that may result in cache misses, adding to latency. On the other hand, the SAAT methods conduct no optimization whatsoever, relying on a tight inner processing loop to provide *predictable computation*, which allows the CPU to optimize instruction throughput without stalls and cache misses [31].

Score Quantization. One important distinction to make with other works is that we are operating on quantized indexes. This means that the scoring operation is simply a sum of impact scores, rather than the computation of a non-trivial relevance function such as BM25, which involves multiple floating point operations. Hence, when index quantization is employed, the cost of scoring documents is much lower than in traditional indexes, making the benefits of skipping much lower. This was directly observed in our experiments, with the BMW algorithm performing worse in some circumstances than the WAND algorithm.

Stopping. For this study, we opted to keep stopwords, as we argued that they are necessary for higher-order models that may be used in later ranking stages. However, the impact of stopping on the effectiveness of candidate generation is small, since the contribution of stopwords does not largely impact the rankings. A better approach is to simply stop the *queries*, which allows the

postings to be kept for use in later stages while improving the efficiency of the first-stage candidate generation. Since SAAT methods process the highest impact postings first, early termination is likely to skip the scoring of stopwords. However, the equivalent DAAT approaches may suffer from processing stopwords, as more documents may be scored than necessary. This phenomenon should be examined more closely in future work.

Indexing and Ranking Considerations. While the SAAT algorithms have proved to be very good for efficient candidate generation, they have some disadvantages that may make them difficult to use in practice. Firstly, since impact-ordered indexes require all documents to be scored and quantized during index building, they are much more difficult to update. While some work has examined the effects of quantization and index layout on dynamic collections [39, 40, 42, 188], there has not been a comparison between document- and impact-ordered indexes. Secondly, by quantizing the index, alternative scoring methods cannot be applied on-the-fly. With regular document-ordered indexes, only term-frequencies are stored within the postings lists, which means that the ranker (or parameters of the ranker) can be easily changed without recomputing the entire index. Even with dynamic pruning methods, such as WAND and BMW, score upper-bounds can be estimated independently from the ranker, allowing the ranker to be changed efficiently [159, 160]. Future work should investigate these operational trade-offs that are important for real IR systems in practice. Thirdly, one final disadvantage for impact-ordered indexes is that they are currently unable to support efficient Boolean operations. For example, both plain and ranked conjunctive queries cannot be efficiently processed over impact-ordered indexes, since each postings segment would need to be accessed in order to determine whether a match is made or not. Indeed, the accumulator table could be augmented with a secondary counter which counts the number of terms that were matched for each document, but this is likely to be slower than processing the same query across a document-ordered index. The above observations add even more nuance to our empirical analysis, but solving them is left for future work.

Further Improvements. Since this research was conducted, a range of improvements have been made for DAAT traversal. For example, using *variable-sized* blocks for index traversal can result in faster than ever top-k retrieval performance [172, 173]. Other recent work has shown that the implementation of the underlying algorithms can have a notable impact on efficiency, with certain optimizations allowing considerable improvements to both DAAT and SAAT algorithms [127, 243]. A more recent comparison of DAAT methods [175] showed that the MaxScore dynamic pruning approach can actually outperform other approaches for long queries and when the value of k is large – a problem identified for the WAND and BMW methods in our experimentation. It would be interesting to compare MaxScore to JASS for processing long queries and for large results sets. A final aspect which is worth considering further is how document identifier re-assignment changes the observations of these experiments. New methods for document identifier re-assignment have shown very large improvements for both the space consumption of document-ordered indexes [83, 171], and the retrieval efficiency of DAAT algorithms [175]. However, it is unclear whether these improvements can be carried over to impact-ordered indexes.

4.3 Improved Heuristics for Score-at-a-Time Index Traversal

In the previous experimental analysis, we examined how both DAAT and SAAT systems perform for the top-k candidate generation task. Aggressive SAAT traversal was shown to provide a tight bound on tail latency by limiting the number of postings that could be scored for each query based on a pre-defined and constant value. As such, the efficiency of SAAT retrieval was not substantially impacted by the length of the query, or by the number of results required, but was correlated instead with the *number of postings* that are processed. However, one issue with using a constant value is that the system will process different fractions of postings for different queries. This can make it difficult to guarantee the effectiveness of the ranking, especially as the number of candidate postings increases. Consider the following examples, based on the ClueWeb12B collection, where we set $\rho = 10^8$ (10 million postings). Now, assume the input query is *"melbourne coffee"*:

- *melbourne* appears in 401,213 documents.
- *coffee* appears in 1,373,924 documents.
- The total number of candidate postings is 1,775,137.

Clearly, since ρ exceeds the total number of postings for this query, every single posting will be processed, meaning that this query was evaluated in a rank safe manner. Now consider the situation where the input query is *"melbourne city view"*:

- *melbourne* appears in 401,213 documents.
- *city* appears in 7,408,632 documents.
- *view* appears in 18,489,503 documents.
- The total number of candidate postings is 26,299,348.

In this case, less than 50% of the candidate postings will be processed. As further motivation, consider Figure 4.6, which plots the percentage of candidate postings processed for different fixed settings of ρ across varying query lengths. As query length increases, the total proportion of processed postings decreases; although SAAT traversal processes the *highest impact* segments first, there may be negative implications on the effectiveness of queries with a high number of postings, and there is currently little understanding on the impact that the value of ρ has on the *effectiveness* aspect of top-k retrieval. We aim to address this issue by proposing new methods of setting ρ, and exploring how they impact both query processing efficiency and result effectiveness.

4.3.1 Heuristics for Score-at-a-Time Processing

We now discuss some heuristic methods for setting ρ for SAAT processing, and the inherent trade-offs that arise from each of these methods.

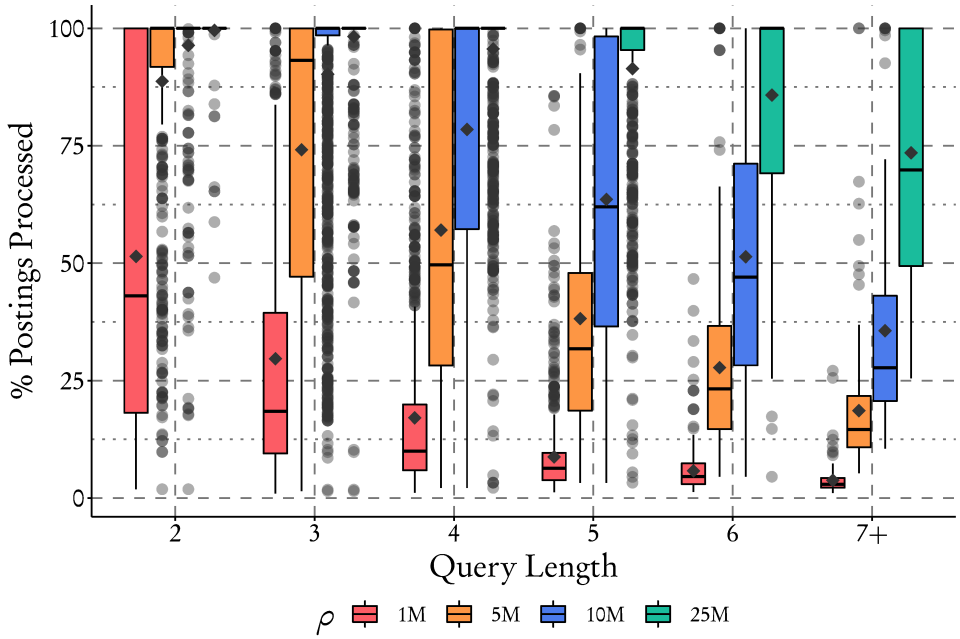


Figure 4.6: The percentage of total candidate postings processed for varying query lengths across the UQV100 queries for four different fixed values of ρ.

Rank Safe Processing. For SAAT traversal, rank safety can be guaranteed by exhaustively processing all candidate postings. For a given query q made up of a set of n unique terms $\{t_1, t_2, \ldots, t_n\}$ with corresponding postings lists $\{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n\}$, exhaustive processing requires setting ρ as:

$$\rho_{exhaustive} = \sum_{i=1}^{n} |\mathcal{L}_i|.$$

In practice, this setting will cause processing efficiency to be sensitive to the number of query terms provided, and more directly, the length of the corresponding postings lists.

While there are early termination methods that can terminate processing when a new document cannot enter the top-k results list, the documents within the top-k set are not guaranteed to be ranked correctly [37]. This optimization compares the kth score to the $k+1$th score; if the $k+1$th document score, summed with the remaining term upper bounds, does not exceed the kth document score, then no new documents can enter the top-k heap. Unfortunately, this is rarely the case, and this optimization has been shown to perform poorly in practice [94].

Fixed Cost Heuristics. One way to enforce aggressive early termination is to fix the value of ρ to some constant. This heuristic setting for ρ can be expressed as:

$$\rho_{fixed} = c, \text{ where } c > 0.$$

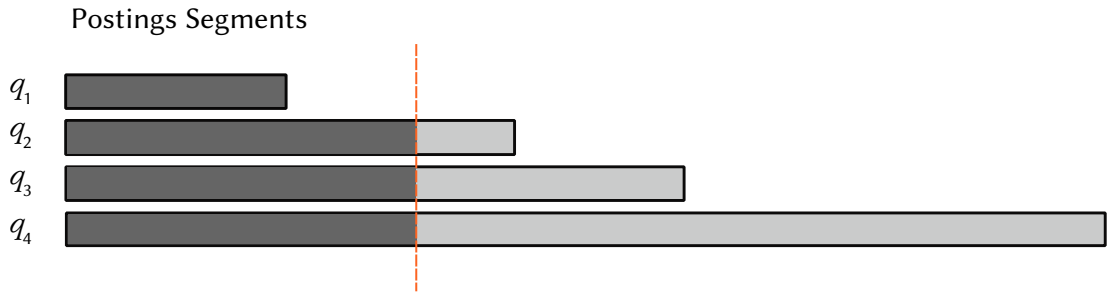


Figure 4.7: An example of JASS processing with a fixed setting of ρ. The darker postings segments represent those that have been processed, and the dashed line represents the selected value of ρ.

Figure 4.7 shows a pictorial example of this heuristic across a set of four different queries. Based on empirical observations from a set of web topics, Lin and Trotman [148] found that setting $c = 0.1 \cdot |D|$, yielded an efficient configuration without sacrificing much effectiveness. This setting was shown to provide efficient and effective results across many different collections [68, 148, 150]. With this heuristic, the processing time becomes largely independent of query length or the term statistics of query terms, as the algorithm will simply back out once it has processed ρ postings. This phenomenon was observed in Section 4.2.4.

Variable Cost Heuristics. Another possible approach is to set ρ based on a percentage of the candidate postings on a per-query basis:

$$\rho_{percentage} = \frac{z}{100} \cdot \sum_{i=1}^{n} |\mathcal{L}_i|, \text{ where } z > 0.$$

Using this configuration, JASS will process a variable number of postings depending on both the value of z, the number of query terms, and the length of the corresponding postings lists. Figure 4.8 shows a pictorial example of this heuristic across a set of four different queries. Intuitively, as z increases, so too would the effectiveness of the traversal, whereas the efficiency would decrease. This setting has not yet been explored in the literature, and is explored in this work.

4.3.2 Parallel Extensions to Score-at-a-Time Traversal

In order to efficiently process queries with a large number of candidate postings, we propose a simple extension to the JASS system which enables concurrent postings traversal. Similar extensions have been proposed for DAAT dynamic pruning algorithms [215], but are more difficult to implement, as dynamic pruning effectiveness depends on tracking dynamic thresholds which are then used to avoid processing documents that can not make the top-k results. Hence, advanced threshold sharing synchronization schemes must be implemented to achieve maximal efficiency in parallel DAAT algorithms. On the other hand, SAAT traversal is much more amenable to parallelization, as the various threads need not communicate during processing.

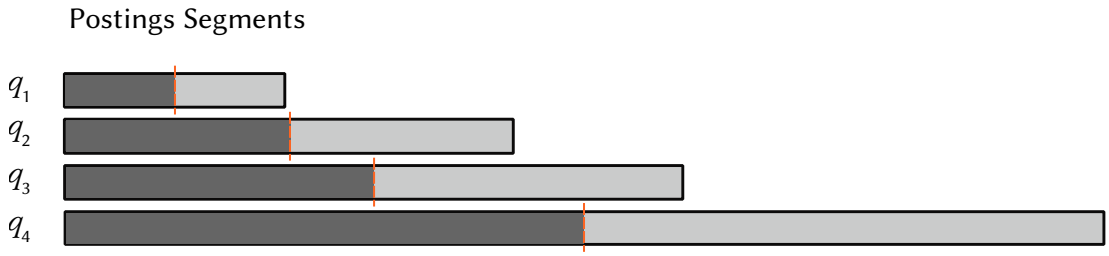


Figure 4.8: An example of JASS processing with a percentage-based ρ, with ρ set to 50% of the candidate postings for each query. The darker postings segments represent those that have been processed, and the dashed lines represent the value of ρ for each query.

Single-Threaded Flow Control. Before outlining the parallel extensions, we reiterate how JASS processes queries in a single-threaded manner.

Given a set of candidate postings lists to process, each consisting of a metadata structure followed by an ascending list of document identifiers, the first step is to extract out the segment metadata and store it as a vector. This vector contains an $\langle \text{impact}, \text{length}, \text{offset} \rangle$ tuple for each segment in the provided set of postings lists, which stores the segment impact value, the number of elements in the segment, and the byte offset in which the segment begins in the postings file. This organization was previously discussed in the background section, and so we refer the reader to Chapter 3.1.2 for a reminder of this index organization.

The metadata vector is then sorted in *descending* order of the impact values, with ties broken by length (shorter segments before longer segments). Postings processing is initiated by iterating through the sorted vector, and processing the next segment if and only if the sum of the processed postings plus the length of the candidate segment do not exceed the upper-bound number of postings to process, ρ. After each segment is processed, the processed postings counter is updated by adding the length of the segment that was just processed. Once the stopping rule is applied, a top-k heap which is maintained during processing can be iterated to efficiently return the top-k highest scoring documents.

Multi-Threaded Flow Control. In the multi-threaded extension of JASS, the main processing loop is modified to be *thread-safe*. Firstly, the metadata vector is obtained as in the single-threaded version. Next, we divide the processing load by allowing each thread P_n to process every nth segment in the segment vector, each updating a *local* processed postings counter. To this end, the early termination heuristic is on a *per-thread* level rather than a global level as in the single-threaded implementation, which allows threads to avoid synchronizing a global counter.

Multi-Threaded Segment Processing. Segment processing is similar to the single-threaded version, with a few additional caveats (Algorithm 4.1). Since multiple segments can be processed in parallel, there is no guarantee that an accumulator can be safely updated — other threads may be trying to update the accumulator at the same time. To rectify this issue, we associate an

Algorithm 4.1: Multi-threaded Segment Processing

```
   Input  : A segment metadata tuple, MetaData, and the accumulator table, Accumulators.
   Output : None
1  Documents ← MetaData.decompresssegment( )
2  for CurrentDoc in Documents do
3  |   while Accumulators.getlock(CurrentDoc) do
4  |   |   continue
5  |   end
6  |   Accumulators.addscore(CurrentDoc, MetaData.impact())
7  |   Accumulators.releaselock(CurrentDoc)
8  end
9  return
```

atomic flag with every docID. When a thread wishes to update the accumulator of a particular document, it must first obtain the flag (lines 3–5), at which point the thread is safe to update the accumulator (line 6), then release the flag (line 7). The atomic flag is implemented in user space to avoid expensive kernel calls, and allows each active thread to remain on its respective processor without an expensive context switch occurring. This synchronization technique can be thought of as a lightweight spinlock. The process of 'locking' and 'unlocking' the atomic flag is indeed an atomic operation, and is therefore thread-safe.

Managing the Accumulator Table. By default, JASS manages its accumulators using the efficient accumulator initialization method of Jia et al. [120]. The main idea of this approach is to break the global accumulator table into a number of logical subsets, each associated with a single bit within a bitvector. When an accumulator needs to be updated, the corresponding bit within the bitvector is checked. If the bit is not set, it means that the current subset of accumulators has not been cleared (and contains values from the previous query). In this case, the associated subset of accumulators are reset to 0, and the associated bit is set. At the end of query processing, the entire bitvector is cleared. Figure 4.9 shows an example of this organization, where two logical subsets have been cleared during the current processing, and four subsets are yet to be accessed or cleared.

The key problem with this approach is that adding a contribution to an accumulator involves checking (and potentially setting) the corresponding bit, which is not thread safe. Indeed, synchronization methods could be applied to the reading/writing of the bit, but this will result in a performance hit (as it is much more likely to be a point of contention between multiple worker threads). For this reason, we revert to using a simple, global accumulator table, with the overhead of initializing *all* accumulators before processing. Using `uint16_t` accumulators, and the vectorized STL `std::fill` operation, this took 13 ± 2 ms across 100 individual benchmarks.

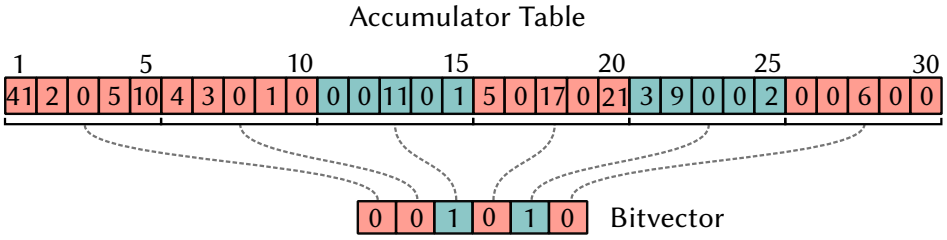


Figure 4.9: An example of the efficient accumulator initialization method of Jia et al. [120]. The accumulator table is broken into subsets of 5 elements, each associated with a bit inside a bitvector. In this example, the regions containing documents [11..15] and [21..25] are not required to be cleared before updating the accumulators within that range.

Managing the Heap. Another major issue with concurrent accumulator access is that the top-k heap cannot be efficiently maintained. At any time, there may be an arbitrary number of threads updating unique accumulator values. If we wish to maintain a top-k heap, then the heap updates must be made thread-safe, which results in a large efficiency decay due to contention. To remedy this, we do away with the score heap entirely, and opt for iterating the accumulator table once the postings have been traversed, collecting the top-k documents, and returning them once the process is complete. This can be done in $\mathcal{O}(n \log k)$ comparisons, where $n = |D|$. Prior work showed that increasing k does not greatly impact the performance of JASS, since the only practical difference in processing is the number of heap operations that will occur [68]. By removing the score heap, this is taken one step further — the value of k has no notable difference on processing efficiency since a constant overhead is added, irrespective of the value of k. Across three sets of 100 individual benchmarks with $k = \{10, 100, 1{,}000\}$, finding the top-k documents from the accumulator table took 71 ± 3ms, with very little variance between the different values of k.

Putting it Together. The general design of the concurrent JASS traversal is based on minimizing contention as much as possible. As discussed previously, JASS is efficient due to its *predictable* processing structure, which allows a high throughput of CPU instructions. Thus, keeping pipeline stalls to a minimum is of high importance. Figure 4.10 shows an example of how contention is mostly avoided in parallel SAAT processing: the only instance of locking occurs when adding an impact to the accumulator table. Since impacts are added to the accumulator table *millions* of times for a query, lightweight atomic spinlocks are used for efficiency.

4.3.3 Experimental Setup

Hardware. Timings were performed in memory on an idle Red Hat Enterprise Linux Server (v7.2) with two Intel Xeon E5-2690 v3 CPUs and 256GB of RAM. Each CPU has 12 physical cores with hyper-threading enabled, resulting in 48 available processing cores. Algorithms were compiled using GCC 6.2.1. All timings are reported in milliseconds unless otherwise stated, and are the average of 3 runs.

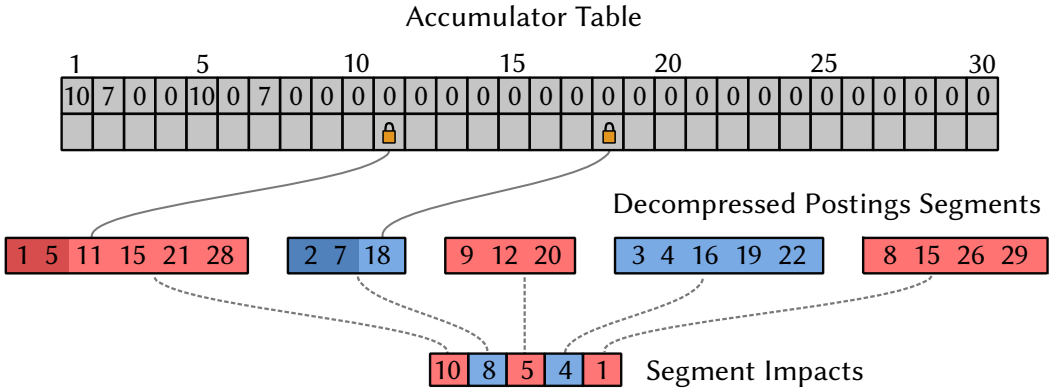


Figure 4.10: An example of JASS processing with multiple threads. In this figure, there are two threads in operation, denoted in red or blue shading. The darker postings segments represent those that have been processed, and the processed values can be seen in the accumulator table. At the current point of processing, P_1 is about to add impact 10 to the accumulator for document 11, and P_2 is about to add the impact 8 to the accumulator for document 18. Note that both threads have obtained the required lock.

Software. We use the original implementation of JASS (`http://github.com/lintool/JASS`) to run all single threaded experiments. Our multi-threaded implementation of JASS was derived from this codebase, and is made available: `http://github.com/JMMackenzie/Multi-JASS`. Threads were spawned using C++ STL threads, and the spinlocks are based on the STL `std::atomic_flag`.

Document Collections and Indexing. Experiments were conducted on the standard TREC ClueWeb12B collection, which contains 52,343,021 web documents. We used the same methodology as earlier for building the index: ATIRE indexes the raw corpus and builds the postings with 9 bit quantization [66] after applying the BM25 similarity model, and the JASS index was then compressed using the QMX compression scheme [242, 244]. Although recent work has shown that *uncompressed* indexes can outperform compressed indexes when using SAAT retrieval strategies, the additional space overhead for such gains can be prohibitive [149]. In any case, the findings presented here are orthogonal to the compression scheme, as the same compression scheme is used for all instances of the search system.

Queries and Relevance. In order to test our hypothesis that a fixed ρ setting reduces system effectiveness as query length increases, we use the UQV100 collection of Bailey et al. [18]. After normalization, which included *s*-stemming, stopping, and removal of duplicate terms within a query, 5,682 queries of varying lengths remained (Table 4.3). Note that in our analyses, we group all queries with 7 or more terms together.

Query Length	2	3	4	5	6	7+	Total
Count	620	1,744	1,881	891	363	183	5,682

Table 4.3: The number of queries for each length across the UQV100 collection. Stopwords and duplicate terms were removed from queries before processing, and 83 single term queries were dropped.

4.3.4 Experimental Analysis

Early Termination Trade-offs. Our first experiment explores the relationship between the early termination heuristic, and the effectiveness of the processed query. In order to measure the effectiveness loss, we first ran all 5,682 queries exhaustively using $\rho_{exhaustive}$. Next, we ran JASS using the percentage based heuristic $\rho_{percentage}$ for $z = \{5, 10, \ldots, 95\}$. We also ran JASS using the fixed heuristic ρ_{fixed} for various values of c between 250 thousand and 75 million. Then, for each heuristic configuration, we retrieved the top-10 documents for each query, and computed the difference in NDCG@10 with respect to the exhaustive result. Next, we calculated the proportion of wins, ties, and losses that the heuristic traversal had with respect to the exhaustive traversal. Since some improvements or losses may have been small, we consider deltas $\leq 10\%$ of the exhaustive value to be a tie. Figure 4.11 shows the density of wins, losses, and ties when comparing both fixed and percentage-based heuristics with the exhaustive run, respectively, for various query lengths. Generally speaking, as the length of the query increases, so too does the magnitude of postings that must be processed to achieve a close-to-exhaustive performance. This indicates that setting a fixed value of ρ may result in reduced effectiveness as the query length (and, more importantly, the number of candidate postings) increases. Additionally, the percentage setting appears to give a more predictable effectiveness trade-off than the fixed setting. Note that approximate processing can occasionally *improve* the effectiveness of the search, since the highest impact segments are processed first.

Our next experiment explores the behavior of the different heuristic settings with respect to execution time. From each heuristic, we select four configurations, and each retrieves the top-10 documents from the index. For the fixed setting, we set c as $c = \{5, 10, 15, 25\}$ million postings, and for the percentage setting, we set $z = \{20, 40, 60, 80\}$. We also run an exhaustive instance of JASS. The results are shown in Figure 4.12. As expected, fixing ρ to be a constant value allows for strict control of the upper-bound processing time, whereas using percentage-based settings may result in more variance, confirming our expectations.

To briefly summarize this experiment, we observe that the added control on the effectiveness trade-off provided by using percentage-based heuristics results in a loss of control on the tail latency. Conversely, fixing ρ, while reducing effectiveness, allows very strict control of the tail-latency.

Impact of Threading on Efficiency. For the next experiment, we test the efficiency of our proposed parallel extension to the JASS system, and in particular, explore how the number of

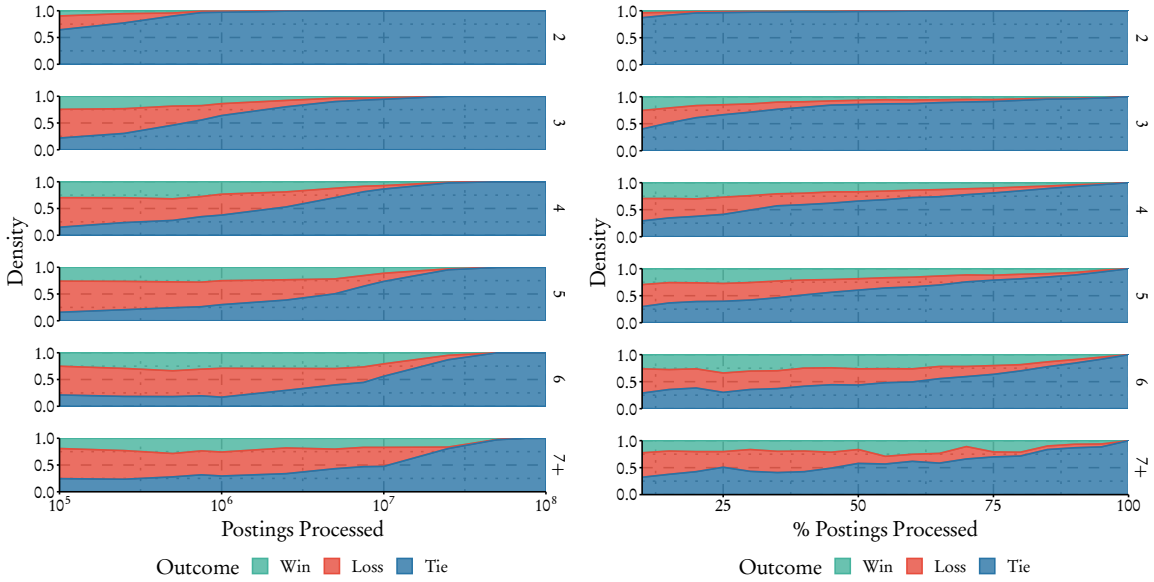


Figure 4.11: Density of wins, ties, and losses, when comparing the approximate retrieval to the exhaustive retrieval across all test queries for the fixed-postings heuristic (left) and the percentage-postings heuristic (right).

threads impacts the efficiency of the processing for varying query lengths. For simplicity, we assume that when processing in parallel, we will exhaustively process all candidate postings lists. We run the parallel version using between 4 and 40 threads inclusive, increasing the number of threads between each run. We then calculate the speedup as the average percentage improvement of the threaded run with respect to the exhaustive, single-threaded run, across each query length. The results are reported in Table 4.4.

Somewhat unsurprisingly, the speedup for a given number of threads generally increases as the length of the query increases. This is because the processing cost for longer queries is dominated by the postings traversal, whereas the cost for shorter queries is more often a combination of both the index initialization, and the accumulator traversal.

More surprisingly, however, is the very low rate in which the speedup increases when adding more threads. Since the workload is divided by allocating segments to threads, there may be cases where all threads are not utilized (for example, when there are 24 worker threads, but only 8 segments). This is due to the *inter-segment* method in which the processing is divided – a single segment will only ever be processed with a single thread. This implies that processing will be at least as slow as the slowest running segment, a potential bottle-neck which may be alleviated with *intra-segment* processing, whereby a segment can be processed by multiple threads. We leave further exploration into this approach as future work.

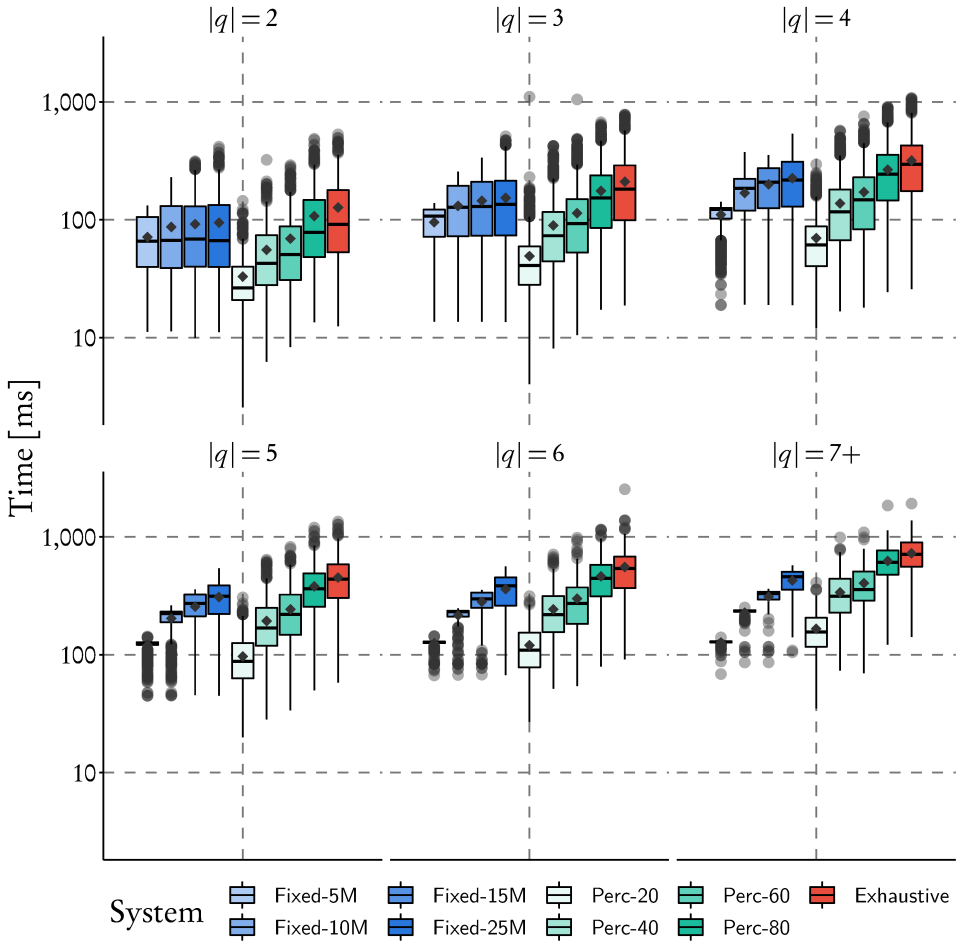


Figure 4.12: Efficiency of the different heuristics for ρ across all queries. Fixed values of ρ tend to exhibit very little variance as the query length increases, whereas the percentage based heuristics are indeed impacted by query length.

Meeting Service Level Agreements. From the lessons learned in the prior experiments, we now attempt to add selective parallelization [119] to the JASS system in order to improve efficiency and effectiveness while meeting two pre-specified SLAs. We select these SLAs to be realistic for a candidate generation stage, and hence set strict time budgets on both the 95th (P_{95}) and 99th (P_{99}) percentile response latency:

- $P_{95} \leq 200$ms, and
- $P_{99} \leq 250$ms.

As an additional quality assurance, we aim to minimize effectiveness degradation as much as possible while still meeting the target SLAs.

Threads	Query Length					
	2	3	4	5	6	7+
4	0.58	0.91	0.64	1.11	1.13	1.24
8	0.78	1.25	1.03	1.74	1.81	1.92
12	0.91	1.11	1.28	2.09	1.49	2.32
16	1.03	1.60	1.57	2.44	2.59	2.75
20	1.08	1.69	1.73	1.96	2.68	3.09
24	1.13	1.50	1.81	2.70	2.22	3.04
32	1.18	1.75	1.98	**2.73**	2.98	3.27
40	**1.20**	**1.76**	**2.08**	2.51	**3.04**	**3.37**

Table 4.4: Average speedup when adding threads to the processing load. As the length of the query increases, so too does the speedup, as the parallel portion of processing becomes larger for longer queries.

Efficiency Modeling. To efficiently predict the approximate running time for a query with ρ postings, we follow Lin and Trotman [148] in building a simple linear model. We sampled 1,000 web queries from an MSN query log [71], and ran them across our ClueWeb12B index using various fixed settings of ρ, collecting both the number of postings processed and the corresponding efficiency in milliseconds. We then derive our model by conducting a linear regression on the time taken and the postings processed for each query. The linear model is a good fit for the sample queries, with a coefficient of determination $R^2 = 0.926$. Our model can be used to estimate the processing time t (in milliseconds) given a value of ρ for the *single-threaded* instantiation of JASS:

$$M_t(\rho) = 35.541 + \rho \cdot 2.28 \times 10^{-5}$$

Conversely, our model can be used to predict the largest value of ρ that is acceptable for a given time bound t:

$$M_\rho(t) = \frac{t - 35.541}{2.28 \times 10^{-5}}$$

We note that many more advanced prediction models have been explored in the literature, especially for quantifying the processing time of DAAT or TAAT traversal [119, 129, 162, 166]. These models are almost certainly overly complicated for SAAT traversal prediction, as SAAT traversal is sensitive only to the number of postings to be processed [68].

Baseline Systems. The most obvious baseline is the exhaustive, single-threaded system. This system exhaustively processes all queries in a single-threaded manner, and provides the desired effectiveness that other systems should aim to achieve. Given the various time/effectiveness profiles that can be achieved using the different aggressiveness heuristics, we employ several instances of each as baseline. For the ρ_{fixed} baseline, we set $c = 5$ million as suggested by Lin

and Trotman [148], as well as $c = 7$ million (which corresponds to the calculated upper-bound ρ to meet the P_{95} SLA). These systems are denoted Fixed-c, where c corresponds to the fixed constant used. For $\rho_{percentage}$, we let $z = \{20, 40, 60, 80\}$, and denote these systems as Percentage-z (or Perc-z).

Selective Parallelization. Next, we propose some hybrid approaches which use selective parallelism to accelerate queries which are predicted to run slowly. The key idea behind selective parallelization is to only parallelize queries which are predicted to exceed the given time budget, as parallelizing short queries is often a waste of resources [119]. Based on our model, we know that queries with more than 7 million postings are likely to exceed the 200ms time budget, so we parallelize any query with over 7 million candidate postings. We exhaustively process *all* queries (and thus, lose no effectiveness), and call this system *Selective Parallelization* (SLP-E). Another approach is to use *aggressive* processing on top of the selective parallel system, where the parallelized queries will be used in conjunction with the fixed heuristic. We use 3 large values of c, namely $c = \{20, 50, 100\}$ million, as this value is divided equally among the processing threads (that is, each of the n threads will have a local ρ of $\frac{\rho}{n}$). This approach is called *Aggressive Selective Parallelization* (SLP-A-c), where c denotes the global ρ value selected.

We run all aforementioned systems across all 5,862 queries, with each system retrieving the top-10 documents. Table 4.5 shows both the efficiency and effectiveness results for this experiment for different numbers of worker threads.

Effectiveness Analysis. In general, the SLP-E and SLP-A systems outperform the fixed and percentage heuristic systems with respect to effectiveness, as they often process more postings than their counterparts. For example, the SLP-E system exhaustively processes all queries, and thus loses no effectiveness. In addition, the Fixed-7M system outperforms the Fixed-5M system and all of the percentage based systems except for Perc-80 with respect to wins, ties, and losses. The SLP-A systems were generally at least, if not more, effective than the Fixed-7M system, except for SLP-A-20M with 32 and 40 threads, which had slightly worse effectiveness. Note also that the SLP-A systems tend to become less effective as more threads are added. This is due to the value of ρ being divided and distributed across each thread, enforcing early stopping more often. We remind the reader that approximate processing can actually improve effectiveness. For example, consider the Perc systems; processing only 20% of the candidate postings results in 21 queries outperforming the exhaustive system, but also results in 27 queries performing worse than the exhaustive system.

Efficiency Analysis. First, we examine the baselines presented in Table 4.5. The Fixed-5M configuration, based on the recommended value of c from Lin and Trotman, was able to meet the efficiency SLAs quite easily. In addition, the linear prediction model was quite accurate in predicting that processing 7 million postings is possible within the given budget, as the Fixed-7M model was also able to meet both SLAs. Conversely, the percentage based systems violate both

System	Time [ms]				NDCG@10	
	Mean	P_{50}	P_{95}	P_{99}	Mean	% W/T/L
1 Thread						
Exhaustive	309.9	270.4	736.5	973.6	0.159	-/-/-
Fixed-5M†	103.8	120.7	129.7	132.1	0.153	11/74/15
Fixed-7M†	135.2	156.3	184.6	189.6	0.154	8/81/11
Perc-20†	69.3	56.5	171.2	231.0	0.150	21/52/27
Perc-40	134.2	106.5	350.4	486.4	0.154	15/67/17
Perc-60	167.6	132.3	423.2	593.9	0.157	12/77/11
Perc-80	260.3	222.8	625.0	842.6	0.159	7/87/6
8 Threads						
SLP-E	207.9	186.1	414.6	630.4	0.159	-/-/-
SLP-A-100M	187.3	190.5	354.1	423.1	0.159	2/96/2
SLP-A-50M	179.6	182.5	328.1	379.1	0.158	4/92/4
SLP-A-20M	153.5	167.8	241.4	255.3	0.156	8/84/8
24 Threads						
SLP-E	146.8	151.1	251.7	316.0	0.159	-/-/-
SLP-A-100M	138.5	147.1	227.7	257.4	0.158	2/95/3
SLP-A-50M	136.8	147.7	214.9	243.7	0.157	4/91/5
SLP-A-20M†	120.4	135.6	164.5	172.2	0.156	9/81/10
40 Threads						
SLP-E	134.9	139.3	216.3	273.3	0.159	-/-/-
SLP-A-100M†	128.8	137.5	199.3	229.6	0.158	3/94/3
SLP-A-50M†	124.9	135.5	183.6	207.3	0.157	5/89/6
SLP-A-20M†	109.7	120.3	150.6	169.4	0.153	9/78/13

Table 4.5: Time and effectiveness trade-offs for the various approaches. † denotes that a system did not violate the SLAs. W/T/L corresponds to the percentage of Wins/Ties/Losses with respect to the exhaustive system.

SLAs with the exception of Percentage-20. Next, we examine the efficiency of both the selective parallelization and the aggressive selective parallelization systems while varying the number of worker threads. As expected, selective parallelization is able to accelerate the exhaustive processing. For example, the rank safe SLP-E systems are between 1.5× and 2.3× faster than exhaustive processing when considering the mean latency, and 1.5× to 3.5× faster when considering the 99th percentile latency, *with no effectiveness loss*. However, all of the SLP-E variants violated the service level agreements, as they could not effectively control the tail latency of the postings traversal. Fortunately, the SLP-A methods were generally able to meet the SLAs, while achieving a higher

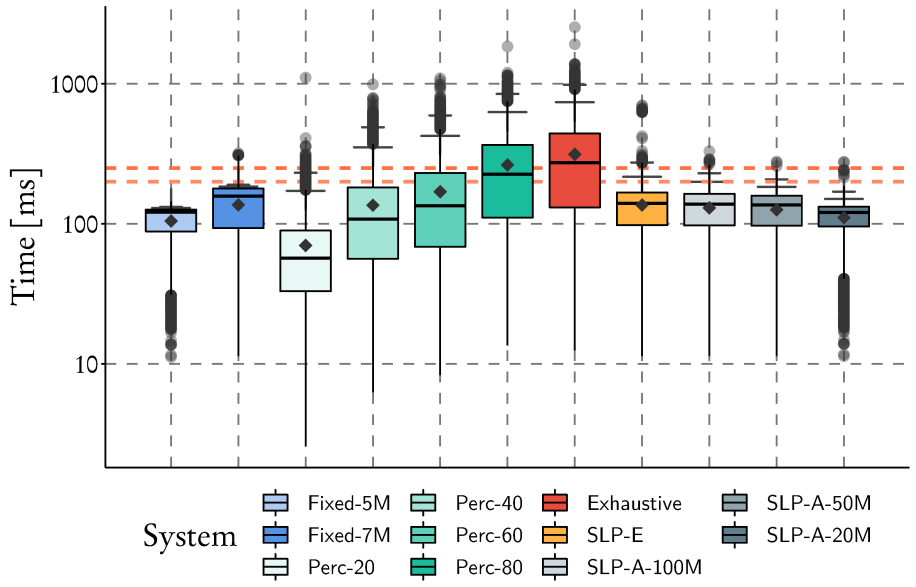


Figure 4.13: Efficiency characteristics of the baseline and parallel approaches for all 5,682 queries. Dashed horizontal lines denote the P_{95} and the P_{99} SLA targets. For each system, the P_{95} and the P_{99} values are denoted by a long and short horizontal bar, respectively. Threaded systems are shown using the maximum number of allowed threads (40).

effectiveness than the fixed systems. For convenience, Figure 4.13 summarizes the timings for the tested systems, including the mean, P_{50}, P_{95}, and P_{99} latency.

Based on our findings, the best approach for efficiently processing queries with more postings *without* sacrificing considerable effectiveness is to use selective parallel processing, which can keep latency low while improving effectiveness over fixed-parameter systems.

Scalability for Large Candidate Sets. As a final experiment, we run our multi-threaded implementation of JASS exhaustively across all queries, retrieving the top-10,000 documents for each query, using 8, 16, 24, 32 and 40 threads. The aim of this experiment is to understand the scalability of the multi-threaded implementation when retrieving large sets of candidate documents, as is necessary in modern, multi-stage retrieval systems [105, 192]. In comparing the runs which retrieve the top-10,000 documents to those which retrieve the top-10 documents, we find that the large increase in k only results in increases of between 5.5ms and 10.3ms when considering the mean and median latency. This makes sense given that the same amount of work was conducted in processing the postings lists (which dominates the processing time) regardless of k. Hence, the parallel approach scales to any arbitrary value of k.

4.3.5 Discussion

We now discuss the findings and implications from our experimentation with various early termination heuristics and parallel processing.

Processing Many Postings. While using a constant value of ρ allows for finely controlled latency, it can have a negative impact on the effectiveness of queries with a large number of postings. To remedy this issue, we explored how parallel processing can be used such that a larger number of postings can be evaluated while keeping within a competitive time budget. To keep resource usage low, selective parallelization only parallelizes queries that are predicted to exceed the processing budget [119, 129]. We showed that these ideas, previously explored within DAAT processing frameworks, can be easily extended to SAAT systems.

Efficiency vs Consistency. In designing our parallel SAAT algorithm, a number of design choices were made that seem possibly counter-intuitive. The first was to do away with the efficient accumulator initialization technique [120] in favor of a single, iterative, initialization loop. The second involved removing the top-k heap, and instead traversing the entire accumulator table to find the top-k documents. The key aspect of these decisions is to trade efficiency for consistency. For example, clearing the entire accumulator table at the beginning of processing was shown to incur an overhead of around 13ms, and traversing the entire accumulator table at the end of processing was shown to incur an overhead of around 71ms. Hence, we would not expect a query to finish in under roughly 85ms, assuming it had at least one posting to process. However, the variance in the efficiency of both operations does not depend on the input query, meaning that we have much more control over the latency of query processing.

Scalability. Another important consideration for SAAT systems is the *size* of the collection. Since their efficiency and effectiveness depends mostly on the number of postings that are processed, one simple way to improve SAAT traversal is to limit the size of the search shard by distributing the collection across multiple servers [20, 131]. In doing so, the average *proportion* of postings lists that can be traversed increases, while allowing the search process to remain efficient.

4.4 Discussion and Conclusion

Efficient top-k search is a fundamental problem within information retrieval. Perhaps the most ubiquitous application of top-k search in modern IR systems is within multi-stage search, where a set of candidate documents must be efficiently identified and returned for a more expensive and accurate ranker to operate on. In this chapter, we investigated the use of both document-ordered and impact-ordered index organizations in this context. While a large amount of prior research has focused on improving the efficiency of DAAT and SAAT search systems, this is the first study that compares the two methods directly in a fair way. This comparison was facilitated by first removing the many confounding factors within each of the query processing frameworks, including the

compression algorithm, the scoring computation, and the processing of the underlying index, resulting in an experimental framework that was as comparable as possible. Furthermore, our analysis focuses on tail-latency, looking beyond commonly examined summary statistics.

From experimentation across three commonly used collections, we identified a range of factors that influence the efficiency modern DAAT dynamic pruning algorithms such as WAND and BMW, as well as early termination within SAAT search, as implemented by the JASS system. For example, the efficiency of the DAAT algorithms is influenced by the size of the collection, the number of terms within the input query, and the size of the desired results set, irrespective of whether rank safe or aggressive traversal is used. Aggressive traversal improved the mean and median latency with little effectiveness loss, but was unable to improve the tail latency of these approaches. Exhaustive SAAT retrieval was generally not competitive with respect to latency, as its run time depends directly on the number of candidate postings to process. However, exhaustive SAAT does guarantee rank safe results if the underlying quantization is of sufficient resolution [66], as it was in our experiments. On the other hand, the efficiency of the early-termination SAAT approach is only impacted by the number of postings traversed, which is a system parameter. By processing fewer postings, retrieval latency can be minimized with a potential loss in effectiveness. These observations answer RQ1.1.

Further investigation into approximate SAAT retrieval heuristics showed that although the tail can be controlled well for the fixed-cost heuristics, this can lead to effectiveness loss for queries with a high number of candidate postings. Dynamic heuristics that select ρ on a per-topic basis were shown to provide a better control on system effectiveness, but were less effective at controlling latency. To achieve the best of both worlds, a parallel SAAT algorithm was outlined, which is able to process many postings segments at once, resulting in competitive efficiency and effectiveness (RQ1.2). Taking this organization further, the notion of selective parallelism was applied, where only queries that were predicted to take longer than a pre-defined time budget would be parallelized [119]. Results across the large ClueWeb12B collection showed that combining selective parallelism with aggressive early termination allowed for efficient query processing while keeping the overall effectiveness high, answering RQ1.3 in the affirmative.

There is a wealth of future work to be conducted on the many topics explored in this chapter. Since the time of this study, the state-of-the-art in both DAAT and SAAT retrieval has been advanced through a number of recent works [172, 173, 174, 197, 243]. It would be interesting to conduct another comparison study to determine if the observations from this study are impacted by these innovations. Another interesting question is how the candidate generation stage can go beyond simple conjunctive or disjunctive matching, perhaps exploiting *soft Boolean conjunctions* [210] or mixed-mode processing for enhanced efficiency/effectiveness trade-offs. For DAAT systems, improving the control of the tail latency would be a very useful focus for future research [167]. As discussed, document-ordered indexes are much more flexible than impact-ordered indexes as they can support additional Boolean matching criteria, can be efficiently updated, and the underlying ranking function is not required to be static. These problems are yet to be solved for impact-ordered indexes.

5 Efficiently Processing Query Variations

Efficiently processing bag-of-words queries is an important requirement for supporting efficient end-to-end search in large-scale IR systems. Chapter 4 focused on this problem in particular, under the assumption that the input query was the exact query that the system would execute. However, it has been shown that there is a high variability in the queries that different users generate for the *same* information need [186]. For example, consider Table 5.1, which shows the most commonly submitted *user query variations* (UQVs) for an information need where users must try to identify a *Norway Spruce* tree [18]. Even for such a simple information need, there are clearly a multitude of ways to formulate a reasonable query.

Prior work has focused on using query variations to improve the quality of search. Most of these works have employed *rank fusion* techniques to merge a set of results lists arising from processing a set of query variations. For example, Belkin et al. [21] found that combining independently generated Boolean variations increased the effectiveness of their search system, arriving at a general rule of thumb: the more lists fused, the better. Similar findings were also reported in recent work from Bailey et al. [19], who showed large improvements to ad-hoc search quality by using rank fusion across top-k results lists arising from disjunctively processing query variations. Benham and Culpepper [23] showed that rank fusion across query variations and systems can improve effectiveness with a relatively low risk of significant harm with respect to a baseline system, indicating that fusion across query variations is a simple yet robust approach for improving search effectiveness. Other combinations of query variations, rank fusion, and search systems have been explored too, becoming the basis of a number of TREC runs in recent years [24, 26]. Clearly, leveraging rank fusion and query variations to improve search effectiveness is an attractive solution for search practitioners, especially when considering its simplicity and robustness.

In this chapter, we explore the implications of processing such groups of related query variations, known as *multiple queries* (multi-queries).[1] In particular, we focus on efficient solutions for computing and fusing a multi-query to generate a single, effective, ranked list of documents, and explore whether such approaches are efficient enough for real-time search.

[1] This term was first coined by Catena and Tonellotto [51].

Backstory: "Many tree species look alike, and so it is easy to mistake one type of tree for another. Your friend has said repeatedly that they have a Norway Spruce growing in their garden, but you don't think it is. You decide to try and identify whether the tree is what your friend says it is."

Query	Count	Percentage
Norway Spruce	19	17.6
Norway Spruce images	5	4.6
Norway Spruce trees	3	2.8
Norway Spruce tree	3	2.8
Norway Spruce picture	3	2.8
Norway Spruce identification	3	2.8
How to identify a Norway Spruce	3	2.8
How to identify Norway Spruce	3	2.8
Other	66	61

Table 5.1: An example of the variability of user queries for the same information need. This example shows the associated backstory for UQV100 topic 290 (above) and the most popular query variations submitted (below). A total of 108 query variations were submitted for this topic.

Problem Definition. Let Q denote a set of n unique queries, $Q = \{q_1, q_2, \ldots, q_n\}$, and let Fuse represent a rank fusion function which consumes an arbitrary set of ranked results lists $R = \{r_1, r_2, \ldots, r_n\}$, and outputs one single representative list r'. The aim of this problem is to efficiently compute the result of a given fusion function Fuse(R). There are two main components of the problem:

1. Processing each query $q \in Q$ efficiently to yield R, and
2. Efficiently fusing each $r \in R$ together to yield a fused list, r'.

The fusion component of the processing operation is expected to be extremely efficient; we are given a set of n ranked lists, each containing no more than around $k = 10{,}000$ elements, and we must traverse each of these lists to fuse the rankings together. On the other hand, the query processing component could be costly; as we have seen in the earlier chapters, it can be difficult to efficiently process even a *single* query within a reasonable time. The major problem then becomes how to process each query in Q such that the overall latency is minimized. To complicate matters further, the rank fusion process has a data dependency on the query processing stage, which may lead to the fusion operation being delayed by *stragglers* — slow running queries from Q.

Our Contributions. In this chapter, we study the problem of efficiently processing multi-queries as formulated previously, focusing on the following research question:

RQ2: *What is the most efficient way to process multi-queries over inverted indexes?*

We also examine the following sub-questions:

RQ2.1: *What are the inherent trade-offs of the multi-query processing algorithms?*

RQ2.2: *Can multi-queries be processed efficiently enough for real-time search?*

Since we are the first to study this problem in modern retrieval architectures, we propose a number of approaches for solving this problem based on different processing paradigms. One such approach involves computing each query in parallel, and then conducting a fusion step at the end of processing, denoted as *Parallel Fusion* (PF). We show how PF can be implemented in both DAAT and SAAT search engines, and explore the trade-offs between both approaches. While PF is shown to be efficient with respect to latency, this approach must use a large amount of system resources, and accesses the same postings lists multiple times. In order to reduce the CPU cost, we propose a family of novel multi-query processing algorithms, denoted *Single Pass* (SP), that can leverage DAAT dynamic pruning algorithms to efficiently traverse all candidate postings *once*, reducing the amount of redundant accesses that must be made to the postings. Furthermore, we extend the SP approach for SAAT systems which allows for efficient *early termination*.

We conduct an experimental analysis across the ClueWeb12B collection and the associated UQV100 topics, demonstrating a range of trade-offs that are possible from employing our approaches. In particular, we find that we can achieve low latency at a high CPU cost through the use of the PF approach. On the other hand, we show that our novel SP methods are able to achieve significant CPU cost reductions with respect to the parallel approaches, at the cost of a small increase in latency (RQ2.1). In any case, we conclude that certain instances of our proposed algorithms are suitable for use in real-time scenarios (RQ2.2).

5.1 Related Work

The problem presented in this chapter draws on many aspects of indexing and retrieval. In particular, we refer the reader to Sections 3.1 and 3.2 for the overview of index architectures and query processing across both DAAT and SAAT indexes. Here, we briefly review concepts related to rank fusion and query variations.

5.1.1 Rank Fusion

As discussed previously, the goal of rank fusion is to take an arbitrary number of ranked results lists $R = \{r_1, r_2, \ldots, r_n\}$ and output a single representative list r'. Each list r consists of ranked documents, whereby a document d in the ith list has a relevance score of $\mathcal{S}_{i,d}$. We now discuss the core idea behind rank fusion, and how common fusion techniques combine information to produce enhanced rankings.

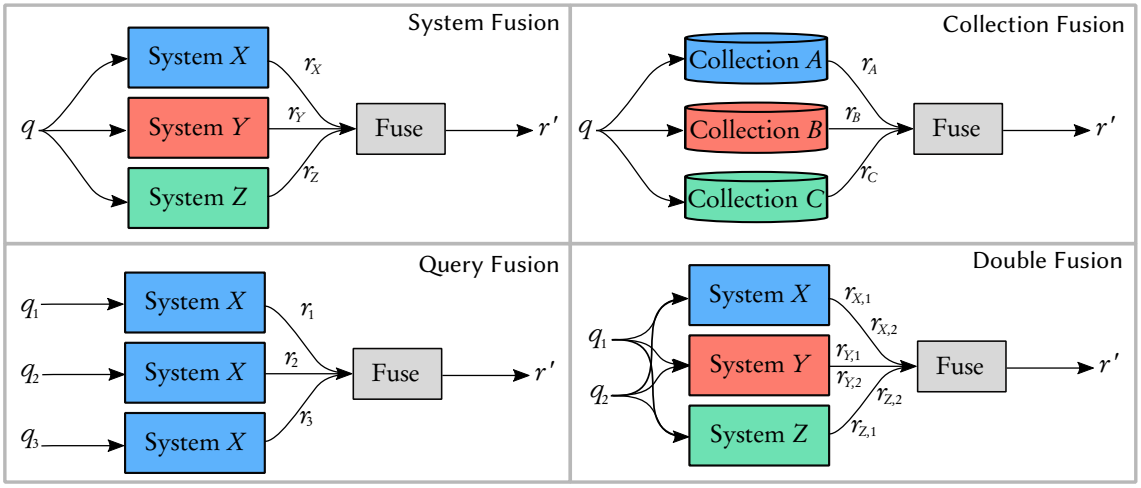


Figure 5.1: Four types of rank fusion. On the top left, a single query is processed on three unique systems to generate ranked lists for fusion (system fusion). On the top right, a single query is processed across three different collections and the results lists fused together (collection fusion). On the bottom left, three unique input queries are ran on a single system, and the results fused together (query fusion). On the bottom right, two unique input queries are provided to three different systems to generate six lists which are then fused together (double fusion). Note that other combinations can be used for double fusion, and *triple fusion* is also possible.

Sources of Evidence. Rank fusion aims to generate a ranking based on *combined evidence* with the aim of improving both the robustness and the effectiveness of the final ranked list. Indeed, a large amount of prior work has studied this problem [19, 65, 95, 116, 136, 146, 189, 190, 225, 251, 261]. For information retrieval systems, there are a multitude of sources of evidence for ranking. Prior work has examined rank fusion across *systems*, *collections*, *queries*, and combinations thereof [139, 261]. System fusion involves processing a given query q across a set of different query processing systems, resulting in a set of results lists to be fused. The idea behind system fusion is to leverage different ranking formulations to build evidence of document relevance. Collection fusion is a less commonly used approach, where a given query is issued to the same system searching across a set of different collections. Query fusion involves submitting a set of queries to a single IR system, and fusing the resultant output lists together. Different combinations of system, collection, and query fusion can be leveraged together [19]. Figure 5.1 shows a pictorial example of each approach, with an example of *double fusion* across queries and systems. For this chapter, we are focused entirely on query fusion. Thus, we use the term rank fusion interchangeably with query fusion.

Fusing Ranked Lists. The key hypothesis behind fusing multiple lists is that the resultant list should be more effective than using any of the constituent lists individually. Improvements in results from rank fusion can been explained through a few different effects [84, 250]:

- *Skimming:* different retrieval runs retrieve different relevant items, so a fusion approach that takes the top ranking items from each constituent list will push irrelevant items down the ranking.
- *Chorus:* different retrieval runs retrieving the same documents indicates that these documents are relevant, and should be ranked highly.
- *Dark Horse:* a given retrieval run may be much more effective than other runs for at least some of the relevant items.

Interestingly, many of the ideas behind rank fusion were founded well before search engines even existed. In the past, an analogous problem to rank fusion was that of devising a technique to combine votes during an election such that the most preferred candidate would be elected. The *Borda count* method was one approach for solving this problem. Although the idea was independently devised several times, Borda count was named after Jean-Charles de Borda, who discussed the idea around 1770 [79]. Centuries later, this approach was extended for use in IR systems as the BordaFuse algorithm [15, 79].

Broadly speaking, there are two families of rank fusion algorithms. *Unsupervised* fusion algorithms do not depend on training data, whereas *supervised* algorithms do [116, 190, 225]. For this work, we are interested in only *unsupervised* fusion techniques due to their simplicity and computational efficiency. Within unsupervised fusion there are two families of algorithms [23, 97], namely rank-based and score-based.

Rank-Based Fusion. Rank-based fusion approaches utilize only the *rank* of each document within each input list to form the final fused list. To this end, rank-based fusion algorithms can be considered as *voting* algorithms. Indeed, many rank-based fusion methods have origins from problems surrounding the aggregation of votes for an election, including BordaFuse and Condorcet [264]. BordaFuse is a very simple method, which assigns a score to a document d as:

$$\mathsf{BordaFuse}(d) = \sum_{i=1}^{n} \frac{k_i - p_{i,d} + 1}{k_i}, \tag{5.1}$$

where k_i is the depth of the ith list, and $p_{i,d}$ is the rank position of d in the ith list [79]. Intuitively, the BordaFuse method applies a linear discount to the weight assigned to each successive document down the ranking. Other approaches such as *Reciprocal Rank Fusion* (RRF) use a *reciprocal* distribution to apply a super-linear discount to documents further down the ranking [65]. A similar idea is used in the recently proposed *Rank-Biased Centroids* (RBC) approach [19], which employs a *geometric decay* on the contribution of each document down the ranked list. Using an increasing discount places greater emphasis on documents that are higher in each $r \in R$ under the assumption that such documents should carry a higher weight, aiming to balance both the chorus and skimming effects.

Score-Based Fusion. Score-based fusion methods use the actual document scores within each input list to yield the final fused list. Some of the most simple yet effective score-based fusion approaches were proposed by Fox and Shaw [95], and are commonly known as the *Combination* (Comb) algorithms. Many versions were proposed, including CombMIN, CombMAX, CombMED, CombSUM, and CombMNZ. CombMIN and CombMAX assign document d the minimum (or maximum) of the scores observed for d across all $r \in R$. While seemingly counter-intuitive, both algorithms were designed to prevent special cases; CombMIN avoids ranking a non-relevant document highly, while CombMAX tries to minimize the number of highly-relevant documents being ranked low in r'. However, as observed by Fox and Shaw, both algorithms solve a specific corner case with little regard to the average case. CombMED, which sets the score of document d to the median score observed for d across all $r \in R$, aims to remedy this issue.

A major issue with the previously mentioned approaches is that they rely on the score of a document from a *single* ranked list to determine the final ranking. On the other hand, CombMNZ and CombSUM use all of the scores present in each list to generate the final output ranking. CombSUM simply calculates the score for d as the sum the value of $\mathcal{S}_{i,d}$ across each ranked list $r \in R$, where $\mathcal{S}_{i,d}$ is the score of document d in the ith list (or $\mathcal{S}_{i,d} \equiv 0$ if $d \notin r_i$):

$$\mathsf{CombSUM}(d) = \sum_{i=1}^{n} \mathcal{S}_{i,d} \,. \tag{5.2}$$

CombMNZ performs the same computation as CombSUM, but multiplies the value of the CombSUM score by the number of lists in R in which d was present, providing higher weights to documents that occur in many ranked lists, which biases the result to favor the chorus effect:

$$\mathsf{CombMNZ}(d) = |d \in R| \sum_{i=1}^{n} \mathcal{S}_{i,d} \,. \tag{5.3}$$

In their studies, Fox and Shaw [95] found CombSUM and CombMNZ to be the best performing fusion approaches. More recent studies also show good performance from the CombSUM and CombMNZ approaches [19, 23], making them appealing due not only to their performance, but also to their efficiency and simplicity.

Score Normalization. All of the aforementioned methods have assumed a constant weight for each $r \in R$, meaning that each *input list* is considered to be equally important to the final ranking. However, this may not always be desirable. For example, fusion across *systems* may incorporate both strong and weak systems, and so the weight given to stronger systems should be higher than the weaker systems. This is of particular importance for balancing the dark horse effect with the chorus effect. Fortunately, a simple approach based on *linear interpolation* [261] can be used, whereby each list r_i is associated with a weight α_i. While this approach works with both rank- and score-based fusion techniques, there is a subtle issue with score-based fusion techniques; the individual scores *within* each list may have different magnitudes, and hence may implicitly

weight the final fused list. Indeed, this is the case for most score-based fusion techniques, even in the absence of a per-list weighting scheme as outlined above. To avoid this issue, the scores within each list can be *normalized* prior to the fusion operation. The most commonly used normalization approach is the MinMax approach [189]. Given a list r with a maximum score $\mathcal{S}_{\text{Max}}$ and a minimum score $\mathcal{S}_{\text{Min}}$, the normalized score of document d can be computed as

$$\text{MinMax}(d) = \frac{\mathcal{S}_d - \mathcal{S}_{\text{Min}}}{\mathcal{S}_{\text{Max}} - \mathcal{S}_{\text{Min}}}, \tag{5.4}$$

where $\mathcal{S}_d$ denotes the *unnormalized* score of d. Thus, after normalizing a list with MinMax, the scores of the documents will be linearly normalized within the range $[0, 1]$.

We refer the interested reader to the work of Wu et al. [261], Benham and Culpepper [23], and Kurland and Culpepper [139] for a more detailed overview of rank fusion.

5.1.2 Query Variations

Query variations are simply different ways of expressing a query for the same information need. In a study from 1988, Saracevic and Kantor [220] noted that there was a low degree of agreement in both the search terms used, and consequently, the documents retrieved, when a set of users were asked to formulate and execute search queries for a set of given tasks. Interestingly, this study also showed that the more commonly an item appeared in the results lists for different users, the more likely it was to be relevant – the same observation that drives rank fusion. Of course, there is certainly a relationship between query variations and rank fusion: fusion algorithms commonly use query variations as a means to produce a set of ranked input lists. Recent work has confirmed that users are as diverse as systems [186], meaning that the input query can have as large-an effect on the result ranking as using a different IR system. Therefore, query variations are a very important consideration for production IR systems. In particular, questions of *result consistency* and *system robustness* arise when expressing an information need with different query variations; it is desirable for a system to produce similar results for similar queries, and the performance of the system should not vary wildly for similar queries. Bailey et al. [19] examined this problem by using *rank fusion* to generate a consistent ranking across a set of query variations, and then measuring other systems against the fused run. Consistency was found to vary between topics and systems, with consistency generally decreasing as the complexity of a topic increases. Nevertheless, leveraging query variations and rank fusion seems to be a promising approach for improving search robustness beyond single queries [74].

Leveraging Query Variations. Most early work on query variations involved combining them with rank fusion to improve search effectiveness [21, 22]. Recently, a renewed interest on query variations has emerged, which is possibly encouraged by the creation of the UQV100 collection, which contains query variations for a set of 100 topics over the large ClueWeb12B web collection [18]. For example, rank fusion across query variations was revisited for various submissions to the TREC CORE track [24, 26]. Other recent work has examined how the best performing

query variation can be selected via query performance prediction [221, 238], how retrieval systems can be compared in the presence of query variations [275], and how fusion over query variations impacts the *risk* of retrieval systems [23].

Generating Query Variations. One of the lesser studied aspects of query variations is how they can be generated. Most of the aforementioned studies rely on *real users* to generate the query variations. For example, the UQV100 collection was created by first generating a so-called *backstory* for each of the 100 topics. The backstory would outline an information need, and crowdworkers were asked to generate a set of query variations for each [18]. A similar approach was used in the TREC CORE runs from Benham et al. [24, 26], where the TREC topic description was used as the backstory. Another interesting approach for generating query variations is to ask crowdworkers to generate a query variation based on an *image* or *video*. Stanton et al. [232] used this approach for generating user queries that various describe health problems depicted in either an image or a short video clip displayed to the crowdworker. While these techniques are robust in general, they require direct human interaction.

Benham et al. [25] use a sampling-based system to automatically generate query variations based on relevance modeling over external corpora. This approach involves inducing relevance models for a given query from a set of external corpora, which can then be sampled iteratively to generate random bag-of-words queries. Then, these bag-of-words queries are processed concurrently on the target collection and fused using RRF. Their results showed this approach to be competitive with a well-tuned RM3 query expansion run [1] at a lower cost.

A more realistic method of mining *real* query variations is presented where user query- and click-logs are available. For example, any reasonably popular commercial search engine will have a huge amount of query and click data at their disposal, and this data can be easily used to mine or generate similar queries. The core idea is to build a representation (usually a bi-partite graph) between user input queries and clicked documents, and use this representation to discover clusters of queries that result in clickthroughs to common or similar documents [257]. This method relies on the assumption that similar queries will result in users clicking on related documents. Various methods of using this data for mining query clusters or variations have been explored, including using random-walks [69], weighted bi-graph clustering [135], and co-visitation algorithms [262]. Other approaches have explored fusion of results from query reformulations [225].

Properties of Query Variations. Because query variations pertain to a common information need, they often vary only slightly from one another. Consider Figure 5.2, which plots the percentage of variations containing the most common 60 terms for each topic in the UQVRobust query variations [23, 24], and in the UQV100 collection. On average, the most popular term in each topic will appear in almost *all* submitted variations, with a long-tail of terms appearing in just one or two submitted queries. A concrete example for the UQV100 collection can be observed in Table 5.1, where the terms *Norway* and *Spruce* appeared in each of the most popular 8 submitted variations. In fact, *Norway* appeared in 99, and *Spruce* in 103 of the 108 submitted variations for that topic.

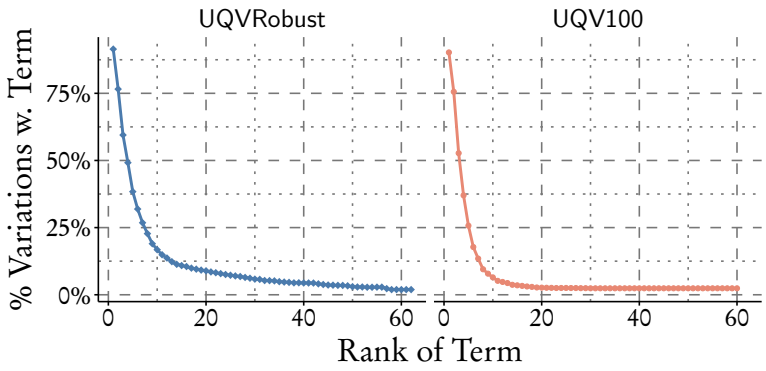


Figure 5.2: Average percentage of UQVs that terms of varying popularity appear in, averaged on a per-topic basis, across both the UQVRobust and UQV100 query variations. Clearly, information needs generally have a few central terms that appear in most variations, and a long tail of rare terms appearing in very few query variations.

5.2 Efficient Multi-Query Processing

Given the proven effectiveness of rank fusion across query variations [18, 21, 23, 24, 26], we now propose a few different approaches for solving the multi-query processing problem. Each approach takes a multi-query Q, containing n unique queries, as input, and returns a ranked list r'. Our solutions aim to be as efficient as possible, and should be suitable for use where strict service level agreements must be met, such as in commercial web search engines [82].

5.2.1 Parallel Fusion

The most obvious approach is to simply spawn a single thread for each $q \in Q$, and execute all of these (or as many as possible) in parallel. Once each thread returns its corresponding results list, the rank fusion algorithm can be applied to assemble the final results list (Figure 5.3).

Assuming that sufficient CPU cores are available, this method should remain efficient. However, the latency for processing Q will be roughly bounded by the *longest running* query in Q, making it vulnerable to high tail latency. Even worse cases can exist for this approach: assume that we have a machine with m CPUs, and $n > m$. In this case, the latency is not bounded by the slowest query in Q, but rather, by the slowest execution path taken by a single processing core. To be more concrete, let $m = 2$ and $n = 5$. Let us also assume that the latency for processing each query is the same. Then, the execution path would be as follows:

- **Epoch 1:** CPU 1 $\rightarrow q_1$, CPU 2 $\rightarrow q_2$.
- **Epoch 2:** CPU 1 $\rightarrow q_3$, CPU 2 $\rightarrow q_4$.
- **Epoch 3:** (CPU 1 or CPU 2) $\rightarrow q_5$.

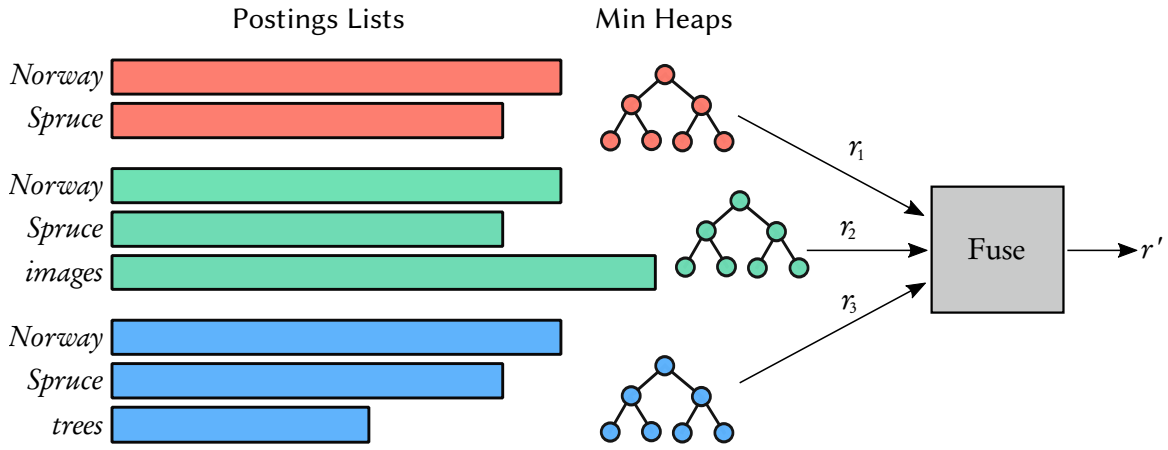


Figure 5.3: An example of the PF technique for a size 3 multi-query. Each query inside the multi-query is processed independently, in parallel, and each result list is then passed through to the Fuse algorithm, yielding the final results list, r'.

In this toy example, the total latency for processing Q is actually 3× larger than processing any single $q \in Q$, which is undesirable for real-time search scenarios. One obvious solution to this problem is to process only up to m queries. Then, latency is guaranteed to be bounded by the slowest query in the set that was processed. However, the problem of selecting a subset of queries to process from a multi-query is a non-trivial problem that is analogous to query performance prediction and the *model-selection* problem [3, 206, 238]. This problem has implications on both the efficiency and the effectiveness of PF, and is left for future work. Another potential solution for latency reduction in PF is to run the fusion algorithm on a partial set of result lists once a given time budget is exceeded, at the cost of a loss in effectiveness. Indeed, a deeper *scheduling* problem exists, since the processing time for each query is not known ahead of time, and is not likely to be uniformly distributed. We do not focus on this problem, and it is left for future work.

Since this approach does not assume any particular index traversal algorithm, we refer to specific instantiations of the *Parallel Fusion* approach as PF-*a*, where *a* corresponds to the index traversal algorithm used (such as WAND, for example). However, since SAAT approaches involve *early termination*, we explore two SAAT approaches. The first, denoted PF-JASS-E, processes *all* postings for each candidate query. The second, PF-JASS-A, will process a maximum of 10 million postings per query ($\rho = 10^7$).

5.2.2 Single Pass DaaT

One disadvantage of the PF approach is that it traverses many of the same postings lists multiple times, as each query may contain similar terms. Consider Figure 5.2, which plots the percentage of query variations that contain the most to least popular terms for the given information need across two separate sets of query variations. As discussed earlier, there appears to be a subset of *popular* terms which appear in many of the submitted variations, with a long tail of other terms

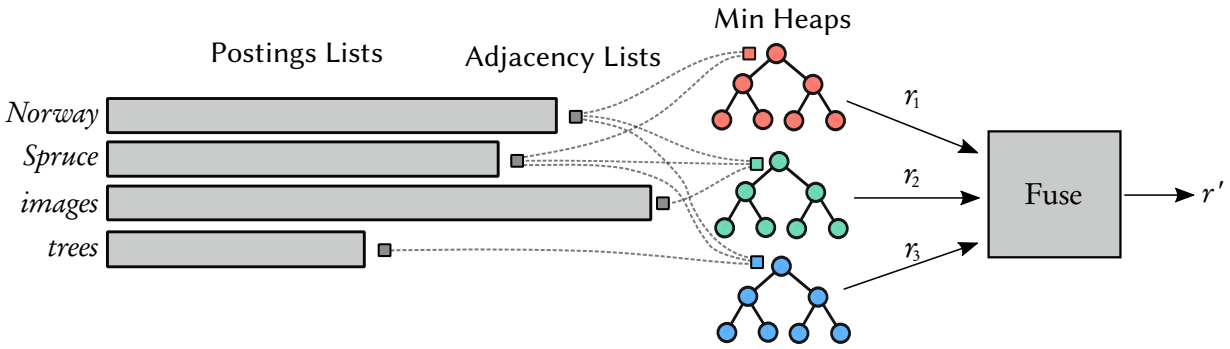


Figure 5.4: An example of the exhaustive SP technique for a size 3 multi-query. Each postings list is traversed simulataneously in a DAAT manner, with scoring taking place for each relevant query. A heap for each query is stored and updated independently. Once all postings lists are exhausted, each result list is then passed through to the Fuse algorithm, yielding the final results list, r'.

that occur in only a single submitted query. Therefore, algorithms which can avoid accessing each posting multiple times should be cheaper and more efficient. We now focus on designing algorithms that limit the number of redundant postings operations.

The first alternative we propose is the exhaustive Document-at-a-Time *Single Pass* algorithm (SP-Exhaustive). Unlike PF, SP-Exhaustive iterates the postings lists for all candidate terms just *once*, utilizing an exhaustive DAAT traversal on all postings lists concurrently. First, a cursor is opened on each of the postings lists in Q. The *pivot* document is the minimum document ID across all of the cursors. At this point, the pivot document is scored, with a score being computed *once* for each term contained in the pivot, and the appropriate scores being accumulated for each of the unique input queries. In order to store the top-k documents for each query, a unique min-heap is built for *each* query $q \in Q$. An adjacency list is associated with each postings list, storing a pointer to each top-k heap that is associated with this list, allowing only the relevant heaps to be updated after a scoring operation. Figure 5.4 shows this organization for a 3 query multi-query, where the first two terms appear in all 3 queries, and the third and fourth terms appear in the second and third queries, respectively. After all postings lists are exhausted, the results lists are fused together to create r'.

While this approach is guaranteed to score each unique term-document pair just once, it does not leverage dynamic pruning. To this end, the PF algorithms are able to *skip* documents that are unable to make it into the respective top-k results, leaving an obvious question: is it better to process many postings lists multiple times with skipping enabled, or to exhaustively process each postings list just once? We answer this question in the following experimental analysis. However, an approach that could leverage dynamic pruning while also only processing each postings list once is a seemingly 'best of both worlds' solution. We now explore the possibility of such an algorithm.

5.2.3 Single Pass CombSUM

As discussed above, it is desirable to process each list just once while also avoiding computing the score of documents that are unable to make the top-k results. Modern dynamic pruning algorithms rely on *additive rankers* to establish upper-bound scores which can be used to determine whether a new document should be evaluated. The monotonic nature of scoring allows such estimations to be made (Section 3.2.4). Recall that the CombSUM fusion function is also strictly monotonic if the lists being fused have positive numeric scores. Using this observation, we now outline how we can combine dynamic pruning with the single pass algorithm, thus creating a new approach for processing multi-queries with CombSUM fusion, known as *Single Pass CombSUM*, or SP-CS for short.

For convenience, we reproduce Equation 5.2 here. Given a set of ranked lists with positive numeric scores for each document in each list, $R = \{r_1, r_2, \ldots, r_n\}$, the CombSUM score for a document d can be computed as the sum of the scores of d, computed over its appearances in each $r \in R$. Assuming the score of some document d in the ith of the lists is given by $\mathcal{S}_{i,d}$, and $\mathcal{S}_{i,d} \equiv 0$ if $d \notin r_i$, then

$$\mathsf{CombSUM}(d) = \sum_{i=1}^{n} \mathcal{S}_{i,d}\,.$$

If the scoring computation that generated the component scores $\mathcal{S}_{i,d}$ for a given query q_i is *additive*, then

$$\mathcal{S}_{i,d} = \sum_{t \in q_i} \mathcal{W}(d,t)\,,$$

where $\mathcal{W}(d,t)$ is the term-document score contribution for the term t in the document d according to the chosen retrieval model. Taking these together gives

$$\mathsf{CombSUM}(d) = \sum_{i=1}^{n} \left(\sum_{t \in q_i} \mathcal{W}(d,t) \right). \tag{5.5}$$

Now let $c_t \equiv |\{q_i \mid 1 \le n \le \ell \wedge t \in q_i\}|$, the number of input queries containing term t. Equation 5.5 can thus be rewritten as

$$\mathsf{CombSUM}(d) = \sum_{t \in Q} c_t \cdot \mathcal{W}(d,t)\,, \tag{5.6}$$

making it clear that for additive similarity scoring mechanisms, the CombSUM score for a set of query variations can be computed by counting term frequencies c_t across the variations, and then evaluating a single *super-query* against the index of the collection by taking a linear sum of the individual contributions of the ranking function $\mathcal{W}$. This approach only works for similarity models that are additive and yield positive scores, as is the case for most dynamic pruning algorithms.

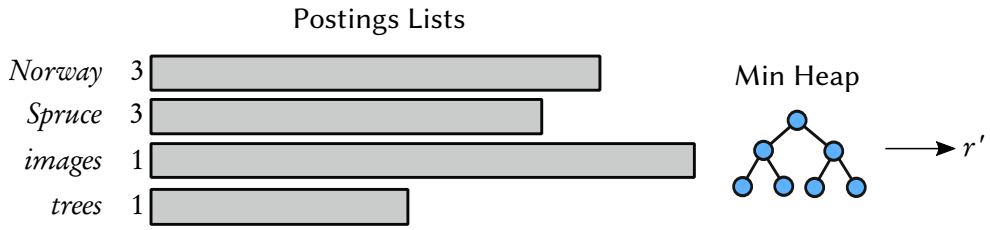


Figure 5.5: An example of the SP-CS technique for a size 3 multi-query. The multi-query is transformed into a single *super-query*, where each term has its weight multiplied by the number of times it was observed in the multi-query. Then, standard dynamic pruning algorithms can be used to compute the CombSUM fusion across the super-query, resulting in the final fused list, r'.

Document-at-a-Time Processing. In order to employ Equation 5.6 to apply DAAT dynamic pruning to the super-query, the index traversal process must be slightly modified. For each candidate term, t, the upper-bound score U_t must also be multiplied by c_t. This ensures that the bound estimation for dynamic pruning is correct with respect to the current ranking. For the block-based approaches, the c_t multiplier must be applied to each block-max score, $U_{b,t}$, as each block is encountered. Query processing does not differ in any other way to the regular bag-of-words algorithms – a single top-k heap is maintained during processing, and contains the result of both the *query processing* and the *rank fusion* stages once the algorithm terminates. Figure 5.5 shows an example of this organization.

Score-at-a-Time Processing. Interestingly, Equation 5.6 can also be applied within a SAAT search system. Recall that in a SAAT index, the value of $\mathcal{W}(d,t)$ is pre-computed, quantized, and stored as an *impact* ($i_{d,t}$). Thus, when traversing the index, each impact can be multiplied by c_t to generate the modified CombSUM impact. Traversal of the index is the same as in the standard bag-of-words SAAT traversal, and at the end of processing, the top-k heap (and subsequent accumulator table) will contain the fused results list.

Score Normalization. As mentioned in Section 5.1.1, CombSUM does not require an explicit score normalization step when the underlying results lists come from the *same* source [95]. This is of particular importance to the SP-CS approach, as normalization would require each query to be processed independently, which defeats the benefit achieved through Equations 5.5 and 5.6. Indeed, one pass fusion would not be possible as described here, since the full results list for each query would be required to depth k before any normalization can be applied. Hence, we do not use any score normalization in our algorithms. In practice, normalization is important for the fusion of ranked lists arising from *different* rankers [23, 27], but is less important when fusing lists based on the same ranking function. Indeed, incorporating normalization into the SP algorithms would allow for alternative rank fusion algorithms to be deployed, but is a surprisingly difficult problem which should be explored further in the future. We conduct a preliminary exploration on the impact of score normalization in Section 5.4.1.

Rank Safety. Another interesting aspect of SP-CS is that it allows for a truly rank safe ranked list to be generated by CombSUM. Usually, fusion algorithms will assemble a fused list from a set of input lists that are generated to some depth k. In this situation, there is no score contribution being generated by documents that exist at any position $\geq k+1$ in any of the lists, as they are inferred to have a score of zero. In contrast, the SP-CS method computes an exact solution of r', mirroring the outcome of giving CombSUM each candidate ranking r to an arbitrary depth. We note however that the rank safety of the SAAT methods depends on processing the candidate postings *exhaustively*. If early termination is employed, rank safety is not guaranteed.

Similar to the PF algorithm, the SP-CS algorithm is generalizable to all index traversal strategies. Hence, we denote each instantiation as SP-CS-a, with a representing the index traversal algorithm. Again, since the SAAT algorithms can use early termination to improve efficiency, we distinguish between two SAAT approaches: SP-CS-JASS-E processes all postings for a given query, whereas SP-CS-JASS-A processes a maximum of 10 million postings for a given query ($\rho = 10^7$).

5.3 Experimental Setup

Hardware. Experiments were performed on an otherwise idle Red Hat Enterprise Linux Server with 512 GiB of RAM and two Intel Xeon E5-2690 v4 CPUs. Each CPU has 14 cores with hyperthreading enabled, resulting in a total of 56 processing units. Indexes are stored entirely in main memory, and are warmed up prior to experimentation, which involves accessing each candidate postings list prior to processing. Timings are the average of 3 independent runs.

Software. Algorithms were implemented using C++11 and were compiled with GCC 7.3.1. Where multiple threads were used, up to 56 threads were spawned using C++ STL threads. The DAAT algorithms were implemented within the state-of-the-art VBMW codebase [173] and the SAAT algorithms were implemented within the JASS codebase [148]. Our extensions are made publicly available at `https://github.com/RMIT-IR/centroid-boost`.

Document Collection and Indexing. The experiments in this section use the ClueWeb12B corpus and the UQV100 query variations and judgments. ClueWeb12B contains close to 52 million web documents, crawled in 2012. We use Indri to index the collection, apply Krovetz stemming and the default Indri stoplist, and then convert the Indri index into the format expected by the VBMW codebase. Next, we reordered the postings lists using the state-of-the-art recursive graph bisection approach [83, 171]. We then build our VBMW index, setting the λ parameter such that we had a mean block size of 40 elements per block [173], and compress the index using the Partitioned Elias-Fano mechanism [191]. A JASS index was also built using the *same* base index after quantizing the index with 512 quantization levels following best practices [66].

Queries and Relevance. To generate a set of reasonable multi-queries, we employ the UQV100 collection [18]. UQV100 provides a set of *query variations* across a set of 100 single-faceted topics,

Documents	52,343,021
Topics	100
Total queries	10,835
Unique queries	4,175
Mean queries per topic	90.1

Table 5.2: The ClueWeb12B and UQV100 resources used. Note that the queries were stopped and stemmed, reducing the number of distinct queries for each topic.

and relevance judgments for these topics on the ClueWeb12B corpus. In total, 10,835 queries were submitted in total from 263 crowdworkers, where 5,764 of these variations were unique. Due to using both stopping and stemming, this total is reduced to 4,175 unique query variations in our experiments. Table 5.2 summarizes some basic statistics concerning the test collection. Ranking was done using the BM25 similarity function as outlined in Section 2.2, and CombSUM [95] is used as the fusion algorithm. Where search effectiveness is measured, we employ two *early-precision* metrics, namely NDCG@10, and RBP with $\phi = 0.8$.

5.4 Preliminary Experiments

5.4.1 Normalization and Effectiveness

Our first experiment explores the question of whether score normalization has a positive impact on the effectiveness of CombSUM fusion. Earlier, we argued that score normalization is not required in our multi-query processing algorithms because the underlying similarity function, BM25, is the same across all ranked lists. However, consider the BM25 computation as shown in Section 2.2. Since the computation of BM25 is additive across each term in the query, the BM25 function is *unbounded*, meaning that BM25 can give *infinitely large* scores as the length of the query tends to infinity. This observation means that, in general, *longer* queries will have a larger contribution to the CombSUM ranking, as long queries are more likely to result in documents with higher scores. A toy example that exhibits this problem is presented in Table 5.3.

In order to test this conjecture, we select a number of *pairs* of queries from each multi-query, fuse them together using CombSUM both with and without normalizing the individual results lists first, and then record the effectiveness of the fused list. Normalization was conducted using the simple MinMax scaling approach as shown in Equation 5.4 (Section 5.1.1). Pairs of queries are selected such that one of the *shortest* and one of the *longest* unseen queries are selected, randomly. After each iteration, the next-shortest (and next-longest) unseen queries are added to the fusion operation from the previous step, without replacement. Table 5.4 shows the results. As the number of pairs increases, the results improve irrespective of whether normalization was applied. However, we do note that for all cases other than when just a single pair of queries were fused, normalizing the input runs results in improved effectiveness. Furthermore, the *uncertainty* in the ranking is lower when normalization is applied, which indicates that there is less noise in

	Ranked list r_1		Ranked list r_2		Fused list r'	
Rank	DocID	Score	DocID	Score	DocID	Score
1	A	2.8	B	9.2	B	11.8
2	B	2.6	D	7.3	D	9.4
3	C	2.4	A	6.1	A	8.9
4	D	2.1	C	4.1	C	6.5
1	A	1.00	B	1.00	B	1.71
2	B	0.71	D	0.63	A	1.39
3	C	0.43	A	0.39	D	0.63
4	D	0.00	C	0.00	C	0.43

Table 5.3: An example of the CombSUM rank fusion procedure across two ranked lists, r_1 and r_2, showing the resultant fused list r'. On the top, no normalization is applied. On the bottom, the same input lists are MinMax scaled prior to fusion. In this example, the two final rankings differ, since r_2 contributes more to the final ranking than r_1 when normalization is not applied.

the ranking. Thus, it seems that query length may bias CombSUM if score normalization is not considered. However, we note that the effect observed on the UQV100 collection is not very large, which is most likely because the lengths of individual query variations for each topic are quite uniform. Hence, the bias is likely to be higher when there is more variance in the length of the queries within each multi-query. However, we do not have data to analyze this phenomenon, and leave further investigation for future work.

5.4.2 Query Log Caching Effects

Since many of the query variations contain similar terms, there may be caching effects which artificially improve query processing efficiency. Thus, before evaluating the efficiency of the proposed algorithms, we first wish to quantify the impact of the *order* of the input queries to determine if we observe any cache effects that may skew our results. To do so, we took the UQV100 query log, and we generated three permutations; the default order of the log, a random ordering of the log, and a sorted version, where we lexicographically sort all query variations. Next, we process the queries, one at a time, using the VBMW algorithm, for various values of k. Each query batch is ran three times, and the per-query average is taken. The results are shown in Figure 5.6. Although we observed some slight efficiency improvements for the default and sorted query logs, these improvements were negligible with respect to the total processing time. For example, the largest difference in mean latency observed was 7ms. Similar results were observed for the SAAT traversal algorithms. Thus, we conclude that the query ordering will not impact our findings going forward.

	NDCG@10		RBP$\phi = 0.8$	
Pairs	Unnormalized	Normalized	Unnormalized	Normalized
1	0.178	0.168	0.406 (0.133)	0.364 (0.129)
2	0.213	0.226	0.491 (0.089)	0.505 (0.042)
3	0.229	0.243	0.521 (0.063)	0.547 (0.031)
4	0.239	0.256	0.521 (0.051)	0.554 (0.020)
5	0.234	0.259	0.518 (0.049)	0.550 (0.018)
10	0.248	0.268	0.535 (0.037)	0.569 (0.007)
All	0.263	0.277	0.547 (0.015)	0.569 (0.005)

Table 5.4: CombSUM fusion effectiveness with and without score normalization. The values in parentheses are RBP residuals, recording the maximum extent of the RBP score uncertainty.

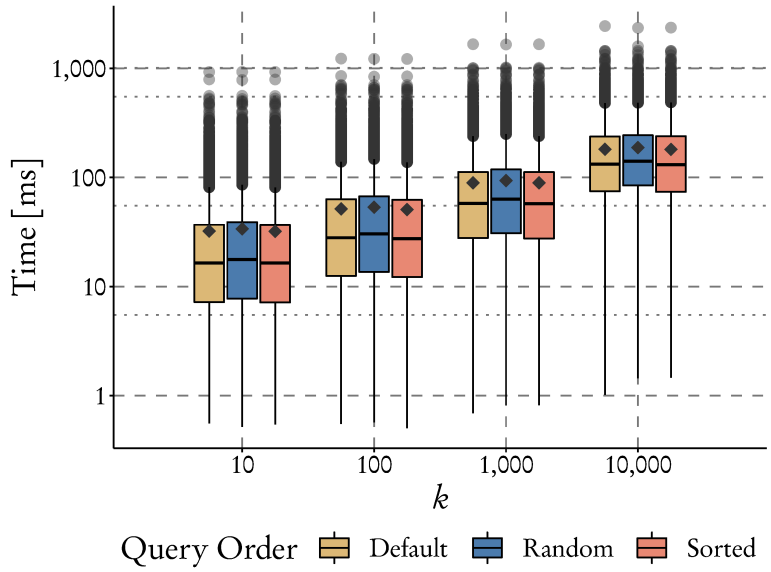


Figure 5.6: The efficiency of query processing with respect to the query log order, for various values of k. Caching of terms has a very small impact on efficiency.

5.5 Experiments

5.5.1 Real-Time Fusion Performance

Our first main experiment aims to examine how efficiently the proposed algorithms can process multi-queries. We take all of the query variations for each of the 100 topics in the UQV100 collection, and generate 100 multi-queries, each with an average size of $|Q| = 42$ (Q is a set of 42 unique variations). Then, we measure the cost of computing the fused result set, r', to a depth of 100 documents. In order to compute r', the representative results list from each $q \in Q$ is

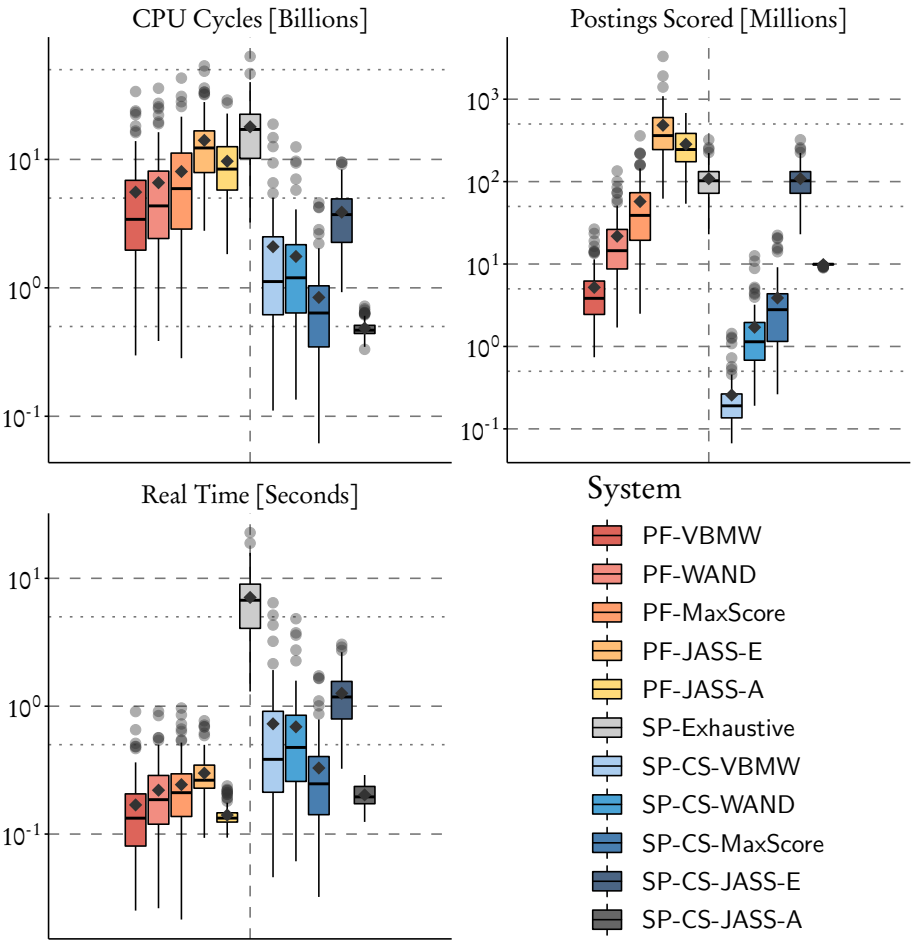


Figure 5.7: Efficiency of rank fusion, assuming that the starting point is a multi-query. The panes show the total cost in CPU cycles; total cost in terms of postings scored; and the query latency, in seconds.

generated to a depth of $k = 1{,}000$ documents, and is then fused using CombSUM (except in the case of the SP-CS algorithms, who compute the fusion as a part of the index traversal). Three performance indicators are measured and reported: the number of CPU cycles consumed; the number of postings scored; and the response latency. In the case of the parallel approaches, we compute the total CPU cycles and the postings scored by taking the sum of these values across each thread. We also ignore the CPU cost associated with managing the parallel processes, such as any `fork()`, `join()`, or `lock()` operations.

Figure 5.7 shows the results across each measure. The first pane, on the top left, shows the total CPU time required for each of the algorithms. The SP-Exhaustive approaches exhibits a larger CPU cost than all other approaches, indicating that dynamic pruning or early termina-

tion is a highly important aspect for efficiently processing multi-queries. The PF methods are generally more expensive than the SP-CS methods, with SP-CS-MaxScore the most cost effective DAAT approach, and SP-CS-JASS-A the most cost effective SAAT method. Interestingly, using MaxScore in the SP-CS framework yields the most cost effective DAAT solution, whereas using MaxScore in the PF framework results in the least cost effective DAAT solution. We explain this phenomenon shortly. Unsurprisingly, both approximate SAAT approaches outperform their exhaustive counterparts.

The second pane, on the top right, shows the number of postings scored. Clearly, using VBMW within either the PF or SP-CS algorithms results in a significantly lower number of postings evaluated. MaxScore, although the most cost effective DAAT approach (with respect to CPU cycles within each processing family), actually scores more postings than the respective WAND or VBMW algorithms. The explanation for this is that the WAND and VBMW algorithms reduce the number of postings scored at the expense of more non-posting related bookkeeping, which becomes expensive when processing multi-queries. For the SAAT methods, the aggressive JASS approaches process only 10 million postings per query. However, since the PF approaches process each query individually, the postings cost is much higher for PF-JASS-A than for the corresponding SP-CS-JASS-A approach. The redundant processing of postings can also be observed by comparing the exhaustive SAAT methods: PF-JASS-E processes every postings list for every term in the multi-query, whereas SP-CS-JASS-E processes every postings list for every *unique* term in the multi-query.

Finally, the third pane (bottom left) shows the elapsed wall-clock time. Generally, the PF approaches are the fastest, due to their extensive use of parallel processing. Even though each multi-query has its execution time dictated by the slowest-running execution path, execution on average is fast, especially considering the workload undertaken by processing such large multi-queries. However, this efficiency comes at a large CPU cost, as shown in the first pane. Again, both of the aggressive JASS algorithms outperform the exhaustive versions. We note, however, that these aggressive approaches may lose some effectiveness with respect to the ground truth ranking, since they rely on early termination. We explore this trade-off further in Section 5.5.3. Interestingly, we see that deploying MaxScore within the SP-CS framework results in a faster execution time than using either WAND or the state-of-the-art VBMW algorithm. This can be explained by the bookkeeping process undertaken by WAND-like algorithms. In particular, WAND-like algorithms *sort* their respective postings lists by the document identifiers underneath each cursor before each round of pivot selection (Section 3.2.4). When processing multi-queries, these sort operations become very expensive, especially as the number of postings lists increases. On the other hand, the MaxScore algorithm does not require the postings to be sorted, as it does not rely on postings to be ordered, but rather utilizes *essential* and *non-essential* lists for pivot selection (Section 3.2.4). This aligns with recent work that showed the performance of MaxScore to be better than the state-of-the-art VBMW method when result sets are large, or queries are long [175].

Algorithm	Mean	P_{50}	P_{95}	P_{99}
PF-VBMW	168.9	132.9	466.2	654.4
PF-WAND	220.3	186.4	500.9	844.5
PF-MaxScore	243.4	210.0	545.1	854.8
PF-JASS-E	299.3	263.4	588.8	711.1
PF-JASS-A	140.8	133.1	210.2	223.4
SP-Exhaustive	7,083.8	6,716.1	14,678.9	18,824.3
SP-CS-VBMW	727.7	358.1	1,915.9	5,155.5
SP-CS-WAND	689.5	469.5	1,582.9	3,747.0
SP-CS-MaxScore	328.5	246.0	870.3	1,659.9
SP-CS-JASS-E	1,261.1	1,175.1	2,549.3	2,893.7
SP-CS-JASS-A	202.4	195.2	258.8	283.7

Table 5.5: Summary of the latency of the multi-query processing algorithms, in milliseconds. The PF approaches have the best latency across the board, with PF-JASS-A being the best choice for ensuring a low tail latency.

5.5.2 Tail Latency in Real-Time Fusion

We now explore how the different algorithms compare when tail latency is considered. We refer to the *service level agreements* (SLAs) discussed in Section 4.3.4 as a guide for acceptable tail latency [119, 129, 170, 265]. Those SLAs were set to be realistic, and provide a strict time budget on the tail latency of query processing:

- 95th percentile (P_{95}) $\leq$ 200ms, and
- 99th percentile (P_{99}) $\leq$ 250ms.

For convenience, we tabulate the mean, median (P_{50}), P_{95}, and P_{99} latency in Table 5.5. Of all the tested algorithms, the PF approaches retain their advantage with respect to latency, with the fastest values across the board. The PF-JASS-A approach is the fastest of all the proposed multi-query processing methods, with a 95th percentile latency of 210ms and a 99th percentile latency of 223ms. Although this approach violates the proposed P_{95} SLA by around 10ms, it is likely to be fast enough for consideration in real-time search scenarios. One way in which this approach could be made more efficient is to further relax the number of postings to score with a potential loss in effectiveness. We also remind the reader that the PF algorithms use a large number of threads, and hence, CPU cycles. On the other hand, the single-threaded SP-CS-JASS-A method can deliver results within 260ms at the 95th percentile, and 285ms at the 99th percentile, while using an order of magnitude less CPU resources than the PF methods.

No. Variations	NDCG@10	RBP$\phi = 0.8$
1	0.212	0.502 (0.033)
2	0.239	0.541 (0.010)
5	0.247	0.536 (0.017)
10	0.263	0.548 (0.021)
20	0.268	0.550 (0.014)
50	0.267	0.547 (0.014)
All	0.263	0.547 (0.015)

Table 5.6: Fusion effectiveness as the number of query variants is increased. Variants are selected from the query clusters such that the most commonly submitted variations are taken first. The values in parentheses are RBP residuals, recording the maximum extent of the RBP score uncertainty.

5.5.3 Efficiency and Effectiveness Trade-offs

As seen in the previous experiment, the DAAT approaches are competitive with respect to both mean and median latency, but do not have the same control over tail latency that the SAAT approaches have. However, the fastest SAAT approaches sacrifice some effectiveness for efficiency. Furthermore, the previous experiment considers processing multi-queries made up of *all* available variations. However, efficiency may be improved by processing multi-queries made of just a subset of the available variations. To explore these subtle trade-offs further, we generate a new set of multi-queries. Following the methodology of Bailey et al. [19], we generate a multi-query from the most popular (up to) v submitted variations for each topic, $v \in \{1, 2, 5, 10, 20, 50, \text{all}\}$. We use this set of multi-queries in the following two experiments.

Multi-Query Size vs Effectiveness. First, we examine the impact on effectiveness as the number of queries included in each multi-query is increased. For each of the multi-query sizes, we computed the CombSUM rankings using the rank safe SP-CS method. Table 5.6 shows the results. Similar to prior literature [19, 21], we observed that adding variations improves effectiveness, but the improvements diminish as more variations are included in the fusion.

Efficiency vs Effectiveness. Since effectiveness improvements diminish as variations are added, we are interested to see how the proposed systems differ in both efficiency and effectiveness as the number of input variations increases. In particular, it seems that by processing a smaller multi-query, the various methods may have improved efficiency without losing considerable effectiveness. To examine this trade-off, we plot the median, 95th, and 99th percentile latency with respect to the resultant NDCG@10 score for the proposed systems as the size of the multi-query is increased. Figure 5.8 shows the results, plotting only the most competitive systems for clarity.

While the DAAT methods (SP-CS-VBMW, SP-CS-MaxScore and PF-VBMW) are efficient when the size of the multi-query is small, they do not scale as effectively as the SAAT methods.

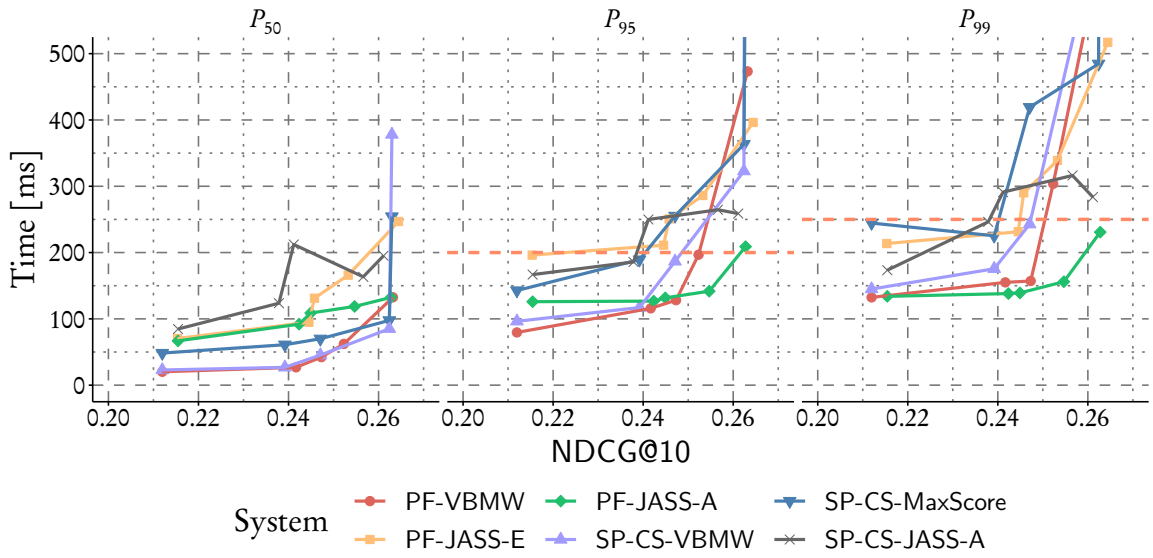


Figure 5.8: Efficiency/effectiveness trade-off for various multi-query processing algorithms. Time is shown for the median (P_{50}), 95th percentile (P_{95}) and 99th percentile (P_{99}) latency, and effectiveness is calculated with NDCG@10. SLAs are denoted by horizontal dashed lines.

This is intuitive, as the aggressive JASS based systems specify a hard limit on the cost of processing. Indeed, the PF-JASS-A approach gives the best trade-off with respect to efficiency and effectiveness. However, there are circumstances where using multi-core processing is not desirable. For example, when query load is high, there is unlikely to be sufficient resources to facilitate processing a query on multiple threads [34, 117]. In these circumstances, the SP-CS-JASS-A method is the best choice. Using only a single thread, this method can still achieve a competitive tail latency, while matching the effectiveness of the rank safe methods.

5.6 Discussion and Conclusion

In this chapter, we investigated three approaches for solving the problem of efficiently processing multi-queries, whereby a set of query variations must be processed efficiently with their results being fused into a single ranked list. The PF and SP-Exhaustive methods can be applied to any rank fusion algorithm, but are more costly than the SP-CS approaches. However, the improvements in cost from SP-CS come at the cost of constraints on both the ranker and the fusion method, which must be additive. We found that the PF methods use almost an order of magnitude more CPU cycles than the respective SP-CS methods, making them less appealing when the computing cost is considered. On the other hand, the PF algorithms shine when keeping latency low is important, with the best representative processing multi-queries within reasonable time bounds for real-time search, thereby answering RQ2.1, and confirming RQ2.2. The most efficient approaches were

those that used SAAT index traversal with early termination. These methods were always the fastest, and had a comparable effectiveness to the rank safe approaches, making them difficult to beat in practice, thus answering RQ2. We also confirmed that processing more query variations generally leads to an improved effectiveness, and showed that score normalization can improve the performance of CombSUM fusion.

There are many avenues for future work on this topic, some of which are already being explored. For example, Catena and Tonellotto [51] focus on alternative strategies for condensing multi-queries into representations that are efficient to process. In particular, they focus on *query factoring*, which involves converting the input multi-query into a more compact yet equivalent representation, which can be used to process sub-queries in *conjunctive* mode, and fuse the results in *disjunctive* mode. Another interesting extension to the problem of processing multi-queries is how the fused output list can be used to improve ad-hoc ranking beyond returning just the fused results list. Benham et al. [27] focus on this problem, assuming that the multi-queries are processed *offline* and cached in a suitable data structure, and propose a number of novel solutions that blend fused results lists with a list that is more closely related to the direct information need of the user. Another related line of work on evaluating complex Boolean expressions [93] may provide new techniques for fusion over query variations. Other open questions include how to select the most effective query variations when building multi-queries, how normalization can be included in the fusion of multi-queries, and how to optimally schedule the processing of a multi-query across many threads.

6 Reducing Tail Latency via Query Driven Algorithm Selection

The competing goals of maximizing both the efficiency and the effectiveness of large-scale IR systems continues to challenge both academic and industry search practitioners. So far, we have explored the causes of tail latency, and methods of improving latency, in the candidate generation stage of cascaded search architectures for both singular bag-of-words queries (Chapter 4) and sets of related queries (Chapter 5). However, these studies ignored the subsequent cost of both feature generation and re-ranking which are important (and possibly expensive) components of the canonical two-stage retrieval system [252]. In fact, most works that study multi-stage retrieval assume that the candidate generation process will generate a *large, fixed* number of candidates, irrespective of the input query itself. Recently, Culpepper et al. [75] made the observation that prior works have mostly focused on optimizing the size of the candidate pool, k, on a global basis. However, differences in query difficulty and information needs result in optimal values of k that vary on a query-by-query basis. Furthermore, the value of k directly impacts *efficiency* in all stages of the retrieval pipeline: as k increases, dynamic pruning algorithms are less effective [68, 196], features must be extracted for more documents [12, 13], and the LtR ranker must be applied on more feature vectors. However, the value of k also impacts the *effectiveness* of multi-stage retrieval: a larger value of k means that the *recall* of the candidate generation stage is potentially increased, which could give the LtR ranker more relevant documents to promote to the top of the final SERP. So, there is a delicate trade-off between efficiency and effectiveness in cascade rankers.

In this chapter, we focus on the problem of predicting the cutoff depth k for the candidate generation stage of a multi-stage retrieval system. Extending the work of Culpepper et al. [75], we cast this as a *regression* problem, and show how a *reference list* approach [61] can be used to accurately label training examples for building effective regression models. We then show how these ideas can be used for prediction of other efficiency/effectiveness parameters such as ρ, the aggression parameter for early-termination SAAT algorithms. Finally, we exploit these ideas together to form a *hybrid* processing pipeline which aims to improve both the tail latency of the

first phase candidate generation stage while simultaneously minimizing the value of k, without significantly impacting effectiveness. Experiments over the ClueWeb09B collection validate the efficacy of these methods.

Motivation. In most prior work on cascaded retrieval, the candidate generation stage has been assumed to generate a fixed number of candidate documents for all input queries. Macdonald et al. [164] outlined that a number of earlier works have suggested different values for k:

> "There is a great variation in the literature and existing test collections about how many documents should be re-ranked when learning models or deploying previously learned models, with large sample sizes such as "tens of thousands" [56], 5,000 [70], 1,000 [205], as well as small samples such as 200 [270] or even 20 [54] observed."

Furthermore, Macdonald et al. showed that small candidate samples can result in significant effectiveness degradations in the final re-ranked list, as the *recall base* is not sufficient.

Consider the following motivating example, where we are trying to optimize our end-to-end pipeline to minimize *cost* but maximize *effectiveness*. Let us define our target effectiveness metric $\mathcal{M}$ as

$$\mathcal{M}(r) = \frac{g(d_1) + g(d_2)}{2},$$

where $g(d_n)$ is the *utility* of the nth document in ranked list r, and

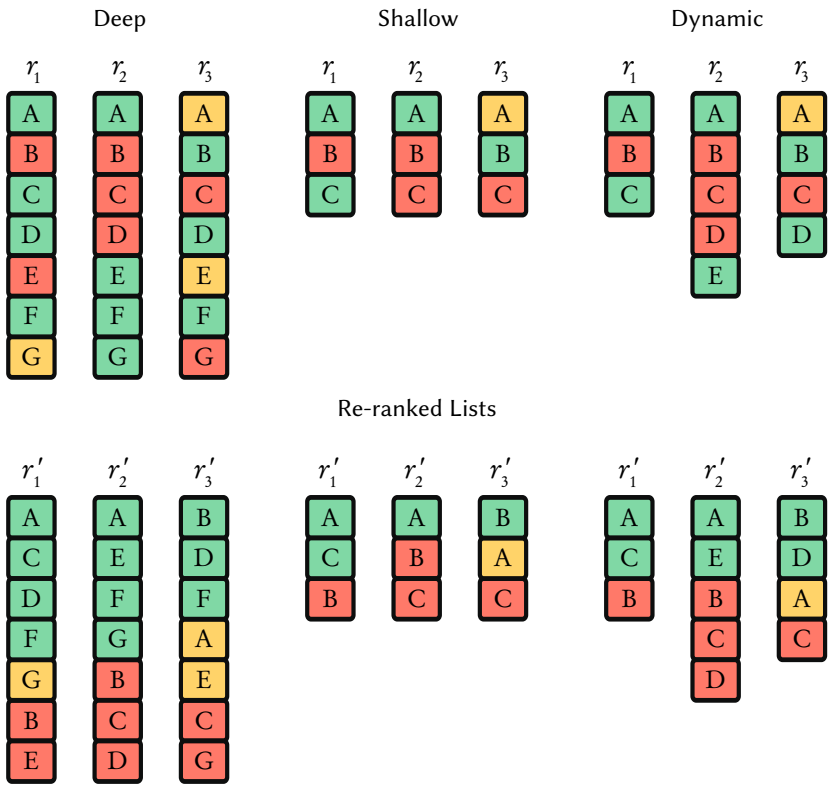
$$g(d) = \begin{cases} 1, & \text{if } d \text{ is highly relevant} \\ 0.5, & \text{if } d \text{ is marginally relevant} \\ 0, & \text{otherwise.} \end{cases}$$

Let the *cost* $\mathcal{C}$ of generating and re-ranking a set of candidates be defined as

$$\mathcal{C}(r) = |r|.$$

Furthermore, imagine that we had a *perfect* method of re-ranking documents, such that any input ranked list r could be permuted to create the best possible ranking, r', with respect to $\mathcal{M}$.

Now consider Figure 6.1 (a), which shows the candidate generation and re-ranked results for three different cutoff selection methods across three queries. The first approach uses a *deep* candidate pool for re-ranking. The second uses a *shallow* pool. The third method, which we advocate for in this chapter, selects the pool depth *dynamically*. Figure 6.1 (b) shows the subsequent effectiveness and cost achieved by each system, assuming that the candidate lists were re-ranked by the *perfect* re-ranker as described above. This example demonstrates the merit in predicting the k cutoff on a per-query basis; fixed samples that are large result in effective results but wasted resources, and fixed samples that are small are cheap but result in a loss of effectiveness.



(a) Candidate generation of three systems over the same three queries, followed by the re-ranked lists (as re-ranked by a perfect LtR system). Green documents are highly relevant, yellow documents are marginally relevant, and red documents are not relevant.

System	Average $\mathcal{M}$	Average $\mathcal{C}$
Deep	$\frac{1+1+1}{3} = 1$	$\frac{7+7+7}{3} = 7$
Shallow	$\frac{1+0.5+0.75}{3} = 0.75$	$\frac{3+3+3}{3} = 3$
Dynamic	$\frac{1+1+1}{3} = 1$	$\frac{3+5+4}{3} = 4$

(b) The average effectiveness and cost of the three systems.

Figure 6.1: A toy example of three different methods for selecting the k cutoff in a multi-stage retrieval system over three queries. Dynamic, which uses dynamic cutoff prediction, provides improved cost over Deep without a reduction in effectiveness. It also outperforms Shallow with respect to effectiveness, while only costing slightly more on average.

Problem Outline. The key problem being solved in this chapter is to predict, on a per-query basis, the number of documents that must be retrieved during candidate generation to satisfy a user, based on the output of a higher-order re-ranker. Given a bag-of-words query, q, and a

target evaluation metric $\mathcal{M}$ for the final ranked results list r', we wish to predict the required number of candidate documents that must be retrieved, k, such that the value of k is *minimized* while simultaneously maximizing the effectiveness of the re-ranked candidate set, $\mathcal{M}(r')$. This implicitly minimizes the cost, C, of feature extraction and re-ranking via LtR. A concurrent goal is to *minimize* the tail latency of the candidate generation process.

Our Contributions. We study the broad problem of efficiently predicting performance critical parameters for the candidate generation phase in a multi-stage retrieval system. To this end, we study the following primary research question:

> **RQ3:** *How can we predict performance sensitive parameters on a query-by-query basis to minimize tail latency during candidate generation?*

We also focus on the following sub-questions:

> **RQ3.1:** *What is the best way to use reference lists to accurately perform dynamic parameter predictions in early-stage retrieval?*
>
> **RQ3.2:** *How can we build predictors that minimize effectiveness loss while also minimizing cost?*
>
> **RQ3.3:** *How can our prediction framework be used to optimize tail latency during candidate generation?*

In order to build efficient and effective prediction models without relevance judgments or human annotations, we employ a *reference list* training framework that can be used to automatically generate training examples. Using this method, we show that models can be trained to effectively predict performance parameters that impact both efficiency and effectiveness on a query-by-query basis (RQ3.1). In order to achieve savings without negatively impacting result effectiveness, we employ a quantile regression method which allows us to generate risk-averse predictions (RQ3.2). We then apply these models to the *candidate generation* phase of a multi-stage retrieval architecture, and show how we can reduce the *tail latency* of this phase, while also implicitly improving the *cost* of downstream stages. In order to achieve these improvements, we experiment with a *hybrid* query processing architecture, where a broker node can decide whether to process the incoming query on a document-ordered or impact-ordered index. The advantage of this approach is that it allows us to achieve the *best of both worlds* — processing queries that require few answers via efficient DAAT algorithms, and processing those that require many answers via early-termination SAAT algorithms (RQ3.3). Our contributions are as follows:

- A unified framework for training and predicting performance-sensitive parameters for multi-stage retrieval systems.
- A highly tunable search architecture that leverages known performance characteristics to provide an improved efficiency/effectiveness trade-off for candidate generation in multi-stage retrieval.

- ✦ A general approach for allowing more fine-tuned optimization techniques that can be deployed on a query-by-query basis, and the tools necessary to implement and test systems that leverage this approach.

Finally, we discuss some of the nuances of our findings in this chapter, discuss some other applications of the ideas explored in this chapter, and provide some future directions for balancing trade-offs in large-scale IR systems.

6.1 Related Work

The problem explored in this chapter involves end-to-end processing on a two-stage retrieval system, where a candidate generation stage identifies and returns a set of likely relevant documents, which are then re-ranked via a more expensive LtR model. More information on index architecture and query processing can be found in Section 2.4 and Chapter 3. Here, we discuss aspects of query performance prediction, evaluation without ground truth labels, and training effective predictors.

6.1.1 Query Performance Prediction

Query performance prediction (QPP) is a classic problem within information retrieval which involves estimating effectiveness without having any relevance judgments [45]. As discussed by Raiber and Kurland [206], there are three major lines of work on QPP. The first involves generating pre-retrieval prediction methods that rely on only the input query and features of the underlying document corpus [3, 110, 111, 238]. The second line of inquiry allows the retrieved results list to be used for prediction, resulting in a richer feature space and enhanced methods of QPP [47, 73, 112, 214, 226, 271]. The third and final broad line of work focuses on the actual prediction framework itself, with various formalizations of the problem being explored, including probabilistic frameworks [140], distance-based measures [47], and neural approaches [266]. Recently, ideas from QPP have been adapted to other prediction tasks in IR, such as predicting the *efficiency* of a given query [162], which is analogous to the problem we study in this chapter. Since these types of prediction must be done *efficiently*, we focus on the ideas related to *pre-retrieval* QPP in this chapter.

Motivation. Variance in the robustness of search quality (and indeed, efficiency) can make it difficult for search systems to provide good results consistently [17, 19, 106]. Even high performing retrieval systems occasionally fail at returning relevant results for certain queries [23, 86, 87]. Hence, it is useful for a system to be able to make an estimation of how well it will perform for a given query *before* actually running it, as this would allow the system to trigger alternative actions depending on the outcome of the prediction. For example, if the predicted quality is low, the system could generate a new query that is expected to perform better than the original query, use an alternative ranking method, or add certain *cards* to the SERP [121], resulting in a better user experience.

Common Features. Since we are interested in pre-retrieval prediction, features must be derived from the collection or query itself rather than results lists arising from a retrieval run (as is the focus for post-retrieval QPP). To this end, many studies have discussed a wide range of simple features that can be generated based only on query terms and the underlying document corpus [3, 45, 111, 238]. Hauff et al. [110] conducted a small survey of such features, showing that the most powerful features depend on the given collection and the underlying ranker. Furthermore, these features are generally cheap, as they can be pre-computed offline for each term, and can be combined as a query is received.

Effectiveness Prediction. The main task in QPP involves predicting how *effective* the result for a given query q over a corpus D will be. Most prior works have focused on creating pre-retrieval QPP features and evaluating their predictive power, as discussed above. A major disadvantage of pre-retrieval QPP models is they are unable to take the retrieval method into consideration when making a prediction, as no information about the ranked list is available. To this end, *post-retrieval* QPP has been shown to make more robust predictions [45]. Recent work from Thomas et al. [238] suggests that pre-retrieval query performance predictors are actually correlated with topic effects, suggesting that they are actually operating as *topic difficulty* predictors. They argue that pre-retrieval predictors are essentially not useful for estimating *query* difficulty, and are only marginally better performing than a random guess. Zendel et al. [268] extend this idea further, showing how QPP can be improved by considering a *set* of possible queries that relate to the same information need, thus capturing the complexities of an information need. This is still an ongoing line of research. In other recent works, the idea of using pre-retrieval features beyond difficulty prediction has shown promise. We discuss some of these ideas shortly.

Applications of Pre-Retrieval QPP. Given a well performing QPP model, various actions can be triggered to improve the search experience. One such application of QPP is the notion of *selective query expansion*; pseudo-relevance feedback has been shown to improve the majority of queries, but significantly degrade the effectiveness of others [64]. QPP models have been exploited to selectively expand queries which are deemed to be improved via expansion [3, 113]. Similar methods have been explored for deciding when to interact with the user [138], selecting search engines in the context of meta-search [258], and selecting the best query variation to use for retrieval [221, 238]. The key theme in all of these approaches is *selectivity*, where an alternative method can be used depending on the outcome of the prediction.

We refer the reader with further interest in QPP to the tutorial and subsequent synthesis lecture from Carmel and Yom-Tov [45, 46], and the Ph.D dissertation of Hauff [109].

6.1.2 Query Efficiency Prediction

A similar problem to QPP is that of predicting the latency of a given query. In simple search engines, efficiency prediction can be quite trivial. For example, Moffat et al. [185] state that

> "The two chief determinants of system load in a text query evaluation engine are the number of terms to be processed, and the length of each term's inverted lists."

This has proven true for SAAT systems, where processing efficiency is correlated with the number of postings that are considered during processing, making efficiency prediction straightforward in practice [148, 170]. However, modern dynamic pruning algorithms, which use advanced processing mechanics, are less predictable [162, 239].

Latency Prediction for Improving Efficiency. Recent work has explored how the general ideas behind QPP can be used for latency prediction, especially for more advanced processing schemes. Tonellotto et al. [239] were the first to explore *query efficiency predictors*, which are inexpensive, pre-retrieval features. They show that their predictors are very effective across all query lengths, but only when combined using linear regression. A follow-on study shows how these predictors can be leveraged for improved scheduling within a large-scale IR system [162].

Other work focuses on reducing the tail latency of candidate generation in large-scale IR systems. Jeon et al. [119] employ selective parallelization to accelerate queries that are predicted to exceed a given time budget. Similar work was conducted by Kim et al. [129], who targeted *extreme* tail latency (at the 99.99th percentile). They showed that *dynamic* prediction features can improve latency prediction. These features are collected by processing a query for a small, fixed period of time (such as 5ms). During this time, features which depend on the actual query processing algorithm, such as how far through the candidate postings lists the cursors are, are collected. Since these features provide a deeper insight into the progress of query processing, the subsequent predictions were shown to be more accurate than using simple pre-retrieval features.

Similar ideas have been applied to *conjunctive matching*, where the fastest conjunctive algorithm can be selected on a query-by-query basis through cost modeling [130].

Latency Prediction for Improving Efficiency/Effectiveness Trade-offs. Broccolo et al. [34] examine a range of efficiency/effectiveness trade-offs that arise from processing queries with different levels of aggression, based on pre-defined time budgets. A simple prediction model is used to decide which instantiation of the index traversal algorithm is used to process the candidate query. Their work examines the real-world phenomenon of peak demand periods, where the workload of the search engine is much higher at particular times of the day. By trading off effectiveness for efficiency, they show that improved trade-offs can be made during peak loads without the need to scale up the search architecture.

Another related work attempts to balance the efficiency/effectiveness trade-off in cascaded rankers by predicting either effectiveness or latency before query processing, and selecting the parameters of the retrieval algorithm based on such predictions [240]. In particular, different configurations of WAND are explored, which are selected from a pre-defined set of WAND instances. Each instance has a different value of either F (the aggression parameter) or k (the number of candidate documents to retrieve), and the instance that is deployed is based on a prior efficiency or effectiveness prediction on a query-by-query basis. The authors found that using the

predicted efficiency to bound the latency had a better trade-off than using the estimated query difficulty to select the run-time configuration.

6.1.3 Optimizing Ranking Cascades

The problem of optimizing ranking cascades has received a lot of attention over the recent years. Since cascade ranking involves a sequence of stages, different approaches for optimizing these stages have been explored. Here, we give a brief overview of each major area of focus.

Candidate Generation. Most of the fundamental work on efficient top-k retrieval directly translated to candidate generation, as the underlying goal is very similar. A few works focused directly on examining the underlying trade-offs within candidate generation. For example, efficiency and effectiveness trade-offs between index traversal strategies and Boolean matching criteria was one major area of focus [10, 13, 61, 162]. While there is a wealth of prior work on top-k retrieval and, in turn, optimizing candidate generation, this was discussed primarily in Chapter 4.

Feature Extraction. Once a set of candidate documents is identified, the features for each document must be extracted efficiently such that the candidates can be re-ranked by the LtR model. Depending on the nature of and the number of the features, feature extraction can be very expensive. However, many features can be pre-computed and stored within a fast-access data structure, which enables rapid look-ups. On the other hand, some features such as query-dependent features are too expensive to store, and must be computed at query time. As such, a few different methods for extracting features have been examined previously.

Perhaps the most obvious way of generating query-dependent features is to *re-traverse* the inverted index for the documents in the top-k heap. However, this method is expensive, as it requires the candidate postings to be traversed again, potentially invoking decompression operations as candidates are accessed [241].

Macdonald et al. [165] outline how Terrier [161] generates features efficiently in what they call *fattening*. During candidate generation, a document that is admitted to the top-k heap has its matching postings cached within the heap. Thus, at the end of traversal, the term postings for all top-k candidate documents are available for feature computation. A disadvantage of this mechanism is that additional terms cannot be readily used, as only the matching terms from the query have their postings cached.

An alternative method to the postings-based approaches is to use a *forward index* or *document vector* representation, where the contents of a document can be accessed efficiently. Asadi and Lin [12] examined these organizations, including those that store *positional* information to efficiently compute phrase- and window-based features [155]. The clear disadvantage of this approach is that both the inverted index and the forward index must be stored, which results in a much higher space occupancy than the previously discussed methods. However, forward indexes come with many advantages. Firstly, they facilitate much more efficient feature generation, as documents can be accessed directly. Secondly, terms other than those contained in the input query can be

used for feature generation, making them much more flexible (for example, if query rewriting was to take place *after* candidate generation). Thirdly, forward indexes can be used to facilitate other operations efficiently, such as field-based scoring [105] and relevance modeling [25].

Document Ranking. Once a set of candidate documents have been identified, and their features extracted, the learned model needs to re-rank them. The most common LtR models use a *forest of trees* to produce rankings based on decision trees, and a large amount of prior work has focused on how to make faster predictions from such models [14, 78, 158]. Other works have focused on early-exit optimizations [43] and tree pruning [11] for speeding up evaluation. While optimizing the evaluation of learned models is an interesting problem in its own right, it is not directly related to the problem presented in this chapter. We refer the reader to the survey from Tonellotto et al. [241] for a more detailed discussion on these innovations.

End-to-End Cost Reduction. We have briefly outlined a range of improvements that have been made for improving the efficiency of the core stages in a cascade ranker. However, there are some improvements that translate to savings across the *entire* ranking cascade. Since both feature extraction and ranking on higher-order models are non-trivial operations, reducing the workload of these operations can result in large savings.

Culpepper et al. [75] address the problem of selecting the optimal value of k on a query-by-query basis in the context of a cascaded architecture. They argue that minimizing the value of k is of critical importance to multi-stage retrieval systems, as feature extraction and document ranking can be expensive. In order to predict a suitable value of k, they employ a classifier cascade that makes binary decisions on the sufficiency of a set of k values. Each value of k is tested, starting from small to large, and the value of k selected depends on the exit node of the classifier cascade [208]. To this end, the cascade could select values of k from a bag-of-values, and the classifiers within the cascade were trained to minimize the loss in end-to-end effectiveness. Hence, the cascade would minimize the value of k while also ensuring the results remain of high quality.

The other way of reducing the cost of feature extraction (and in turn, document evaluation) is to utilize only a subset of the available features at a given stage of the ranking pipeline. This was the motivation for the seminal work of Wang et al. [252], who proposed the notion of cascade ranking. The key idea is that, as the system advances into a more expensive ranking stage, more expensive features should be used to rank *fewer* documents, thus balancing the tension between efficiency and effectiveness. More recent work on this problem incorporates the feature cost into the learned models to more accurately balance this trade-off [57, 102].

6.1.4 Training Models with Reference Lists

A difficult problem that exists within many different contexts of information retrieval is how effective models can be trained in the absence of ground truth labels. While a breadth of related work has focused on using human interaction data such as click logs to label training instances [122], we focus only on the use of *reference lists* in this chapter.

List Comparison Methods. Before outlining the use of reference lists for generating labels, we must first examine approaches for *comparing* lists, as these form the basis of the labeling approach. Many list similarity measures have been explored in the literature, and they can be characterized into different categories which are suitable for different tasks. As described by Webber et al. [256], list similarity measures can be *unweighted* or *top-weighted*, and may require *conjointness* or may support *non-conjoint* rankings.

Unweighted comparison approaches assume that the elements within the lists being compared are all *equally important*, whereas top-weighted methods assign a higher weight to elements occurring in earlier ranks. Indeed, the notion of weight is important to ranked retrieval, as users typically examine documents in a top-down approach when presented with a results page. Conjoint comparison methods assume that the lists being compared contain the same elements. However, there are many cases where the input rankings are non-conjoint, and must be handled accordingly. In particular, ranked lists arising from search engines are typically truncated to depth k, which means the lists are often non-conjoint. To aid the following discussion, assume that we have two ranked input lists, r_A and r_B, and that we wish to compare the similarity between these lists.

An elementary method for computing the difference between r_A and r_B is to leverage Overlap:

$$\mathsf{Overlap}(r_A, r_B) = \frac{|r_A \cap r_B|}{|r_A \cup r_B|}. \tag{6.1}$$

However, Overlap is an *unweighted* measure, and therefore does not account for the fact that documents appearing higher in the ranking generally have higher importance as these are the first documents observed by the user.

Webber et al. [256] proposed *rank-biased overlap* (RBO), motivated by the lack of appropriate list similarity methods for ranked lists that are commonly found in IR tasks. In particular, RBO computes a *top-weighted* overlap between two possibly *non-conjoint* sequences, providing a much more robust way to measure list similarity for ranked retrieval. In order to set the weight for each position in the ranked list, a geometric probability distribution is assumed, which captures the *patience* of a user. The parameter ϕ is the probability that the user will *continue* to view the next position, so values of ϕ close to 1 represent a patient user (who will go deep in the ranking), whereas smaller values of ϕ represent an impatient user who assigns a very high weight to very few highly ranked documents. RBO can be computed as:

$$\mathsf{RBO}(r_A, r_B) = (1-\phi)\sum_{i=1}^{\infty} \phi^{i-1} \left[\frac{|\mathit{prefix}(r_A, i) \cap \mathit{prefix}(r_B, i)|}{i} \right]. \tag{6.2}$$

Taking this idea one step further, Tan and Clarke [235] propose the *maximized effectiveness difference* (MED) family of rank similarity methods. Given an effectiveness metric $\mathcal{M}$, and the two ranked lists r_A and r_B, MED assigns a set of pseudo relevance judgments such that the effectiveness

difference between r_A and r_B is *maximized*, and returns this difference:

$$\mathsf{MED}(r_A, r_B) = |\mathcal{M}(r_A) - \mathcal{M}(r_B)|. \tag{6.3}$$

Therefore, the *smaller* the value of MED between two lists, the less likely that a user would notice any effectiveness difference between such lists. Clarke et al. [61] further examined the correlation between MED and true relevance, and found that MED values below 0.20 usually indicated very little effectiveness difference between the input lists, especially for *early precision* metrics such as RBP with $\phi = 0.8$. While initially only suitable for utility-based metrics, Moffat [179] outlined how MED can be extended to recall-based metrics such as AP and NDCG.

Measuring Effectiveness Loss. The main goal for the candidate generation stage of a multi-stage retrieval system is to generate a large set of *possibly relevant* documents. On the other hand, the primary goal for the complex ranker is to identify the relevant documents within the large set of candidates, and push them to the top of the ranked list, allowing the user to easily find them. Hence, there is a mismatch in goals between the stages of retrieval. This makes it difficult to evaluate each stage of the search process individually, as suitable metrics may require a large number of relevance annotations which is expensive in practice. Consider the case for evaluating the effectiveness of the first stage of retrieval — *recall* is the most important aspect. However, measuring the true recall of a document set requires annotating the *entire corpus* for each information need, which is intractable in practice. However, a much more elegant approach was proposed by Clarke et al. [61]. They show how rank similarity metrics (such as MED) can be used to evaluate any arbitrary stage of a multi-stage retrieval system, which we outline now.

Firstly, it is assumed that there exists a system that can produce an *effective* final ranked list for any input query. In the context of multi-stage rankers, this would be the machine-learned late-stage ranker, which uses a large amount of higher-order features to accurately rank documents. The main criteria for such ranker is that it is *trusted* to give good results, as we will see shortly. The output list from this ranker, denoted r_B, is considered as a *reference list* or *gold standard* ranking for the given query. For example,[1] consider the following ranking that arises from processing a query:

$$r_B = \langle 12, 71, 23, 9, 54, 32, 7, 22, 42, 17, \ldots \rangle.$$

Now, suppose we wanted to test a new candidate generation algorithm, which for the same query as above returns a candidate list of documents which contains the subsequence

$$r_A = \langle 12, 23, 54, 7, 22, 42, 17, \ldots \rangle.$$

Now imagine that we wished to evaluate the effectiveness of the final ranking using RBP with $\phi = 0.8$. By the definition of MED and RBP, the maximum effectiveness difference between two input lists arises when common documents between the lists are considered *non-relevant*,

[1] This example is inspired by the example in the work of Clarke et al. [61].

and documents that are in r_B but not in r_A are considered *relevant*. Note that we assume any document occurring in r_A but not in r_B as non-relevant. For our above examples, that leads to

$$\begin{aligned}\mathsf{RBP}(r_B) &= \mathsf{RBP}(\langle 0,1,0,1,0,1,0,0,0,0\rangle)\\ &= 0.2(0.8^1+0.8^3+0.8^5)\\ &= 0.328\\ \mathsf{RBP}(r_A) &= \mathsf{RBP}(\langle 0,0,0,0,\ldots,0\rangle)\\ &= 0.\end{aligned}$$

Hence, if the candidate set r_A is used to generate the final ranked list r_B, 0.328 is the upper-bound of effectiveness loss that will be experienced with respect to RBP $\phi = 0.8$. As another example, consider a different candidate generation algorithm which generates the list r_A, and $r_B \subseteq r_A$. In this case, the candidate generation phase identified *every* document in the final ranked list r_B, resulting in *no* effectiveness loss, as

$$\begin{aligned}\mathsf{RBP}(r_B) &= \mathsf{RBP}(\langle 0,0,\ldots,0\rangle)\\ &= 0\\ \mathsf{RBP}(r_A) &= \mathsf{RBP}(\langle 0,0,\ldots,0\rangle)\\ &= 0.\end{aligned}$$

Conversely, the worst case scenario arises where the candidate generation stage identifies no documents that appear in the final ranked list, that is, $r_A \cap r_B = \emptyset$. In this case, the value of MED-RBP will approach 1. Hence, this method allows us to evaluate the *effectiveness loss* that arises between any given stage and the *expected* final results list in a multi-stage retrieval system. Furthermore, this approach was shown to align well with true relevance judgments in practice [61].

Leveraging Reference Lists for Labeling. While MED was shown to be a useful tool for evaluating multi-stage retrieval systems in the absence of relevance labels, Culpepper et al. [75] took this notion even further and showed how reference lists can be used in conjunction with MED to annotate training data. Their task, which is studied in this chapter, was to predict the number of candidate documents that must be retrieved on a query-by-query basis in order to minimize cost in multi-stage retrieval systems. To label training examples, the authors used an initial candidate set, r_A, and a reference list, r_B. The key idea was to find the *shortest prefix* of r_A which, when evaluated against r_B in terms of MED, yielded an acceptable margin of error (denoted as ϵ henceforth). Intuitively, the idea is to leverage MED to ask *"how many documents from the candidate generation stage do I need to pass along to the final stage such that I will observe at most only a very small loss in effectiveness?"* As such, the *optimal* value of k can be used as a training label.

6.2 Training Effective Prediction Models

In the previous section, we discussed a multitude of ideas from different areas of IR, including query performance prediction, query efficiency prediction, list similarity measures, and obtaining training labels in the absence of relevance annotations. Now, we show how these ideas can be leveraged, together, to form a unified framework that can be used to balance efficiency and effectiveness for multi-stage retrieval systems, while also reducing the tail latency of candidate generation.

6.2.1 Generating Labels

The work of Culpepper et al. [75] outlined a novel method for using reference lists and MED to label training data without requiring relevance data or human annotation. However, they cast the problem of predicting k as a multi-label classification problem, which involved predicting k from a fixed set of 9 values. While this method was shown to work well in practice, we argue that predicting the exact value of k should provide even better cost savings, and hence consider it as a regression problem.

Building Effective Reference Lists. Before describing our prediction models, we first describe how we build effective reference lists. In the prior work from Clarke et al. [61] and Culpepper et al. [75], the gold-standard systems which were used to generate the final top-k reference list rankings were selected by taking the *best performing runs* from the respective TREC tasks. One problem with this approach is that the initial candidate generation stage may not completely align with the final stage runs, meaning that high MED values could be observed for some queries due only to confounding factors such as indexing differences. To avoid this problem, we build our own late-stage ranker, denoted as URisk-Ideal, which can be used to generate a reference list for any arbitrary query. Since our model needs to be *robust*, we train a risk-sensitive Lambda-Mart model [103] via a risk-sensitive NDCG@10 loss function. In particular, URisk [86, 87] with $\alpha = 1$ was used to weight a degradation in effectiveness 2 times heavier than an improvement, resulting in a conservative model which is less likely to *hurt* the effectiveness of the initial candidate set when re-ranking it [253]. This is important since the model will be used to generate top-k rankings for queries without relevance judgments. The model uses around 150 standard features which were generated using the publicly available system described by Chen et al. [57] and Gallagher et al. [102],[2] and a description of the features can be found within that codebase. For training, we used 687 queries from the 2009 million query track which contain shallow relevance judgments. As input, we retrieve the top 10,000 documents using the BM25 ranker. When deploying our model across the 2009 ClueWeb09B ad-hoc query topics (1–50), we observe significant effectiveness improvements over the BM25 baseline (Table 6.1).

[2] https://github.com/rmit-ir/tesserae

System	NDCG@10	ERR@10	RBP$\phi = 0.8$
BM25	0.205	0.096	0.307 (0.173)
URisk-Ideal	0.310	0.135	0.425 (0.219)

Table 6.1: The effectiveness of the candidate generation BM25 stage with respect to the re-ranked LtR stage. LtR significantly outperforms the baseline for all metrics with $p < 0.01$ using a two-sided student's t-test.

Building effective and robust gold-standard rankers is an interesting topic in its own right, but analogous to the work presented here. Thus, we refer the interested reader to Liu [152] and to Macdonald et al. [164, 165] for further information on building a competitive LtR system.

Labeling for Regression. Now that we have a candidate set r_A to depth $k = 10{,}000$ and a subsequent re-ranked reference list r_B for each query, we can follow the method of Culpepper et al. [75] to compute the minimal prefix of r_A such that

$$\mathsf{MED}(\mathit{sortedprefix}(r_A, k, r_B), r_B) \leq \epsilon, \tag{6.4}$$

where $\mathit{sortedprefix}(r_A, k, r_B)$ returns the first k documents from r_A, sorted with respect to r_B. Note that the sorting operation ensures that the following MED computation gives the correct effectiveness difference, as the input list may be ordered differently to the reference list. When the problem is formulated as a multi-label classification problem, the value of Equation 6.4 can be computed by taking a prefix of the candidate list in ascending order of the fixed values of k, sorting it, and measuring MED until the desired effectiveness bound is met. However, our problem is more involved — we need to find the true optimal value of k, which is the smallest value of k which satisfies Equation 6.4. In order to do this efficiently, we *iteratively* increase the number of candidate documents that are considered by MED until we find the optimal. While a binary search could be used for this process, our iterative method can cache the partial data from the previous MED computation, meaning that the subsequent MED computation is extremely trivial. Algorithm 6.1 shows this process.

6.2.2 Training and Prediction

Now that we have labels for our training examples, we will outline the different models that are trained for the prediction task, and the features used to make these predictions.

Features. A critical aspect of the features used for the given prediction tasks is that they are efficient to compute. This is because the prediction task occurs prior to any query processing, so any latency added during feature generation or prediction will have a direct impact on the query latency. As such, we opt to use a set of features that can be computed very efficiently based on

Algorithm 6.1: Finds the smallest value of k such that effectiveness loss is minimized between a candidate list and subsequent re-ranked list.

Input : A candidate list, r_A, a reference list r_B (the gold-standard), and a desired MED threshold, ϵ.

Output : The smallest value of k such that $\text{MED}(\textit{sortedprefix}(r_A, k, r_B) r_B) \leq \epsilon$.

```
CurrentMED ← 1.0
k ← 0
CandidateSet ← ∅
while CurrentMED > ε do
    k ← k + 1
    CandidateSet ← sortedprefix(r_A, k, r_B)
    CurrentMED ← MED(CandidateSet, r_B)
end
return k
```

pre-computed term and score statistics. These features have been used for similar prediction tasks, and have been shown to work well in practice [75, 119, 129, 187, 197].

Table 6.2 describes the features used. The left table reports the statistics that are stored in the feature database for each ranker. In our study, we deploy 7 common bag-of-words rankers, namely TF·IDF, BM25 [212], Query Likelihood [204], DPH, DFR, PI2, and Bose-Einstein [2]. The right table shows how these statistics are combined to create the final feature vector. In total, 147 features are used in our predictive models. While all features are unlikely to be useful, we leave further ablation studies for future work.

Regression Models. Regression models aim to predict a target variable y given a vector of independent variables (features), $\vec{x}$. In order to train a regression model, input data of the form $\langle(\vec{x_1}, y_1), (\vec{x_2}, y_2), \ldots, (\vec{x_n}, y_n)\rangle$ is optimized by the model to minimize some *loss function*, such as the *least squares* function, which minimizes the residual between the predicted value of y, and the true value of y (and is a good estimator of the mean). This is the approach used by an out-of-the-box regression algorithm, such as the *random forests* [33] method (RF). However, a pitfall of standard regression algorithms is that they can become biased in the presence of heavily skewed distributions, reducing the predictive power of the model. One reason for this is that the underlying distribution of y is assumed to be normal. However, this assumption does not always hold in practice.

One simple way to deal with this problem is to employ *quantile regression* [133] (QR), which estimates a given *quantile* of the response variable, y. If y has a cumulative distribution function of

$$F_y(z) = p(y \leq z),$$

	Term Statistics
1	No. documents containing term t (f_t)
2	No. occurances of t in the collection (F_t)
3	Maximum similarity score
4	P_{25} similarity score
5	P_{50} similarity score
6	P_{75} similarity score
7	Arithmetic mean of similarity scores
8	Harmonic mean of similarity scores
9	Variance of similarity scores
10	Interquartile range of similarity scores

Feature Description
Arithmetic mean of $\{3,5,7,8,9,10\}$
Maximum of $\{3,4,5,6,7,8,9\}$
Minimum of $\{3,4,5,6,7,8,9\}$
Query length
Average of $\{1,2\}$
Minimum of $\{1,2\}$
Maximum of $\{1,2\}$

Table 6.2: An overview of the QPP features used for our prediction tasks. This left table shows the pre-computed statistics that can be stored to efficiently generate the QPP features. The right tables show how these statistics can be aggregated to generate features for an input query, with the top half representing features that depend on pre-computed score information, and the bottom half representing score-independent features that depend only on the query terms.

then the τth quantile of y is given by the quantile function

$$F_y^{-1}(z) = \inf\{z : F_y(z) \geq \tau\}.$$

As such, a new loss function can be defined that minimizes the loss with respect to a given quantile, τ, and the subsequent model will make predictions that estimates the value of the τth quantile of y given the feature vector $\vec{x}$. Another benefit of quantile regression is that the aggression of the predictor can be changed by predicting different quantiles. For example, if k was being predicted and very little loss in effectiveness is permitted, the quantile regression model could be deployed such that it is more likely to over predict the true value of k, resulting in a more costly but more effective retrieval run. To this end, quantile regression allows for effective *risk sensitive* prediction, which is the goal of RQ3.2.

Random Forests. Random forests build several decision trees using attribute bagging — each tree learns using only a subset of the data, and the final predictions are the *average* of the value predicted by each tree. During training, individual trees are learned by branching data down the tree, and only a subset of features are used for splitting at each node. This results in trees that are less likely to be correlated to each other, improving the robustness of the predictions at each tree. At the end of the training phase, the model M is an *ensemble* of decision trees, where the leaf nodes of each tree contain a *prediction* value.

In order to make a single prediction, M consumes an input feature vector $\vec{x}$ which is used to traverse the ensemble of trees, $\mathcal{T}$. For each tree $T \in \mathcal{T}$, the features within the feature vector are looked up to traverse the decision tree until a *leaf node* is reached. The predicted value $M(\vec{x})$ of

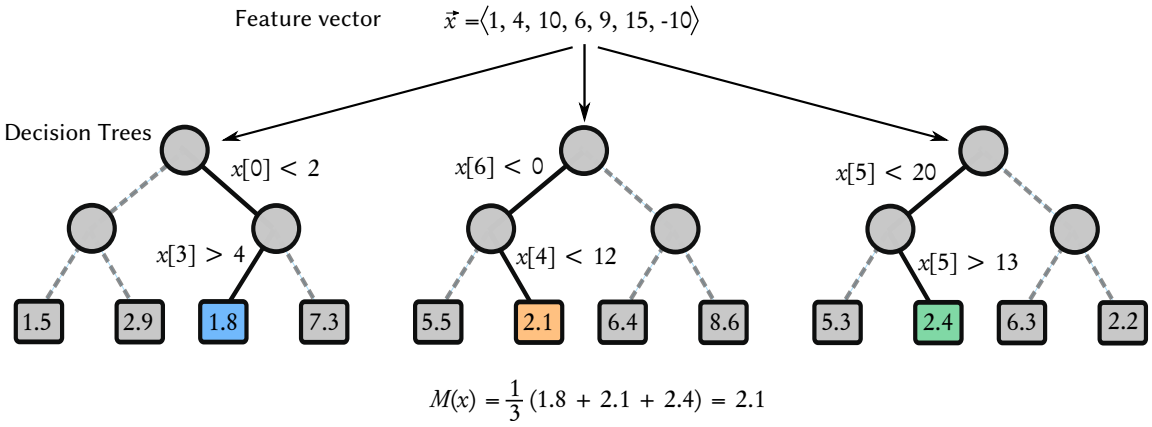


Figure 6.2: A toy example of computing the output of a random forests regression model, M, made up of a set of three decision trees. For each tree $T \in \mathcal{T}$, the exit node is found by walking T with respect to $\vec{x}$, and the output of each is averaged to form the prediction.

the ensemble $\mathcal{T}$ is computed as the *mean* of the predictions of each tree:

$$M(\vec{x}) = \frac{1}{|\mathcal{T}|} \sum_{i=1}^{|\mathcal{T}|} T_i(\vec{x}).$$

A pictorial example of this process is shown in Figure 6.2, where $|\mathcal{T}| = 3$.

Quantile Regression. For quantile regression, we use *gradient boosted regression trees* (GBRTs) [99, 100]. Similar to random forests, GBRTs utilize an ensemble of decision trees to generate an output. However, instead of training via data and feature sampling, as is done with random forests, GBRTs learn a single tree at a time, and each subsequent tree aims to minimize the remaining regression residual (error). At the end of the training phase, the learnt model M is an ensemble of decision trees, and each of the leaf nodes contains a prediction value. Unlike random forests, each tree T_n may also be associated with a *weight* $w_n \in \mathbb{R}$.

To make a prediction over M, a similar process to the random forests model is followed. First, M consumes a feature vector $\vec{x}$ which is used to traverse each tree T in the ensemble $\mathcal{T}$. At the exit node of each tree, the prediction value is output, and the final predicted value of the model, $M(\vec{x})$, is computed as a *weighted sum*:

$$M(\vec{x}) = \sum_{i=1}^{|\mathcal{T}|} w_i \cdot T_i(\vec{x}).$$

In our learning framework, the weight for each tree is optimized using line search. To further reduce the sensitivity of the model to outliers, the Huber loss function is employed [114], which penalizes extreme values linearly rather than quadratically, as is the case with squared loss.

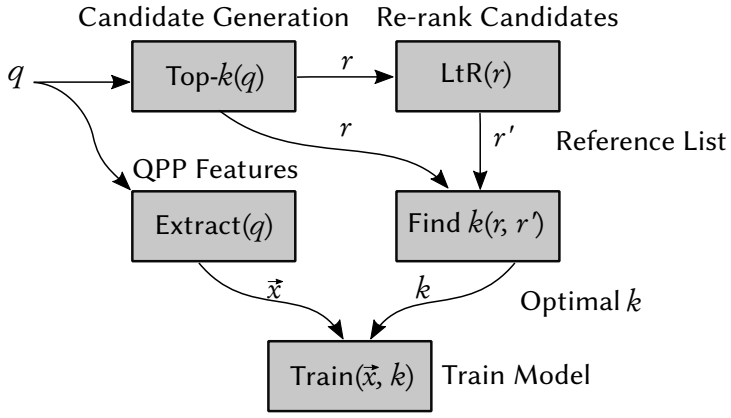


Figure 6.3: The process for building a training example for the M_k prediction model. In our pipeline, we use an LtR model to re-rank a set of candidate documents, r, to yield the reference list, r'. Both r and r' are then used to find the optimal k — the shortest prefix of r that would achieve a value of MED below a given value of ϵ. Then, the optimal value of k (the label) is paired with the QPP feature vector, $\vec{x}$, to generate a training instance.

Putting it Together. Figure 6.3 outlines our entire end-to-end training process for a single query, assuming the existence of the LtR model. First, a rank safe candidate generation algorithm will generate the top 10,000 candidate documents, and pass them to the LtR algorithm. Both of these lists will then be taken as input to Algorithm 6.1, which will output the smallest prefix of the candidate list which achieves a MED score $\leq \epsilon$. At the same time, the original query terms are used to generate a vector of features, $\vec{x}$. Finally, the feature vector $\vec{x}$ and optimal k are consumed by the training regime, which permutes the model. This process is repeated for all training queries, and then the model for predicting k, M_k, can be deployed. In our experiments, we use MED-RBP with $\phi = 0.95$ as our list comparison method, and we target a very low effectiveness loss margin with $\epsilon = 0.001$. It is important to note that the LtR system could be replaced with any other well performing system.

6.3 Experimental Setup

Hardware. All experiments were executed on an otherwise idle 12 core Intel Xeon E5-2690 v3 with 512 GiB of RAM hosting Red Hat Enterprise Linux. All experiments were conducted entirely in main memory, and the postings lists warmed up prior to experimentation. Only a single processing thread was used for experiments in this chapter. All reported timings are in milliseconds, and are the average of 5 runs.

Software. ATIRE [246] was used for indexing the raw corpus, and then this index was dumped into formats suitable for both the DAAT and SAAT systems. Timings were conducted using

publicly available implementations of BMW[3] and JASS,[4] the same systems that were used in Chapter 4. These systems were compiled using GCC 6.2.1. The learning-to-rank model (LtR), which was used to generate our reference lists, was built using JForests.[5] The random forests (RF) regression model was built using Weka.[6] For quantile regression (QR), we used XGBoost.[7]

Document Collection and Indexing. We use the ClueWeb09B collection, which contains around 50 million web documents, crawled during 2009. The index was stopped using the default Indri stoplist, and stemmed using an *s*-stemmer. Normalization included converting malformed UTF-8 characters to spaces, stripping XML comments and tags, and segmenting numeric and alphabetic characters. Postings were compressed using QMX compression [242, 244] and then scored via the BM25 similarity model. Quantization was employed, using 9 bits to represent scores [66].

Queries and Relevance. For the prediction tasks, we use the 2009 Million Query Track (MQT) queries [49]. Single term queries were filtered from all test, train, and validation sets, as they can be answered trivially by taking the first k documents from the relevant postings list of the impact-ordered ISN. In addition, we filtered out queries which contained out-of-vocabulary terms, and the queries which were used to train the LtR gold-standard system, resulting in a set of 26,959 unique queries. For all predictions, queries were randomly assigned to 10 folds, and standard 10-fold cross validation was performed to produce the query predictions.

6.4 Preliminary Experiments

While we expect that predicting the value of k on a per-query basis will greatly improve the overall efficiency of the search system, it may not greatly improve the tail latency. In Chapter 4, it was shown that DAAT dynamic pruning algorithms were faster for smaller values of k, but queries with high latency could occur for any value of k. This observation held even when *aggressive* dynamic pruning schemes were used. The alternative approach is to use a SAAT algorithm such as JASS. However, aggressive early termination was shown to possibly hurt effectiveness, especially for queries with many candidate postings. So, rank safety is another confounding factor for improving the efficiency/effectiveness trade-offs in cascade rankers. To re-illustrate these effects, we run both rank safe and approximate versions of BMW and JASS across the ClueWeb09B collection and MQT queries for large values of k. We remind the reader that the rank safe algorithms are denoted as BMW-E and JASS-E, and the aggressive instances are denoted as BMW-A and JASS-A. We examine a few different aggression levels for BMW-A, so we opt to add the corresponding F value to each system identifier. Figure 6.4 shows the outcome, and confirms our earlier conclusions:

[3] https://github.com/JMMackenzie/Quant-BM-WAND
[4] https://github.com/lintool/JASS/
[5] https://github.com/yasserg/jforests
[6] https://cs.waikato.ac.nz/ml/weka
[7] https://github.com/dmlc/xgboost

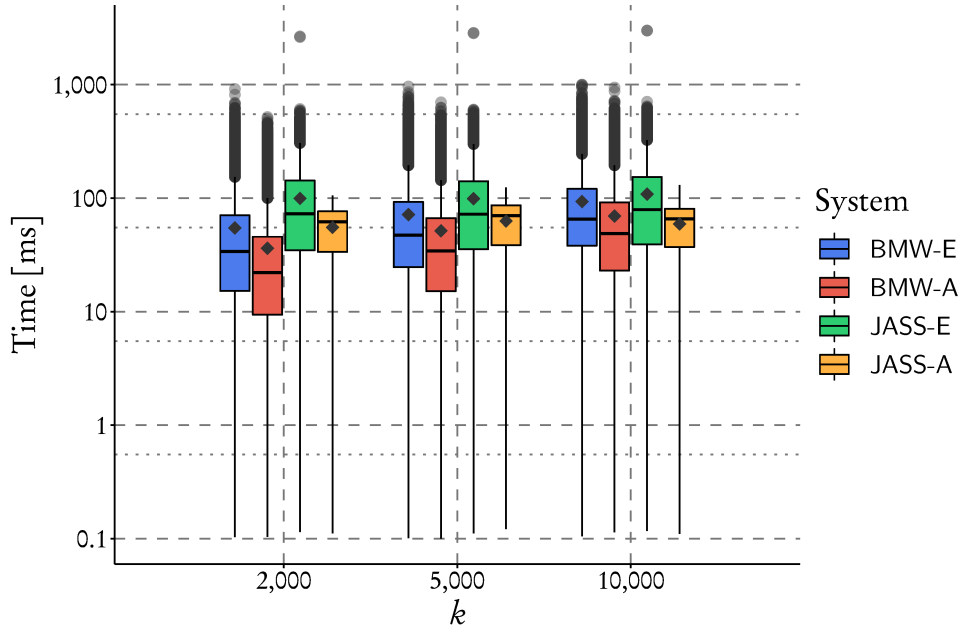


Figure 6.4: Efficiency comparison of across the 26,959 MQT queries using both rank-safe and aggressive processing schemes for BMW and JASS. Aggressive BMW used a value of $F = 1.2$, and aggressive JASS used a value of $\rho = 5 \times 10^6$.

boosting the aggression parameter for DAAT dynamic pruning algorithms generally increases the efficiency, but the high tail latency is difficult to reduce.

To further explore the relationship between the tail latency of the DAAT and SAAT systems, we conduct a simple overlap analysis on the queries exceeding the P_{95} latency. Table 6.3 shows the results. Exhaustive JASS, exhaustive BMW, and aggressive BMW traversal algorithms all tend to share similar queries in the slowest 5% band. This indicates that these queries are inherently *expensive* to process. On the other hand, the aggressive JASS traversal shares a much smaller number of queries with the other systems. As previously shown, the aggressive JASS algorithm has its efficiency impacted only by the value of ρ.

In light of this new evidence, a pragmatic hypothesis emerges: can we combine the best properties of both the JASS and BMW traversal methods to create a hybrid approach that outperforms either system in isolation?

6.5 Hybrid Index Architecture

We now propose a hybrid search architecture that tries to combine both DAAT and SAAT index traversal methods to improve search efficiency without negatively impacting effectiveness.

	BMW-A $F = 1.1$	BMW-A $F = 1.2$	JASS-E	JASS-A
BMW-E	86.0	61.7	56.2	16.7
BMW-A $F = 1.1$	-	67.4	53.1	18.3
BMW-A $F = 1.2$	-	-	42.3	24.2
JASS-E	-	-	-	8.0

Table 6.3: The percentage overlap of queries that exceed the 95th percentile efficiency band for $k = 2{,}000$. Making BMW more aggressive may improve timings, but outliers are still present. On the other hand, it is less common for JASS and BMW to have overlapping queries that contribute to tail latency, especially when a non-exhaustive ρ value is used.

6.5.1 Index Traversal Trade-offs

As discussed in Chapter 4, both DAAT and SAAT processing mechanisms have a range of trade-offs between them that make them attractive for use in varying circumstances. While early-termination SAAT algorithms can bound processing latency, they are sometimes subject to a loss in effectiveness. On the other hand, DAAT dynamic pruning algorithms such as BMW are generally more efficient for smaller values of k, but are susceptible to high tail latency. Our preliminary experiments above confirm these ideas, showing that JASS-A is the clear winner with large values of k and tail latency, and BMW is faster for smaller values of k and has a better mean and median latency.

6.5.2 Index Replication

Prior work on distributed IR systems has shown that an effective approach to scaling efficiency is to *replicate* popular indexes (or index shards) across multiple index server nodes (ISNs) [39, 82, 96, 131, 210]. We leverage this observation, and assume that replicas can be optimized for a specific index traversal operation. In other words, when building replicas, we may opt to build a document-ordered index, which is suitable for DAAT traversal, or we may build an impact-ordered index, which is suitable for SAAT traversal. We comment more on the implications of this approach in the discussion (Section 6.7).

6.5.3 Hybrid Pipeline

We now describe our *hybrid* processing pipeline, which aims to leverage query performance prediction to predict performance-sensitive parameters on a per-query basis. Backed by both DAAT and SAAT traversal algorithms, our approach can limit the disadvantages of each mechanism to achieve improved trade-offs for end-to-end retrieval.

The first step in our approach is to predict the k cutoff. If k is small, we will use the document-ordered shard to process the query using BMW, as we are confident that it will be efficient and has the benefit of returning rank safe results. On the other hand, the latency of SAAT traversal is not impacted by the value of k, so we farm out queries with large values of k to the impact-ordered

shard. If the SAAT system is used, we will also predict the value of ρ, allowing more aggressive processing which will further reduce tail latency, with an upper-bound of $\rho = 5 \times 10^6$. This value is selected as it requires less than 200ms on our current hardware configuration, as does not result in significant effectiveness loss for early precision metrics across ClueWeb09B [68, 148].

Predicting ρ. The process for training a model for predicting ρ, M_ρ, is the same as the process discussed for M_k. Since the effectiveness of JASS depends on both the value of k and the value of ρ, we use the *optimal* value of k when conducting training (which we derived from the process of training M_k). Next, the JASS system was modified to output the ranked list that arises after processing *every* segment for the given query. For example, if the candidate postings contained 14 segments, we would iteratively process the query and derive *up to* 14 top-k results lists, where k is based on the optimal value of k for the given query. Early in processing, if there are not yet k results present in the top-k heap, we do not output these results lists, as we do not want the value of k to be a confounder for obtaining the optimal value of ρ. Following the same methodology as earlier, we then compare each of these lists with the reference list to find the *lowest* value of ρ which achieves a value of MED $\leq \epsilon$. To improve the efficiency of prediction, the feature vectors are the same for both M_k and M_ρ, so features do not need to be recomputed for predicting ρ.

Full Hybrid Pipeline. Our end-to-end hybrid pipeline is shown in Figure 6.5. We denote this system as Hybrid-k, and report the results for three different instances of this hybrid pipeline, discussed shortly. Since the algorithm selection component is required to decide whether to select the DAAT or SAAT method based on the value of k, we use this selection threshold to generate different trade-offs within each of the hybrid systems. In order to set the selection criteria to target a specific MED-RBP $\epsilon \in \{0.04, 0.06, 0.08, 0.10\}$, we used the same folds as listed previously to tune the selection threshold such that the given ϵ value was achieved via 10-fold cross validation. The mean value was then used as the *switching* point between using the DAAT and SAAT system. As such, the *algorithm selection* component of the pipeline is driven directly by the predicted value of k.

Cost of Prediction. In the experiments discussed henceforth, we assume the cost of prediction is *free*. That is, when reporting timings, we do not account for the time taken to predict the value of k or ρ. Hence, the timings reported correspond only to the query processing component of the framework. However, recent work has shown that prediction costs of models similar in nature to ours are *less than 0.75ms* [119], *less than one ms* [129], and *between 0.6 and 0.9ms* [197]. While building optimized prediction models is an interesting problem in its own right, it is orthogonal to the ideas presented here.

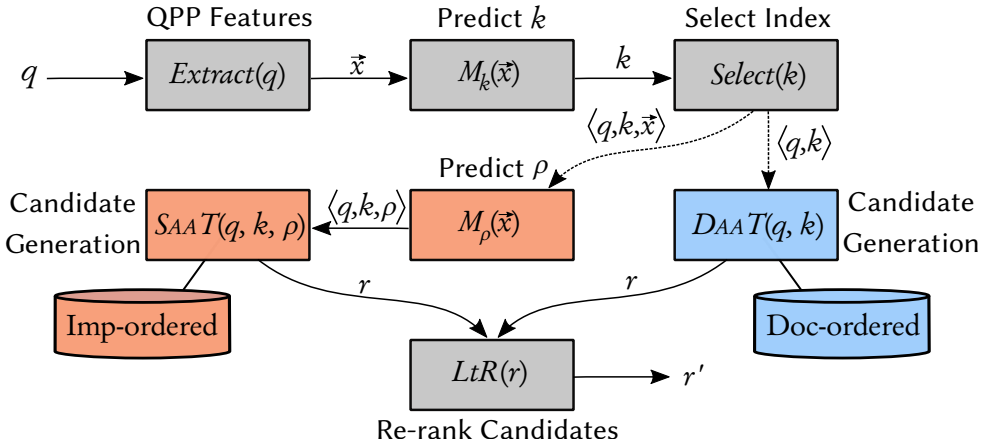


Figure 6.5: The hybrid architecture deployed for reducing tail latency. First, the QPP features are extracted, and the value of k is predicted from M_k. Based on this value, either DAAT or SAAT traversal will be used to generate the list of candidate documents, r. If SAAT traversal is selected, the value of ρ is predicted from M_ρ. Finally, the LtR model re-ranks the candidate set to yield a new ranking, r'.

6.6 Experimental Analysis

We now describe the experimental analysis which examines the predictive performance of both the random forests and quantile instances of M_k and M_ρ. We also show how our hybrid system compares to highly tuned fixed parameter systems in a realistic query processing task.

6.6.1 Predictive Models

First, we validate our predictive models. In particular, we explore the performance differences between the Fixed, RF, QR, and Oracle predictors for both k and ρ.

Prediction Distributions. The first experiment aims to gain intuition on whether the QR predictor is more robust at estimating the true value of the predicted variable than the RF predictor. As discussed earlier, the RF predictor targets the mean value of the distribution based on the input feature vector. On the other hand, the QR predictors are more robust to outliers, as they predict for a specified quantile of the distribution. Figure 6.6 shows the distributions of the RF, QR, and Oracle predictors for both the k and ρ prediction tasks across all queries. Clearly, the QR methods are able to more accurately capture the underlying distribution of values, since they are not biased by outliers. As such, we expect the QR method to be more accurate for both prediction tasks.

Predicting k. While the last experiment gave us intuition that the QR predictor is a better candidate than the RF predictor, we did not consider the trade-off that occurs when the value of ϵ

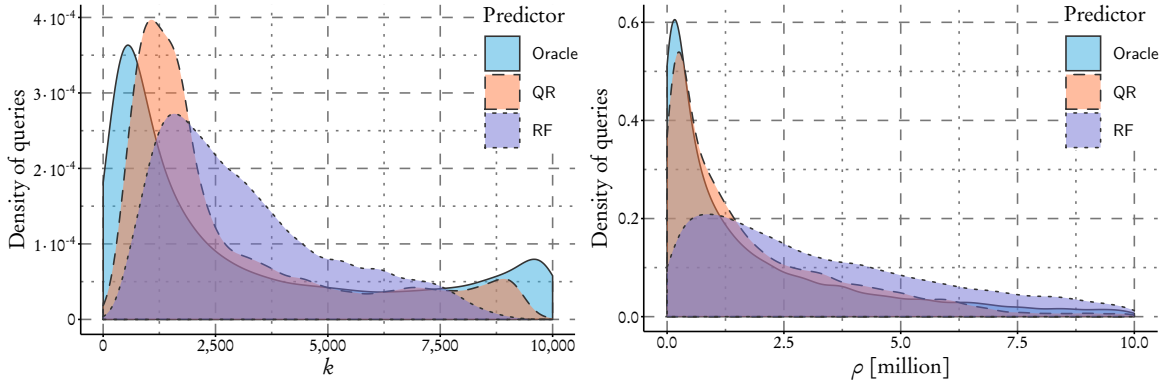


Figure 6.6: The distribution of predicted values of k (left) and ρ (right). The QR predictor uses a quantile of $\tau = 0.55$ for predicting k and a quantile of $\tau = 0.45$ for predicting ρ. The RF predictor predicts the mean value for predicting both k and ρ. Training examples optimize the MED-RBP$\phi = 0.95$ error $\epsilon < 0.001$.

is relaxed (which results in less effective, yet more efficient traversal). To examine this trade-off space, we train the RF predictor for a number of MED values, with $\epsilon = [0.001, 0.20]$. We also compare this with the QR model trained for $\epsilon = 0.001$, and vary the aggression of this model via sweeping the quantile parameter $\tau = [0.10, 0.75]$. Finally, we include the trade-off that occurs from using a fixed value of k for all queries, and the oracle value of k on a query-by-query basis. Figure 6.7 shows the trade-off for the mean value of k (left) and the median value of k (right). Since the distribution of ideal k values is skewed (Figure 6.6), the median plot captures the trade-off more accurately. Both the RF and QR predictor have very similar trade-offs when considering the mean value of k, but the QR predictor outperforms the RF predictor when considering the median value of k. While these predictors give a better trade-off than simply using a fixed value of k, there is still clearly room for improvement when observing the oracle. In the following experiments, we employ QR regression for predicting k.

Predicting ρ. We repeat the above experiment for predicting the value of ρ. Again, we expect the QR predictor to outperform the RF predictor based on the fit of the prediction distributions from Figure 6.6. We train the RF model to target a minimal loss in effectiveness, aiming for a MED-RBP $\epsilon = 0.001$. The QR model is also trained on minimizing effectiveness loss with $\epsilon = 0.001$, and the aggression of this model is varied by predicting different quantiles $\tau \in [0.15, 0.75]$. We also plot the true oracle point, and the fixed ρ curve. Figure 6.8 shows the results. Again, we note that the RF and QR predictors perform similarly, and are able to outperform using a fixed cutoff across all queries. However, the predictive models cannot achieve effectiveness close to the oracle value, indicating that there is still room for improvement in this prediction task. For all predictions of ρ in subsequent experiments, we use the QR method with $\tau = 0.45$, as we observed it to fit well with the true distribution of values in Figure 6.6.

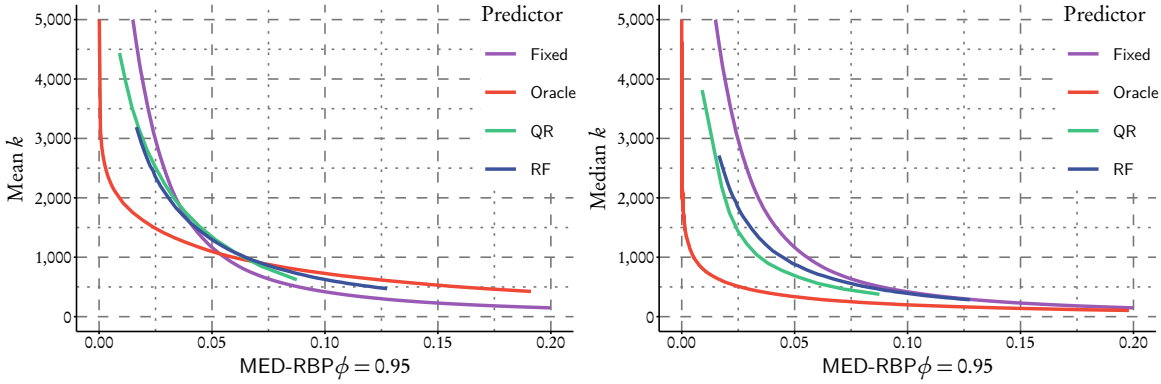


Figure 6.7: Fixed, oracle and predicted k vs MED-RBP trade-off for both mean (left) and the median (right) values of k.

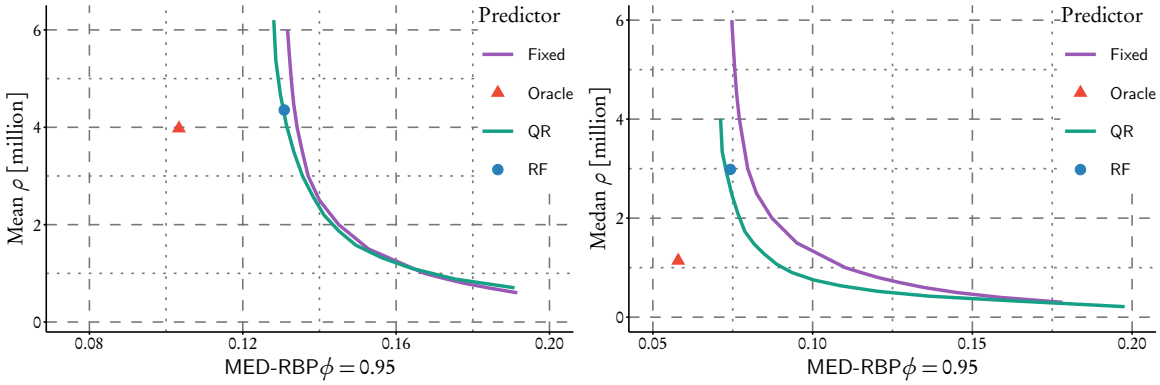


Figure 6.8: Fixed, oracle and predicted ρ vs MED-RBP trade-off for both mean (left) and the median (right) values of ρ.

6.6.2 Hybrid Pipeline

Our next experiment aims to show the performance gains that are possible by deploying the Hybrid model as shown in Figure 6.5. We instantiate three Hybrid methods; Hybrid-k-25, Hybrid-k-45, and Hybrid-k-55, where the value following k represents the percentile that was learnt by the QR algorithm for predicting k. This allows us to explore the trade-offs between aggressive and cautious predictions of k. As a baseline, we deploy the BMW-E algorithm with rank safe processing. We also generate two JASS baselines which either exhaustively process each query (JASS-E) or process up to 5 million postings for each query (JASS-A). The baseline systems use a fixed value of k across all queries. Each system is configured to achieve a mean MED-RBP $\epsilon \in \{0.04, 0.06, 0.08, 0.10\}$ to demonstrate different trade-off points.

Candidate Generation Profile. Figure 6.9 shows the efficiency of the candidate generation stage for the baselines and hybrid systems. Where very little effectiveness difference between the final output and the candidate generation stage is allowed (such as when $\epsilon = 0.04$), the Hybrid algorithms tend to perform similarly to the BMW-E baseline, but have slightly lower mean, median, and tail response latency. JASS-A has a higher mean and median than the hybrid and BMW-E systems, but has a much better bound on the tail. However, as the effectiveness constraints are relaxed, the Hybrid algorithms begin to balance the positives of both DAAT and SAAT traversal. For example, when $\epsilon = 0.10$, the Hybrid-k-45 system is able to reduce the mean, median, and tail latency with respect to JASS-A while only being slightly slower in mean or median response with respect to the BMW-E system. This is because the Hybrid approaches tend to select the document-ordered ISN more often for situations when effectiveness loss must be minimized, as the impact-ordered ISN and JASS-A processing are only approximations of the true results list. On the other hand, when the effectiveness constraints are relaxed, the Hybrid methods opt to select the SAAT traversal method more often, resulting in a slightly more approximate results set but also a much lower tail latency. Interestingly, the Hybrid approaches could be fine tuned even further — training for different values of τ in the M_ρ predictor, or explicitly setting the algorithm selection bound are two ways of doing so. However, we do not wish to overfit our models — the main take-away is that the Hybrid systems allow for much tighter controls on the inherent trade-offs between low median or mean latency, and keeping the tail latency under control.

Cost Trade-offs. The previous discussion focused directly on the candidate generation stage of the end-to-end pipeline. However, the goal of predicting k is to reduce the number of candidate documents that need to be evaluated in all subsequent stages of the cascaded pipeline. To directly observe the complex trade-offs that exist between the latency of candidate generation, the cost of re-ranking (implicitly through the value of k), and the system effectiveness, we plot a series of summary statistics in Table 6.1, where each sub-table corresponds to one of the MED-RBP targets (and hence, a facet of Figure 6.9). Note that we drop the JASS-E system from this analysis as it is outperformed by the BMW-E system in all dimensions.

Consider the case where effectiveness loss is minimized ($\epsilon = 0.04$). Here, the rank safe DAAT and SAAT methods must retrieve around 1,600 candidate documents. However, the best Hybrid approach can achieve the same effectiveness difference by returning only 1,148 documents, resulting in large down-stream savings for subsequent retrieval stages. This system also has the lowest median response latency. On the other hand, the aggressive JASS-A system has the best tail latency, but must retrieve almost 3,000 documents to achieve the same expected effectiveness.

Where there is less restriction on effectiveness difference ($\epsilon = 0.10$), the BMW-E and JASS-E approaches retrieve the lowest number of candidates. However, the best hybrid system, Hybrid-k-25, has a much faster response latency across the mean, median, 95th, and 99th percentile latency when compared to the BMW-E. In particular, Hybrid-k-25 can generate the candidates 1.7× faster when considering the mean latency, 2.6× faster at P_{95}, and 3.7× faster at P_{99}. A range of other trade-offs can be observed as the Hybrid algorithms tend towards the DAAT or the SAAT ISN.

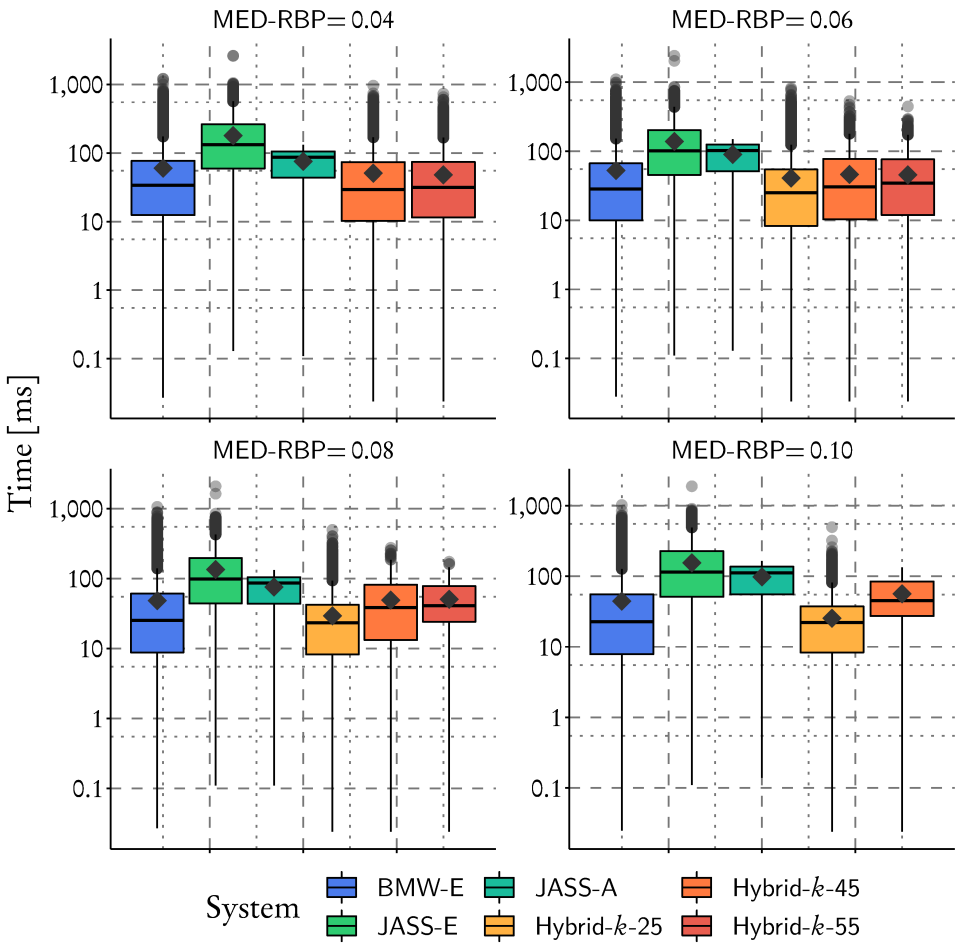


Figure 6.9: Response time for each system for different bands of MED-RBP in the candidate generation stage. Different configurations of the hybrid approach provide trade-offs that bridge the gap between the BMW and JASS systems.

While our Hybrid methods can outperform the fixed systems in most dimensions, it is clear that there is extreme tension between efficiency and effectiveness. For example, improvements in tail latency come at the expense of a larger k, which could translate to a slower system overall. On the other hand, when k is minimized, it is difficult to also minimize the efficiency of the candidate generation stage. However, we have shown that useful trade-offs are possible through our Hybrid pipeline.

6.6.3 Validation

As a final validation step, we deploy our end-to-end processing pipeline on the (previously unseen) 2009 ClueWeb09B Web Track topics. As comparison points, we show the effectiveness of the

System	Median k	Mean time	P_{50}	P_{95}	P_{99}
MED-RBP = 0.04					
BMW-E	1,598	60.2	34.0	205.6	377.1
JASS-A	2,970	75.6	87.1	**114.1**	**123.8**
Hybrid-k-45	**1,148**	50.8	**29.3**	149.9	281.0
Hybrid-k-55	1,660	**48.1**	31.4	126.9	220.1
MED-RBP = 0.06					
BMW-E	889	52.7	28.4	183.4	343.9
JASS-A	1,913	89.9	102.5	139.7	143.7
Hybrid-k-25	**631**	**40.9**	**25.0**	132.8	244.3
Hybrid-k-45	1,148	46.2	30.6	121.9	146.0
Hybrid-k-55	1,659	45.6	34.5	**110.0**	**116.9**
MED-RBP = 0.08					
BMW-E	**578**	48.3	25.2	171.2	326.4
JASS-A	1,324	75.6	86.6	119.8	124.7
Hybrid-k-25	631	**29.3**	**23.3**	**73.4**	135.2
Hybrid-k-45	1,148	49.1	38.4	120.9	126.7
Hybrid-k-55	1,659	50.3	41.0	109.3	**113.6**
MED-RBP = 0.10					
BMW-E	**419**	44.3	22.6	158.2	302.2
JASS-A	959	97.7	111.5	152.2	157.2
Hybrid-k-25	631	**25.4**	**22.1**	**59.8**	**81.0**
Hybrid-k-45	1,148	56.2	45.2	120.7	126.3

Table 6.4: Summary statistics for k and time. Each sub-table corresponds to a facet of Figure 6.9 and the best values are bold. The Hybrid approaches generally outperform the fixed parameter systems with respect to all time dimensions, while also sometimes improving the median k value.

BM25 ranker (without re-ranking), and the idealized late-stage run, URisk-Ideal. We also compare *exhaustive* and *aggressive* fixed parameter systems, and our Hybrid-k-45 system. Table 6.5 shows the results. Remarkably, our hybrid pipeline has no measurable effectiveness loss with respect to the URisk-Ideal system. We employed the *Two One-Sided Tests* (TOST) procedure [223] with an acceptable range of inequality of ±1%. We found that the runs are significantly equivalent ($p < 0.01$). Interestingly, the hybrid system was actually able to perform better than the baseline, but this is of course due to chance. In any case, we are confident that our approach does not sacrifice any effectiveness for its improved efficiency, and we remind the reader that we specifically trained our models to minimize effectiveness loss.

System	MED-RBP	NDCG@10	ERR@10	RBP$\phi = 0.8$
BM25	-	0.205	0.096	0.307 (0.173)
URisk-Ideal	-	0.310	0.135	0.425 (0.219)
Fixed (BMW-E, JASS-E)	0.04	0.303	0.130	0.418 (0.224)
Fixed (JASS-A)	0.04	0.303	0.129	0.400 (0.250)
Hybrid	0.04	0.311	0.136	0.422 (0.222)
Fixed (BMW-E, JASS-E)	0.06	0.305	0.132	0.421 (0.221)
Fixed (JASS-A)	0.06	0.306	0.130	0.403 (0.253)
Hybrid	0.06	0.314	0.136	0.426 (0.232)
Fixed (BMW-E, JASS-E)	0.08	0.308	0.134	0.423 (0.216)
Fixed (JASS-A)	0.08	0.305	0.131	0.407 (0.248)
Hybrid	0.08	0.314	0.135	0.425 (0.231)
Fixed (BMW-E, JASS-E)	0.10	0.294	0.127	0.411 (0.220)
Fixed (JASS-A)	0.10	0.309	0.132	0.411 (0.245)
Hybrid	0.10	0.314	0.135	0.425 (0.231)

Table 6.5: Effectiveness measurements taken across the held-out query set. No statistical significance was measured between the hybrid systems with respect to the ideal system (URisk-Ideal), using the two one-sided test with $p < 0.05$. We used the Hybrid-k-45 system for this comparison, although the results are not significantly different to the other Hybrid parameterizations.

6.7 Discussion and Future Work

We have presented a unified framework for building predictive models based on a reference list framework, and leveraged this framework to show how a hybrid query processing pipeline can be deployed for reducing tail latency and improving cost in the context of multi-stage retrieval. Now, we discuss some of the key components of the training framework and query processing pipeline, and comment on future work in these areas.

Ground Truth Selection. One of the most important aspects of the reference list training framework is selecting a ground truth system to build the reference lists. The model training process involves finding a configuration which will yield a *similar* outcome to the ground truth. Hence, the ground truth ranker should be both effective *and* robust, as it is assumed to always give a high quality result. One advantage of this method is that it can be deployed in any *production* IR system — if a particular search company has a ranker which is serving results to real users, then they are presumably confident in the performance and robustness of the ranker. Then, our general approach involves reducing the *cost* of their production pipeline without negatively impacting the effectiveness of it. One property of MED which was ignored in this work is that it can be used in conjunction with partial relevance judgments, should they be available [235]. Hence, it would

be interesting to see if the interaction data available in industrial search systems would allow for an even better instance labeling mechanism, whereby *soft* relevance signals such as clickthrough data [122] could help inform MED to build even better predictors.

Improved Prediction Performance. Another way of improving the performance of the parameter prediction methods is to experiment with other regression models. There is a wealth of regression algorithms that could be used for this problem [92], and future work could focus on comparing and tuning them for better performance. A similar line of work could involve conducting an ablation study to gain a deeper understanding of the best features to use for these efficiency prediction problems [110].

Load Balancing. Since the selection of which ISN (and index traversal method) will be used to process a query depends on the predicted k (and the threshold within the *algorithm selection* decision), our method could result in an unbalanced workload across processing ISNs. While we employed cross validation to select a threshold to hit a particular MED target, a real-world system would need to carefully assign the processing based on system loads [34, 131, 132]. Furthermore, it is likely that large-scale IR systems would deploy more than just two replicas for each index, which was the scenario explored here. Hence, an interesting point of further examination is how to best decide on the *type* of index that will be built when replicating an existing shard.

Alternative Pipelines. While we presented one particular pipeline which involved selecting the query processing algorithm (or ISN) based on the value of k, many alternatives are possible. For example, predicting the time required to process the query could drive the algorithm selection decision [119, 129, 162]. Another interesting idea would be to select the query processing algorithm *within* each ISN. Recent work showed that while VBMW is the state-of-the-art in efficient query processing, it can be outperformed by the MaxScore algorithm when k is large or when the query contains many terms [173, 175]. Hence, selecting a processing algorithm could go beyond using a fixed index traversal method at each ISN [130].

End-to-End Savings. The Hybrid pipeline was shown to reduce the tail latency in the candidate generation stage of processing, and can allow fewer candidate documents to be evaluated by the LtR system. However, this work did not directly quantify the cost savings in the feature extraction or re-ranking stages of the learning-to-rank process, but rather just assumed implicit savings from a reduction in the workload of the LtR model. Work on cascade rankers has investigated the cost of different features and how the actual feature set used for ranking can be different for each stage of the cascade. In particular, more expensive feature sets are generated for smaller sets of documents later in the cascade [57, 102, 252]. It would be interesting to see if our framework could be used to predict the required document cutoff in a more advanced cascade, such as those investigated by Chen et al. [57], which can have up to 5 ranking stages. Similar ideas based on QPP could be used to predict a *dynamic* feature set (and hence, a dynamic ranker) within each

stage. In fact, since cascade rankers operate on smaller subsets of an initial candidate set, it may even be possible to use ideas from *post-retrieval* QPP for more accurate modeling. These ideas are left for future investigation.

6.8 Conclusion

We have presented and validated a unified framework for tuning multi-stage retrieval systems to improve cost and efficiency while retaining competitive effectiveness, thus answering our primary research question, RQ3. The framework uses a strong ranking system to generate *reference lists* which are then used in conjunction with MED [235] to generate training annotations for predictive models, thus answering RQ3.1. We experimented with a *quantile regression* model, which was shown to provide risk-sensitive predictions (RQ3.2). Based on preliminary experimentation (and observations from Chapter 4), we propose a hybrid query processing system that minimizes effectiveness loss while improving the latency and cost of candidate generation. This Hybrid system was shown to yield competitive efficiency and effectiveness trade-offs with respect to well tuned fixed parameter systems (RQ3.3).

There are many open questions related directly to this research and related areas, and some of these ideas have already been explored. In recent work, Mohammad et al. [187] showed that the ideas presented in this chapter (and in the work of Culpepper et al. [75]) can effectively solve the problem of dynamic *shard* cutoff prediction, where a predictor is built to decide how many index shards must be examined to achieve effective results in distributed multi-stage retrieval architectures. It would be interesting to see if this idea could be deployed in tandem with predicting k, especially in the selective search architectures explored by Siedlaczek et al. [227] and Hafizoglu et al. [107]. More recently, similar ideas were used by Petri et al. [197] to train a model which can predict the final threshold of a dynamic pruning algorithm to accelerate query processing *without* requiring relevance judgments. We hope that future research can continue to apply and improve on the ideas presented here.

7 Conclusion

7.1 Summary

Due to the ever-increasing amount of data being generated, supporting efficient and scalable search continues to be a major area of focus in the field of information retrieval. Building large-scale IR systems requires a relentless focus on efficiency in order to maintain the sub-second response times which users have come to expect. A key aspect of efficiency, which has been largely overlooked until now, is the impact of *high percentile tail latency*, which characterizes the worst-case latency experienced by search engine users. In particular, a range of recent studies have outlined the negative impacts of slow response times on users, confirming that efficiency is not just a performance metric, but is absolutely essential to user satisfaction and engagement. Unfortunately, there has been little focus on increasing the understanding of tail latency in the literature, and, as such, there is little understanding of the causes of tail latency within various index organizations, processing schemes, and contexts. The aim of this dissertation was to close this gap in understanding, and to provide a view on the inherent trade-offs that exist between these different query processing methods.

Chapter 4 presented a large-scale empirical analysis of the state-of-the-art index organizations and algorithms for supporting efficient top-k query processing, including DAAT dynamic pruning methods [36, 88] and SAAT early-termination approaches [148]. While DAAT dynamic pruning algorithms are efficient on average, they exhibit less control than SAAT algorithms, making tail latency more difficult to control. On the other hand, the SAAT algorithms are able to provide strict guarantees on tail performance. A further analysis on the under-studied SAAT retrieval approach revealed that strict latency constraints may result in considerable losses in search effectiveness. Alternative heuristics for balancing time/quality trade-offs were then proposed, and a concurrent SAAT traversal algorithm outlined, which was able to provide better trade-offs at a higher processing cost.

Motivated by research on effective rank fusion over *query variations* [19, 23], Chapter 5 explored the novel problem of batch processing a *set* of related queries to improve search effectiveness through result list fusion. The core contribution of this chapter was the proposal of three separate processing paradigms for efficiently handling *multi-queries*, and an analysis of these paradigms

with respect to flexibility, latency, and processing cost. The *parallel fusion* and exhaustive *single pass* approaches can be applied to any given rank fusion algorithm, but are more costly than the *single pass CombSUM* approach, which leverages the additive properties of the CombSUM fusion method [95] to accelerate processing.

Finally, Chapter 6 examined the problem of tail latency in the context of end-to-end multi-stage retrieval. Based on the work of Culpepper et al. [75], we outlined a method for building predictive models that can improve the efficiency and cost of multi-stage retrieval systems. We employed this framework to predict the number of documents required, k, and the number of postings that must be processed, ρ, on a query-by-query basis. Next, we explored how a hybrid query processing system could leverage these predictive models to improve the tail latency of the candidate generation phase of retrieval, while also reducing the downstream costs of the LtR system. This hybrid approach was shown to provide competitive efficiency and effectiveness trade-offs with respect to the fixed parameter approaches that are currently used in practice.

7.2 Future Directions

While this thesis provides various contributions towards further understanding tail latency and the trade-offs between various index organizations and query processing algorithms, there is still a number of interesting open research directions for developing efficient and effective large-scale IR systems.

7.2.1 Impact-ordered Indexing and Score-at-a-Time Traversal

A major finding in this dissertation was that impact-ordered indexes and SAAT traversal allow tail latency to be controlled in a fine-grained manner in many retrieval contexts. This property is attractive for large-scale IR systems, where controlling latency is extremely important for meeting competitive SLAs. However, impact-ordered indexes have many limitations which make them less practical for production search systems.

Flexibility. A major issue with SAAT retrieval is that Boolean matching cannot be done efficiently. This is due to the way in which impact-ordered indexes organize the postings as segments. However, there may be some methods for making efficient Boolean search possible in SAAT systems. For example, Anh and Moffat [6] showed that impact-ordered indexes can be searched in a DAAT-like manner, by opening a cursor on *each* candidate segment, and traversing them at once. A major issue with this approach is that, since each segment is compressed individually, *all* segments would need to be decoded prior to processing. An alternative method could encode the postings within each segment using fixed-length blocks, and decode them on-demand. It would be interesting to further examine the merits of this approach on modern architectures.

Index Updates. Another inherent difficulty posed by impact-ordered indexes is that the impact for each document/term pair must be computed and quantized during indexing. This means that

the ranking method is *fixed* for the remainder of the indexes lifetime. Hence, when adding a new document to the index, the impact score may be inaccurate (since the ranker may depend on the underlying global index statistics). Mohammed et al. [188] recently showed that, for append-only collections, quantization settings from previous time-frames can be applied to new documents without a significant loss in effectiveness. It would be interesting to extend these ideas to totally dynamic indexes, where deletions are also supported, and to examine how both ranking quality and quantization are affected.

Another problem with dynamic impact-ordered indexes is how the relevant segment within a given postings list can be efficiently updated. Since a postings list is a sequence of compressed segments, updating a single segment may require re-organizing the entire postings list. Some possible solutions already exist, including the *postings pools* method from Twitter [39]. In any case, dynamic indexes are a challenging direction for future research.

7.2.2 Cascade Ranking

Much of the work in this dissertation was conducted with regard to utilizing multi-stage retrieval systems to balance efficiency and effectiveness. While a simple two-stage retrieval process was employed in this work, systems with an arbitrary number of ranking and filtering stages have been explored. However, there are still many open directions for improving multi-stage retrieval.

Quantifying End-to-End Cost. Chapter 6 explored how predictive models can be used to improve tail latency during candidate generation, and reduce the cost of the re-ranking stage in a multi-stage retrieval system. However, the cost savings were only assumed implicitly. On the other hand, recent work has focused directly on the cost of the feature extraction and re-ranking phases of cascade rankers, but assume the presence of the initial candidate set [57, 102, 252]. It would be useful to further examine the complex trade-offs between latency, cost, and effectiveness, across all stages of the multi-stage architecture.

Tail Tolerant Cascades. The work of Gallagher et al. [102] explores how *feature cost* can be considered directly in the optimization of machine-learnt ranking cascades, resulting in cascades that are both efficient and effective. An interesting extension to this idea would be to further examine how *latency* may be considered as an optimization goal. Furthermore, to the best of our knowledge, tail latency has not been considered at all for feature extraction or document re-ranking tasks. Clearly, the latency of these operations is crucial for maintaining competitive end-to-end search efficiency, and further exploration into these processes is warranted.

Unified Processing Pipelines. In a production IR system, SERPs are built using information from a number of different indexes and subsystems, including web-page retrieval, question-answering, ad retrieval, image retrieval, and so on. Generally, each of these subsystems is assumed to be isolated, and the final SERP generated by blending components (documents) from each

component together. It would be interesting to further explore how these pipelines could be unified, which may lead to cost and efficiency improvements for large-scale IR systems.

7.2.3 Tail Latency Beyond Web Search

The main theme of this thesis focused on tail latency for large-scale search systems, with a focus on the common web search problem. However, with the increasing popularity of social media and e-commerce, there are many other applications of large-scale search systems with vastly different requirements. For example, social media documents are often much shorter than web documents, indexes may require rapid updates, and query types may be significantly different. In addition, the ranking challenges are different too, as social media often requires temporal ranking, analogous to the price-oriented rankings found on e-commerce sites. It would be interesting to revisit the indexing and retrieval approaches presented in this thesis in the context of large-scale social media or e-commerce collections.

7.2.4 Cost Sensitive Information Retrieval

A final aspect of information retrieval which has been largely ignored is the cost of various search processes and paradigms. While large IR companies may be willing to sacrifice computational resources for small improvements in latency or effectiveness, there are many situations where computational cost cannot be considered infinite. Consider cases such as enterprise search, where computational resources may be proportionally low with respect to the amount of data that must be indexed and searched, or energy-conscious IR systems, which may aim to limit or reduce their power consumption. In these contexts, the priorities of fast and effective search may be in tension with operating at a reduced computational cost. While some work has been conducted on energy-efficient IR [52, 98] and resource-constrained retrieval [131], there are many open directions in this area. Perhaps the most pertinent, however, is how *green* IR systems can be developed to enable sustainable information retrieval systems for future generations [59]. We look forward to focusing on these issues in the future.

BIBLIOGRAPHY

[1] N. Abdul-Jaleel, J. Allan, W. B. Croft, F. Diaz, L. Larkey, X. Li, M. D. Smucker, and C. Wade. UMass at TREC 2004: Novelty and HARD. In *Proc. Text Retrieval Conf. (TREC)*, 2004. ⟨96⟩

[2] G. Amati and C. J. Van Rijsbergen. Probabilistic models of information retrieval based on measuring the divergence from randomness. *ACM Trans. on Information Systems*, 20(4): 357–389, 2002. ⟨11, 127⟩

[3] G. Amati, C. Carpineto, and G. Romano. Query difficulty, robustness, and selective application of query expansion. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 127–137, 2004. ⟨98, 117, 118⟩

[4] G. Amati, E. Ambrosi, M. Bianchi, C. Gaibisso, and G. Gambosi. FUB, IASI-CNR and University of Tor Vergata at TREC 2007 blog track. In *Proc. Text Retrieval Conf. (TREC)*, 2007. ⟨11⟩

[5] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005. ⟨63⟩

[6] V. N. Anh and A. Moffat. Pruning strategies for mixed-mode querying. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 190–197, 2006. ⟨146⟩

[7] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 372–379, 2006. ⟨37⟩

[8] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Software Practice & Experience*, 40(2):131–147, 2010. ⟨63⟩

[9] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 35–42, 2001. ⟨27, 38, 64⟩

[10] N. Asadi and J. Lin. Fast candidate generation for two-phase document ranking: Postings list intersection with Bloom filters. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 2419–2422, 2012. ⟨60, 120⟩

[11] N. Asadi and J. Lin. Training efficient tree-based models for document ranking. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 146–157, 2013. ⟨121⟩

[12] N. Asadi and J. Lin. Document vector representations for feature extraction in multi-stage document ranking. *Information Retrieval*, 16(6):747–768, 2013. ⟨29, 57, 113, 120⟩

[13] N. Asadi and J. Lin. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 997–1000, 2013. ⟨57, 61, 66, 113, 120⟩

[14] N. Asadi, J. Lin, and A. de Vries. Runtime optimizations for prediction with tree-based models. *IEEE Trans. on Knowledge and Data Engineering*, pages 2281–2292, 2012. ⟨121⟩

[15] J. A. Aslam and M. Montague. Models for metasearch. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 276–284, 2001. ⟨93⟩

[16] X. Bai and B. B. Cambazoglu. Impact of response latency on sponsored search. *Information Processing & Management*, 56(1):110–129, 2019. ⟨19⟩

[17] X. Bai, I. Arapakis, B. B. Cambazoglu, and A. Freire. Understanding and leveraging the impact of response latency on user behaviour in web search. *ACM Trans. on Information Systems*, 36(2):1–42, 2017. ⟨19, 22, 54, 117⟩

[18] P. Bailey, A. Moffat, F. Scholer, and P. Thomas. UQV100: A test collection with query variability. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 725–728, 2016. ⟨18, 79, 89, 95, 96, 97, 102⟩

[19] P. Bailey, A. Moffat, F. Scholer, and P. Thomas. Retrieval consistency in the presence of query variations. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 395–404, 2017. ⟨89, 92, 93, 94, 95, 109, 117, 145⟩

[20] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003. ⟨2, 18, 87⟩

[21] N. J. Belkin, C. Cool, W. B. Croft, and J. P. Callan. The effect of multiple query variations on information retrieval system performance. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 339–346, 1993. ⟨89, 95, 97, 109⟩

[22] N. J. Belkin, P. Kantor, E. A. Fox, and J. A. Shaw. Combining the evidence of multiple query representations for information retrieval. *Information Processing & Management*, 31 (3):431–448, 1995. ⟨95⟩

[23] R. Benham and J. S. Culpepper. Risk-reward trade-offs in rank fusion. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 1.1–1.8, 2017. ⟨89, 93, 94, 95, 96, 97, 101, 117, 145⟩

[24] R. Benham, L. Gallagher, J. Mackenzie, T. T. Damessie, R-C. Chen, F. Scholer, A. Moffat, and J. S. Culpepper. RMIT at the 2017 TREC CORE track. In *Proc. Text Retrieval Conf. (TREC)*, 2017. ⟨89, 95, 96, 97⟩

[25] R. Benham, J. S. Culpepper, L. Gallagher, X. Lu, and J. Mackenzie. Towards efficient and effective query variant generation. In *Proc. Conf. on Design of Experimental Search & Information Retrieval Systems (DESIRES)*, pages 62–67, 2018. ⟨96, 121⟩

[26] R. Benham, L. Gallagher, J. Mackenzie, B. Liu, X. Lu, F. Scholer, A. Moffat, and J. S. Culpepper. RMIT at the 2018 TREC CORE track. In *Proc. Text Retrieval Conf. (TREC)*, 2018. ⟨89, 95, 96, 97⟩

[27] R. Benham, J. Mackenzie, A. Moffat, and J. S. Culpepper. Boosting search performance using query variations. *ACM Trans. on Information Systems*, 37(4):41.1–41.25, 2019. ⟨4, 101, 111⟩

[28] G. Bennett, F. Scholer, and A. Uitdenbogerd. A comparative study of probabilistic and language models for information retrieval. In *Proc. Australasian Database Conf. (ADC)*, pages 65–74, 2008. ⟨52⟩

[29] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. ⟨60⟩

[30] P. Boldi and S. Vigna. Compressed perfect embedded skip lists for quick inverted index lookups. In *Proc. Symp. on String Processing and Information Retrieval (SPIRE)*, pages 25–28, 2005. ⟨51⟩

[31] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Communications of the ACM*, 51(12):77–85, 2008. ⟨71⟩

[32] F. Borisyuk, K. Kenthapadi, D. Stein, and B. Zhao. Casmos: A framework for learning candidate selection models over structured queries and documents. In *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 441–450, 2016. ⟨30⟩

[33] L. Breiman. Random forests. *Journal of Machine Learning*, 45(1):5–32, 2001. ⟨127⟩

[34] D. Broccolo, C. Macdonald, S. Orlando, I. Ounis, R. Perego, F. Silvestri, and N. Tonellotto. Load-sensitive selective pruning for distributed search. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 379–388, 2013. ⟨55, 110, 119, 142⟩

[35] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000. ⟨34⟩

[36] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 426–434, 2003. ⟨4, 47, 50, 51, 52, 57, 61, 145⟩

[37] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 97–110, 1985. ⟨38, 52, 74⟩

[38] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 89–96, 2005. ⟨21, 60⟩

[39] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at Twitter. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 1360–1369, 2012. ⟨7, 18, 72, 133, 147⟩

[40] S. Büttcher and C. L. A. Clarke. Indexing time vs. query time: Trade-offs in dynamic information retrieval systems. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 317–318, 2005. ⟨72⟩

[41] S. Büttcher and C. L. A. Clarke. A document-centric approach to static index pruning in text retrieval systems. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 182–189, 2006. ⟨51⟩

[42] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and evaluating search engines*. MIT Press, Cambridge, MA, 2010. ⟨17, 72⟩

[43] B. B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 411–420, 2010. ⟨121⟩

[44] D. Carmel and E. Amitay. Juru at TREC 2006: TAAT versus DAAT in the terabyte track. In *Proc. Text Retrieval Conf. (TREC)*, 2006. ⟨51⟩

[45] D. Carmel and E. Yom-Tov. Estimating the query difficulty for information retrieval. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 2(1):1–89, 2010. ⟨117, 118⟩

[46] D. Carmel and E. Yom-Tov. Estimating the query difficulty for information retrieval. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 911–911, 2010. ⟨118⟩

[47] D. Carmel, E. Yom-Tov, A. Darlow, and D. Pelleg. What makes a query difficult? In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 390–397, 2006. ⟨117⟩

[48] C. Carpineto and G. Romano. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys*, 44(1):1.1 – 1.56, 2012. ⟨12⟩

[49] B. Carterette, V. Pavluy, H. Fang, and E. Kanoulas. Million query track 2009 overview. In *Proc. Text Retrieval Conf. (TREC)*, 2009. ⟨18, 131⟩

[50] C. Castillo. *Effective Web Crawling*. PhD thesis, University of Chile, 2004. ⟨24⟩

[51] M. Catena and N. Tonellotto. Multiple query processing via logic function factoring. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 937–940, 2019. ⟨89, 111⟩

[52] M. Catena, C. Macdonald, and N. Tonellotto. Load-sensitive CPU power management for web search engines. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 751–754, 2016. ⟨148⟩

[53] K. Chakrabarti, S. Chaudhuri, and V. Ganti. Interval-based pruning for top-k processing over compressed lists. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 709–720, 2011. ⟨50⟩

[54] O. Chapelle and Y. Chang. Yahoo! learning to rank challenge overview. In *Journal of Machine Learning Research: Conference and Workshop Proceedings*, pages 1–24, 2010. ⟨114⟩

[55] O. Chapelle, D. Metzler, Y. Zhang, and P. Grinspan. Expected reciprocal rank for graded relevance. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 621–630, 2009. ⟨16⟩

[56] O. Chapelle, Y. Chang, and T-Y. Liu. Future directions in learning to rank. In *Journal of Machine Learning Research: Conference and Workshop Proceedings*, pages 91–100, 2010. ⟨114⟩

[57] R-C. Chen, L. Gallagher, R. Blanco, and J. S. Culpepper. Efficient cost-aware cascade ranking in multi-stage retrieval. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 445–454, 2017. ⟨21, 57, 121, 125, 142, 147⟩

[58] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 219–228, 2009. ⟨34⟩

[59] G. Chowdhury. An agenda for green information retrieval research. *Information Processing & Management*, 48(6):1067–1077, 2012. ⟨148⟩

[60] C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski. An algebra for structured text search and a framework for its implementation. *Computer Journal*, 38(1):43–56, 1995. ⟨13⟩

[61] C. L. A. Clarke, J. S. Culpepper, and A. Moffat. Assessing efficiency—effectiveness tradeoffs in multi-stage retrieval systems without using relevance judgments. *Information Retrieval*, 19(4):351–377, 2016. ⟨61, 113, 120, 123, 124, 125⟩

[62] C. Cleverdon. The Cranfield tests on index language devices. *Aslib proceedings*, 19(6): 173–194, 1967. ⟨14⟩

[63] C. Cole. A theory of information need for information retrieval that connects information to knowledge. *Journal of the American Society for Information Science and Technology*, 62 (7):1216–1231, 2011. ⟨8⟩

[64] K. Collins-Thompson and J. Callan. Estimation and use of uncertainty in pseudo-relevance feedback. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 303–310, 2007. ⟨118⟩

[65] G. V. Cormack, C. L. A. Clarke, and S. Büttcher. Reciprocal rank fusion outperforms Condorcet and individual rank learning methods. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 758–759, 2009. ⟨92, 93⟩

[66] M. Crane, A. Trotman, and R. O'Keefe. Maintaining discriminatory power in quantized indexes. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 1221–1224, 2013. ⟨28, 62, 64, 79, 88, 102, 131⟩

[67] M. Crane, A. Trotman, and D. Eyers. Improving throughput of a pipeline model indexer. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 2.1–2.4, 2015. ⟨24⟩

[68] M. Crane, J. S. Culpepper, J. Lin, J. Mackenzie, and A. Trotman. A comparison of document-at-a-time and score-at-a-time query evaluation. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 201–210, 2017. ⟨75, 78, 83, 113, 134⟩

[69] N. Craswell and M. Szummer. Random walks on the click graph. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 239–246, 2007. ⟨96⟩

[70] N. Craswell, D. Fetterly, M. Najork, S. Robertson, and E. Yilmaz. Microsoft research at TREC 2009: Web and relevance feedback tracks. In *Proc. Text Retrieval Conf. (TREC)*, 2009. ⟨114⟩

[71] N. Craswell, R. Jones, G. Dupret, and E. Viegas. Web search click data workshop. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, 2009. ⟨83⟩

[72] N. Craswell, B. Billerbeck, D. Fetterly, and M. Najork. Robust query rewriting using anchor data. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 335–344, 2013. ⟨12⟩

[73] S. Cronen-Townsend, Y. Zhou, and W. B. Croft. Predicting query performance. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 299–306, 2002. ⟨117⟩

[74] J. S. Culpepper. Single query optimisation is the root of all evil. In *Proc. Conf. on Design of Experimental Search & Information Retrieval Systems (DESIRES)*, pages 100–100, 2018. ⟨95⟩

[75] J. S. Culpepper, C. L. A. Clarke, and J. Lin. Dynamic cutoff prediction in multi-stage retrieval systems. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 17–24, 2016. ⟨5, 113, 121, 124, 125, 126, 127, 143, 146⟩

[76] V. Dang and W. B. Croft. Query reformulation using anchor text. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 41–50, 2010. ⟨12⟩

[77] V. Dang, M. Bendersky, and W. B. Croft. Two-stage learning to rank for information retrieval. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 423–434, 2013. ⟨61⟩

[78] D. Dato, C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini. Fast ranking with additive ensembles of oblivious and non-oblivious regression trees. *ACM Trans. on Information Systems*, 35(2):15.1–15.31, 2016. ⟨121⟩

[79] J. C. de Borda. Mémoire sur les élections au scrutin. *Histoire de l'Academie Royale des Sciences pour 1781 (Paris, 1784)*, 1784. ⟨93⟩

[80] L. L. S. de Carvalho, E. S. de Moura, C. M. Daoud, and A. S. da Silva. Heuristics to improve the BMW method and its variants. *Journal of Information & Data Management*, 6 (3):178–191, 2015. ⟨51⟩

[81] J. Dean. Challenges in building large-scale information retrieval systems: Invited talk. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 1–1, 2009. ⟨31⟩

[82] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013. ⟨1, 3, 54, 58, 97, 133⟩

[83] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 1535–1544, 2016. ⟨34, 35, 62, 72, 102⟩

[84] T. G. Diamond. Information retrieval using dynamic evidence combination, 1998. Unpublished PhD Dissertation Proposal. ⟨92⟩

[85] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. Optimizing top-*k* document retrieval strategies for block-max indexes. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 113–122, 2013. ⟨63⟩

[86] B. T. Dinçer, C. Macdonald, and I. Ounis. Risk-sensitive evaluation and learning to rank using multiple baselines. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 483–492, 2016. ⟨117, 125⟩

[87] B. T. Dinçer, C. Macdonald, and I. Ounis. Hypothesis testing for the risk-sensitive evaluation of retrieval systems. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 23–32, 2014. ⟨117, 125⟩

[88] S. Ding and T. Suel. Faster top-*k* document retrieval using block-max indexes. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 993–1002, 2011. ⟨4, 35, 50, 52, 57, 61, 62, 64, 145⟩

[89] J. Duda. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding, 2013. arXiv:1311.2540. ⟨31⟩

[90] A. Elbagoury, M. Crane, and J. Lin. Rank-at-a-time query processing. In *Proc. Int. Conf. on Theory of Information Retrieval (ICTIR)*, pages 229–232, 2016. ⟨40⟩

[91] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Trans. on Information Systems*, 2(4):267–288, 1984. ⟨29⟩

[92] M. Fernández-Delgado, M. S. Sirsat, E. Cernadas, S. Alawadi, S. Barro, and M. Febrero-Bande. An extensive experimental survey of regression methods. *Neural Networks*, 111: 11–34. ⟨142⟩

[93] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitski, E. Vee, S. Venkatesan, and J. Zien. Efficiently evaluating complex boolean expressions. pages 3–14, 2010. ⟨30, 111⟩

[94] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien. Evaluation strategies for top-*k* queries over memory-resident inverted indexes. *Proc. Conf. on Very Large Databases (VLDB)*, 4(12):1213–1224, 2011. ⟨38, 45, 52, 74⟩

[95] E. A. Fox and J. A. Shaw. Combination of multiple searches. In *Proc. Text Retrieval Conf. (TREC)*, pages 243–252, 1993. ⟨92, 94, 101, 103, 146⟩

[96] G. Francès, X. Bai, B. B. Cambazoglu, and R. Baeza-Yates. Improving the efficiency of multi-site web search engines. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 3–12, 2014. ⟨133⟩

[97] H. D. Frank and I. Taksa. Comparing rank and score combination methods for data fusion in information retrieval. *Information Retrieval*, 8(3):449–480, 2005. ⟨93⟩

[98] A. Freire, C. Macdonald, N. Tonellotto, I. Ounis, and F. Cacheda. A self-adapting latency/power tradeoff model for replicated search engines. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 13–22, 2014. ⟨148⟩

[99] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000. ⟨129⟩

[100] J. H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38 (4):367–378, 2002. ⟨129⟩

[101] L. Gallagher, J. Mackenzie, R. Benham, R-C. Chen, F. Scholer, and J. S. Culpepper. RMIT at the NTCIR-13 We Want Web task. In *Proc. NII Testbeds and Community for Information Access Research (NTCIR)*, 2017. ⟨13⟩

[102] L. Gallagher, R-C. Chen, R. Blanco, and J. S. Culpepper. Joint optimization of cascade ranking models. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 15–23, 2019. ⟨121, 125, 142, 147⟩

[103] Y. Ganjisaffar, R. Caruana, and C. Lopes. Bagging gradient-boosted trees for high precision, low variance ranking models. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 85–94, 2011. ⟨125⟩

[104] S. Geva and C. M. de Vries. TopSig: Topology preserving document signatures. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 333–338, 2011. ⟨29⟩

[105] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He. Bitfunnel: Revisiting signatures for search. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 605–614, 2017. ⟨29, 61, 86, 121⟩

[106] L. Grauer and A. Lomakina. On the effect of "stupid" search components on user interaction with search engines. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 1727–1730, 2015. ⟨117⟩

[107] F. Hafizoglu, E. C. Kucukoglu, and I. S. Altingovde. On the efficiency of selective search. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 705–712, 2017. ⟨143⟩

[108] D. Harman. Is the Cranfield paradigm outdated? In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 1–1, 2010. ⟨15⟩

[109] C. Hauff. *Predicting the effectiveness of queries and retrieval systems*. PhD thesis, University of Twente, 2010. ⟨118⟩

[110] C. Hauff, D. Hiemstra, and F. de Jong. A survey of pre-retrieval query performance predictors. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 1419–1420, 2008. ⟨117, 118, 142⟩

[111] B. He and I. Ounis. Inferring query performance using pre-retrieval predictors. In *Proc. Symp. on String Processing and Information Retrieval (SPIRE)*, pages 43–54, 2004. ⟨117, 118⟩

[112] B. He and I. Ounis. Query performance prediction. *Journal of Information Systems*, 31(7): 585–594, 2006. ⟨117⟩

[113] B. He and I. Ounis. Combining fields for query expansion and adaptive query expansion. *Information Processing & Management*, 43(5):1294–1307, 2007. ⟨118⟩

[114] P. J. Huber. Robust estimation of a location parameter. *Annals of Mathematical Statistics*, 35(1):73–101, 1964. ⟨129⟩

[115] D. A. Hull. Stemming algorithms: A case study for detailed evaluation. *Journal of the American Society for Information Science and Technology*, 47(1):70–84, 1996. ⟨24, 25⟩

[116] S. Huo, M. Zhang, Y. Liu, and S. Ma. Improving tail query performance by fusion model. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 559–658, 2014. ⟨92, 93⟩

[117] S-W. Hwang, S. Kim, Y. He, S. Elnikety, and S. Choi. Prediction and predictability for search query acceleration. *ACM Trans. on the Web*, 10(3):19.1–19.28, August 2016. ⟨22, 110⟩

[118] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Trans. on Information Systems*, 20(4):422–446, 2002. ⟨16⟩

[119] M. Jeon, S. Kim, S-W. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: Taming tail latencies in web search. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 253–262, 2014. ⟨4, 19, 54, 59, 82, 83, 84, 87, 88, 108, 119, 127, 134, 142⟩

[120] X-F. Jia, A. Trotman, and R. O'Keefe. Efficient accumulator initialization. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 44–51, 2010. ⟨37, 77, 78, 87⟩

[121] J. Jimmy, G. Zuccon, B. Koopman, and G. Demartini. Health cards for consumer health search. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 35–44, 2019. ⟨117⟩

[122] T. Joachims. Optimizing search engines using clickthrough data. In *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 133–142, 2002. ⟨121, 142⟩

[123] S. Jonassen and S. E. Bratsberg. Efficient compressed inverted index skipping for disjunctive text-queries. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 530–542, 2011. ⟨45⟩

[124] R. Jones, B. Rey, O. Madani, and W. Greiner. Generating query substitutions. In *Proc. Conf. on the World Wide Web (WWW)*, pages 387–396, 2006. ⟨12⟩

[125] T. Kaler, Y. He, and S. Elnikety. Optimal reissue policies for reducing tail latency. In *Proc. ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 195–206, 2017. ⟨55⟩

[126] A. Kane and F. W. Tompa. Skewed partial bitvectors for list intersection. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 263–272, 2014. ⟨30⟩

[127] A. Kane and F. W. Tompa. Split-lists and initial thresholds for WAND-based search. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 877–880, 2018. ⟨51, 72⟩

[128] S. Karimi, S. Pohl, F. Scholer, L. Cavedon, and J. Zobel. Boolean versus ranked querying for biomedical systematic reviews. *BMC Medical Informatics and Decision Making*, 10(1): 58–78, 2010. ⟨9⟩

[129] S. Kim, Y. He, S-W. Hwang, S. Elnikety, and S. Choi. Delayed-Dynamic-Selective (DDS) prediction for reducing extreme tail latency in web search. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 7–16, 2015. ⟨54, 83, 87, 108, 119, 127, 134, 142⟩

[130] S. Kim, T. Lee, S w. Hwang, and S. Elnikety. List intersection for web search: Algorithms, cost models, and optimizations. *Proc. Conf. on Very Large Databases (VLDB)*, 12(1):1–13, 2018. ⟨119, 142⟩

[131] Y. Kim, J. Callan, J. S. Culpepper, and A. Moffat. Efficient distributed selective search. *Information Retrieval*, 20(3):1–32, 2016. ⟨18, 87, 133, 142, 148⟩

[132] Y. Kim, J. Callan, J. S. Culpepper, and A. Moffat. Load-balancing in distributed selective search. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 905–908, 2016. ⟨142⟩

[133] R. W. Koenker and G. Bassett. Regression quantiles. *Econometrica*, 46(1):33–50, 1978. ⟨127⟩

[134] R. Kohavi, A. Deng, B. Frasca, T. Walker, Y. Xu, and N. Pohlmann. Online controlled experiments at large scale. In *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 1168–1176, 2013. ⟨2⟩

[135] J. Kong, A. Scott, and G. M. Goerg. Improving semantic topic clustering for search queries with word co-occurrence and bigraph co-clustering. *Google Inc*, 2016. URL `https://ai.google/research/pubs/pub45569.pdf`. ⟨96⟩

[136] A. K. Kozorovitsky and O. Kurland. Cluster-based fusion of retrieved lists. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 893–902, 2011. ⟨92⟩

[137] A. Kulkarni and J. Callan. Selective search: Efficient and effective search of large textual collections. *ACM Trans. on Information Systems*, 33(4):17.1–17.33, 2015. ⟨18⟩

[138] G. Kumaran and J. Allan. Selective user interaction. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 923–926, 2007. ⟨118⟩

[139] O. Kurland and J. S. Culpepper. Fusion in information retrieval: SIGIR 2018 half-day tutorial. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 1383–1386, 2018. ⟨92, 95⟩

[140] O. Kurland, A. Shtok, S. Hummel, F. Raiber, D. Carmel, and O. Rom. Back to the roots: A probabilistic framework for query-performance prediction. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 823–832, 2012. ⟨117⟩

[141] P. Lacour, C. Macdonald, and I. Ounis. Efficiency comparison of document matching techniques. In *ECIR Workshop on Efficiency Issues in IR*, 2008. ⟨51⟩

[142] A. Lawler. Writing gets a rewrite. *Science*, 292(5526):2418–2420, 2001. ⟨7⟩

[143] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software Practice & Experience*, 41(1):1–29, 2015. ⟨57, 62, 63⟩

[144] D. Lemire, L. Boytsov, and N. Kurz. SIMD compression and the intersection of sorted integers. *Software Practice & Experience*, 46(6):723–749, 2016. ⟨30, 62⟩

[145] D. Lemire, N. Kurz, and C. Rupp. Stream VByte: Faster byte-oriented integer compression. *Information Processing Letters*, 130:1–6, 2018. ⟨31⟩

[146] S. Liang, Z. Ren, and M. de Rijke. Fusion helps diversification. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 303–312, 2014. ⟨92⟩

[147] J. Lin. The neural hype and comparisons against weak baselines. *SIGIR Forum*, 52(2): 40–51, 2019. ⟨52⟩

[148] J. Lin and A. Trotman. Anytime ranking for impact-ordered indexes. In *Proc. Int. Conf. on Theory of Information Retrieval (ICTIR)*, pages 301–304, 2015. ⟨4, 39, 55, 61, 68, 75, 83, 84, 102, 119, 134, 145⟩

[149] J. Lin and A. Trotman. The role of index compression in score-at-a-time query evaluation. *Information Retrieval*, 20(3):199–220, 2017. ⟨40, 79⟩

[150] J. Lin, M. Crane, A. Trotman, J. Callan, I. Chattopadhyaya, J. Foley, G. Ingersoll, C. Macdonald, and S. Vigna. Toward reproducible baselines: The open-source IR reproducibility challenge. In *Proc. European Conf. on Information Retrieval (ECIR)*, 2016. ⟨11, 75⟩

[151] B. Liu, X. Lu, O. Kurland, and J. S. Culpepper. Improving search effectiveness with field-based relevance modelling. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 11.1–11.4, 2018. ⟨13⟩

[152] T-Y. Liu. Learning to rank for information retrieval. *Foundations & Trends in Information Retrieval*, 3(3):225–331, 2009. ⟨14, 126⟩

[153] X. Lu. *Efficient and Effective Retrieval Using Higher-Order Proximity Models*. PhD thesis, RMIT University, 2018. ⟨17⟩

[154] X. Lu, A. Moffat, and J. S. Culpepper. How effective are proximity scores in term dependency models? In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 89–92, 2014. ⟨13⟩

[155] X. Lu, A. Moffat, and J. S. Culpepper. On the cost of extracting proximity features for term-dependency models. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 293–302, 2015. ⟨13, 63, 120⟩

[156] X. Lu, A. Moffat, and J. S. Culpepper. Efficient and effective higher order proximity modeling. In *Proc. Int. Conf. on Theory of Information Retrieval (ICTIR)*, pages 21–30, 2016. ⟨13⟩

[157] X. Lu, A. Moffat, and J. S. Culpepper. The effect of pooling and evaluation depth on IR metrics. *Information Retrieval*, 19(4):416–445, 2016. ⟨17, 64⟩

[158] C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini. Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 73–82, 2015. ⟨121⟩

[159] C. Macdonald and N. Tonellotto. Upper bound approximation for BlockMaxWand. In *Proc. Int. Conf. on Theory of Information Retrieval (ICTIR)*, pages 273–276, 2017. ⟨72⟩

[160] C. Macdonald, I. Ounis, and N. Tonellotto. Upper-bound approximations for dynamic pruning. *ACM Trans. on Information Systems*, 29(4):17.1–17.28, 2011. ⟨72⟩

[161] C. Macdonald, R. McCreadie, R. Santos, and I. Ounis. From puppy to maturity: Experiences in developing terrier. In *Proc. SIGIR 2012 Workshop on Open Source Information Retrieval*, 2012. ⟨120⟩

[162] C. Macdonald, N. Tonellotto, and I. Ounis. Learning to predict response times for online query scheduling. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 621–630, 2012. ⟨55, 83, 117, 119, 120, 142⟩

[163] C. Macdonald, N. Tonellotto, and I. Ounis. Effect of dynamic pruning safety on learning to rank effectiveness. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 1051–1052, 2012. ⟨50, 66⟩

[164] C. Macdonald, R. L. T. Santos, and I. Ounis. The whens and hows of learning to rank for web search. *Information Retrieval*, 16(5):584–628, 2013. ⟨14, 57, 114, 126⟩

[165] C. Macdonald, R. L. T. Santos, I. Ounis, and B. He. About learning models with multiple query-dependent features. *ACM Trans. on Information Systems*, 31(3):11.1–11.39, 2013. ⟨14, 57, 120, 126⟩

[166] C. Macdonald, N. Tonellotto, and I. Ounis. Efficient & effective selective query rewriting with efficiency predictions. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 495–504, 2017. ⟨83⟩

[167] J. Mackenzie. Managing tail latencies in large scale IR systems. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 1369–1369, 2017. ⟨88⟩

[168] J. Mackenzie, F. M. Choudhury, and J. S. Culpepper. Efficient location-aware web search. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 4.1–4.8, 2015. ⟨63⟩

[169] J. Mackenzie, R-C. Chen, and J. S. Culpepper. RMIT at the TREC 2016 LiveQA track. In *Proc. Text Retrieval Conf. (TREC)*, 2016. ⟨61⟩

[170] J. Mackenzie, F. Scholer, and J. S. Culpepper. Early termination heuristics for score-at-a-time index traversal. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 8.1–8.8, 2017. ⟨108, 119⟩

[171] J. Mackenzie, A. Mallia, M. Petri, J. S. Culpepper, and T. Suel. Compressing inverted indexes with recursive graph bisection: A reproducibility study. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 339–352, 2019. ⟨34, 35, 62, 72, 102⟩

[172] A. Mallia and E. Porciani. Faster BlockMax WAND with longer skipping. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 771–778, 2019. ⟨72, 88⟩

[173] A. Mallia, G. Ottaviano, E. Porciani, N. Tonellotto, and R. Venturini. Faster BlockMax WAND with variable-sized blocks. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 625–634, 2017. ⟨50, 52, 72, 88, 102, 142⟩

[174] A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel. PISA: Performant indexes and search for academia. In *Proc. of the Open-Source IR Replicability Challenge (OSIRRC) at SIGIR 2019*, pages 50–56, 2019. ⟨11, 88⟩

[175] A. Mallia, M. Siedlaczek, and T. Suel. An experimental study of index compression and DAAT query processing methods. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 353–368, 2019. ⟨33, 52, 62, 72, 107, 142⟩

[176] B. Mansouri, M. S. Zahedi, R. Campos, and M. Farhoodi. Online job search: Study of users' search behavior using search engine query logs. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 1185–1188, 2018. ⟨7⟩

[177] M. Mayer. In search of a better, faster, stronger web. Velocity, 2009. ⟨19⟩

[178] D. Metzler and W. B. Croft. A Markov random field model for term dependencies. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 472–479, 2005. ⟨13⟩

[179] A. Moffat. Computing maximized effectiveness distance for recall-based metrics. *IEEE Trans. on Knowledge and Data Engineering*, 30(1):198–203, 2018. ⟨123⟩

[180] A. Moffat and M. Petri. ANS-Based index compression. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 677–686, 2017. ⟨31, 32⟩

[181] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, 2000. ⟨31⟩

[182] A. Moffat and J. Zobel. Self indexing inverted files for fast text retrieval. *ACM Trans. on Information Systems*, 14(4):349–379, 1996. ⟨51⟩

[183] A. Moffat and J. Zobel. Rank-biased precision for measurement of retrieval effectiveness. *ACM Trans. on Information Systems*, 27(1):2, 2008. ⟨16⟩

[184] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory-efficient ranking. *Information Processing & Management*, 30(6):733–744, 1994. ⟨27, 38⟩

[185] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, 10(3):205–231, 2007. ⟨118⟩

[186] A. Moffat, F. Scholer, P. Thomas, and P. Bailey. Pooled evaluation over query variations: Users are as diverse as systems. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 1759–1762, 2015. ⟨89, 95⟩

[187] H. R. Mohammad, K. Xu, J. Callan, and J. S. Culpepper. Dynamic shard cutoff prediction for selective search. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 85–94, 2018. ⟨127, 143⟩

[188] S. Mohammed, M. Crane, and J. Lin. Quantization in append-only collections. In *Proc. Int. Conf. on Theory of Information Retrieval (ICTIR)*, pages 265–268, 2017. ⟨72, 147⟩

[189] M. Montague and J. A. Aslam. Relevance score normalization for metasearch. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 427–433, 2001. ⟨92, 95⟩

[190] A. Mourão and J. Magalhães. Low-complexity supervised rank fusion models. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 1691–1694, 2018. ⟨92, 93⟩

[191] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 273–282, 2014. ⟨31, 57, 102⟩

[192] J. Pedersen. Query understanding at Bing. In *Invited Talk: Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, 2010. ⟨3, 21, 60, 86⟩

[193] S. A. Perry and P. Willett. A review of the use of inverted files for best match searching in information retrieval systems. *Journal of Information Science*, 6(2-3):59–66, 1983. ⟨38⟩

[194] M. Persin. Document filtering for fast ranking. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 339–348, 1994. ⟨26, 38, 51⟩

[195] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency sorted indexes. *Journal of the American Society for Information Science and Technology*, 47(10): 749–764, 1996. ⟨26, 38, 51⟩

[196] M. Petri, J. S. Culpepper, and A. Moffat. Exploring the magic of WAND. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 58–65, 2013. ⟨51, 53, 113⟩

[197] M. Petri, A. Moffat, J. Mackenzie, J. S. Culpepper, and D. Beck. Accelerated query processing via similarity score prediction. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 485–494, 2019. ⟨51, 88, 127, 134, 143⟩

[198] G. E. Pibiri. *Space and Time-Efficient Data Structures for Massive Datasets*. PhD thesis, University of Pisa, 2018. ⟨xi, 33⟩

[199] G. E. Pibiri and R. Venturini. Clustered Elias-Fano indexes. *ACM Trans. on Information Systems*, 36(1):2.1–2.33, 2017. ⟨31⟩

[200] G. E. Pibiri and R. Venturini. Techniques for inverted index compression, 2019. arXiv:1908.10598. ⟨33⟩

[201] G. E. Pibiri and R. Venturini. On optimally partitioning variable-byte codes. *IEEE Trans. on Knowledge and Data Engineering*, to appear, 2019. ⟨31⟩

[202] G. E. Pibiri, M. Petri, and A. Moffat. Fast dictionary-based compression for inverted indexes. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 6–14, 2019. ⟨31, 32, 62⟩

[203] J. Plaisance, N. Kurz, and D. Lemire. Vectorized VByte decoding. In *Proc. Int. Symp. on Web AlGorithms (iSWAG)*, 2015. ⟨31⟩

[204] J. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 275–281, 1998. ⟨12, 57, 127⟩

[205] T. Qin, T-Y. Liu, J. Xu, and H. Li. Letor: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval*, 13(4):346–374, 2010. ⟨114⟩

[206] F. Raiber and O. Kurland. Query-performance prediction: Setting the expectations straight. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 13–22, 2014. ⟨98, 117⟩

[207] V. Ramaswamy, R. Konow, A. Trotman, J. Degenhardt, and N. Whyte. Document reordering is good, especially for e-commerce. In *Proc. of the SIGIR 2017 Workshop on eCommerce*. ⟨35⟩

[208] V. C. Raykar, B. Krishnapuram, and S. Yu. Designing efficient cascaded classifiers: Tradeoff between accuracy and cost. In *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 853–860, 2010. ⟨121⟩

[209] D. Reinsel, J. Gantz, and J. Rydning. The digitization of the world: From edge to core. IDC White Paper #US44413318, 2018. ⟨1⟩

[210] K. M. Risvik, T. Chilimbi, H. Tan, K. Kalyanaraman, and C. Anderson. Maguro, a system for indexing and searching over very large text collections. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 727–736, 2013. ⟨13, 18, 20, 60, 88, 133⟩

[211] S. E. Robertson and H. Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations & Trends in Information Retrieval*, 3:333–389, 2009. ⟨11, 13⟩

[212] S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *Proc. Text Retrieval Conf. (TREC)*, 1994. ⟨11, 57, 127⟩

[213] S. E. Robertson, H. Zaragoza, and M. Taylor. Simple BM25 extension to multiple weighted fields. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 42–49, 2004. ⟨13⟩

[214] H. Roitman. An enhanced approach to query performance prediction using reference lists. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 869–872, 2017. ⟨117⟩

[215] O. Rojas, V. Gil-Costa, and M. Marin. Efficient parallel block-max WAND algorithm. In *Proc. European Conf. on Parallel and Distributed Computing (Euro-Par)*, pages 394–405, 2013. ⟨75⟩

[216] G. Salton. The use of citations as an aid to automatic content analysis, 1962. Technical Report. ISR-2, Section III, Harvard Computation Laboratory, Cambridge, MA. ⟨10, 57⟩

[217] G. Salton. *Automatic Index Organization and Retrieval*. McGraw-Hill, New York, NY, 1968. ⟨10, 57⟩

[218] M. Sanderson. Test collection based evaluation of information retrieval systems. *Foundations & Trends in Information Retrieval*, 4(4):247–375, 2010. ⟨17⟩

[219] M. Sanderson and W. B. Croft. The history of information retrieval research. *Proceedings of the IEEE*, 100:1444–1451, 2012. ⟨7⟩

[220] T. Saracevic and P. Kantor. A study of information seeking and retrieving. III. Searchers, searches, and overlap. *Journal of the American Society for Information Science and Technology*, 39(3):197–216, 1988. ⟨95⟩

[221] H. Scells, L. Azzopardi, G. Zuccon, and B. Koopman. Query variation performance prediction for systematic reviews. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 1089–1092, 2018. ⟨96, 118⟩

[222] H. Scells, G. Zuccon, and B. Koopman. Automatic boolean query refinement for systematic review literature search. In *Proc. Conf. on the World Wide Web (WWW)*, pages 1646–1656, 2019. ⟨9⟩

[223] D. J. Schuirmann. A comparison of the two one-sided tests procedure and the power approach for assessing the equivalence of average bioavailability. *J. Pharmacokinetics and Biopharmaceutics*, 15(6):657–680, 1987. ⟨140⟩

[224] E Schurman and J Brutlag. Performance related changes and their user impact. Velocity, 2009. ⟨2, 19⟩

[225] D. Sheldon, M. Shokouhi, M. Szummer, and N. Craswell. LambdaMerge: Merging the results of query reformulations. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 795–804, 2011. ⟨92, 93, 96⟩

[226] A. Shtok, O. Kurland, and D. Carmel. Query performance prediction using reference lists. *ACM Trans. on Information Systems*, 34(4):19.1–19.34, 2016. ⟨117⟩

[227] M. Siedlaczek, J. Rodriguez, and T. Suel. Exploiting global impact ordering for higher throughput selective search. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 12–19, 2019. ⟨143⟩

[228] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. European Conf. on Information Retrieval (ECIR)*, pages 101–112, 2007. ⟨34, 62⟩

[229] A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Engineering Bulletin*, 24(4):35–43, 2001. ⟨7, 9, 57⟩

[230] A. F. Smeaton and C. J. van Rijsbergen. The nearest neighbour problem in information retrieval: An algorithm using upperbounds. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 83–87, 1981. ⟨38⟩

[231] P. Sondhi, M. Sharma, P. Kolari, and C. Zhai. A taxonomy of queries for e-commerce search. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 1245–1248, 2018. ⟨7⟩

[232] I. Stanton, S. Ieong, and N. Mishra. Circumlocution in diagnostic medical queries. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 133–142, 2014. ⟨96⟩

[233] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. SIMD-based decoding of posting lists. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 317–326, 2011. ⟨31⟩

[234] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 175–182, 2007. ⟨51⟩

[235] L. Tan and C. L. A. Clarke. A family of rank similarity measures based on maximized effectiveness difference. *IEEE Trans. on Knowledge and Data Engineering*, 27(11):2865–2877, 2015. ⟨122, 141, 143⟩

[236] J. Teevan, D. Ramage, and M. R. Morris. #twittersearch: A comparison of microblog search and web search. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 35–44, 2011. ⟨7⟩

[237] L. H. Thiel and H. S. Heaps. Program design for retrospective searches on large data bases. *Information Storage and Retrieval*, 8(1):1–20, 1972. ⟨31⟩

[238] P. Thomas, F. Scholer, P. Bailey, and A. Moffat. Tasks, queries, and rankers in pre-retrieval performance prediction. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 11.1–11.4, 2017. ⟨96, 98, 117, 118⟩

[239] N. Tonellotto, C. Macdonald, and I. Ounis. Query efficiency prediction for dynamic pruning. In *Proc. 9th Workshop on Large-scale and Distributed Information Retrieval*, pages 3–8, 2011. ⟨119⟩

[240] N. Tonellotto, C. Macdonald, and I. Ounis. Efficient and effective retrieval using selective pruning. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 63–72, 2013. ⟨54, 119⟩

[241] N. Tonellotto, C. Macdonald, and I. Ounis. Efficient query processing for scalable web search. *Foundations & Trends in Information Retrieval*, 12(4-5):319–500, 2018. ⟨38, 42, 120, 121⟩

[242] A. Trotman. Compression, SIMD, and postings lists. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 50.50–50.57, 2014. ⟨31, 63, 64, 79, 131⟩

[243] A. Trotman and M. Crane. Micro- and macro-optimizations of SaaT search. *Software Practice & Experience*, 49(5):942–950, 2019. ⟨40, 72, 88⟩

[244] A. Trotman and J. Lin. In vacuo and in situ evaluation of SIMD codecs. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 1–8, 2016. ⟨31, 63, 79, 131⟩

[245] A. Trotman, J. Degenhardt, and S. Kallumadi. The architecture of eBay search. In *Proc. of the SIGIR 2017 Workshop on eCommerce*. ⟨7, 18⟩

[246] A. Trotman, X-F. Jia, and M. Crane. Towards an efficient and effective search engine. In *Proc. of the SIGIR 2012 Workshop on Open Source Information Retrieval*, pages 40–47, 2012. ⟨11, 62, 63, 130⟩

[247] A. Trotman, A. Puurula, and B. Burgess. Improvements to BM25 and language models examined. In *Proc. Australasian Document Computing Symp. (ADCS)*, pages 58–65, 2014. ⟨11, 52⟩

[248] H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing & Management*, 31(6):831–850, 1995. ⟨37, 42, 45, 51, 52, 57⟩

[249] S. Vigna. Quasi-succinct indices. In *Proc. ACM Int. Conf. on Web Search and Data Mining (WSDM)*, pages 83–92, 2013. ⟨32⟩

[250] C. C. Vogt and G. W. Cottrell. Predicting the performance of linearly combined IR systems. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 190–196, 1998. ⟨92⟩

[251] C. C. Vogt and G. W. Cottrell. Fusion via a linear combination of scores. *Information Retrieval*, 1(3):151–173, 1999. ⟨92⟩

[252] L. Wang, J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 105–114, 2011. ⟨21, 57, 60, 113, 121, 142, 147⟩

[253] L. Wang, P. N. Bennett, and K. Collins-Thompson. Robust ranking models via risk-sensitive optimization. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 761–770, 2012. ⟨125⟩

[254] L. Wang, J. Lin, D. Metzler, and J. Han. Learning to efficiently rank on big data. In *Proc. Conf. on the World Wide Web (WWW)*, pages 209–210, 2014. ⟨21⟩

[255] Q. Wang, C. Dimpoloulos, and T. Suel. Fast first-phase candidate generation for cascading rankers. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 295–304, 2016. ⟨61⟩

[256] W. Webber, A. Moffat, and J. Zobel. A similarity measure for indefinite rankings. *ACM Trans. on Information Systems*, 28(4):20.1–20.38, 2010. ⟨122⟩

[257] J. R. Wen, J. Y. Nie, and H. J. Zhang. Query clustering using user logs. *ACM Trans. on Information Systems*, 20(1):59–81, 2002. ⟨96⟩

[258] R. W. White, M. Richardson, M. Bilenko, and A. P. Heath. Enhancing web search by promoting multiple search engine use. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 43–50, 2008. ⟨118⟩

[259] A. F. Wicaksono. Measuring job search effectiveness. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 1453–1453, 2019. ⟨7⟩

[260] H. E. Williams, J. Zobel, and P. Anderson. Fast phrase querying with combined indexes. *ACM Trans. on Information Systems*, 22(4):573–594, 2004. ⟨29⟩

[261] M. Wu, D. Hawking, A. Turpin, and F. Scholer. Using anchor text for homepage and topic distillation search tasks. *Journal of the American Society for Information Science and Technology*, 63(6):1235–1255, 2012. ⟨92, 94, 95⟩

[262] G-R. Xue, H-J. Zeng, Z. Chen, Y. Yu, W-Y. Ma, W. Xi, and W. Fan. Optimizing web search using web click-through data. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 118–126, 2004. ⟨96⟩

[263] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. Conf. on the World Wide Web (WWW)*, pages 401–410, 2009. ⟨31, 63⟩

[264] H. P. Young. Condorcet's theory of voting. *American Political Science Review*, 82(4): 1231–1244, 1988. ⟨93⟩

[265] J-M. Yun, Y. He, S. Elnikety, and S. Ren. Optimal aggregation policy for reducing tail latency of web search. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 63–72, 2015. ⟨55, 108⟩

[266] H. Zamani, W. B. Croft, and J. S. Culpepper. Neural query performance prediction using weak supervision from multiple signals. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 105–114, 2018. ⟨117⟩

[267] H. Zaragoza, N. Craswell, M. Taylor, S. Saria, and S. E. Robertson. Microsoft Cambridge at TREC 2004: Web and HARD track. In *Proc. Text Retrieval Conf. (TREC)*, 2004. ⟨13⟩

[268] O. Zendel, A. Shtok, F. Raiber, O. Kurland, and J. S. Culpepper. Information needs, queries, and query performance prediction. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 395–404, 2019. ⟨118⟩

[269] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. Conf. on the World Wide Web (WWW)*, pages 387–396, 2008. ⟨31⟩

[270] M. Zhang, D. Kuang, G. Hua, Y. Liu, and S. Ma. Is learning to rank effective for web search? In *Proc. SIGIR 2008 Workshop on Learning to Rank for Information Retrieval (LR4IR)*, 2009. ⟨14, 114⟩

[271] Y. Zhou and W. B. Croft. Ranking robustness: A novel framework to predict query performance. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 567–574, 2006. ⟨117⟩

[272] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996. ⟨39⟩

[273] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38 (2):6:1–6:56, 2006. ⟨12, 25, 57⟩

[274] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. on Database Systems*, 23(4):453–490, 1998. ⟨29, 61⟩

[275] G. Zuccon, J. Palotti, and A. Hanbury. Query variations and their effect on comparing information retrieval systems. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 691–700, 2016. ⟨96⟩

[276] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 59–71, 2006. ⟨63⟩