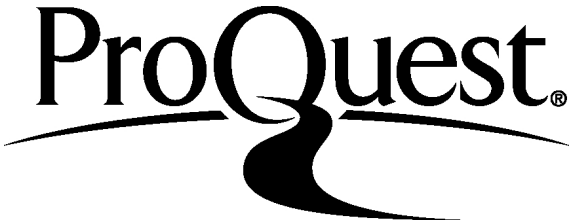
IMPROVING EFFICIENCY AND FLEXIBILITY OF INFORMATION RETRIEVAL SYSTEMS

by

Hao Wu

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Winter 2016



IMPROVING EFFICIENCY AND FLEXIBILITY OF INFORMATION RETRIEVAL SYSTEMS

by

Hao Wu

Approved: _____

Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____

Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____

Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Hui Fang, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Chengmo Yang, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Stéphane Zuckerman, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Ben Carterette, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my advisor, Hui Fang. who gave me the opportunity for graduation study and led me into the field of information retrieval. I had no prior experience in this area and she taught me the knowledge and skills with great patience. It is a great pleasure to work with her and I learnt pretty much through the process. Besides, she greatly encouraged and supported my interests in IR efficiency and IR toolkit development. Without her help I cannot finish this thesis.

I also thank other members in my thesis committee, Chengmo Yang, Ben Carterette, Guang R. Gao and Stphane Zuckerman. They gave me a lot of useful suggestions and helped me to make the thesis more solid and completed.

I would also thank my collaborators. Wei Zheng and Xitong Liu gave me a lot of help when I first came to the lab and I could not setup and learn so quickly without their help. I would like to thank Peilin Yang, Yue Wang, Kuang Lv and Wei Zhong as well, It is a great pleasure to discuss and work with them.

I would like to thank parents for their unconditional love and support throughout my whole life. Finally I would like to give special thanks to my wife Zhuo Liu, who encouraged me seek graduation education abroad. It is a great luck that we got admissions from the same university and we spent a lot of pleasure time in the small city of Newark. Pursuing Ph.D. degree is a long and lonely journey. Her understanding and supports is one of the energy which drives me finish this adventure.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The development of information retrieval (IR) (the search engine) is one of the revolutionary techniques of the past century. It changes the way people communicate and share knowledge, and it frees people up from the hassles of seeking and remembering information, in addition to saving time and energy that can be used more effectively elsewhere.

As the amount of information grows exponentially, so do the complexities and the costs. Search-engine efficiency, which is related to user experience and the provider's revenue, becomes more and more important. Existing techniques such as dynamic pruning can help improve the efficiency of IR systems. However they do not solve the whole puzzle and in many situations (e.g. long queries, large number of returned documents and etc.) their improvement of efficiency is far from enough.

To improve the efficiency of query processing, we create an analytical model to explain query processing time of IR systems. This model uses few features and it is more accurate than previous. Inspired by the model, we try to solve various IR efficiency problems. First, to improve the query processing time when k (e.g. the number returned documents) is large, we propose a document prioritizing methods which can better improve efficiency than state-or-art methods without hurting effectiveness. Second, we studied a special case of long query processing called pseudo-relevance feedback and we improve its efficiency by providing a new incremental approach. Third, we further explore the trade-off between efficiency and effectiveness and propose several methods which can improve efficiency at the cost of acceptable effectiveness loss in a time-constrained environment.

In additional to query processing efficiency, we also explore another kind of efficiency: how easily people can implement different search-engines. Current toolkits

can help implementing various retrieval functions with their API -based framework. However their APIs are usually complicated and it is still difficult for inexperienced users to implement retrieval functions.

To improve this situation, we introduced an information retrieval toolkit called Virtual IR Lab. When we compared it to existing IR toolkits, it applied a simpler but more efficient architecture. By applying automatic code-generating techniques, the toolkit can help users implement various retrieval functions conveniently. Its friendly and flexible design makes it a good fit for both education and research.

Chapter 1

INTRODUCTION

Information retrieval (IR) as performed by search engines is one of the revolutionary techniques of the past century. This wonderful achievement was founded on the improvement of effective communication and knowledge sharing. Across space and time, even from within a hazardous environment, information can be transmitted in seconds and connect human minds across the globe. New ideas erupted and disrupted our lives as never before.

Search engines changed the way people access information [44, 47, 51, 57, 60, 62, 104, 112, 118]. In the past, it was painful, sometimes expensive, and at times even impossible to search certain information. However, searching has now become so easy that anyone can get information in seconds by typing a few words in a search box. New information can be spread around the world in minutes, and people can access such knowledge at a very low cost. Search engines free people from the hassle of remembering information and save time and energy to explore new ideas.

However, as the amount of information grows exponentially, the complexity and cost of IR systems also increase exponentially. A commercial search engine may involve thousands of machines or more. As a result, query efficiency (i.e., how fast the IR system can process queries) becomes increasingly important [23, 32, 87].

First, efficiency is directly related to user experience. Search users usually expect search engines to display results in a short time in response to their search requests (i.e., within seconds). If the query delay is too long, the user may feel inconvenienced and switch to a different search engine. Fewer query delays also mean more search requests; users are likely to perform additional search if the IR system displays results more quickly. More search requests lead to more revenue for search engine providers [87].

Second, efficiency is highly related to costs. To handle large amounts of data and reduce query delay, modern commercial search engines usually employ a distributed architecture involving thousands of machines. If we can double search efficiency with other unchanged environmental factors (amount of data, query delay, etc.), a company can reduce the need for at least half of the hardware as well as energy costs [23]. This is surely of great importance to the IR industry.

Third, efficiency is also related to search effectiveness. Some techniques, such as pseudo relevance feedback , have been proven useful for improving search effectiveness (i.e., how search results meet users' information needs). However, due to their high computational costs, their application to industry is limited [25, 42, 63]. Improving efficiency may help break the limitation and lead to further improvement in search engine effectiveness. In summary, efficiency is becoming more and more important, and improving efficiency means cheaper, faster, better communication and knowledge sharing.

An IR system firstly gathers web pages and documents and builds an index from them. It then uses the index to serve users by outputting corresponding information to the queries. The first part can be done offline and is less time sensitive. A few studies [36] focus on improving efficiency on this part, e.g. reducing indexing delay. The second part in which IR systems grade and rank candidate documents based users' input queries and represent the retrieval results to users is known as query processing. Since query processing must be performed online, its delay is directly present to users and it has more impacts on users' satisfaction. Therefore, in this dissertation, we focus on the efficiency of the online part, i.e. how to reduce the computational costs and delays of query processing.

The basic query processing is to grade and rank all candidate documents. However as the data collection becomes large, basic query processing may lead to low efficiency and long delays. Previous work [10, 18, 38, 40, 49, 50, 78, 90, 97] explored different methods to improve query processing efficiency. The basic idea of these methods is trying to avoid unnecessary candidate documents grading so that the query processing

time can be reduced. Although they are proved to be useful to improve the efficiency of IR systems query processing, they don't solve the whole puzzle. In some situations, query processing time can still be very long (e.g. long queries, large number of returned documents).

In addition to query processing efficiency, we also try to explore another kind of IR efficiency which has been rarely studied before: how fast an IR system may implement and switch different retrieval functions. People are not satisfied with seeing only the data and search results provided by several centralized general search engines; they are also seeking methods to manage and share their own data. For example, a company may need a customized search engine to search for data in its private database. An electronic merchant may want to provide a search engine that is customized for a website to help customers find products. As a result, a toolkit that provides a framework for basic IR functions is needed. Such a toolkit is also important for IR research and education. Researchers keep introducing new methods and they need implementations to test their design. Students want to know how retrieval results are generated and they need opportunities to control and study IR process. To meet these requirements, a complicated industrial system is usually impractical, whereas a flexible simple toolkit can work well. Fortunately, there are several IR toolkits, such as Indri [3], Lucene [7], and Ivory [4], and they have been proven useful in providing search related functionality. However, even with these toolkits, exploring new retrieval functions and implementing customized search engines is still complicated. They require users to understand their mechanisms and to modify or write a great deal of application programming interface codes to implement basic functions. This process is even more painful for inexperienced users. To solve this problem, we propose a new framework for creating IR toolkits that use a very flexible and easy-to-use process to implement search engines.

To solve these two kinds of efficiency problems, in this thesis we make following contributions:

(1) First, we show a web service called Virtual IR Lab which is designed to

improve IR system development efficiency. Virtual IR Lab is based on a simple framework and uses automatic code generation techniques to help users implement their own search engines in a very convenient and efficient way. We also provide a platform for them to test and compare search engines so that they can easily improve the methods. In additional to these, the architecture of Virtual IR Lab is simple but efficiency. Compared with existed IR toolkits such as Indri [3], it is much faster in the term of query processing. We describe the architecture of Virtual IR Lab as a good example of how we can improve query processing efficiency by simplifying and optimizing the design. It should also help readers to better understand basic concepts and mechanism of IR systems so that they can understand the following chapters easily. (See Chapter 3)

(2) Second, in order to improve query processing efficiency of IR systems, we study features affecting efficiency. By figuring out the key features and how they influence query processing time, we build an analytical model which can more accurately predict processing time for each query. This model and observations in the chapter are used to develop methods to improve query processing efficiency in following studies. (See Chapter 4)

(3) Third, we study how to improve efficiency for IR systems which return a large number of documents per query. As we know, current dynamic pruning techniques [49, 50, 90, 97] can help to greatly reduce query processing time when the number of returned documents is small. As the number of returned documents becomes larger, the efficiency gain through current dynamic pruning techniques becomes smaller. To solve this problem, we propose a novel document prioritization strategy which can out-perform a very strong baseline. (See Chapter 5)

(4) Fourth, we seek methods to reduce processing time of long queries. Specifically, we focus on a well-used technique called pseudo relevance feedback which is suffering from low efficiency of long query processing. By introducing new incremental approach, we greatly improve speed of pseudo relevance feedback with almost no effectiveness loss. (See Chapter 6)

(5) Finally, we explore trading off effectiveness for efficiency in time-constrained

environment. Commercial IR systems may set up time-out boundaries for query processing. It is for the purpose of improving system stability as well as user experience. In order to meet such an efficiency requirement, we may need to further sacrifice effectiveness. To reduce effectiveness loss, we provide some laws to select good trade-off strategies. We then propose and compare several effectiveness and efficiency trade-off methods.(See Chapter 7)

Chapter 2

BACKGROUND

With the increasing amount of online information and the rapidly growing number of queries, commercial search engines have leveraged many performance optimization techniques, including parallel computing [23, 26, 28, 71, 72], caching [22, 56], index compression [15, 16, 116, 122], and dynamic pruning [18, 49, 50, 90, 97] to meet strict performance constraints: high throughput and low latency. We now provide some background about the topics most relevant to our study.

2.1 Basic Query Processing

The main approach to improving information-retrieval-system efficiency is to reduce data access and computing during query processing. To avoid accessing whole collections, IR systems perform query processing based on inverted indices [31, 66, 73, 96, 124]. An inverted index is a data structure that enables efficient access to documents that contain specific terms. In particular, an inverted index consists of an inverted list for each term in the collection, and each inverted list contains a list of postings with more detailed information about the occurrences of the corresponding term in the collection. Typically, a posting of a term provides the information about the terms occurrence in a document, such as the document ID and the frequency of the term in the document. Postings of an inverted list can be sorted based on either the document IDs [124] or their impact, i.e., the terms contribution to the relevance scores of the corresponding documents [13, 14, 17, 18, 90].

With the help of inverted indices, IR systems can access and evaluate a subset of documents related to the query terms. When processing a query, IR systems must traverse the inverted lists of all query terms and compute the relevance score of each

document with respect to the query based on a retrieval function. The query-processing strategies can be classified into two classes:

- **Term-At-A-Time (TAAT):** Sequentially processes the inverted lists of all query terms and accumulates the partial document scores contributed by each term [35, 76, 124]. Each document requires an accumulator to record the accumulated relevance score.
- **Document-At-A-Time (DAAT):** Processes the inverted lists of all query terms in parallel. The IR system must fully evaluate a document based on the contribution of all query terms that occur in it before moving to the next document. [29, 49, 50, 70, 91, 97].

DAAT and **TAAT** have their own advantages. Because **TAAT** processes inverted lists sequentially, it results less cache miss and can process single inverted lists very quickly. However, to summarize the contributions of all the query terms for the documents, **TAAT** must maintain a large hash table that stores all candidate documents and their accumulated scores . As a result, TAAT may consume a large amount of additional memory and computational power in maintaining and operating the hash table.

DAAT on the other hand, requires less memory and only maintains a heap that stores just the top documents list. However, to localize all the information for the same documents, **DAAT** requires synchronization operations over all inverted lists at significant computational cost. Comparing the two query processing techniques, **DAAT** is more widely used simply due to its ability to handle more complex queries [29, 91]. As a result, in the thesis, we will mainly focus on DAAT query processing.

2.2 Static Document Pruning

One thinking to reduce computational cost is not to evaluate postings which are less likely to affect retrieval results. We can estimate the impacts of postings off-line and prune low impact ones. Various static pruning methods [10, 38, 40, 78] have been proposed and their difference are at how they prune postings. Some [40] use term-based features such as impact score. Some methods [10, 38] concentrate on documents

and try to keep most representative terms for each documents. While the others [78] use both term and document features. Static pruning methods can greatly reduce index size and they can more or less reduce query processing time as well. However due to the facts that static pruning methods cut postings independently from queries, they are not always satisfying: On one hand static pruning methods are unsafe, they might hurt effectiveness and in some situation may be significant due to over-aggressive pruning. On the other hand, the efficiency improvement of static pruning might not be significant enough.

2.3 Top-K Query Processing and Dynamic Pruning

Even with the help of inverted indices, the computational cost may still be too large for the IR system to handle. In a commercial search engine, a common terms inverted list could contain billions of postings; traversing all the postings and accurately computing the relevant score for every candidate document in the collection is very costly and likely impossible. Hence, instead of outputting all candidate documents, a more practical approach is to find the top-k (e.g., top 1000)-ranked results for a given query based on a retrieval function [24,59]. Improving top-k query processing efficiency is one of the most important problems for an IR system.

Various dynamic pruning (i.e., early termination) techniques have been proposed for both DAAT and TAAT, with the aim of reducing the number of documents that need to be fully evaluated by avoiding scoring postings that cannot make the top-K results [18, 49, 50, 90, 97].

2.3.1 Dynamic pruning of DAAT query processing

In recent years, a few pruning methods have been proposed for DAAT-based processing. The main approach these methods take is to skip unpromising documents in candidate-document lookup steps. To implement the idea, the pruning methods must maintain two types of information: (1) a pruning threshold, which records the minimum score required for documents to make the top-k results; and (2) the maximum

expectation score of each candidate document. The pruning threshold is usually easily accessible by reading the score of the top k-th document, which is the very bottom document of the top documents heap. The maximum expectation score of a document is usually estimated as the sum of the maxscores of the query term it contains, where a term maxscore is the highest impact score of all the postings in the terms inverted list. It is worth mentioning that maxscores are bound to a special retrieval function and require pre-processing to get their values.

The Maxscore method [97] first introduces this approach. It dynamically identifies essential terms based on their maxscores and the threshold and then skips the scoring of documents that do not contain any essential terms. The WAND method [29] takes a different approach based on pivoted terms. Specifically, terms are first sorted based on the current document IDs in their inverted lists, and a pivot term is then selected so that the sum of this terms maxscores as well as that for each of the terms ranked in front of it is greater or equal to the pruning threshold. After this, the document ID of the pivot term will be used to skip documents with smaller IDs. To seek for more efficiency improvement, the BMW method [50] stores additional highest impact scores for each blocks. It can estimate the maximum score more accurately for candidate documents than WAND, and consequently more documents can be pruned and higher efficiency can be achieved.

It is worth pointing out that the previous pruning methods we have discussed are rank-safe for top-k results, which means that the results they generate are identical to those generated by exhaustively evaluating every document. In fact, to achieve higher efficiency, we can more aggressively prune more documents at the risk of effectiveness loss. One method for doing so is to artificially change the pruning threshold to a value higher than that of the minimum score required for a document to make the top-k results. The higher the artificial threshold, the higher the efficiency, but the more effectiveness we may lose.

Another way of generating top-k results is to use the *conjunctive* mode, i.e., only evaluating documents that contain all query terms. Previous work has focused

on fast-posting list intersection [20, 95]. The conjunctive mode can be considered a special pruning method that is often more efficient than the disjunctive mode with regard to pruning, but it would hurt the query processs effectiveness as it may miss many relevant documents and generate results with lower recall.

2.3.2 Dynamic pruning of TAAT(SAAT) query processing

In order to skip more posting accesses in TAAT, Score-At-A-Time (SAAT) [18], a variation of TAAT, makes a series of modifications, beginning with Index Organizations. Instead of sorting postings based on their document IDs, indices for SAAT sort postings based on their impact, i.e., their contribution to a documents relevance score. Additionally, indices for SAAT store the impact score, rather than the term frequency, for each posting. These approaches have two advantages: first, a documents relevance score D for query Q can be calculated simply by summing up the impact scores of D in the inverted lists of all the terms in Q. It moves a lot of complex computing (e.g., impact score calculating) offline and reduces delay for query processing systems. Second, it allows the query processing to access most impacted postings at the beginning and lets it skip less impacted postings later. One disadvantage of such index architecture is that indices are bound to specific retrieval functions. If we use a different retrieval function, the whole index must be rebuilt. To further improve efficiency, the impact scores are binned into a smaller number of distinct values and only the binned integer values are stored in the index. This strategy provides two benefits. First, the impact score can be stored using a very small number of bits (e.g., 6 bits for 64 distinct binned integers) instead of using 32 or 64 bits for a floating point variable. Second, with the binned scores, the number of documents with the same score is sufficiently large for us to separate an inverted list into segments, one for each distinct value of the impact scores. Within each segment, documents are then ranked based on their document IDs, and the compression techniques used for document-sorted indexing can be applied here to achieve a high level of compression. Formally, the retrieval score of document D for

query Q is computed as follows,

$$S_{Q,D} = \sum_{w \in Q} B_{w,D}, \tag{2.1}$$

where $B_{w,D}$ is the binned integer score representing the impact of query term w in D. Previous studies have shown that such a binning strategy would not significantly affect the retrieval accuracy [17, 90]. Another benefit of binned-score-index organization is to enable skipping within the postings with the same binned-impact scores. Skips are forward pointers within an inverted list which make it possible to pass over non-promising information on the postings with minimal effort. They are often inserted into the indices and stored as additional information. With the help of the skips, the system can jump to required records without going through the posting list one record by one record.

The second modification is at query processing. Instead of processing the inverted posting lists one after another for each query term, SAAT processes the postings in decreasing order of impact value. It fetches all inverted lists first (as in DAAT), but it determines the most impacted postings and processes them. It does not require synchronization operations in document lookup; however, the hash-based document accumulators are still needed to accumulate partial score contributions from the different inverted lists. To skip unpromising postings and reduce accumulator list size, SAAT query processing usually goes through a four-stage pruning. The four stages are OR, AND, REFINE and IGNORE. The processing begins with the OR stage by processing the postings in the inverted lists based on the decreasing order of the impact values. Document accumulators are created whenever necessary, i.e., when a new document is seen in one of the inverted lists. The processing switches to AND stage when we can prove that we have created accumulators for all the documents that could possibly enter the top n. In the AND stage, we ignore all the documents without accumulators. As the processing continues, and we update the scores for the existing accumulators with newly-processed information. The processing switches to the REFINE stage as soon as we know exactly what the top n documents are without

knowing their exact ranking. The REFINE stage works with the accumulators of the top n documents. Finally, once we can determine the rank order of the top n documents, the process enters the IGNORE stage, which means that we can ignore all the remaining information from the inverted list. The object of the four-stage design is to reduce the number of active accumulators gradually when we know more confidently that other accumulators would not change the final top-k search results. In particular, the OR stage is the only one that can add accumulators; the AND stage only updates existing accumulators; the REFINE stage only processes accumulators that can make the top-k results. Moreover, we further speed up the process by using skipping and accumulator trimming [90]. Skipping enables long postings to be processed quickly, while accumulator trimming can reduce the computational cost in the AND stage by dynamically reducing the number of enabled accumulators.

2.4 Information Retrieval Toolkit

Information retrieval is one of the most important techniques we use in our everyday life. To help people implement such techniques into various applications, several open-source toolkits/libraries have been developed. The Lemur project was started in 2000 by the Center for Intelligent Information Retrieval (CIIR) at the University of Massachusetts at Amherst and the Language Technologies Institute (LTI) at Carnegie Mellon University. Its products, the Lemur toolkit and, later, the Indri [3] toolkit, are widely used in academia. They provide implementations of various language-model approaches and allow people to implement new retrieval methods by using APIs or modifying the core codes of these methods.

Lucene [7] is more popular than either Lemur or Indri. Lucene (or Apache Lucene) is a free open-source information-retrieval-software library supported by Apache Software Foundation. Originally written in Java by Doug Cutting in 1999, Lucene has been well developed and has achieved great success in both academia and industry. It is widely recognized for its utility in implementing Internet search engines and local,

single-site searching. The IT industry has made wide use of the enterprise search server applications Apache Solr and Elasticsearch.

Other information retrieval toolkits, for example, include Terrier (University of Glasgow) [8], Ivory (University of Maryland) [4], etc. However, none of the toolkits or platforms, including Indri and Lucene, provide user-friendly interfaces that enable one to implement and try different retrieval methods. In particular, to implement new retrieval functions in addition to those provided by the toolkits, a user must either modify their core code or write API programs. Modifying core code usually requires a user to have a deep understanding of the toolkits architecture. Re-compiling and debugging is also required, complicating the process. While API programming is preferred to core-code modification, before being able to write usable API code a user is required to read a lot of API documents to use the toolkit-provided API functions. Obviously, neither solution is convenient, especially for beginners who have little knowledge and experience in using such toolkits.

Chapter 3

VIRTUAL IR LAB: A NOVEL PLATFORM FOR IR TEACHING AND RESEARCH

3.1 Introduction

A modern information retrieval (IR) system not only returns to users the documents containing their query terms; most importantly, it ranks the candidate documents and presents users with the most relevant results. Consequently, the scoring mechanism used to determine how the documents are ranked, called the retrieval function (or IR model) [11,53,58,61,82,88,102,108,109,121,125] becomes the soul of search accuracy. Many retrieval functions have emerged in the past several years; however, no single winner exists. In most situations, people have to implement and process experiments to see which one is best for which kind of data collection. Throughout the development of customized search, the importance of different retrieval methods has become evident in the effort to cover different situations and meet various data and goals. On the other hand, teaching IR requires students to implement and try different retrieval functions to enable them to understand IR techniques well. Both situations require a tool that can facilitate the development and testing of different IR models.

IR toolkits such as Indri [1], Terrier [2], and Lucene [3] have achieved significant success by providing ways to transfer IR models to industry and various applications . People can implement their IR models by either modifying the core code of the toolkits or using the API function the toolkit provides . Although this represents

[1] http://www.lemurproject.org/indri/

[2] http://terrier.org

[3] http://lucene.apache.org

major progress from letting users implement IR systems from scratch, either solution has its own weakness: to modify the original code, users need a basic understanding of the toolkits to figure out where and how to insert their codes; to use the API, users require knowledge not only about how to extract term and document statistics from the index but also about how to deal with trivial issues (e.g. importing queries, ranking documents, and presenting results). Consequently, such API programming is a little complicated and requires the user to have certain knowledge of information retrieval and skills in programming. Figure 3.1 shows a small part of a typical Indri query processing API program. This code fragment looks for the next candidate document by going through the posting list of each query term. However, it is just a small part of the whole query processing program. Therefore it is obvious that implementing retrieval functions through API is not easy either.

Both modifying the core code or using API to implement retrieval functions are complicated and time-consuming, especially for beginning users. The complexity could discourage people from using the programs to test more functions over more collections.

This study introduces a new IR framework. It is different from previous IR toolkits in that people can implement different retrieval functions within minutes. However, its implementation of efficient top-k query processing functionality is even faster than that of mainstream IR toolkits. More important, by applying a novel architecture and dynamic code generation techniques, it allows users to implement and test customized retrieval functions with minimal effort (e.g. a few lines of C/C++ coding). We will introduce the design in the rest of the chapter.

3.2 Methodology

3.2.1 Architecture

The basic functionality of typical IR systems is usually divided into two parts: index building and query processing. The purpose of index building is to organize the documents so that the information can be accessed efficiently. The Virtual IR Lab follows the paradigm, but it introduces several novel designs to fit its purpose. Figure

```
int findNextDoc()
{
  int minDoc=MaxNum;
  int i,l;

  for(i=0;i<num;i++)
  {
    if(termOccur[i]>0)
    {
      if(diList[i]->hasMore())
      {
         diEntry[i]=diList[i]->nextEntry();
         curDocID[i]=diEntry[i]->docID();
      }
      else curDocID[i]=MaxNum;
    }
  }
  for(i=0;i<num;i++)
  {
    if(minDoc>curDocID[i])
    {
      for(l=0;l<i;l++) termOccur[l]=0;
      minDoc=curDocID[i];
      termOccur[i]=1;
    }
    else if(minDoc==curDocID[i]) termOccur[i]=1;
    else termOccur[i]=0;
  }
  return minDoc;
}
```

Figure 3.1: Code fragment of API based query processing implementation

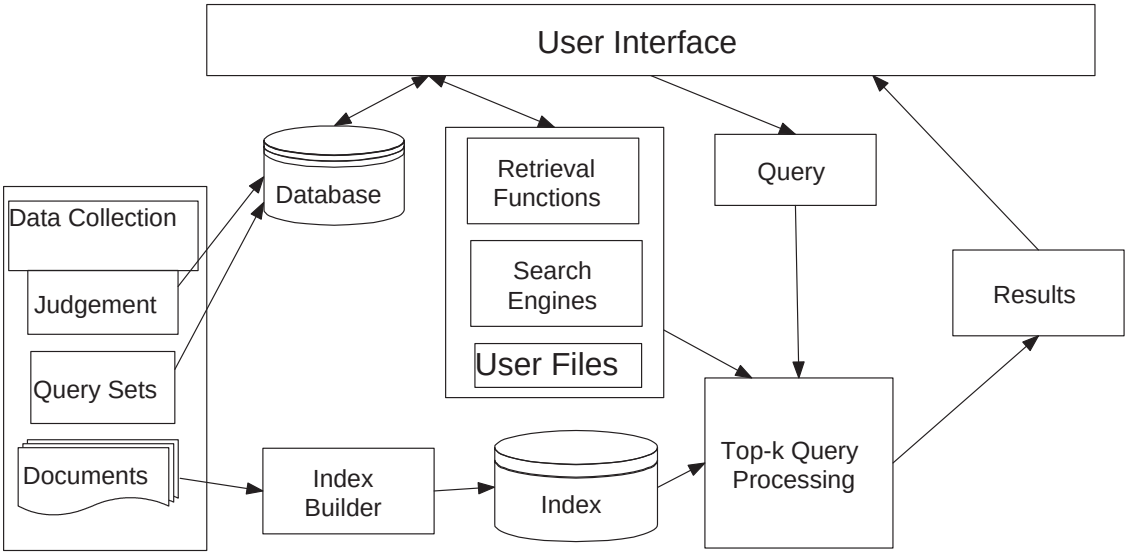


Figure 3.2: The architecture of Virtual IR Lab

3.2 shows the architecture of the Virtual IR Lab system. Virtual IR Lab has a user interface that connects both database and file system. It allows users to log in and customize retrieval functions and search engines. When a query comes into the system, Virtual IR Lab automatically generates code based on both retrieval function definition and query term statistics. The automatically generated code is used to perform top-k query processing through the index. The results are then returned to the user interface for further operations (e.g. display and evaluation). To better explain the design, we will show more details in the following subsections.

3.2.2 Life of an index

Virtual IR Lab is based on a simple but efficient index design. The index in Virtual IR lab has the following features: first, the index architecture is simple and easy to understand and maintain, making it ideal for both education and research. Second, the index is especially efficient for query processing. Experimental results show that top-k query processing can be even faster than other toolkits query processes (e.g. Indri

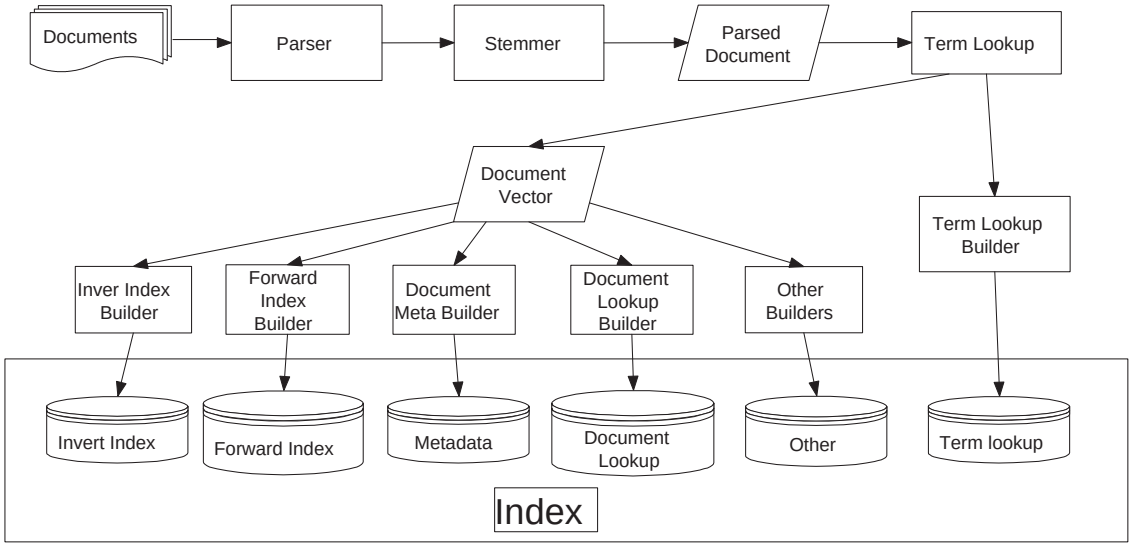


Figure 3.3: The process of index building

and Lemur). Finally, the index is extendable, which enables more data and statistics to be added. The idea of dynamic pruning methods is to using pre-computing information such as maximum impact scores of each posting lists. Virtual IR Lab's index can be easily extended to compute and store such extra data. As a result, it can also support dynamic pruning such as WAND and maximum score as well. In this subsection, we explain how documents are pre-processed and used to build an index in Virtual IR Lab.

We summarize the process in figure 3.3. At the beginning, documents go through a parser, where the punctuation is removed and documents are tokenized into terms. The terms are then further stemmed [80] and transferred into their original forms. For example, the term "does" will be stemmed as "do". The stemmed terms are then passed to the next stage, where they are translated into interval term IDs.

The idea of interval terms is also used in mainstream IR toolkit such as Indri and Lucene. Using interval term IDs instead of text strings gives us several advantages in efficiency. First, the interval term IDs are integers, which use less space than text

strings. Using interval IDs greatly reduces the cost of data storage and transfer. Second, the interval IDs are allocated continually and can naturally help the system to locate the data associated with the terms. Therefore, the interval term ID translation process, called "term lookup", is one of the most important steps in index building. Multiple methods are available for such translation, such as hash tables [45,74]. A hash table is a frequently-used data structure that maps text strings to integer numbers and can be used to build such a dictionary. However, as the vocabulary size becomes large (e.g. millions of terms), a hash table might not be able to store the entire dictionary in memory, making maintaining and accessing the hash table less efficient. Hence, in Virtual IR Lab, we use prefix trees for term lookup. Compared with hash tables, prefix trees [27, 103] can be more easily stored on disk and efficiently accessed. Most important, using prefix trees can help the system locate terms similar to searched terms and is useful for query suggestion and query correction.

After the term lookup stage, a document is translated into a vector of integers. It is then passed to various builders to build indices that store different data and serve different functionalities in query processing. One important index is the inverted index, which stores the posting list and helps the system quickly locate documents containing given query terms. Figure 3.4 shows how the inverted index is built through documents. Based on their term IDs, the terms are passed to corresponding posting list builders. One posting list builder handles one single term and consists of two parts: a temporary posting list and metadata. A temporary posting list is an array that contains the most recent postings of the term. In the current design, the temporary posting list capacity is 64. When the array is full, the temporary posting list will be cleared and the 64 postings will be compressed and stored on disk as data blocks. Because all the posting list builders share the same compressed data block space, we need to store the positions of the blocks that correspond to each term. Additionally, we must also store other metadata, such as number of postings and sum of term frequency. After processing all documents, we further sort the blocks based on termID, then document ID, in order from lowest to highest. The advantage of sorting is obvious: because sorting merges

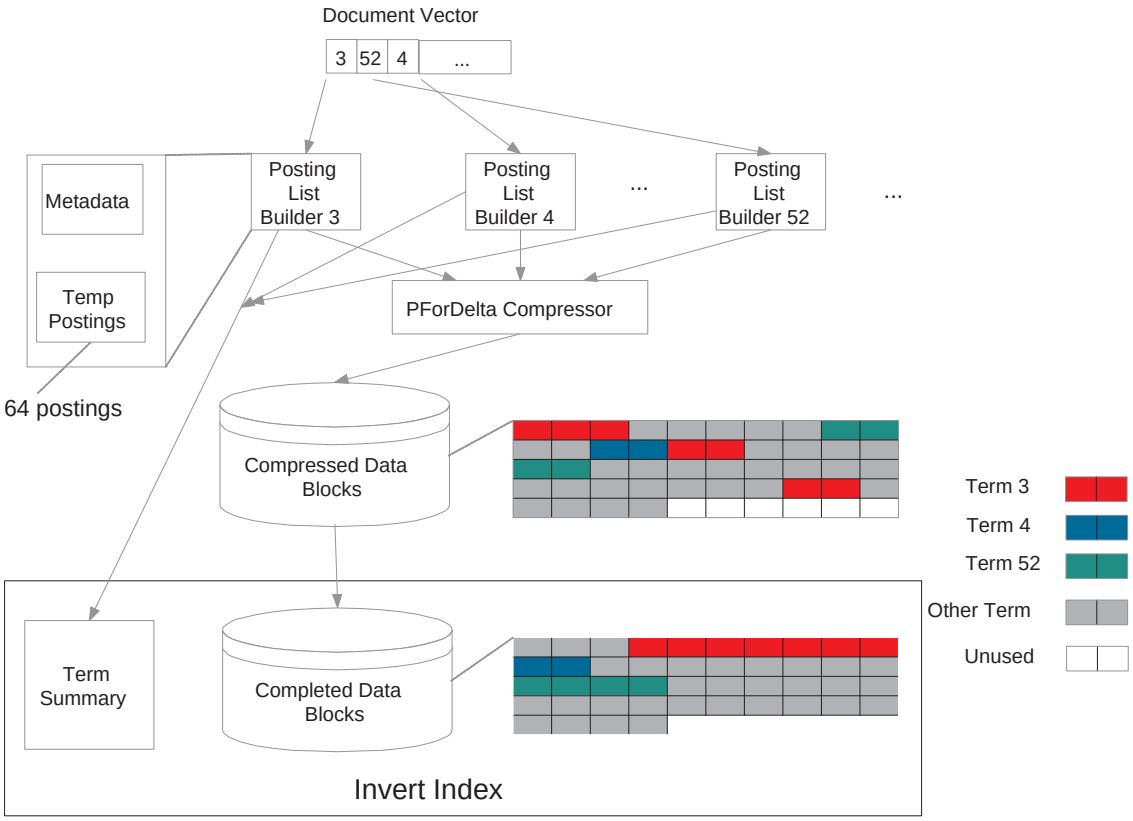


Figure 3.4: The process of invert index building

postings corresponding to the same term, the system can read the posting lists easily, thus greatly reducing the cost of index access during query processing. After all the blocks are sorted, the metadata of each posting list build are also optimized into a new data structure, called a term summary, which contains only the position information to help the system quickly locate the data for each term during query processing. The term summary and completed compressed data block form an inverted index of Virtual IR Lab, and we will show how to read posting lists from the inverted index in the next subsection.

Compared with inverted indexing, the process of forward index building is simpler. In Virtual IR Lab, the forward index stores the term order of each document

and proves useful for recovering document content. Forward indexing can also be used for advanced searches such as those using term proximity. Figure 3.5 shows how a forward index is built in Virtual IR Lab. A forward index builder handles the document vector one after another. Term ID fills the temporary lists in order. When the temporary lists are full (i.e. 64), the term IDs are compressed using the PFD compression technique [116], and the compressed block is then stored on disk. At the same time, we store the position of the start block of each document in an array, which is finally converted into a document summary that helps us locate the forward lists of each document. A forward index is easy to build and maintain because information is stored based on document order and the compressed blocks need not be reorganized.

Both the document metadata index, which stores document-based features such as document length, and document lookup, which recovers internal document IDs into the original documents, are of a design similar to the forward index. Because all information is ordered by document, these indices are similarly easy to build and maintain. The term lookup index differs from other indices in that it optimizes the prefix tree used for term lookup and stores it on disk in depth-first traversal order. Unlike width-first traversal, depth-first traversal can help locate the search path in the same disk segment or one that is adjacent . As a result, it may help reduce the disk seek time for term lookup.

3.2.3 Life of a query

In the previous subsection, we discussed how we can build indices in Virtual IR Lab. In this subsection, we will describe how a query is processed. Figure 3.6 summarizes the process. A query can come from either user direct input or the database; the former is for user-customized search engines and the latter for retrieval function evaluation, which uses official queries and judgment. At the beginning, a query is tokenized into terms, which are then stemmed, passed to the prefix tree-based dictionary, and converted into internal term IDs. At this point, we get a query vector which is consistent with a serial of integer term IDs. Based on the query vector, Virtual IR

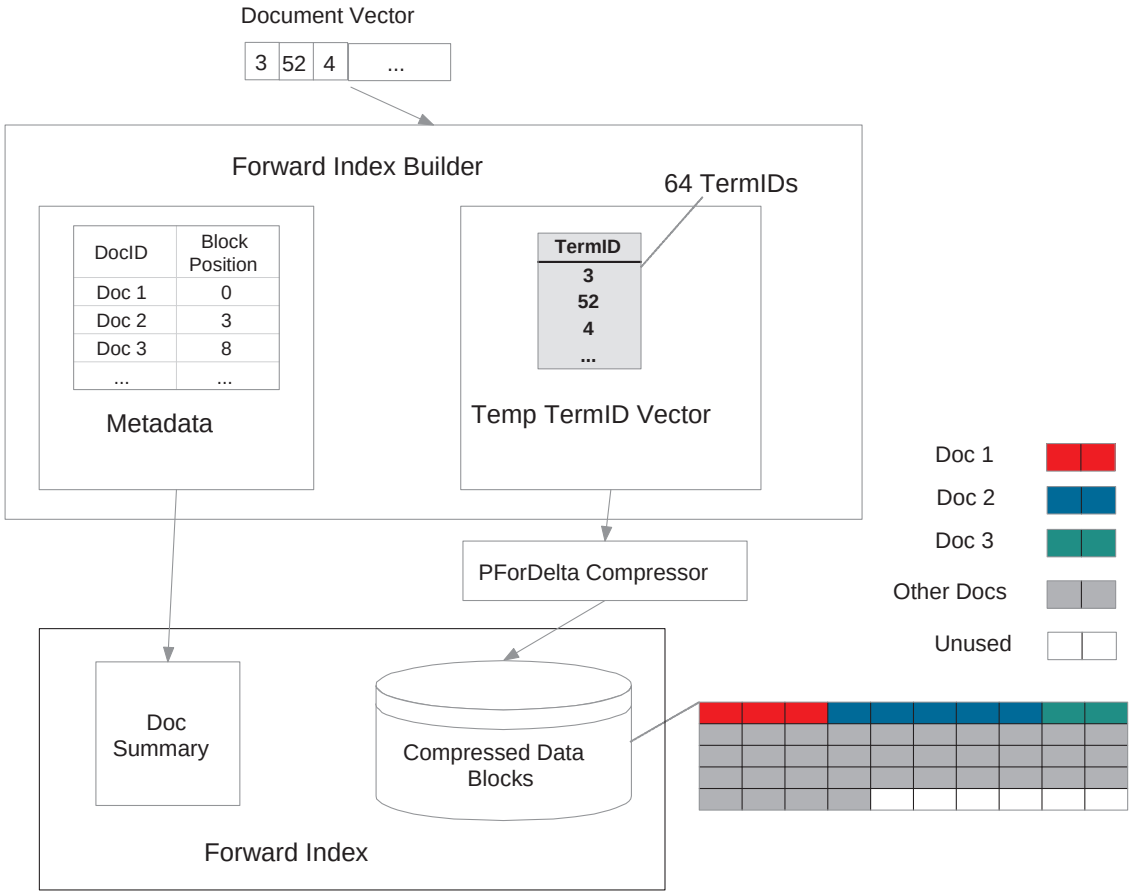


Figure 3.5: The process of forward index building

Lab collects statistics from both the statistics index and the inverted index and uses the statistics index and a user-defined retrieval function to automatically generate a code segment for use in document grading. At the same time, the internal term IDs are used to locate the posting lists in the inverted index. Because the posting lists are sorted by term ID, the process is easy and efficient.

Figure 3.7 demonstrates how to locate the posting list of term 103 which is stored in the position of 5080. First, we go through the term summary, which is essentially a sorted table storing the position of each terms posting list block. In this example, Virtual IR Lab reads row 103 of the term summary and gets the achieved position of 5080. It further directs into position 5080 and fetches metadata specifically for the posting list iteration. This metadata includes the number of postings, which is essentially the number of documents containing the term; the terms total number of occurrences; and, most important, the information needed to decompress the data blocks. As previously mentioned, Virtual IR lab uses the PFD compression technique. To decompress such a block requires the following information: data block position, data block size, parameter A, and parameter B, which are set specifically for the data block. In query processing, Virtual IR Lab first reads block decompression information and stores it in memory; it then uses the information to fetch the compressed blocks from disk and decompress them into posting lists if they are not skipped.

Another optional step is to prepare document skip information, which is only for dynamical pruning methods, such as maximum score, WAND, and BMW, which decide whether to prune documents by comparing documents' maximum score estimation with the cut-off threshold. In order to estimate a document's maximum score, we need to know the maximum impact of each terms postings. To skip more documents, some dynamic pruning methods such as BMW provide even more accurate maximum score estimation by using the maximum score of each compressed block. Obviously, such maximum impacts information needs to be pre-computed and indexed and is binding to retrieval functions. In other words, the maximum impacts information pre-computed from a retrieval function cannot be used to direct the dynamic pruning

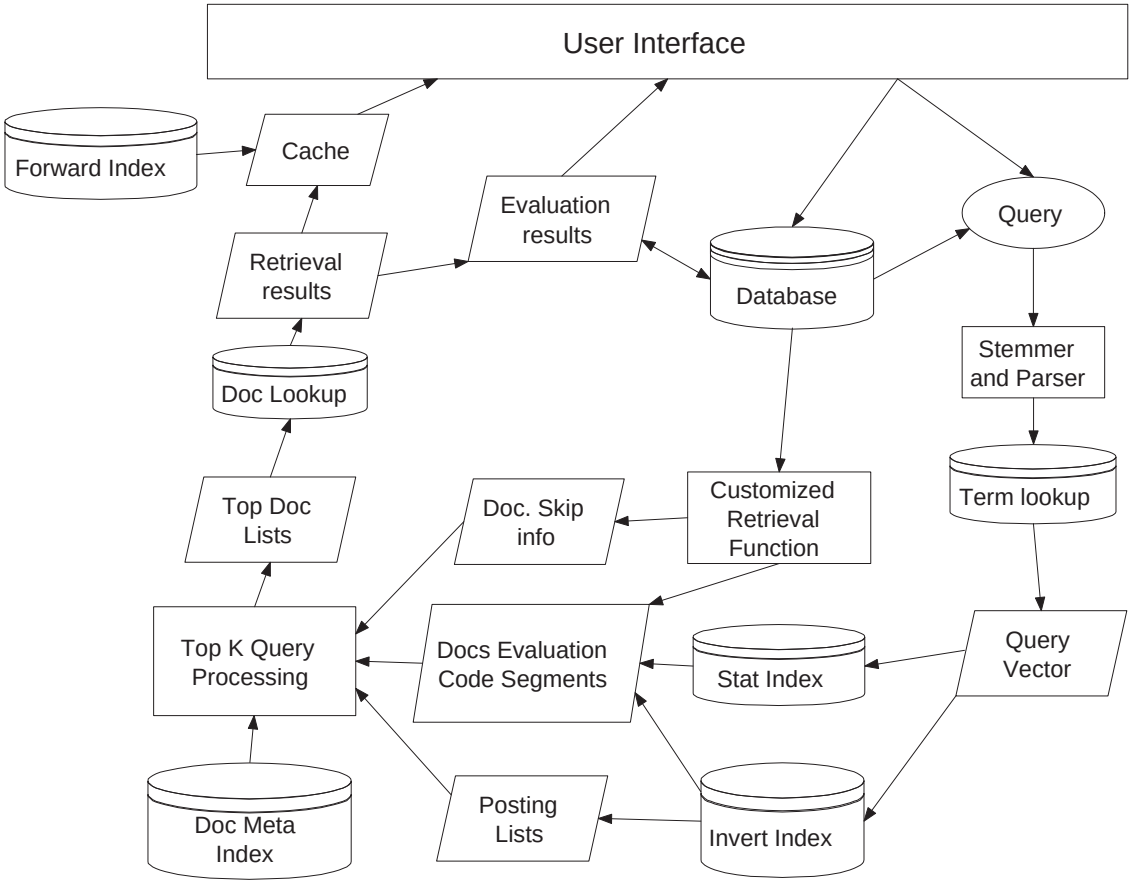


Figure 3.6: Life of a query

process of other retrieval functions; therefore, Virtual IR Lab will try to look for the pre-computed information for the retrieval function. If no pre-computed information is found, the query will be processed using exhaustive methods that grade every document containing at least one query term.

When grading function, posting lists, and document skip information are ready, the top-k query processing begins. Virtual IR Lab applies a typical document-at-a-time (DAAT) query processing mode, which consists of multiple iterations. Each iteration identifies a candidate document through the posting lists and collects information related to that document, such as query term frequency, document length, etc. The

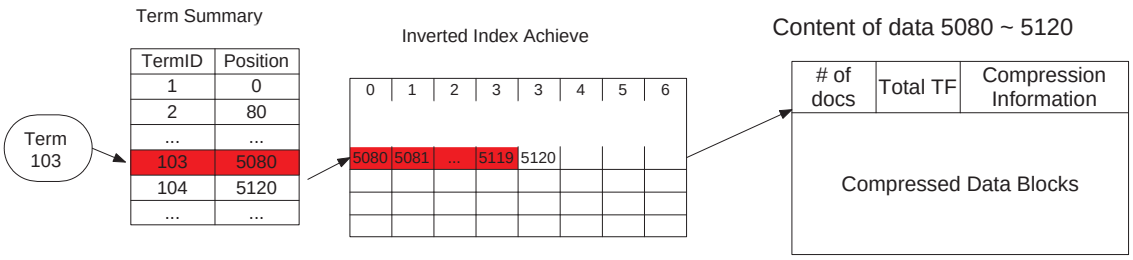


Figure 3.7: The process of posting lists locating

document-related information is then input into the document grading function. If the resulting grade is high enough (i.e. higher than the cut-off threshold), both the document and its grade will be stored in a heap that stores the top-k documents, and the cut-off threshold is modified. The previous chapter, which describes the analytical model of top-k query processing, provides more details about the process. Information such as document length is not stored in the inverted index but instead in the document metadata index. Therefore, in top-k query processing, Virtual IR Lab needs to fetch data from the document metadata index as well.

The output of top-k query processing is a list containing the internal IDs of top-k documents. To present the retrieval results to users, we must convert the internal document IDs into their original titles. The document lookup index handles the task and applies a two-layer architecture. The first layer is Document ID lookup, essentially a table that stores the record position and the length. Based on position and length, we can easily locate and fetch the document name string. Figure 3.8 demonstrates how Virtual IR Lab converts DocID 123 to its original form, first reading row 123 of the summary table and getting the position of 1988 and length of 21, then locating and fetching the record titled "An Introduction of IR". It is also worth mentioning that other indices such as term ID lookup apply a similar design.

After the document ID lookup stage, we finally get the retrieval results. Retrieval results are going to two directions: on one branch, retrieval results are evaluated

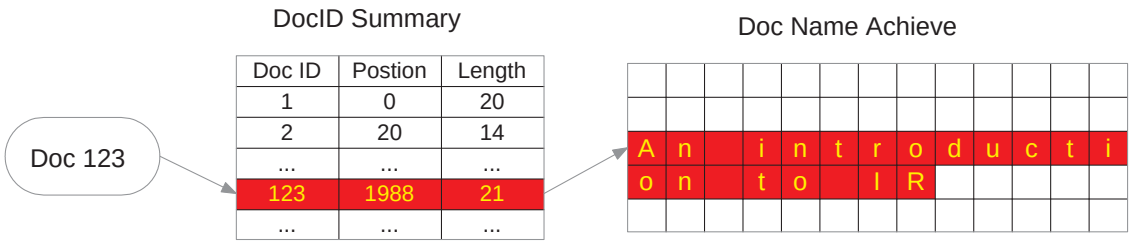


Figure 3.8: The process of document name lookup

by human judgment, and the evaluation score (e.g. mean average precise (MAP) and precision@30) can be used to compare different retrieval functions on the leading board. On the other branch, results are displayed on customized search engines if the query is from the search box. To accomplish this, the retrieval results are first stored in the cache while the retrieval document IDs are sent to the forward index to get snippets. The process of fetching document content from the forward index is similar to that of document ID lookup, and we can easily get the document content vector from the archive. By organizing the terms around the query terms, Virtual IR Lab creates a snippet for each document, and the information is also stored in cache. Finally, the UI fetches the information in cache and displays it to the user, thus completing the query process.

3.3 Discussions

3.3.1 Query processing with dynamical code generation

One special feature of Virtual IR Lab is that it can apply dynamical code generation technique to implement different retrieval function definitions. Mainstream IR toolkits usually implement customized retrieval functions through their APIs. Therefore, it is interesting to compare the two approaches, and we show the difference in their architectures in figure 3.9.

Unlike traditional API-based implementations that use the same program to process all queries, Virtual IR Lab generates embedded programs to deal with different

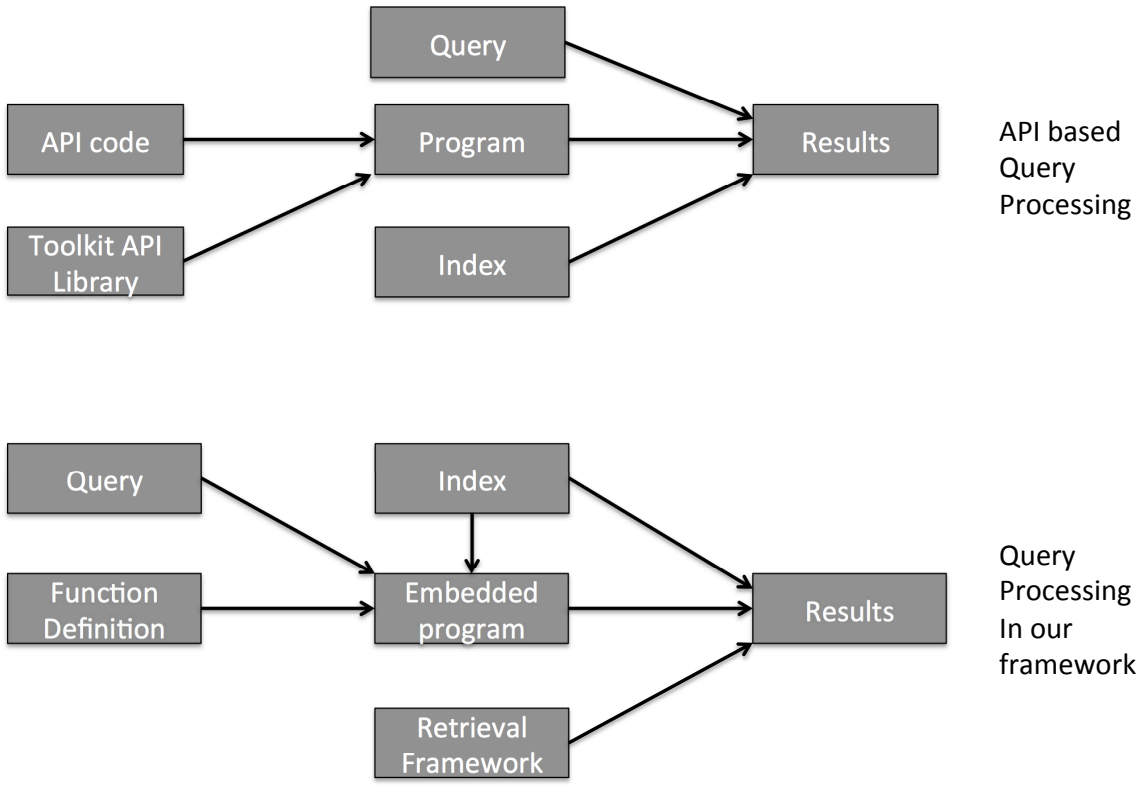


Figure 3.9: Difference between API based query processing and our new approach

```
double BM25()
{
  int docID=curDoc;
  double docLength = theIndex->docLength(docID);

  int i;
  double score=0;
  for(i=0;i<num;i++)
  {
    if(termOccur[i]!=0)
    {
      int tf = diEntry[i]->termCount();
      double idf = log(((docN-docCount[i])+0.5)/((double)docCount[i]+0.5));
      double weight = ((okapiK1+1.0)*tf)/
                  (okapiK1*(1.0-okapiB+okapiB*docLength/docLengthAvg)+tf);
      double tWeight = (okapiK3+1.0)*qf[i]/(okapiK3+qf[i]);
      score+=idf*weight*tWeight*termOccur[i];
    }
  }
  return score;
}
```

Figure 3.10: API-based BM25 document evaluation function implementation

queries. Figure 3.10 displays sample code for implementing the evaluation function of BM25. Obviously, the code is not optimized for efficiency in that some parts are unnecessarily computed repeatedly. For example, in each document evaluation the program will look for the document frequency (DF) of each term and compute its inverted document frequency. Considering that there are usually millions of candidate documents to be evaluated, these IDF computations must be redone millions of times. However it is worth noticing that IDFs do not change during query processing and need be computed only once.

Our design hand can automatically optimize users' code. In its statistics-centered mechanism, user-defined code is translated into "embedded code" by replacing statistics names with real values. The compiler further optimizes the embedded code, thus naturally avoiding redundant computation. Figure 3.11 shows how BM25s retrieval function is implemented for document evaluation: it is similar to but more concise than the process shown in figure 3.10. When processing a query with two query terms whose respective DFs are 100 and 1000, given collection size $N = 5000$ and average document length $avgl = 500$, the corresponding embedded code will look

```
double k1 = 1.2;
double b = 0.2;
double k3 = 1000;
for(occur)
{
        double idf = log((docN-DF[i]+0.5)/(DF[i]+0.5));
        double weight = ((k1+1.0)*tf[i]) /
                (k1*(1.0-b+b*docLength/docLengthAvg)+tf[i]);
        double tWeight = ((k3+1)*qf[i])/(k3+qf[i]);
        score+=idf*weight*tWeight;
}
```

Figure 3.11: BM25 document evaluation function implementation in IR Virtual Lab

like figure 3.12. The compiler will optimize the code in the manner of figure 3.13, where 3.89 and 1.39 are the computational results of $log((5000-100+0.5)/(100+0.5))$ and $log((5000-1000+0.5)/(1000+0.5))$, accordingly. Thus we avoid redundant IDF computing without user effort.

This design is advantageous not only because it is efficient but also because it is more convenient and more flexible. On one hand, users can define retrieval functions in more convenient ways. They do not need to look for API functions and remember their usage; instead, they can direct use feature names. On the other hand, this design allows Virtual IR Lab to explain and implement feature accessing methods separately. Consequently, to deal with additional features Virtual IR Lab can easily introduce new indices or additional computations on existing indices to fetch the statistics.

3.4 Experiments

We have shown the design of Virtual IR Lab in the previous section. The framework is more concise and more convenient for users to implement retrieval functions. In the section, we'd like to compare the Virtual IR Lab with other toolkit (i.e. Indri) to check Virtual IR Lab can produce effective and efficient results. we conduct experiments on the TREC Gov2 collection, which consists of 25.2 million web pages from the .gov domain. We use the queries from the TREC official terabyte track 2004-2006 [37],

```
double grade() {
  double score = 0;
  if (tf[0] > 0) {
    double idf = log((5000-100+0.5)/(100+0.5));
    double weight = ((1.2+1.0)*(tf[0]) /
              (1.2*(1.0-0.2+0.2*docLength/500)+tf[0]);
    double tWeight = ((1000+1)*1)/(1000+1);
    score+=idf*weight*tWeight;
  }
  if (tf[1] > 0) {
    double idf = log((5000-1000+0.5)/(1000+0.5));
    double weight = ((1.2+1.0)*(tf[1]) /
              (1.2*(1.0-0.2+0.2*docLength/500)+tf[1]);
    double tWeight = ((1000+1)*1)/(1000+1);
    score+=idf*weight*tWeight;
  }
  return score;
}
```

Figure 3.12: The embedded code which is generated by replacing variables with actual values

```
double grade() {
  double score = 0;
  if (tf[0] > 0) {
    score += 3.89 *
             ((2.2*tf[0]/(1.2*(0.8+0.2*docLength/500)+tf[0]);
  }
  if (tf[1] > 0) {
    score += 1.39 *
             ((2.2*tf[0]/(1.2*(0.8+0.2*docLength/500)+tf[0]);
  }
  return score;
}
```

Figure 3.13: The code which has been optimized by compilers

Table 3.1: Performance comparison of the three system (MAP@1000)

	TB04	TB05	TB06
Indri	0.250	0.324	0.285
Virtual-d	0.250	0.324	0.285
Virtual-a	0.250	0.324	0.285

denoting them as *TB04*, *TB05*, and *TB06*. Each query set contains 50 queries, and for each query we return 1000 documents.

As in other chapters, all experiments were conducted on a single machine with dual AMD Lisbon Opteron 4122 2.2GHz processors and 32GB DDR3-1333 memory. And we used okapi BM25 [82] as the retrieval function to rank documents. Since Indri has no dynamical mechanism and to maintain the flexibility, we did not apply dynamical pruning techniques for Virtual IR Lab as well and we exhaustively evaluate all the documents which contain at least one query term.

We compare three systems: original Indri runquery application and we use version 5.1. which is denoted as *Indri* [3], Virtual IR Lab with dynamical code generation which is denoted as *Virtual-d* and query processing system implemented by Virtual IR Lab API without dynamical code generation which is denoted as *Virtual-a*. We compare the two Virtual IR Lab system in order to check the affects of dynamical code generation.

3.4.1 Experimental results

First we compare the results generated from the three systems, table 3.1 summaries the effectiveness comparison between the results generated from the three systems on different query sets. It shows that there are no effectiveness difference between the three systems. It indicates that Virtual IR Lab can provide comparable results with Indri.

We then compare the efficiency of the three systems. We compare their efficiency by using the average processing time of queries. Table 3.2 summaries the comparison on

Table 3.2: Efficiency comparison of the three system (ms)

	TB04	TB05	TB06
Indri	2,446	2,074	1,960
Virtual-d	534	460	448
Virtual-a	789	671	624

different query sets. It shows that on each query set, Virtual IR Lab is much faster than Indri [3] implementation. By using Virtual IR Lab API, the average query processing time is less than one third as those of Indri application. By applying dynamical code generation, the code is further optimized and the system is far more efficiency (i.e. more than four times faster than Indri). The large efficiency gain is due to our concise and optimized design which has been shown in the previous sections.

3.5 Conclusion and future work

Virtual IR Lab is a novel IR toolkit that has some advantages over traditional IR toolkits such as Indri: (1) Simplicity. The architecture of Virtual IR Lab is simple and easy to understand, making it ideal for education and research. Moreover, it is installation-free; users need not install the toolkit and can remotely access it anywhere at any time. (2) A friendly user interface. Instead of referring to complicated documentation and typing commands with all kinds of parameters, users can process queries and build search engines simply by clicking the mouse. (3) Efficiency and extendability. Automcatical code generation and an optimized I/O framework make Virtual IR Lab efficient, and it is easy to add new features. Virtual IR Lab has other useful features, such as allowing users to compare their retrieval functions on a leaderboard and find the best ones. Finally, we show the screenshot of the toolkit in figure 3.14 and hope you can try it and give us some feedback [54].

Although Virtual IR Lab has shown some good potential in usability and efficiency, it is still not perfect. One of the main problems of the current version is the index builder. Virtual IR Lab can ingest raw documents and build indices without

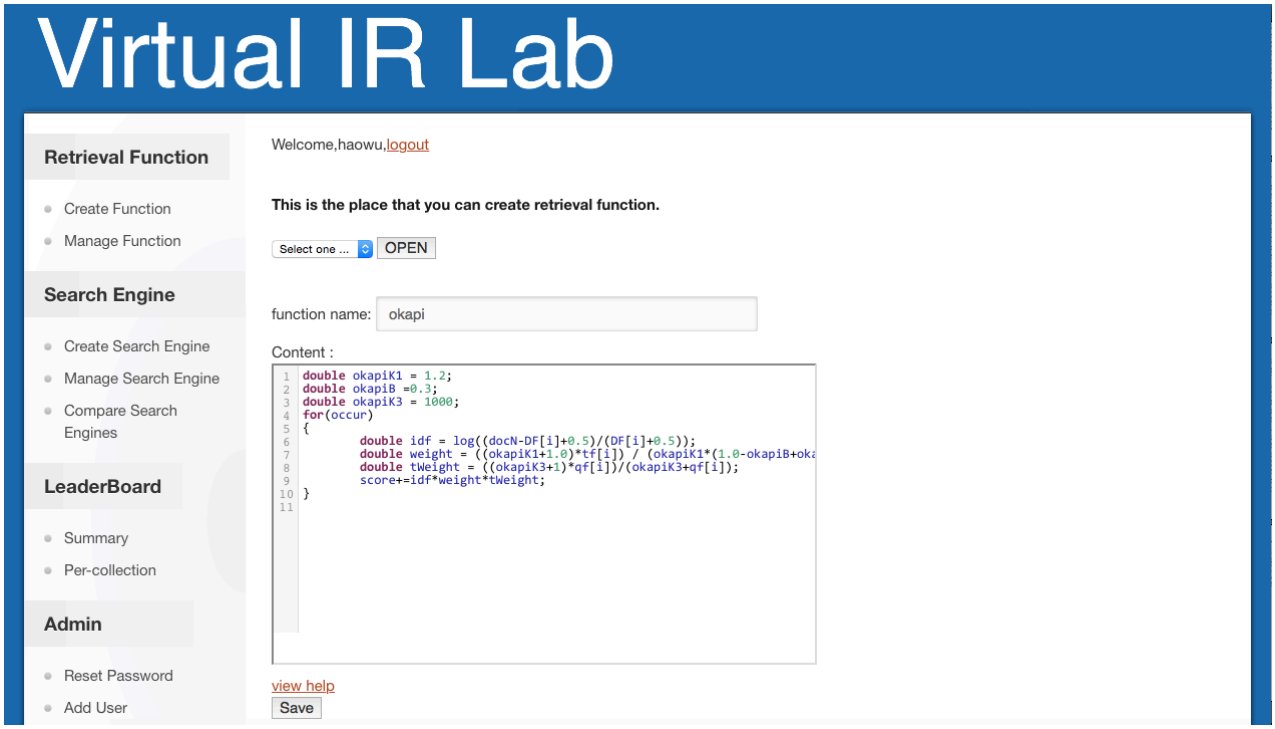


Figure 3.14: The interface of IR virtual lab

help from external sources, however this function is not well-optimized. In particularly, when dealing large collection (e.g. over 10GB), Virtual IR Lab's index building can be very slow and it consumes considerable memory. As a temperate solution, currently for large collection, instead of building index from its own index builder, we recommend to convert Indri [3]'s index to Virtual IR Lab's index. We hope the problem can be improved in future updates.

Virtual IR Lab is one of the many implementations of information retrieval system. What is the common query processing behaviors of these systems? What features affect query processing time? How can improve query processing efficiency? We will try to answer these questions in the following chapters.

Chapter 4

PERFORMANCE MODELING FOR TOP-K QUERY PROCESSING

4.1 Introduction

In the previous chapter, we have shown Virtual IR Lab which contains a good example of DAAT query processing systems. We will try to study the common behavior of DAAT query processing systems and try to improve their efficiency in the following chapters.

A lot of studies have been done for DAAT about how to improve efficiency [29, 49, 50, 91, 97]. The common idea of them is trying to reduce computational cost by dynamically skip documents which cannot be ranked into the final results. Although the methods can improve efficiency in certain situation, they are not good enough. In particularly, when k (i.e. the number of returned document) becomes larger, the cut-off threshold may decrease and less documents will be pruned, which leads to longer query processing time. What is more, dynamic pruning methods are not stable and they cannot guarantee that every query finish on time. To solve this problem, we first need to understand top-k query processing: What are the features which can affect the efficiency of top-k query processing? How do the features influence efficiency? How do we estimate the features of each query before we actually processing it? How do we adjust such features to achieve better efficiency? The questions will be answered in the next sections.

The most related work is the model from Macdonald [70]. They introduce run-time performance prediction framework which can predict top-k query processing time with 42 features. The prediction system was proven to be very useful. First, it can facilitate online query scheduling. To improve the throughput, Web search

engines often replicate the indices so that independent queries can be processed at the same time [23, 28]. Since the execution time varies for different queries, an accurate prediction of query processing time can lead to better load balancing and minimize the query wait time. Second, knowing the query processing time in advance enables us to select retrieval strategies accordingly. For example, if the query processing time would be longer than the necessary response time, a simple yet efficient retrieval strategy, such as the conjunctive mode [92, 95], could be chosen to satisfy the requirement on the response time. But if the query processing time would be shorter, a more expensive but effective retrieval strategy, such as learning to rank [21, 69, 98, 99], could be used. Finally, run-time performance prediction could also be useful in automatically detecting problems in a large-scaled IR system. For example, if a node's actual query processing time is much longer than the predicted value, it could indicate that some faults have happened and the query scheduler might need to assign the task to other nodes.

However the method proposed by Macdonald is still not good enough. First of all, the method is complicated and it requires 42 features. These features are not only hard to pre-compute, but also require a lot of memory and disk space to be stored. The huge resource consumption limits the utilization of the method on large data collections. Second, the prediction result of their method is not accurate enough especially for dynamic pruning methods. Although some features (e.g. total number of postings) can explain the executing time of exhaustive query processing well, the features cannot explain the complicated mechanism of dynamic pruning methods. Therefore the prediction results have very large error comparing to the actual query processing time. Finally, the method does not provide explanation about how the features influence the query processing and interact with each other. With their model, it is still hard to tell how we can improve efficiency or reduce query processing time to meet certain time constraint.

To improve the situation, we propose a novel general performance modeling

framework for top-k query processing methods, including both the exhaustive evaluation and dynamic pruning methods. Specifically, we first conduct a comparative analysis of a few representative top-k query processing methods, identify their commonality and differences, and develop a general model to estimate their run-time performance based on a few identified features. With the developed model, we fit the parameters using a step-by-step method and compute the approximated feature values based on a small set of easily obtained statistics about each query term such as the number of postings, the maximum score of the postings and the minimum score of the postings.

4.2 Model Development

We describe how to develop a general model to estimate top-k query processing time in this section. In particular, we first divide the query processing pipeline into multiple stages, identify the most useful features for cost estimation at each important stage, and then develop the model based on these features accordingly.

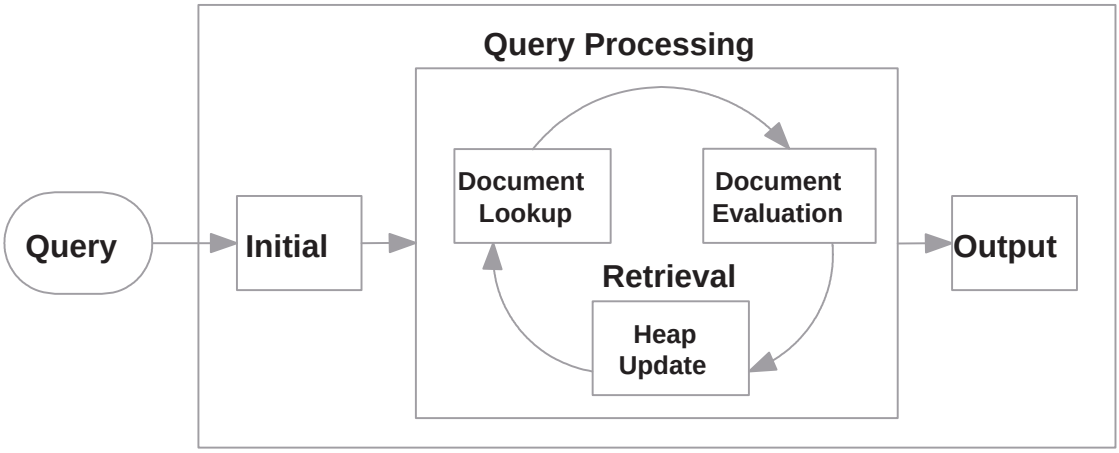
4.2.1 Multiple Stages in Query Processing

Despite the differences among various top-K query processing methods, all of them can be divided into three stages: Initialization, Retrieval and Result generation. The basic idea is illustrated in Figure 4.1.

In the *Initialization* stage, a query is pre-processed and parsed into one or multiple query terms. For each query term, its corresponding inverted list is then located in the index and loaded to the memory. We also need to initialize the variables such as the pointer to the current document IDs in each inverted list for all methods and the maxscore of each list for the pruning methods.

In the *Retrieval* stage, we first traverse the inverted indices to find documents that need to be evaluated, and then compute the relevance score of each document and return the top-K ranked results. It is clear that this stage can be further divided to three steps for each iteration: document lookup, document evaluation and heap update.

Figure 4.1: Multiple stages in the query processing



In the *Result generation* stage, the top-K ranked results could be either displayed to search users or passed to another more complex retrieval methods as input for re-ranking [21]. Since the processing time of this stage is much smaller than the other two stages and it depends on the architecture design of the IR systems (e.g., how to render the search results etc.), we focus on estimating the query processing time for only the first two stages in this study.

4.2.2 Performance Modeling: the *Initialization* stage

The computational cost of the *Initialization* stage is determined by the following two tasks.

The first task is to locate the inverted list of a term based on its numerical ID and load the data to the memory. The process of locating the inverted list can be very efficient with a look-up table, and the time is mainly spent on seeking the posting lists in the disk. Since we need to do it for each query, the time complexity is $O(L(q))$, where $L(q)$ is the number of unique terms in query q. The second task is to initialize variables for the retrieval step. For each query term, we need to initialize the pointer to the current document ID in its inverted list, and then fetch its maxscore when necessary. Thus, the time complexity is also $O(L(q))$. In summary, the time

complexity of the *Initialization* stage is $O(L(q))$.

Thus, we propose to model the processing time T_i of a query q in the *initialization* stage as follows:

$$T_i(q) = \alpha \times L(q) \tag{4.1}$$

where α is a parameter. As shown later, the query processing time of this stage is much smaller than that of the *retrieval* stage.

4.2.3 Performance Modeling: the *Retrieval* stage

The *Retrieval* stage is consisted of three steps: document lookup, document evaluation and heap update. We now discuss the processing time for each step.

In the *document lookup*, we need to traverse the indices and locate candidates documents that need to be evaluated. Since indices are often compressed to reduce the disk space and transferring time [12,115], the first thing we need to do is to decompress the postings in the inverted lists. We denote the time spent on this decompression process as T_d. It is clear that T_d is determined by the amount of data that need to be accessed and how the postings are compressed. In our implementation, we compressed the inverted lists in blocks of 64 postings so that each block can be accessed and decompressed individually [115]. Thus, we can estimate the decompression time T_d of a query q as follows:

$$T_d(q) = \beta_d \times B(q), \tag{4.2}$$

where $B(q)$ is the number of blocks that need to be accessed and decompressed in the query processing for q and β_d is a parameter measuring how fast the system can read from the disk and execute the decompression algorithm.

After decompressing the postings, we need to traverse the indices to locate documents that need to be evaluated. Since we focus on the DAAT-based methods, the pointers to the current document IDs in all inverted indices need to be synchronized so that a document can be fully evaluated before moving to the next one. This process suggests that the query processing time of this step is related to the number of inverted lists, i.e., $L(q)$. Moreover, it is also related to the number of document IDs that need

to be synchronized, which is denoted as $D_s(q)$. The time T_l spent on locating the documents of a query q can be estimated as:

$$T_l(q) = \beta_l \times L(q) \times D_s(q) \tag{4.3}$$

where β_l is a parameter which reflects the machine speed and the complexity of the retrieval function used for the query evaluation. It is worth noting that, in the *exhaustive* evaluation and *maximum score* [97] methods, $D_s(q)$ is essentially the number of documents that need to be evaluated. However, in the *WAND* [29] method, $D_s(q)$ could be much larger than the number of evaluated documents due to the pivot mechanism.

Thus, the processing time of query q in the *document lookup* step can then be estimated as $T_d(q) + T_l(q)$.

Moreover, the computational cost of the *document evaluation step* is related to the number of documents that need to be evaluated as well as the cost of computing the relevance score of a document for the query. Since the cost of computing the relevance score of a document is related to the number of query terms in the document, we can model the processing time of query q in the document evaluation step using:

$$T_e(q) = \beta_e \times P(q), \tag{4.4}$$

where $P(q)$ is the number of postings that need to be evaluated and β_e is a parameter related to the complexity of the retrieval function.

Finally, for top-k query processing, a minimum heap with the size of k can be used to store the information about the top-k documents. With the minimum heap, the update cost can be greatly reduced. In our preliminary experiments in the previous chapter, we find that the time spent on the heap update is usually much smaller compared with other stages unless k is extremely large (e.g., more than 100,000). Thus, in this paper, we ignore the time spent on heap update.

4.2.4 Performance Modeling: Summary

We now summarize the developed model for estimating the query processing time for query q. As described earlier, the model is related to four important features:

- $L(q)$: the number of unique terms in q;
- $B(q)$: the number of blocks that need to be accessed and decompressed when processing q;
- $D_s(q)$: the number of documents whose ID needs to be synchronized when processing q;
- $P(q)$: the number of postings that need to be evaluated when processing q.

Thus, the processing time T of a query q can be estimated as:

$$\begin{aligned} T(q) &= T_i(q) + T_d(q) + T_l(q) + T_e(q) \\ &= \alpha \cdot L(q) + \beta_d \cdot B(q) + \beta_l \cdot L(q) \cdot D_s(q) + \beta_e \cdot P(q) \end{aligned} \tag{4.5}$$

where $\alpha, \beta_d, \beta_l, \beta_e$ are parameters which depends on the machine speed and the retrieval method used for ranking.

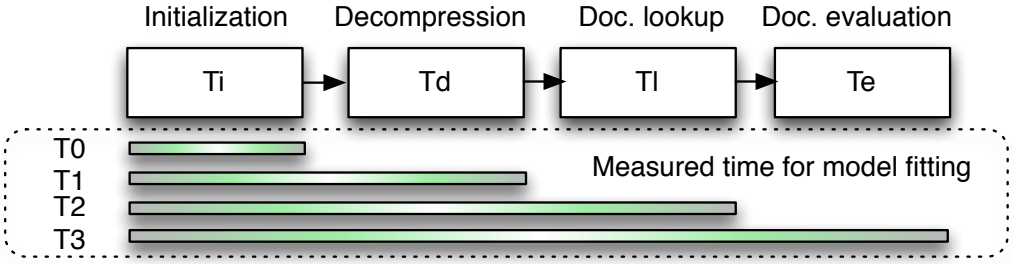
We will discuss how to train the parameter values and how to approximate the feature values based on easily obtained statistics in the following sections.

4.3 Model Fitting

Since the model parameters (i.e., α, β_d, β_l and β_e in Equation (4.5)) depend on the machine configuration and retrieval method used for ranking, we now discuss how to train their values to fit the model based on a training query set.

Here we assume that we have a set of N training queries, i.e., q_1,...,q_N. For each query q_i, we know their actual query processing time $T_3(q_i)$ (refer to figure 4.2) and the corresponding features values, i.e., $L(q_i)$, $B(q_i)$, $D_s(q_i)$ and $P(q_i)$. One simple strategy is to train the values of all the parameters at the same time based on the fitting of Equation (4.5). However, this strategy would not work well since the parameters are not independent. For example, both β_l and β_e are related to the machine speed. Thus, we propose to estimate the parameter values using a step-by-step method so that the parameters can be estimated individually based on Equations (4.1-4.4). This strategy can increase the estimation accuracy, reduce the estimation cost, and enable us to gain a better understanding about what happens at each stage.

Figure 4.2: Measured Time for Model Fitting



4.3.1 Fitting for the exhaustive evaluation method

We start with a discussion on fitting the exhaustive evaluation method since its feature values do not change with the query processing progress. Thus, we can safely divide the entire query processing into a few independent parts and use the actual time used for each part to estimate parameters independently. Figure 4.2 shows the basic idea.

To estimate the parameter α related to the *initialization* stage, we can get the actual initialization time for each query, denoted as $T_0(q_i)$, and then use linear regression to estimate the value of α based on Equation (4.1). We then conduct experiments to get the time taken for initialization and reading the inverted lists of all query terms, say $T_1(q_i)$. It is clear that the actual decompression time can then be computed using $T_1(q_i) - T_0(q_i)$. With this time, we can then apply linear regression and estimate the value of β_d based on Equation (4.2). Next, we conduct another set of experiments to measure the time taken for the first three steps, i.e., $T_2(q_i)$. In particular, for each query, we initialize the process and go through the query document lookup stage to get all the candidate documents without computing the relevance scores. Thus, we can get the actual document lookup time using $T_2(q_i) - T_1(q_i)$, and then estimate the value of β_l in Equation (4.3) using linear regression. Finally, we can the get the actual document evaluation time using $T_3(q_i) - T_2(q_i)$ and then estimate the value of β_e based on Equation (4.4) using linear regression.

4.3.2 Fitting for the dynamic pruning methods

For the dynamic pruning methods such as *maximum score* [97] and *WAND* [29], we cannot apply the same strategy as the exhaustive evaluation method because the stages are not independent. For example, it is impossible to separate the document lookup step from the evaluation since the pruning threshold used in the lookup stage needs to be adjusted by the relevance scores computed in the evaluation stage.

Fortunately, the operations involved in the posting list decompression and document evaluation are identical to the exhaustive evaluation method. Thus, it is reasonable to assume that β_d and β_e are the same as those in the exhaustive evaluation method. And we only need to estimate the value of α and β_l.

In the *initialization* step, the dynamic pruning method needs more operations than the exhaustive evaluation method. For example, they need to find the maxscores for each query term. Thus, the α value would be different, but we can use the similar strategy as described in the previous subsection to estimate its value. For β_l, we can first compute the actual time used for document lookup using

$$T_3(q_i) - T_0(q_i) - \beta_d \times B(q_i) - \beta_e \times P(q_i),$$

and then apply linear regression to learn the value of β_l accordingly.

4.4 Feature Approximation

We have developed the performance model based on four features and discussed how to estimate the parameters in the previous sections. In order to predict a query's processing time before executing the query, we need to know the values of these four features without traversing the indices. Among the four features, $L(q)$ is easier to obtain while the other three features are impossible to directly obtain from the indices. To tackle this challenge, we propose to approximate the real feature values based on easily obtained statistics information and then predict the run-time performance based on the developed model as shown in Equation (4.5).

The statistics information used for the approximation is summarized as follows:

- N, the number of documents in the collection;

- $P_i(q)$, the number of postings in the inverted list of the i-th term in query q;
- $M_i(q)$, the maxscore (i.e., the highest relevance score contribution) of the inverted list of the i-th term in query q;
- $m_i(q)$, the minscore (i.e., the lowest relevance score contribution) of the inverted list of the i-th term in query q.

All these statistics here are either accessible from the indices or can be computed with the minimal cost. For example, $M_i(q)$ and $m_i(q)$ can be computed while building the indices and it only takes less than 20MB space for the GOV2 collection [2]. It is certainly possible to use other statistics information such as the distribution of the postings. However, the more statistics we use, the more space and time will be spent in the prediction. This study focuses on predicting the run-time performance with minimum cost, it should be more accurately prediction if we use more statistics.

4.4.1 Approximation in the exhaustive evaluation method

The exhaustive evaluation method would essentially traverse the postings of all the query terms and compute the relevance score for each document in the postings. Thus, we can easily compute the values of $P(q)$ and $B(q)$ as follows:

$$P(q) = \sum_{i=1}^{L(q)} P_i(q), B(q) = \sum_{i=1}^{L(q)} \frac{P_i(q)}{Bsize}.$$

$Bsize$ is the number of postings in each compressed block, and it is set to 64 in our implementation.

Now the remaining challenge is to approximate the value of $D_s(q)$, which is equal to the number of evaluated documents in the exhaustive evaluation method. Since it is essentially the number of documents containing at least one query term in the collection, its value would be between $max(P_i(q))$ (when query terms always co-occur in the documents) and $\sum P_i(q))$ (when query terms never co-occur in the documents).

Let us start with a simple case when a query has only two terms. It is trivial to show that $D_s(q)$ can be estimated using $P_1(q) + P_2(q) - Con(q)$, where $Con(q)$ is the

number of documents that contain both query terms. Thus, the challenge is converted to the problem of estimating the number of documents containing two terms. Assuming query terms follow the binomial distribution, the probability of observing a query term in a document can be estimated as $\frac{P_i(q)}{N}$. If we assume that the two query terms are independent, the expected number of documents that containing both terms can be estimated as $\frac{P_1(q)}{N} \times \frac{P_2(q)}{N} \times N$. To make it more generalized, when the two terms are not independent and $P_1(q) < P_2(q)$, we propose to estimate the number of documents containing both terms as follows:

$$Con(q) = \frac{P_1(q)}{N} \times (\frac{P_2(q)}{N})^{\gamma} \times N, \tag{4.6}$$

where γ is a parameter which controls how the two terms are related. When γ is equal to 1, the two terms are independent. And when γ is equal to 0, the two terms are closely related (i.e., $P_1(q) \in P_2(q)$). In this paper, we set $\gamma = 0.5$ since query terms are related and tend to occur in the same documents.

The estimation can be easily generated to the queries with more terms, and the algorithm is described in Algorithm 1.

Algorithm 1 Estimation of $D_s(q)$ for the exhaustive evaluation method

```
D = 0
for i = 1 to L(q) do
  if D ≤ P_i(q) then
    Con = D * (P_i(q)/N)^γ
  else
    Con = (D/N)^γ * P_i(q)
  end if
  D+ = P_i(q) − Con
end for
```

4.4.2 Approximation in the pruning methods

The feature approximation for pruning methods are much more complicated since the query processing is controlled by a pruning threshold, θ, whose value keeps changing in the process. Indeed, θ plays a critical role in the query processing method

since it determines when the pruning begins and how much pruning is needed for each inverted list. Thus, we will start with the estimation of θ and then move on to discuss how to approximate the feature values in the *maxScore* and *WAND* methods.

4.4.2.1 Estimating the pruning percentage

The basic idea of dynamic pruning methods is to skip the scoring of documents that can not make to the final top-k results. The selection of these documents is controlled by θ, the smallest relevance score in the intermediate top-K results. The value of θ will monotonously increase during the query processing. To model the pruning behavior, we have to trace the change of θ. Not every possible value of θ could change the pruning behavior, i.e., how much to prune for each inverted list. Instead, the pruning behavior would only change at a few *critical values*.

Let us consider the *maximum score* method first. The pruning would start when θ is equal to or greater than the smallest maxScore value of all the inverted lists. The pruning behavior will change (i.e., prune more postings) again when θ is equal or greater than the sum of the smallest maxScore values. In fact, given a query q, the number of critical values for θ in *maxScore* is $L(q)$. The pruning process in the *WAND* method is similar but with different number of critical values, i.e., $2^{L(q)} - 1$, since it considers the combinations of the inverted lists in the pruning process.

Clearly, the key challenge is to understand the pruning behavior for each critical value in a pruning method. Specifically, we need to estimate the percentage of the postings that will be pruned in each inverted list for every critical value of θ. Algorithm 2 provides a method that can estimate the pruning percentage of a critical value of C when retrieving top-K documents for query q. The basic idea of the algorithm is to estimate the number of the documents $D(C)$ whose score is higher than C. If the estimated number (i.e., $D(C)$)is smaller than K, there is no pruning is necessary and 0 is returned. Otherwise, we know around K out of $D(C)$ documents are not pruned, so the pruning percentage can be computed as $1 - \frac{K}{D(C)}$.

Algorithm 2 The estimation of pruning percentage of a critical value C when retrieving top K documents for q

Function{getPrunePercentage}{C,K,q}
$D(C) = 0$
for $g \in G(q)$ **do**
  $D(C) += D_g * (1 - normalCP(\mu_g, \sigma_g^2, C))$
end for
if $D(C) < K$ **then**
  return 0
else
  return 1-$\frac{K}{D(C)}$
end if
EndFunction

We now explain how to estimate $D(C)$. The basic idea is to divide documents into multiple groups based on the occurrences of query terms. For example, if $q = \{A, B\}$, there would be three document groups corresponding to the ones covering only A, covering only B and covering both terms. A set of these document groups is denoted as $G(q)$. We assume that the number of documents in each group g follow a normal distribution $N(\mu_g, \sigma_g^2)$. $normalCP(\mu, \sigma^2, C)$ denotes the percentage of documents whose value is smaller than C and D_g is the number of documents in the group g. For each group g, we can estimate $D(g)$ following the similar method described in Algorithm 1 and estimate μ_g and σ_g^2 based on the maxscores and minscores of the corresponding query terms for document group g.

4.4.2.2 Feature approximation for maximum score method

For the simplicity, we assume that $M_1(q) \geq M_2(q) \geq ... \geq M_{L(q)}(q)$ for any query q. Algorithm 3 describes how to estimate the features (i.e., $B(q), D_s(q)$ and $P(q)$) in the *maxScore* method.

We first compute all the critical values, i.e., $C_i(q)$, for the *maxScore* method. Note that each critical value corresponds to a query term. After that we go through each terms to calculate its contributions to all the features.

For each term, we first try to estimate the number of documents that contain

Algorithm 3 The approximation of features for maxScore

FUNCTION getmaxScoreFeatures $\{K$,$q\}$
sort$(query)$
$B(q) = 0$
$D_s(q) = 0$
$P(q) = 0$
for $i = L(q)$ **to** 1 **do**
  $C_i(q) = C_{i+1}(q) + M_i(q)$
end for
for $i = 1$ **to** $L(q)$ **do**
  if $D_s(q) \leq P_i(q)$ **then**
    $Con = D_s(q) * (P_i(q)/N)^{\gamma}$
  else
    $Con = (D_s(q)/N)^{\gamma} * P_i(q)$
  end if
  $B_i(q) = \lceil P_i(q)/64 \rceil$
  $Per = getPrunePercentage(C_i(q), K, q)$
  $RB_i(q) = B_i(q) * Per$
  $B(q) = B(q) + B_i(q) * (1 - Per) + RB_i(q) * (1 - ((RB_i(q) - 1)/RB_i(q))^{Con*(1-Per)})$
  $D_s(q) = D_s(q) + (P_i(q) - Con) * (1 - Per)$
  $P(q) = P(q) + P_i(q) * (1 - Per) + Con * Per$
end for
return $B(q)$,$D_s(q)$,$P(q)$
ENDFUNCTION

both the current term and the terms we have already evaluated, i.e., Con. Con is important since these documents will always be accessed no matter whether pruning is applied or not. $B_i(q)$ is the number of blocks in the inverted list of the current term and the block size used in our implementation is 64. After that, we get the pruning percentage based on the corresponding critical value and figure out the percentage of the pruned postings in the inverted list of current term. The contribution to each feature is calculated as two parts: before θ reaches the critical value and after. For example, before reaching the critical value, the contribution to $P(q)$ is $P_i(q) * (1 - Per)$ (where Per is the pruned percentage) since every non-pruned postings will be exhaustively evaluated. After reaching the critical value, the contribution to $P(q)$ is $Con * Per$ since we only access the postings that contain other terms (i.e., Con) after pruning. Moreover, before reaching the critical value, the contribution to $D_s(q)$ is related to the number of postings that do not contain previously evaluated terms and can be computed as $(P_i(q) - Con) * (1 - Per)$, but after reaching the critical value, the corresponding query term will not be eligible to contribute new documents, so its contribution to $D_s(q)$ is 0.

$RB_i(q)$ is the number of remaining blocks after reaching to the critical value. We can then estimate the number of decompressed blocks within them using binomial model. If the pruning happened, not every posting in the posting list will be accessed. And if none of the 64 postings within a block is accessed. The block will be skipped and it will not be decompressed. If we have RB blocks and we accessed p postings from them. For each block, the probability that none of the p postings is from the block can be estimated by the binomial model as $(\frac{RB-1}{RB})^p$. So for all the RB blocks, the expected number of decompressed blocks can be estimated as $RB \times (1 - (\frac{RB-1}{RB})^p)$.

4.4.2.3 Feature approximation for WAND

Different from maxScore, WAND applies a pivot-based pruning strategy to prune more documents. As described earlier, given a query, we can divide documents

into different groups based on the query term occurrences. These documents are denoted as $G(q)$. For each $g \in G(q)$, we denote M_g as the sum of the maxscores of all query terms corresponding to the group g. In fact, each M_g can be considered as a critical value for the *WAND* method. For example, if θ is higher than M_g, documents from the group g will be pruned. Thus, if we know $getPrunePercentage(M_g, K, q)$ is equal to 0.7, we expect the first 30% documents from group g will be evaluated while the remaining 70% will be pruned. Algorithm 4 shows the method used to compute the approximated features values in *WAND*.

Algorithm 4 The approximation of features for WAND

FUNCTION getWANDFeatures {K,q}
$B(q) = 0$
$D_s(q) = 0$
$P(q) = 0$
for $i = 1$ to $L(q)$ **do**
  $E_i(q) = 0$
end for
for $g \in G(q)$ **do**
  $Per = getPrunePercentage(M_g, K, q)$
  for $i = 1$ to $L(q)$ **do**
    if $q_i \in g$ **then**
      $E_i(q) = E_i(q) + D_g * (1 - Per)$
    end if
  end for
end for
for $i = 1$ to $L(q)$ **do**
  $P(q) = P(q) + E_i(q)$
  $D_s^i(q) = (E_i(q)^\lambda * P_i(q))^{1/(1+\lambda)}$
  $D_s(q) = D_s(q) + D_s^i(q)$
  $B_i(q) = \lceil P_i(q)/64 \rceil$
  $B(q) = B(q) + B_i(q) * (1 - ((B_i(q) - 1)/B_i(q))^{D_s^i(q)}$
end for
return $B(q)$,$D_s(q)$,$P(q)$
ENDFUNCTION

The basic idea is to estimate the number of the evaluated postings for each query term (i.e., $E_i(q)$) and then use it to estimate $D_s(q)$ and $B(q)$. Since the pruning of WAND is based on combinations of posting lists, each combination corresponds to

a document group $g \in G(q)$. For each document group, we can then get the pruning percentage based on $getPrunePercentage(M_g, K, q)$, so the number of evaluated documents in each group can be estimated as $D_g * (1 - Per)$. As mentioned earlier, D_g is the number of documents in group g. After iterating through all the document groups, we can get the value $E_i(q)$ for each query term. We can then compute the value of $P(q)$ by summing all the value of $E_i(q)$ together.

The remaining question is how to estimate $D_s^i(q)$. We know that its upper bound is the posting list length (i.e., $P_i(q)$) and its lower bound is the number of evaluated postings for the term (i.e., $E_i(q)$). In this paper, we use a heuristic way to estimate a value between these two bounds, i.e., $D_s^i(q) = (E_i(q)^\lambda * P_i(q))^{1/(1+\lambda)}$. It uses parameter λ to control which bound to approach more. When $\lambda = 1$, it is geometric mean. After we get the number of access postings for each term, we can the estimate the number of decompressed block using the binomial model described earlier.

4.5 Experiments

4.5.1 Experimental design

We evaluate how well the proposed performance model can predict the query processing time for the exhaustive evaluation method using DAAT, denoted as *Exhaustive*, as well as two dynamic pruning methods, i.e., *maximum score* [97] and *WAND* [29].

Regarding the data collection, we use the TREC Gov2 collection [2], which consists of 25.2 million web pages crawled from the .gov domain. For queries, we randomly select 1,000 queries from the efficiency queries used in the TREC 2005 Terabyte track as the training set to fit the proposed model. This data set is denoted as *Train05*. Moreover, we randomly select a different set of 4,000 queries from the same track as the first test collection, denoted as *Test05*, and randomly select 5,000 queries from the efficiency queries used in the TREC 2006 Terabyte track as the second test collection, denoted as *Test06*.

Similar to other sections, all experiments were conducted on a single machine

Table 4.1: Prediction comparison with real feature values: RMSE (ms)

	Exhaustive		*maximum score*		*WAND*	
	Test05	*Test06*	*Test05*	*Test06*	*Test05*	*Test06*
Postings	56.41	25.13	16.46	19.12	38.82	36.37
Model-l	**36.05**	**14.16**	**6.52**	**7.42**	**7.11**	**5.69**
Model-s	39.95	23.93	7.67	9.28	10.01	6.03

with dual AMD Lisbon Opteron 4122 2.2GHz processors and 32GB DDR3-1333 memory. And we use Okapi BM25 [82] as the retrieval function to rank documents. To evaluate the performance of the prediction, we use the RMSE (Root Mean Square Error) as the evaluation measure. A smaller RMSE means the prediction is more accurate. The query processing system is implemented similar to Virtual IR Lab which is described in previous chapter.

4.5.2 Results on performance modeling

The first set of experiments is to examine whether the proposed model fitting method can capture the actual run-time performance well. We use the 1,000 queries from *Train05* as training queries with K, i.e., the number of retrieved documents, is set to 10.

We use two ways to estimate the parameters: (1) Linear regression and (2) the method described in Section 4.3. We denotes them as **Model-l** and **model-s** accordingly. To make a comparison, we trained simple models using the number of evaluated postings which denotes as **Postings**. table 4.1 summarize the results of the comparisons.

Figure 4.3 shows the comparison between the predicted and actual processing time of each query for all the three query processing methods on *Test05* collection. It is clear that the proposed model fitting method is effective, and the predicted query processing time correlates well with the actual query processing time for both the exhaustive evaluation and dynamic pruning methods. The trend on the *Test06* collection

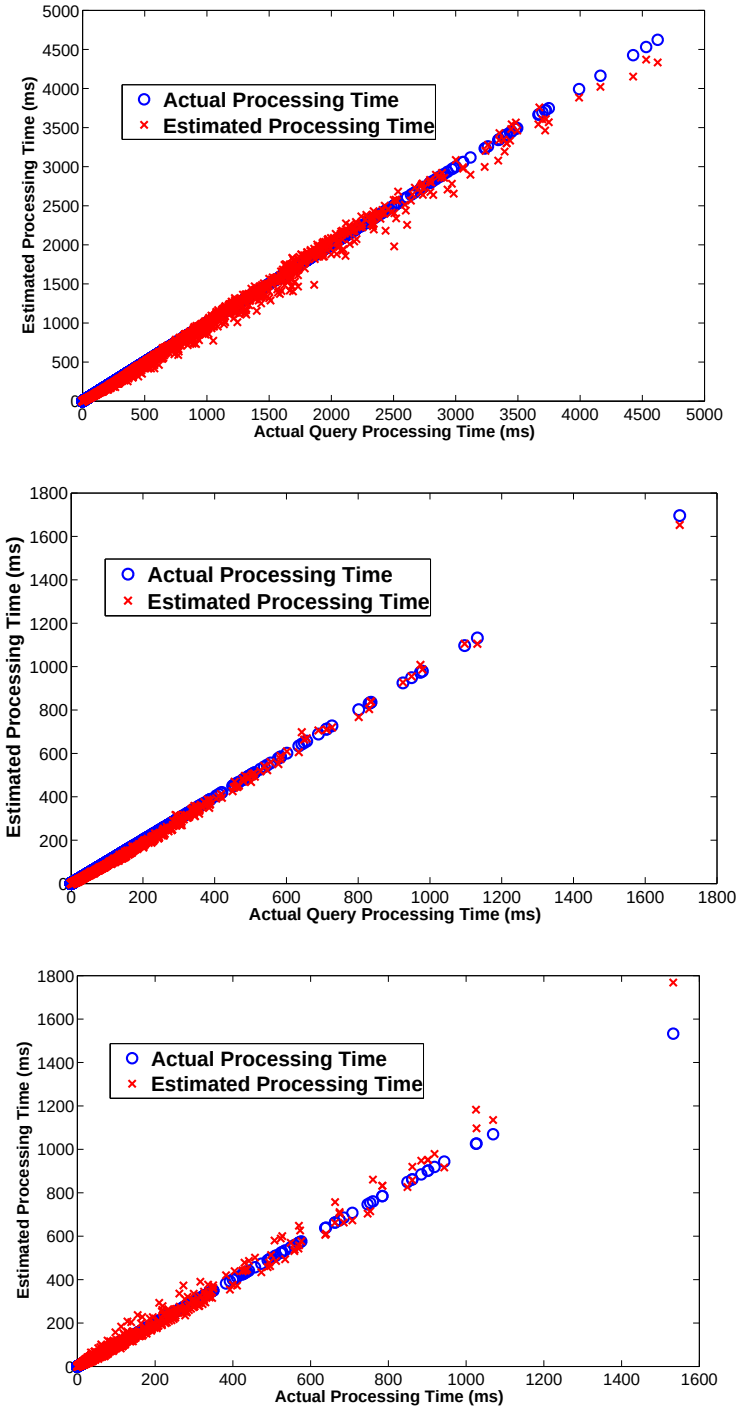


Figure 4.3: Model Fitting of *Exhaustive* (up), *maximum score* (middle) and *WAND* (down) on *Test05*

Table 4.2: Percentage of the time spent on each stage

	Init.	Decompress.	Lookup	Evaluation
Exhaustive	0.13%	5.45%	13.33%	81.08%
maximum score	2.38%	9.80%	16.68%	71.13%
WAND	2.46%	25.87%	25.92%	45.76%

is similar.

The results show that our model can accurately estimate query processing time by using real features. Using linear regression can lead to better performance than training parameters separately. It may due to the inaccurate time measurements at each stage.

Finally, we conduct experiments to understand why the number of evaluated posting lists is not a good performance indicator for dynamic pruning methods. Table 4.2 shows the average percentage of the time spent on each stage of the query processing over the *Train05* collection. When using the *Exhaustive* method, more than 80% of time are spent on the evaluation, which is related to the number of evaluated postings. The *Postings* method can capture the query processing time in the document evaluation stage well. Thus, if a query processing method spent most time on the evaluation step, the *Postings* method might give a reasonable performance prediction. However, for other methods (such as WAND), which spend less time on the evaluation stage, the *Postings* method would not be a good performance indicator.

4.5.3 Results on processing time prediction

In this section, we conduct experiments to evaluate how well the developed model can predict the query processing time in the real-world scenarios, when the real feature values are unknown.

By applying the approximation methods proposed in Section 4.4, we are able to compute the estimated feature values based on a few easily obtained statistics. We can then plug the estimated feature values together with the learned parameter values (as

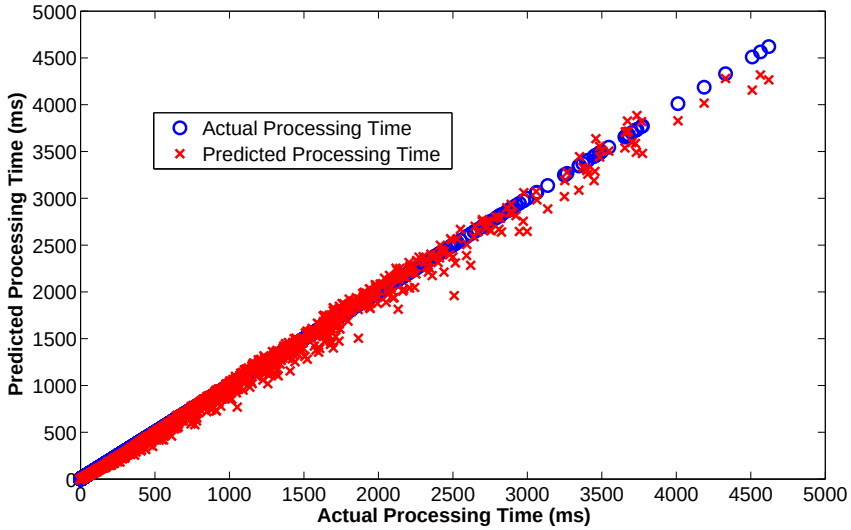


Figure 4.4: Processing time predicted by **Model** for *Exhaustive* on *Test05*

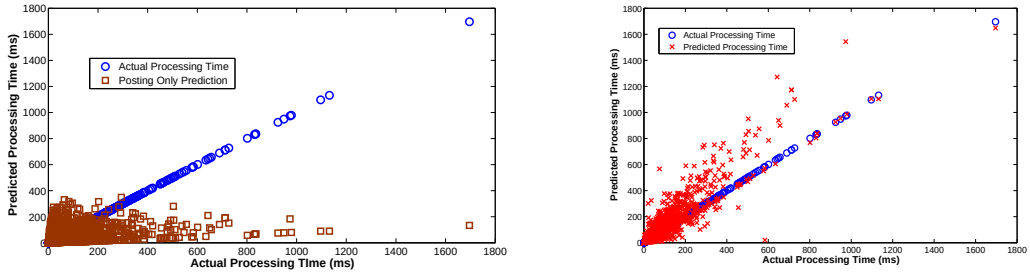


Figure 4.5: Processing time prediction for *maximum score* on *Test05*: **BL** (left) and **Model** (right)

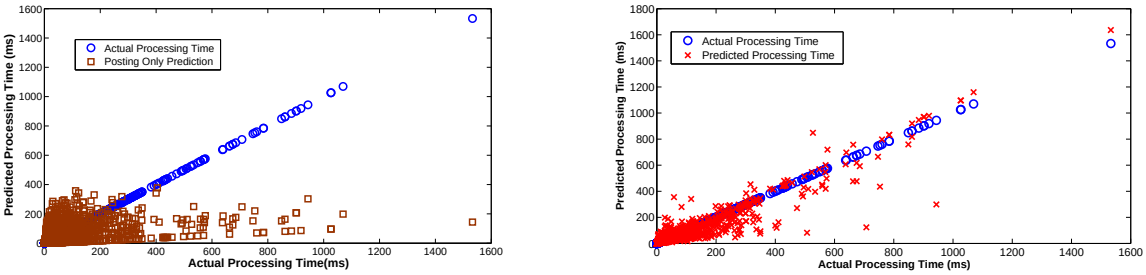


Figure 4.6: Processing time prediction for *WAND* on *Test05*: **BL** (left) and **Model** (right)

Table 4.3: Prediction comparison with estimated feature values: RMSE (ms)

	Exhaustive		*maximum score*		*WAND*	
	Test05	*Test06*	*Test05*	*Test06*	*Test05*	*Test06*
Postings	56.41	25.13	88.52	81.26	82.97	57.69
42 features	37.41	22.54	78.29	66.28	74.66	56.91
Model-l	37.3	**13.71**	74.44	43.43	31.84	26.9
Model-s	40.97	22.85	**50.24**	**29.57**	31.76	28.23
combine	**29.36**	14.66	63.13	38.33	**31.07**	**26.44**

described in the previous subsection) into the proposed performance model as shown in Equation (4.5) and compute the predicted query processing time for each query.

In addition to using the sum of the posting list length of each query terms which denotes as *Postings*, we also text the linear regression model suggested in [70] which denotes as *42 Features*. We also try to combine the 42 features with our model's 4 features and use linear regression to train models of the 46 features which denotes as *Combine*.

Table 4.3 shows the results of the comparisons. The linear model with 42 features is more accurate than using the number of postings alone. The observation is similar to the those by Macdonald [70]. However, our model can more accurately predict query processing time especially for dynamic pruning methods such as *maximum score* and *WAND*. The reason behind these results is that our model can better estimate the pruning behaviour of these dynamic pruning methods. In some tests, training parameters separately can perform better than that using linear regression. Although training parameters separately can suffer from inaccurate time measurement for each stages in parameters estimation, the trained parameters are more stable and they are less sensitive to errors between predicted features and real ones. Combined with the 42 features, the performance can increase a little for most of the test. It is because the 42 features provide some information that our model ignored (e.g. heap behaviour). However the increase is not significant which indicate that our model has captured the main part of query processing cost.

Table 4.4: Prediction results at different query length: RMSE (ms) (Test05)

	1	2	3	4	5	> 5
	model-s					
Exhaustive	7.26	20.44	33.42	52.69	82.81	162.49
maximum score	6.89	50.06	51.28	66.03	72.71	112.22
WAND	5.67	28.30	32.47	25.18	47.47	66.02
	42 feature					
Exhaustive	16.96	19.33	27.00	39.14	72.36	161.80
maximum score	82.12	73.61	60.87	77.04	116.92	136.48
WAND	72.80	76.21	58.66	71.80	110.88	116.68

Table 4.5: Prediction results at different query length: RMSE (ms) (Test06)

	1	2	3	4	5	> 5
	model-s					
Exhaustive	11.56	19.04	23.09	27.47	35.09	52.87
maximum score	12.57	15.54	28.13	40.34	62.52	80.43
WAND	6.44	21.68	26.84	41.30	41.97	57.32
	42 feature					
Exhaustive	35.29	20.11	16.01	22.83	30.60	36.78
maximum score	43.44	48.77	70.48	74.16	112.39	164.31
WAND	39.79	42.81	58.80	63.65	95.62	148.59

We further show the prediction results at different query length at table 4.4 and table 4.5. From the table we see that the prediction is stable at different query length. For longer query, the query processing time is longer and as a result, the prediction error is also increased.

4.5.4 More analysis

One of the most important usage of our model is to help people understand query processing time. For example, what would the query processing time be if we applied the same retrieval strategy to another data collection? How will the query processing time change if we use different query processing functions? How will the query processing time be reduced if we return fewer documents?

Table 4.6: Prediction Results for different K: RMSE(ms) (Test05)

	10	100	1000	10000
	maximum score			
Postings	88.52	108.30	151.20	233.17
42 Features	78.29	104.32	154.32	238.64
Model-l	74.44	78.32	83.00	81.72
Model-s	**50.24**	**52.65**	**60.06**	**64.04**
	WAND			
Postings	82.97	93.86	120.96	188.55
42 Features	74.66	91.73	126.30	196.77
Model-l	31.84	37.10	**47.67**	59.32
Model-s	**31.76**	**36.65**	47.84	**58.31**

To prove these usages and test the robustness of our model, We conduct the following experiments which use the model we trained in previous subsection to apply on different environment.

If we change K, the number of retrieval documents, the pruning behaviour of dynamic pruning method may change and more/less postings will be pruned. To see whether our model can capture these changes, we apply the model trained on $K = 10$ to predict the query processing time at different K value. Since the change of K has little influence for non-pruning method, we only report the prediction results for dynamic pruning methods.

Table 4.6 and table 4.7 summarizes the comparisons on prediction on different K. For two baseline methods, *Postings* and *42 Features* cannot monitor the change of different K and their error become bigger as K increases. Our model on the other hand can capture the change of K very well. Training parameters separately performs a little better since the its parameters are more robust.

The encouraging high accuracy of the cross-K prediction may suggest one interesting application of our model is that we can control the query processing time by changing the number of returned documents based on the model prediction results. We would like to explore the direction in our future work.

Table 4.7: Prediction Results for different K: RMSE(ms) (Test06)

	10	100	1000	10000
	maximum score			
Postings	81.26	110.21	149.29	238.91
42 Features	66.28	96.22	141.39	230.40
Model-l	43.43	44.34	49.92	**54.53**
Model-s	**29.57**	**38.11**	**48.70**	57.03
	WAND			
Postings	57.69	63.39	90.46	158.20
42 Features	56.91	67.56	95.84	161.71
Model-l	**26.90**	**31.39**	41.21	57.67
Model-s	28.23	32.35	**40.47**	**53.25**

Figure 4.7: Average predicted query time for different values of K: *maximum score*

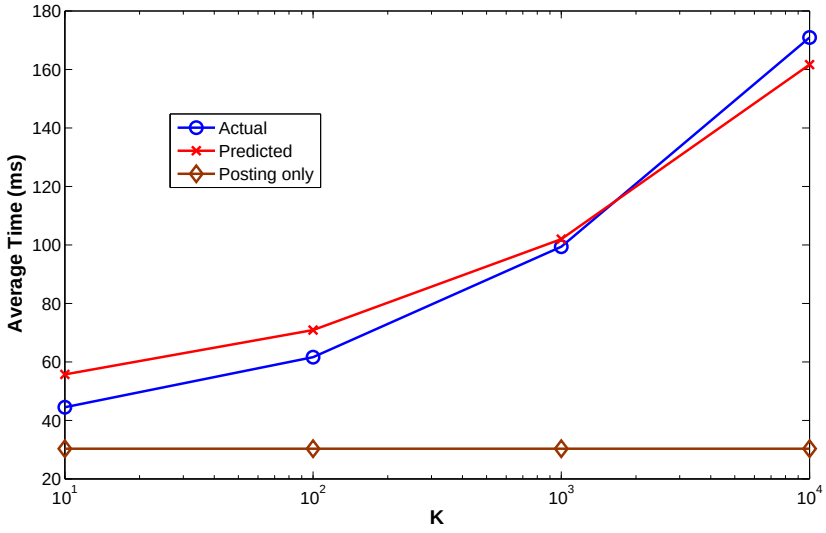


Figure 4.8: Average predicted query time on different values of K: *WAND*

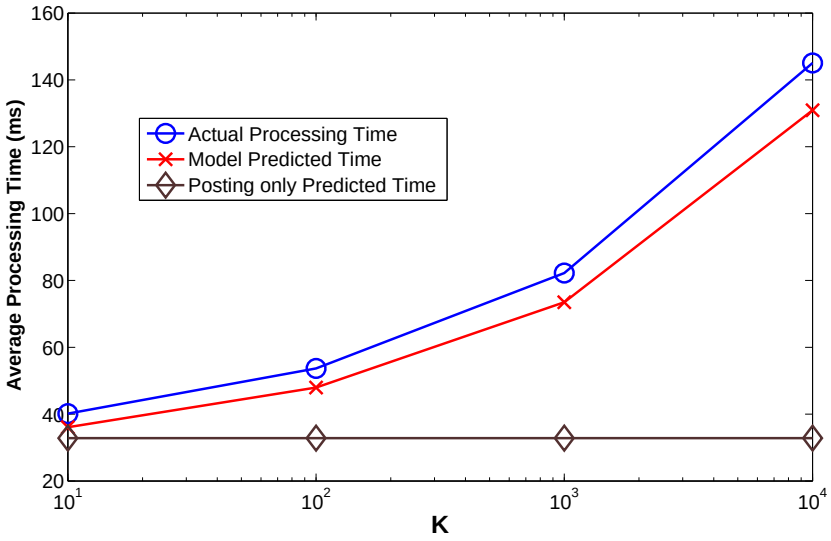


Table 4.8: Cross collection prediction results

	Robust04		*Clueweb09B*	
	maximum score	WAND	maximum score	WAND
Postings	8.83	5.12	323.26	259.72
42 Features	902.82	835.49	407.67	389.78
42 Features op	1.49	1.17	94.05	85.92
model-l	5.01	4.29	119.70	85.96
model-s	**1.86**	**2.39**	**72.91**	**63.86**

To further test the robustness of our model, we apply the models trained on TREC Gov2 [2] collection to test on other two TREC collections: (1) Robust04 [5] collection: a much smaller collection which contains only 528K documents and (2) Clueweb09 [1] category B collection: a larger collection which contains about 50 million documents. We test the collections with 249 official robust 2004 track queries and 200 official 2009-2012 web track queries accordingly. Table 4.8 summarize the comparison results of the two collections.

It is obvious that the model for 42 featured trained on Gov2 collection totally

Table 4.9: Space usage for different prediction method (GB)

Index	postings	42-feature	model
10.8	0	1.8	0.2

fails in the cross collection test. The reason behind this large error is that their parameters are highly collection-dependent. So when we switch to a very different collection (25.2 million to 0.5 million documents), the parameters trained on original collection will hardly apply again. Using the number of total postings may product a fair results especially for small collection, because dynamic pruning plays little role in small collections and number of postings dominate most of the performance.

It is encouraging to see our model performs quite well in the cross-collection test, especially for training parameters separately. Not influenced by the dependence between features, the trained parameters are more robust and collection independent. This encouraging result may suggest one advantage of our prediction model: it can adjust to rapid data collection changes which is important for real time search. It can also help in modern multiple layer index environment since you only need one rather than multiple models to predict query processing time at each ties.

4.5.5 Efficiency of the model

Most of the features in our model can directly access the indices which support a dynamic pruning retrieval method (e.g. posting list length and maximum score of each query term). The only additional feature we need to compute is the minimum score of each query term. We summarize the original index and additional space usage for gov2 collection as table 4.9.

Compared to prediction by 42 features, our model is more space efficient and it uses less than 2% of the original index which may be ignored in most of the applications.

The computational cost of our model is $O(2^{L(q)})$ where $L(q)$ is the number of unique query terms. For most of the queries which are less than 10 terms, the cost

Table 4.10: Average Query Processing time (ms)

	Test05		*Test06*	
	maximum score	WAND	maximum score	WAND
no-prediction	42.73	39.05	59.89	44.53
prediction	42.96	39.21	60.22	44.56

is still ignored compared to the query processing cost which may involve millions or even billions of operations. To test the efficiency of our prediction method, we conduct two experiments: 1. The first experiment is a simulation of the real situation that we built two similar query processing system. The only difference between them is that one predicts processing time before executing the query while the other does not. We test whether doing prediction will slow down the system.

We do both experiments on *Test05* and *Test06* queries on Gov2 collection with the top 10 documents returned for each query which is the same as the previous experiments. We repeat each experiments 5 times and reports the average time as table 4.10.

Compared to query processing time with no-prediction, prediction introduces an insignificant additional cost (e.g. less than 0.6%). Notice that the cost of our prediction method is independent to collection size, we expect the percentage to become smaller for larger collections.

4.6 Conclusions

Modeling the time consuming of top-k query processing system can help us to understand why some queries take longer time than others. It is an important step for us to develop strategies to further improve efficiency of information retrieval systems. Although it is relative easy to predict the performance of the exhaustive evaluation method, it is rather challenging to accurately model and predict the run-time performance for the dynamic pruning methods. Previous studies have attempted to address the problem by either use one feature to approximate the performance or

apply machine learning methods to combine more than 40 loosely related features without conducting deep analysis on the query processing pipeline.

We provide a simpler but more accurate analytic performance model to better explain executing time of top-k query processing systems. In particular, we analyze each query processing step, identify important features related to the query processing time and propose to use a small set of easily obtained statistic information to estimate the features. Experiment results show that the proposed model can predict the query processing more accurately than the state of the art method. In particular, the model is capable of capturing the relation between the value of K and the processing time. Finally, the model can generate fairly accurate prediction for the average processing time of a group of queries.

The model and some observations in this chapter are useful for our following studies which try to improve query processing efficiency. In algorithm 2, algorithm 3 and algorithm 4 we show that how some features such as K (i.e. the number of returned documents) will influence the query processing time for a query when dynamic pruning techniques are applied. It is obvious that the query processing time of a query will increase greatly as K increase and it explains the reason why dynamic pruning methods may become less effective when K is large. In addition to that, the model also shows that the computational cost may increase exponentially as the query becomes longer. So how can we improve the query processing efficiency when K is large? How can we reduce query processing time for long queries? We will try to answer these questions in the next chapter.

Chapter 5

IMPROVE EFFICIENCY OF QUERY PROCESSING THROUGH DOCUMENT PRIORITIZATION

5.1 Introduction

In the previous section, we built an analytical model which can explain query processing time and provide accurate prediction for it. In the model, it is shown that query processing time is related to some features such as query length, number of iterations in document lookup, number of evaluations and etc. The next question is, how can we use the model to improve the efficiency of DAAT query processing.

In order to improve the efficiency of query processing, several dynamic pruning methods have been proposed [29,43,49,50,84,90,97]. Instead of exhaustively evaluating every document that matches at least one query term, dynamic pruning techniques evaluate only a smaller set of documents that have greater potential to make to top K results and bypass the evaluation for the rest of the documents. However all of the techniques share a weakness that when k (e.g. the number of returned document) is large, their improvement of efficiency becomes less significant. By using our analytical model, this phenomenon can be explained more clearly: as more documents need to be returned for each query, the cut-off threshold may decrease. Therefore during query processing, more iterations and document evaluations are required and the query processing increases. This weakness seems unavoidable for all the dynamic pruning methods who uses the cut-off threshold to determine whether to skip documents or not. And this problem may become more serious for advanced dynamic pruning methods such as BMW [50] and Live block [49] which require more efforts in document iterations.

IR system usually applies multi-layer architecture and it requires the number of returned document at top-k query processing to be large enough so that the results quality of later re-ranking stages can be guaranteed. As a result, the weakness of dynamic pruning methods hinders their application on commercial IR system. Alternatively, people uses conjunctive or "AND" query processing technique [21, 92, 95]. Comparing to disjunctive query processing which considers all the documents containing at least one query terms, conjunctive query processing evaluates only documents which contain all the query terms. Conjunctive query processing does not suffer from efficiency loss when k becomes large. However it hurts effectiveness. Not all the relevant documents will contain all the query terms. In reality, some documents may use different terms from the query to describe a same concept. Conjunctive query processing will certainly miss such documents and its results may not be as good as those from disjunctive query processing.

In another word, when the number of retrieval results is large, either conjunctive mode or disjunctive mode can produce both effective and efficient query processing results. To solve this problem, we present a novel and scalable query processing method based on document prioritization. The basic idea is to reduce the number of documents that need to be fully evaluated by prioritizing documents based on the number of matched query terms as well as their importance. To implement this idea, we build a decision tree to quickly classify documents into different buckets based on the matched query terms, and these buckets, together with the documents in the buckets, are then ranked based on the importance of the matched query terms. On one hand, the proposed method includes more documents in the evaluation process than the conjunctive mode, which leads to more effective results. On the other hand, it use a simple strategy to prioritize all the documents matching at least one query term so that the number of documents that need to be fully evaluated is smaller than the disjunctive mode, which means more efficient query processing.

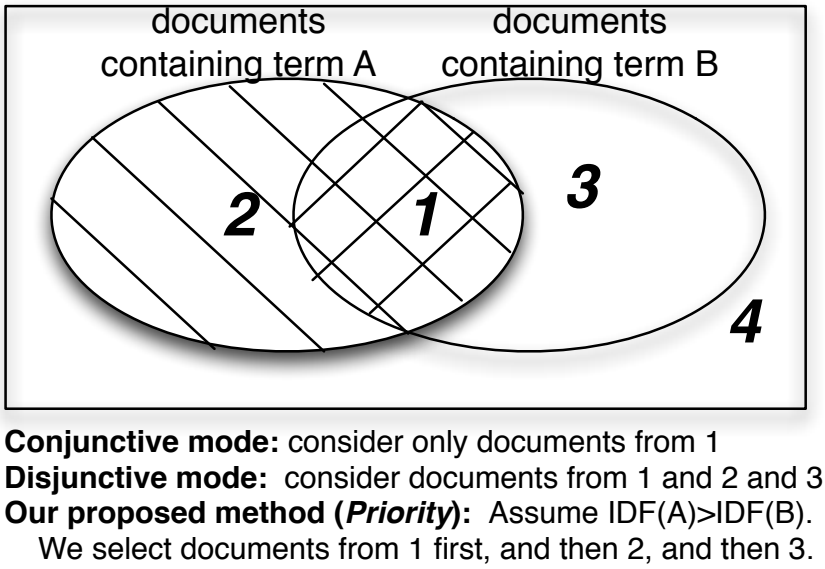


Figure 5.1: Difference between conjunctive mode, disjunctive mode and the proposed method.

5.2 Document Prioritization

5.2.1 Basic idea

Query processing time is closely related to the number of documents that need to be fully evaluated for a given retrieval function, such as Okapi BM25 [82]. The *disjunctive* mode without pruning evaluates all documents matching at least one query term. All these documents are treated *equally*, and their relevance scores with respect to the retrieval function are then fully computed.

Intuitively, not every document passing the Boolean OR filter has the same likelihood of being relevant. The basic idea of document prioritization is to prioritize documents based on an approximation of the relevance likelihood and then compute the accurate relevance scores for the top-K highly prioritized documents instead of accurately computing the relevance scores for all of the documents. Clearly, the key challenge is how to *efficiently* and *effectively* prioritize the documents.

Dynamic pruning methods in the disjunctive mode can be regarded as one specific method of document prioritization. They try to prioritize documents based on

their fully or partially computed relevance score for the given retrieval function. They can produce the exact top-K results, but are not very efficient due to the fact that the relevance scores are costly to compute. Conjunctive mode can be regarded as another specific method of document prioritization. It essentially splits all documents passing the Boolean OR filter into two categories, i.e., whether they contain all query terms or not, and focuses on only on the documents passing the Boolean AND filter. This method is efficient but not very effective.

To overcome the limitations of existing methods, we propose a new way of document prioritization with the goal of using a simple yet effective retrieval function to *approximate* the results generated by the given retrieval function. It is well known that the retrieval effectiveness is related to the use of multiple retrieval signals such as TF (term frequency) and IDF (inverse document frequency) [53]. Existing retrieval functions such as Okapi BM25 often combine these retrieval signals in a complicated way, which requires the traverse of inverted lists of all query terms and the combination of statistics obtained from the lists. To simplify the cost of such evaluation cost, we proposed the a new ranking strategy called "tie-breaking". The main idea of "tie-breaking" is to rank documents first using a single strong signal such as IDF and then use other signals or ranking methods to break the ties. (i.e., re-rank the documents with the same scores computed using the previous signals) In our previous study, we found that "tie-breaking" can lead to comparable or even more effective results.

Therefore, based on the similar idea, we propose to prioritize documents based on the sum of the IDF values of all distinct query terms that occur in the documents. Formally, the priority score of document D for query Q is computed as follows:

$$S_P(Q, D) = \sum_{t \in Q \cap D} IDF(t), \tag{5.1}$$

where t is a query term. $IDF(t)$ is the inverse document frequency of t and can be computed as $log\frac{N+1}{df(t)}$, where N is the number of documents in the collection and $df(t)$ is the number of documents containing term t. It is clear that the documents are prioritized based on the number of *distinct* query terms that they contain and the

importance of the terms. Figure 5.1 shows an example scenario with a two-term query. If the IDF value of A is larger, documents from the area 1 (i.e., those with both terms) would have higher priority than those from area 2 (i.e., documents with only term A), which have higher priority than those from area 3 (i.e., documents with only term B).

This function is chosen to strike a better balance between the efficiency and effectiveness. First, IDF has been shown to be the stronger retrieval signal than others such as TF, since it has higher upper bound performance in our previous study of tie-breaking. Second, the number of distinct values of the priority scores is small, which makes it possible to efficiently process documents using the tree-structure that will be described in the following subsection.

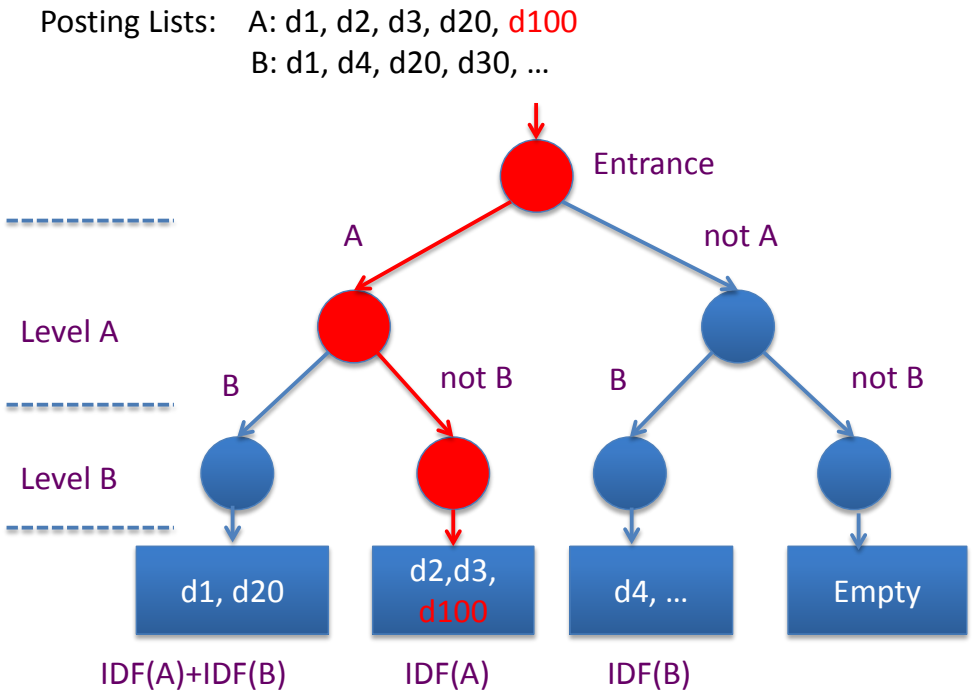
After the document are prioritized based Equation (5.1), we can then take at least K documents with higher priorities and re-rank them with more accurate relevance scores computed based on the given retrieval function. We will explain how to efficiently implement this idea in subsection 5.2.2 and discuss its efficiency, effectiveness and space usage in subsection 5.2.3.

5.2.2 Tree-based prioritization

We have explained the basic idea of document prioritization for query processing in the previous subsection. We now explain how to implement it efficiently based on a tree-based structure.

Recall that the basic idea is to prioritize documents based on the number of distinct query terms that they contain and the importance of the matched terms. Thus, if a query contains N terms, the priority scores computed from Equation (5.1) would have only 2^N different values. Since a collection may contain millions or even billions of documents, it means that many documents would have the same priority scores.

This observation motivates us to use a decision-tree based data structure that can quickly classify documents into different buckets based on the matched query terms



1

Figure 5.2: An example of the decision tree based query processing

and then prioritize the document buckets (i.e., documents with the same priority score) accordingly.

Constructing a decision tree: Given a query, the decision tree can be constructed as follows. All query terms are first ranked based on their IDF values, and their ranks determine which levels of the decision tree they will correspond to. We put terms with higher IDFs at the higher levels because this enables the fast pruning of nodes corresponding to the non-essential terms, which will be discussed soon. Each non-leaf node will have two children. One child includes the documents containing the corresponding term, while the other child includes the documents that do not contain the corresponding term. Each node is associated with a priority score, which is determined by the priority score of the documents belonging to the node. Each leaf node contains a document bucket, in which all the documents have the same priority scores as computed using Equation (5.1). Figure 5.2 shows an example decision tree for a query with two terms A and B. Since $IDF(A)$ is larger than $IDF(B)$, the nodes at the first level corresponds to A and those at the second level correspond to B. Node "A" in the first level has a priority score of $IDF(A)$, and "not A" has a priority score of 0. Moreover, the first leaf node has a priority score of $IDF(A) + IDF(B)$ since this node corresponds to the documents matching both terms.

Tree-based query processing: After building a decision tree for a query, we traverse the indices in a similar way as DAAT. The inverted lists of all query terms are processed in parallel, and the documents are processed in the order based on their document IDs. For each document, instead of directly computing its priority score, we leverage the decision tree and put the document into the corresponding bucket. The priority of a document is the same as that of its corresponding bucket, which is computed when building the tree. The last bucket (the one with the lowest priority score) will always be empty since it corresponds to the documents that do not match any query terms and these documents would not occur in the inverted lists of query terms. As shown in Figure 5.2, when processing $d1$, we would put it to the first bucket based on the constructed decision tree. After processing all the documents, we can

then return all the documents in the first M buckets so that they are the smallest set of buckets that can cover K documents. For example, if $K = 3$, we would return the first two buckets as shown in the Figure 5.2. When putting a document into a bucket, the term statistic information related to the document will also be stored in the decision tree. Thus, after identifying the top M buckets, we will then compute the relevance score of all the documents in the buckets based on the given retrieval function and return top K results.

Dynamic pruning: Since we focus on top-K query processing, it would be more efficient it we could skip some documents that will not make the final top K results. We propose a dynamic pruning strategy that can be combined with the above tree-based query processing. After placing a document into a bucket of the decision tree, we will check whether this bucket together with the buckets with higher priority are big enough to cover top K documents. If so, we will disable all the nodes with buckets that have lower priority. If all the children of a node are disabled, this node will also be disabled. If a disable node corresponds to the occurrence of a term, we can refer to this term as a *non-essential* term. All the documents that contain only non-essential terms will not make the final top K results. Thus, for the inverted list of non-essential term, we can then skip the documents that do not contain any essential terms by moving the current document pointers in the inverted lists of the non-essential terms to the smallest document IDs in the inverted lists of the essential terms, i.e., those corresponding nodes in the tree are not disabled. Note that this idea is similar to the MaxScore method [97] since we also try to exclude non-essential terms from the query processing.

Let us go back to the example shown in Figure 5.2. Assume we focus on retrieving top 2 results. According to DAAT, $d1$ is the first one to be processed, and it will be put into the first bucket since it contains both query terms. $d2$ and $d3$ are the next, and will be put into the second bucket. Since $K = 2$ and the first two buckets already cover 3 documents, we can now disable the last buckets and associated nodes, i.e., all the nodes from the sub-tree of "not A". It means that B is a non-essential term since

the documents containing only document B will not make to the final top K results. Based on our dynamic pruning method, the current document ID in the inverted list of B will be moved to $d100$ since it is the smallest document ID in the inverted lists of essential terms. After processing $d100$, we can put it into the first bucket and return only the documents from the first bucket. Finally, the two documents from the first bucket will be evaluated with the given retrieval function such as Okapi BM25. It is clear that the dynamic pruning makes it possible to skip many documents that can not make to the top K results, i.e., $d4$, $d6$, ..etc.

5.2.3 Discussions

Efficiency: The efficiency of any query processing method is closely related to the number of documents that need to be fully evaluated with the given retrieval function. The main advantage of our approach is to first prioritize the documents using an efficient tree-based query processing technique to reduce the number of documents that need to be evaluated with the given retrieval function. In particular, the number of evaluated documents in the proposed approach is determined by the size of the first M buckets. When the size of the first M buckets is smaller than the number of documents that need to be fully evaluated by existing query processing methods, the proposed method is expected to be more efficient. Since the number of evaluated documents in any query processing method would grow with the value of K and the bucket size in the proposed decision tree would shrink when a query has more terms, we expect that the proposed method could improve the efficiency better when K is larger or the queries are longer.

Effectiveness: The proposed method is similar to the conjunctive mode since both of them try to use a simple method to select a small set of documents that need to be evaluated. However, our method is more effective since it can include more potentially relevant documents than in the conjunctive mode. Since IDF is a very strong retrieval signal, the number of missed relevant documents in our method would be small.

Space Usage: The proposed method requires a decision tree for query processing. Although the number of the nodes in the tree grows exponentially with the number of terms in a query, the entire tree would not take much memory since the information stored for each node is limited - including the corresponding term and the priority score. In addition to the nodes in the tree, we also need to keep the record of all the documents from the top M buckets. These buckets store basic information about the documents including document ID, document length and term statistics. Overall, the space usage of the proposed method is small (a few megabytes for GOV2 collection).

5.3 Experiments

5.3.1 Experimental setup

We compare the proposed method with a couple of the state of the art query processing techniques on two large-scale search domains, i.e., Web search and Microblog search. Due to the high cost associated with creating relevance judgments, there is no single collection that has a large number of queries with relevance judgments. Thus, for each search domain, we have to use different collections for evaluating the effectiveness and efficiency. All experiments are conducted via a single machine with dual AMD Lisbon Opteron 4122 2.2GHz processors and 16GB memory. We use the Okapi BM25 [82] as the retrieval function and retrieve top-K results. The query processing system is implemented similar to VIrtual IR Lab.

Data sets for Web search: We use the TREC Gov2 collection [2] as the data set. The collection contains 25.2 million web pages crawled from the .gov domain. To evaluate the retrieval effectiveness, we use two sets of 50 queries from the TREC 2005-2006 Terabyte tracks [37], which are denoted as *TB05* and *TB06*. To evaluate the efficiency, we randomly choose 1000 queries from the efficiency queries used in the same tracks. They are denoted as *TB05L* and *TB06L*.

Data sets for Microblog search: To evaluate the retrieval effectiveness, we use the TREC Tweets2011 collection as the data set. The collection contains 16 million

tweets posted from 01/23/2011 to 02/08/2011. Two query sets from TREC 2011 and 2012 Microblog tracks [79] are used and denoted as *MB11* and *MB12*. To evaluate the efficiency, we use a larger collection, i.e., a subset of the twitter7 [117] collection, as the data set. The data set contains 253 million tweets from 06/07/2009 to 10/21/2009. We crawled trending queries for each date in the corresponding period from Google Trends and randomly pick 1000 queries to use for our experiments. Although they are not queries submitted to Twitter, they can represents user information needs for the corresponding period. This collection is denoted as *MBL*.

Methods: The proposed tree-based document prioritization method with pruning describe in the previous section is denoted as *Priority*. Specially, to retrieval top-K results, the *Priority* method would first use the proposed prioritization method to identify top ranked blocks covering at least K documents, grade the documents from these blocks using the Okapi BM25 method, and the retrieval top-K documents as the final results. The *Priority* method will be compared with three baseline methods: (1) *BMW [50]:* the state of the art dynamic pruning method for disjunctive mode, which has been shown to be more efficient than the WAND [29] and Maxscore methods [97]; (2) *LB [49]*: an improved version of BMW using live blocks, which has been shown to achieve a speed-up of 2 over BMW for top-10 query processing; (3) *AND*: the conjunctive mode implemented using the WAND method [21, 29].

5.3.2 Performance comparison: efficiency

The first set of experiments is to compare the average query processing time per query for both domains. Figure 5.3 and figure 5.4 show the comparison results over different values of K (i.e. the number of retrieval documents per query) on the two web collections, i.e., TB05L and TB06L. Note that the performance of AND varies with different values of K due to the different cost of heap update, but the differences are too small to see in the figures.

There are a few interesting observations. First, the proposed *Priority* method clearly has a constant factor speed improvement over the state of the art dynamic

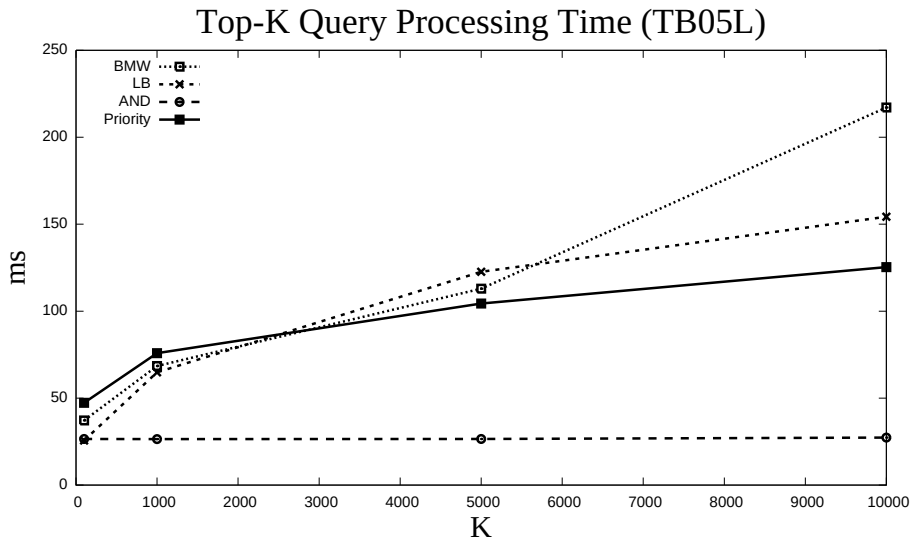


Figure 5.3: Efficiency comparison on *TB05L*: average processing time per query (ms)

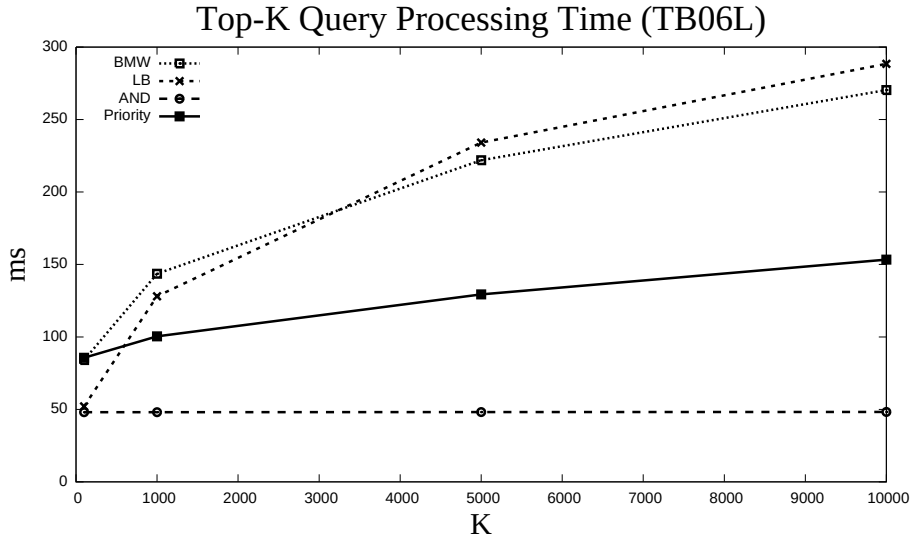


Figure 5.4: Efficiency comparison on *TB06L*: average processing time per query (ms)

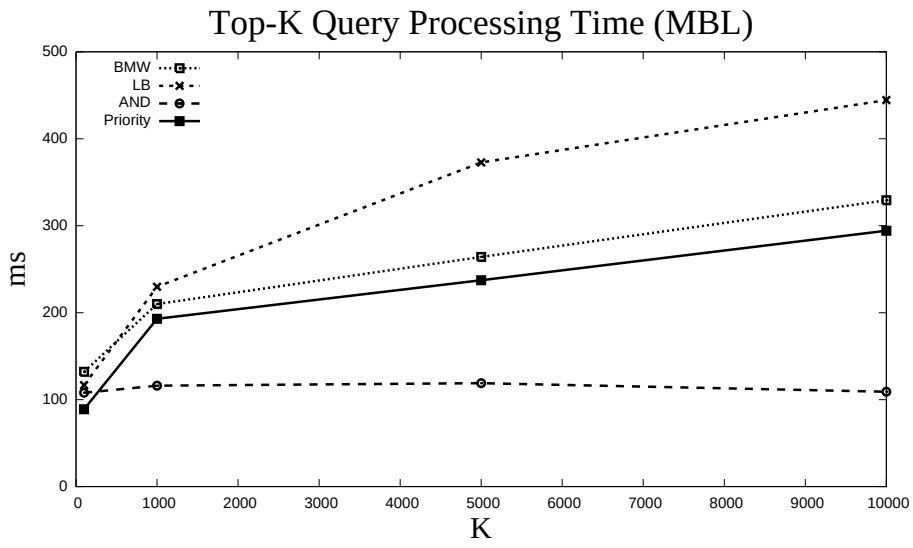


Figure 5.5: Efficiency comparison on *MBL*: average processing time per query (ms)

pruning methods (i.e., BMW and LB) on both collections. In particular, the speed up of the *Priority* method is around 2 when K is large. Second, the *Priority* method does not perform as well as *BMW* when the value of K is smaller. Finally, although *LB* is able to achieve a speed-up of 2 over *BMW* when K is smaller, which is consistent with the previous study [49], it does not always have a clear advantage over *BMW* for larger K.

Similar experiments is done for Microblog search. Figure 5.5 compares the average query processing time of the four methods. It is interesting to see that the *Priority* method is consistently more efficiency than *BMW* and *LB* methods for different retrieval cut-offs, although the speedup is not as large as we observe in the Web domain search.

5.3.3 Performance comparison: effectiveness

The proposed *Priority* method is not rank safe for top K results, since the priority function we used is a simple approximation of the original retrieval function. Thus, it would be interesting to compare it with the two baseline methods in terms of

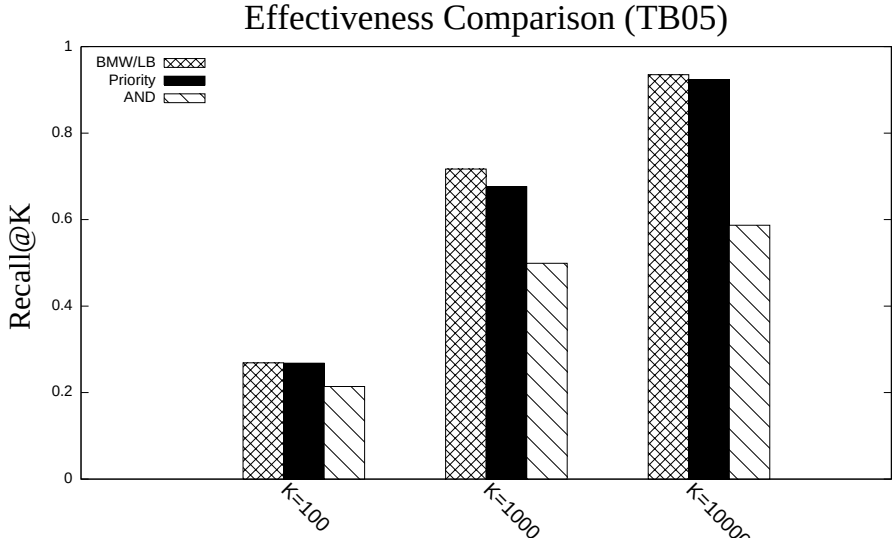


Figure 5.6: Effectiveness Comparison (TB05)

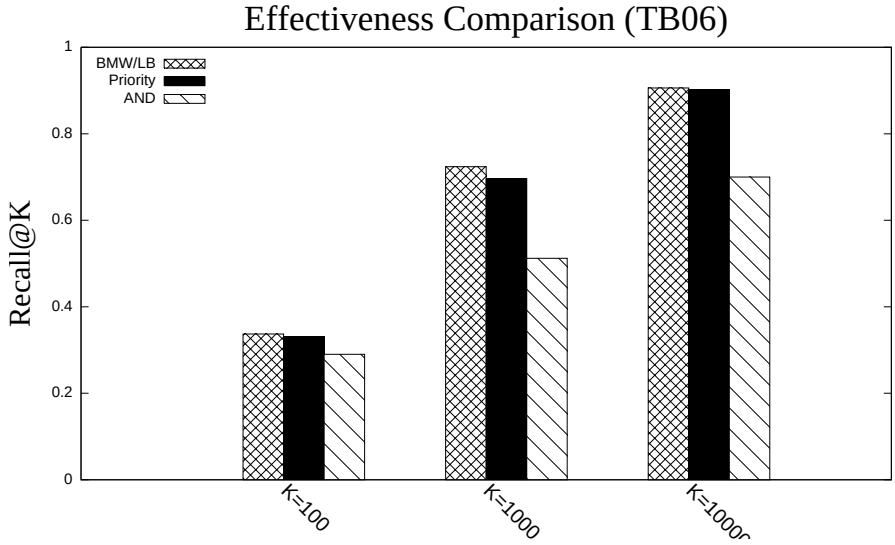


Figure 5.7: Effectiveness Comparison (TB06)

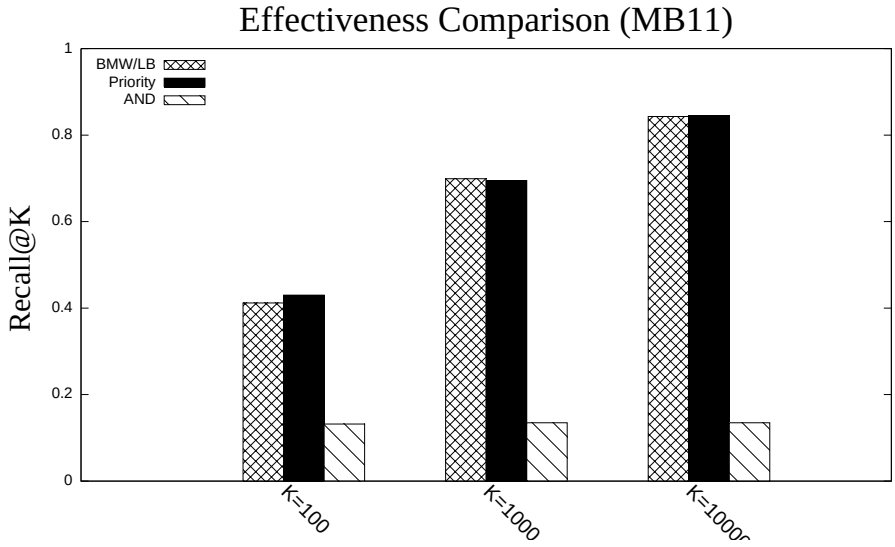


Figure 5.8: Effectiveness Comparison (MB11)

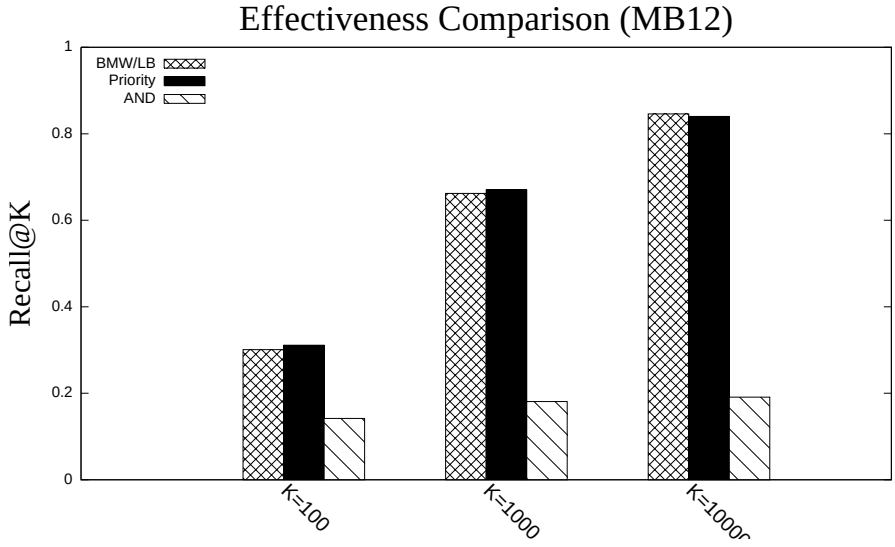


Figure 5.9: Effectiveness Comparison (MB12)

Table 5.1: Effectiveness Comparison (MAP@K)

K		100	1000	10000
TB05	BMW/LB	0.167	0.324	0.343
	AND	0.132	0.236	0.245
	Priority	0.165	0.306	0.322
TB06	BMW	0.188	0.285	0.293
	AND	0.163	0.247	0.252
	Priority	0.187	0.281	0.290
MB11	BMW/LB	0.206	0.245	0.247
	AND	0.077	0.079	0.079
	Priority	**0.221**	**0.265**	**0.268**
MB12	BMW	0.137	0.193	0.203
	AND	0.086	0.094	0.095
	Priority	**0.148**	**0.208**	**0.217**

the retrieval effectiveness. Note that *BMW* and *LB* have the same effectiveness since both of them are rank-safe. Figure 5.6, 5.7, 5.8 and 5.9 show the recall at different retrieval cut-offs on the four web search and microblog search collections. And the performance measures with MAP is also shown in table 5.1.

It is clear that the effectiveness of the *Priority* method is slightly worse than that of *BMW* and *LB* for Web search, but the effectiveness loss is much smaller than that caused by the *AND* method. And such an observation is consistent for different retrieval cut-offs, different effectiveness measures and different collections. Moreover, for the Microblog search, it is interesting to see that the *Priority* method is more effective than the other two methods. The observation suggests that IDF is a very strong retrieval signal for microblog search since the short of length of tweets makes other signals such as TF and document length normalization less effective.

5.3.4 Result analysis

As shown earlier, the proposed method can strike a better balance between the effectiveness and efficiency than the baseline methods by significantly reducing the

Table 5.2: Avg. processing time per query (ms) for different query length on *TB05L* (K=1000)

Query length	2	3	4	5	6	> 6
BMW	36.13	75.01	110.27	133.93	184.17	290.37
Priority	90.93	59.62	64.81	56.41	83.48	122.35
Speed-up	0.4	**1.3**	**1.7**	**2.4**	**2.2**	**2.37**

Table 5.3: Avg. processing time per query (ms) for different query length on *TB06L* (K=1000)

Query length	2	3	4	5	6	7	> 7
BMW	52.41	107.01	146.54	238.06	344.94	383.38	531.81
Priority	82.01	86.95	93.93	126.85	153.17	206.87	268.07
Speed-up	0.64	**1.2**	**1.6**	**1.9**	**2.3**	**1.9**	**2.0**

query processing time without sacrificing too much on the quality of search results. Here we conduct more analysis to better understand the performance differences.

5.3.4.1 Efficiency comparison for different query lengths

To further understand the performance behavior, we report the average query processing time for different query lengths. Table 5.2 shows the average query processing time when retrieval top 1,000 documents on *TB05L* collection. It is interesting to see that the *Priority* method is more efficient than *BMW* when the queries have more than two terms. Moreover, it can achieve a speed-up around 2 when the number of query term is larger than 4. We can make consistent observation on *TB06L* collection as well, as shown in table 5.3. Similar observations can be made on different K values, we show average query processing on different cut-offs in figure 5.10.

In summary, the proposed *Priority* method is more salable than the *BMW* method and it is more efficient for longer queries and larges values of K. This trend can be clearly seen in figure 5.10. Note the trend is similar on the other data set and for the other method (i.e. *LB*).

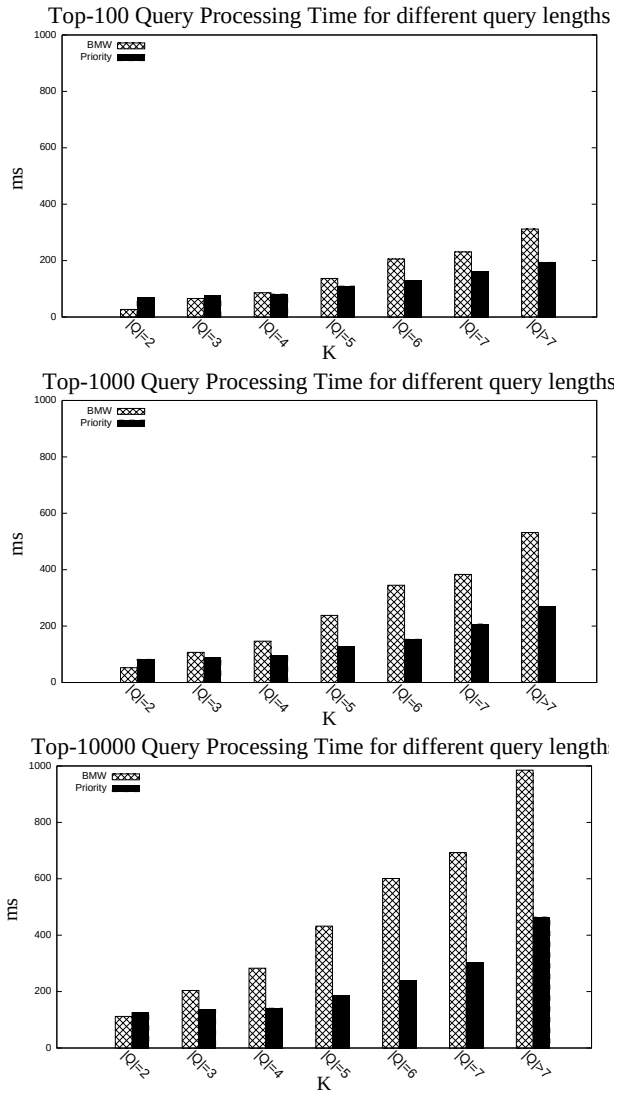


Figure 5.10: Performance comparison on *TB06*: average query processing time per query (ms)

5.3.4.2 Efficiency: the number of evaluated documents

The query processing time is closely related to the number of documents that need to be fully evaluated, the performance gain of *Priority* probably comes from its ability to quickly prune many documents with lower priorities. To verify the hypothesis, we report the number of evaluated documents for *TB06L* as shown in table 5.4. It is clear that the query processing time is indeed related to the number of evaluated documents. When more documents need to be fully evaluated, it would take more time for query processing. It is clear that the number of evaluated documents of *Priority* is much smaller than that of *BMW* when K is larger.

We now discuss how the number of evaluated documents in the *Priority* can be affected by K. The number of evaluated documents is determined by the sizes of all the documents blocks with the highest priority from the decision tree. The size of these blocks varies for different collections. For example, the size of the first block is the number of documents that containing all the query terms in the collection (they are also the very documents will be evaluated with the conjunctive mod). When K is larger than the size of the first block, the number of evaluated documents in *Priority* would often be smaller than those in *BMW* since the documents are prioritized using a simple priority function. However, when K is much smaller, the number of documents from the first block might be already much bigger than the value of K, which means that we need to unnecessarily evaluate many more documents than *BMW*.

When a query has more terms, the size of the blocks with higher priorities is smaller since these blocks require documents with more terms. As a result, these smaller blocks may lead to fewer documents that need to be evaluated by the *Priority* method. Table 5.5 shows the number of evaluated documents for different query lengths, the results are consist with what we observed in table 5.3. Clearly, the results are consist with our analysis, which provides a justification on why our method is more efficient for longer queries.

Table 5.4: Avg num. of evaluated docs on *TB06L*

K	100	1000	5000	10000
BMW	**35,104**	105,277	246,089	357,357
Priority	66,409	**83,506**	**112,996**	**140,688**

Table 5.5: Avg num. of evaluated docs on *TB06L* for different query length (K=1000)

Query length	2	3	4	5	6	7	> 7
BMW	**111,947**	109,859	93,399	107,984	113,466	96,659	100,398
Priority	229,752	**73,535**	**25,242**	**13,967**	**5,270**	**5,822**	**3,547**

5.3.4.3 Impact of the pruning strategy

Finally, we conduct experiments to examine how much the proposed pruning method in the previous section can improve the efficiency. The results are shown in Table 5.6. The *Exhaustive* method refers to the query processing method that exhaustively computes the relevant score for every document which contains at least one query term. It is clear that the prioritization method can improve the query processing significantly, and the proposed pruning method is able to further reduce the query processing time by a factor of at least 2. We also study how the percentage of pruned blocks changes as the query processing continues. The results are shown in Figure 5.11, where y-axis is the percentage of the non-pruned blocks and x-axis is the percentage of the posting lists that have been processed. We can see that the percentage of non-pruned blocks keeps decreasing in the whole process, and around eighty percent blocks are pruned at the end.

Table 5.6: Avg. query processing time (ms) on *TB06L*

K	100	1000	5000	10000
Priority	**85.61**	**100.40**	**129.34**	**153.36**
Priority + No pruning	275.69	284.90	301.54	318.97
Exhaustive	1135.1	1141	1143.3	1154.4

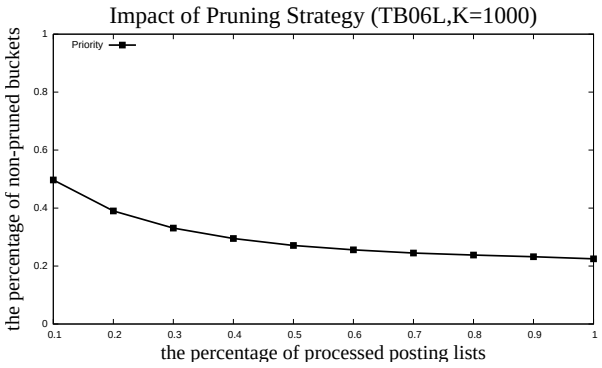


Figure 5.11: Impact of the pruning strategy: the percentage of non-pruned blocks

5.4 Conclusions

Although dynamic pruning techniques can help to reduce query processing time, when the systems need to return more documents for each query, the efficiency improvement of dynamic pruning becomes less significant. On the other hand, conjunctive or AND query processing does not suffer from the problem but it may hurt effectiveness very much.

To solve this problem, we propose a novel tree-based document prioritization method for query processing. The basic idea is to first prioritize documents using a simple yet efficient method and then pick only a small set of documents with higher priority for full evaluation. We propose to implement this idea using a tree-based structure. Experimental results show that the proposed method is more scalable than one of the state of the art dynamic pruning method for disjunctive mode. In particular, when the number of retrieved documents is larger than 1000 or the number of query terms is larger than 2, the proposed method can achieve a speed up around 2 over TREC Web collections with marginal loss in terms of the effectiveness. We also find that it can improve the retrieval efficiency for Twitter collection with close-to-zero effectiveness loss. Overall, the proposed method has been shown to achieve a better trade-off between the efficiency and effectiveness.

We have got a better solution for query processing when the number of returned

documents is large. So how about long query processing?

Chapter 6

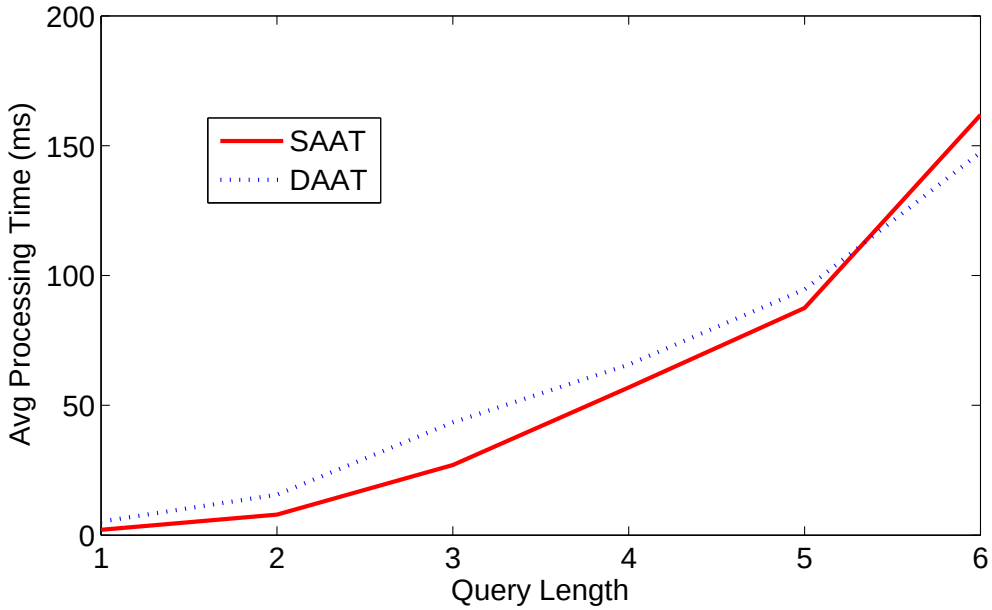
IMPROVE EFFICIENCY OF PSEUDO RELEVANCE FEEDBACK

6.1 Introduction

Query processing time may become longer as the query becomes longer (i.e. containing more unique query terms). It is due to two reasons: on one hand, more query terms may introduce more postings to be evaluated. On the other hand, longer query also increases the maximum score a document can get and as a result dynamic pruning methods may become less effective and the IR system needs to evaluate more documents as well. Figure 6.1 shows how the average query processing time change at different query length for both DAAT and SAAT query processing. For DAAT we apply WAND dynamic pruning strategy [29] while for SAAT we apply the four-stage pruning method [18]. When query is very short (i.e. query length ≤ 2), both DAAT and SAAT can perform very well (i.e. less than 20 ms). However as the query becomes longer, it is obvious that average query processing time increases exponentially. This trend is even more obvious for SAAT query processing. The low efficiency of long query processing affects the utilization of some useful techniques such as pseudo-relevance feedback.

Pseudo-relevance feedback [9, 55, 86, 93, 100, 101, 109] is an important technique that can be used to improve the effectiveness of IR systems. There exists vocabulary gaps between queries and relevant documents. It is may due to the fact that users cannot speak out all the important aspects of their information needs, or simply those relevant documents may use different terms to describe the same things. The vocabulary gaps keep some relevant documents from being retrieval to top lists and as a result it affects effectiveness of information retrieval systems. Pseudo-relevant feedback is one of the most important technique to improve effectiveness by crossing vocabulary gaps.

Figure 6.1: Query processing time on different query length



The main idea of Pseudo-relevant feedback is trying to expand original queries with useful terms to help the system better seeking for relevant documents. Particularly, these relevant documents are selected from top-ranked documents from an initial retrieval run and it assumes that the these top ranked documents are likely to be relevant and their content can help the system to understand users' information needs better.

Different pseudo-relevance feedback methods have been proposed and studied for various retrieval models [33, 64, 81, 83, 85, 113, 120], but they all boil down to a two round retrieval process. In the first round, the original query is used to retrieval an initial set of documents. And the several most representative terms are selected to expand the original query and a second round retrieval is processed with respect to the new expanded query.

Although pseudo-relevance feedback methods can lead to large performance gain in terms of effectiveness, there is one major downside that limits their usability in real retrieval application: its low efficiency [25, 42, 63]. In particular, the second round of retrieval could significantly slow down the performance due to the time spent on

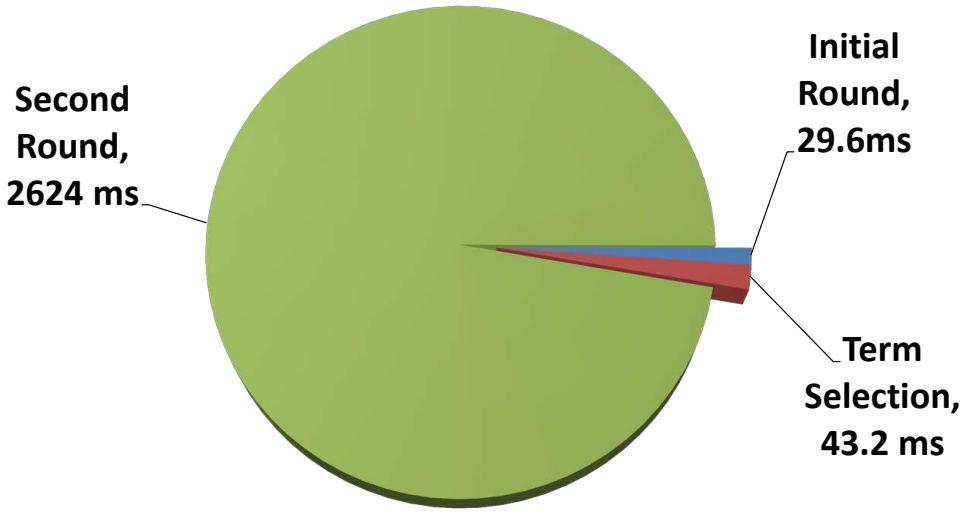
selecting terms for expansion and on executing the expanded query that are often much longer than the original one [42, 63]. It is clear that such drastically increased execution cost limits the applicability of pseudo-relevance feedback methods in real IR applications.

Compared with continual efforts on improving the effectiveness and robustness of pseudo feedback methods in the past decades [33, 34, 39, 46, 48, 64, 65, 67, 67, 68, 75, 81, 83, 85, 94, 113, 114, 120], less attention has been paid to make these methods more efficient [25, 42, 63]. Billerbeck and Zobel [25] proposed a summary based method for efficient term selection from the feedback methods. Lavrenko and Allan [63] proposed a fast relevance model, and Cartright et. al. [42] studied approximation methods to reduce the storage and computational cost for the fast relevance model. These proposed methods were designed for either a specific feedback method [42, 63] or a bottleneck in the feedback implementation, i.e., how to efficiently select expansion terms from the feedback documents [25]. However, it remains unclear whether there is a general solution to address other bottlenecks for efficient pseudo-relevance feedback implementation.

This problem may become more serious for Score-at-a-time SAAT query processing [13, 14, 17, 18] which uses the accumulated list to store the candidate documents and their scores. As the query becomes longer, more postings will be accessed and more more candidate documents may need to be stored in the accumulated list. Therefore the query processing time as well as memory usage may grow exponentially as the query becomes longer. The feature obviously does not fit to pseudo-relevance feedback since the expanded query is usually very long. The shortage actually limits the application of PRF on SAAT query processing.

To overcome this problem, in the following section, we propose a general solution that can improve the efficiency of pseudo-relevance feedback in SAAT query processing. The basic idea is to reduce the executing cost for the expaned query by using the query processing status of the the original query. Existing IR systems often implement the pseudo-feedback methods as a two-round retrieval, where the second round is processed

Figure 6.2: Processing time of the three steps in pseudo-relevance feedback implementation



independently to the first round. However, since the query used in the second round retrieval is an expanded version of the original query used in the first round, it would be beneficial to leverage the results of the first round of retrieval to reduce the query processing time for the second round.

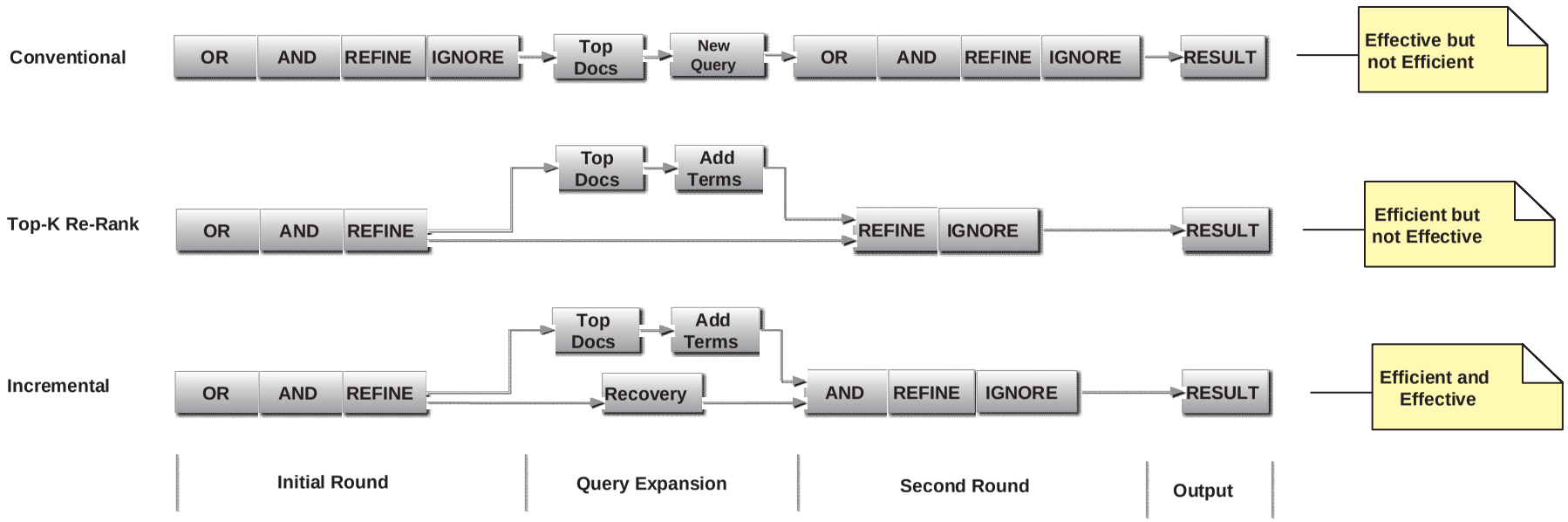
6.2 Efficient Pseudo-relevance Feedback for SAAT

6.2.1 Overview of existing implementation strategy

As described in previous section, many pseudo-relevance feedback methods have been proposed and studied. Despite the differences on how to exploit feedback information, they all require the following three-step implementation:

1. **Initial retrieval:** finding documents relevant to an original query;
2. **Term selection:** identifying useful expansion terms (i.e., relevant information in language modeling framework) from the feedback documents;
3. **Second-round retrieval:** returning documents relevant to the expanded query (i.e., updated query model in the language modeling framework).

Figure 6.3: Computational flows for different implementations of pseudo-relevance feedback methods



We'd like to show the details of implementations of the three steps as following:

The *initial retrieval* step can be implemented with any existing top-k query processing techniques as described in the previous chapter. There have been significant efforts on optimizing the efficiency for this step [13, 14, 17, 18, 29, 35, 52, 76, 89, 90, 124].

The *term selection* step is to select important terms from the feedback documents with the expectation that the selected terms can bring more relevant documents in the second round of retrieval. Traditionally, these terms are selected directly from the top ranked documents. However, this step could takes lots of time when the documents are long and the information about the documents need to be read from the disk. To solve this problem, Billerbeck and Zobel [25] proposed to generate a short tf-idf based summary for each document and select expansion terms from the summaries of the feedback documents. These summaries are small enough to be pre-loaded into the memory, and can lead to more efficient term selection. Their experimental results showed that this strategy is efficient with ignorable loss in terms of the effectiveness. Following their study, we use a similar strategy for term selection. One difference is that we use term probability $p(t|D)$ instead of tf-idf weighting to generate document summaries because the retrieval function used is based on language modeling approach. And we set the length of document summary to 20 terms.

The *second round retrieval* step aims to retrieve final retrieval results with the expanded query. The expanded query is often formulated as a linear interpolation of the original query and the expansion terms selected from the second step. As an example, in the relevance model [64], this step is to retrieve documents with an updated query model (i.e., θ_Q^{new}) by linearly combining the original query model (i.e., θ_Q) with the relevance model estimated from the feedback documents (i.e., $\theta_{\mathcal{F}}$) as follows:

$$\theta_Q^{new} = \lambda \cdot \theta_Q + (1 - \lambda) \cdot \theta_{\mathcal{F}}, \tag{6.1}$$

where the original query model θ_Q is estimated using the maximum likelihood estimation of query Q, the relevance model $\theta_{\mathcal{F}}$ estimated from the feedback documents $\mathcal{F}$

using the methods described in previous study [64], and λ is to control the amount of feedback.

In the second round of retrieval, existing IR systems such as Indri would process the expanded query in the same way as a newly submitted query. In other words, the two rounds of retrieval are processed *independently*.

Figure 6.2 shows how much time each step takes when using the existing implementation methods described above for the relevance feedback method with 20 expansion terms. It is clear that the third step takes the most of computational time while the other two stages share a very small part the time usage. Thus, the key to efficient pseudo-relevance feedback methods is to reduce the execution time for the *second-round retrieval*, which is the focus of this study.

6.2.2 Analyzing the second round retrieval

Compared with the initial retrieval, the expanded query processed in the second round is often much longer than the original query because it has much larger number of query terms to be processed. For example, the average length of Web queries is around 3, while the number of expansion terms is often set to 20. As a result, the computational cost for expanded query is significantly higher than that for the basic retrieval.

One limitation of existing implementation for feedback methods is that the two rounds of retrieval are processed *independently*. Each round starts with an empty set of accumulators and gradually adds new accumulators in the OR stage. When switched to AND stage, accumulators can be updated but no new accumulated can be added. When switched to REFINE stage, only top k accumulators are processes. And in the final IGNORE stage, all the information from the inverted lists can be ignored. Note that the accumulators used the two rounds of retrieval are computed from the scratch separately. This implementation is illustrated in the upper part of Figure 6.3

However, unlike processing a new query, the expanded query in the second round retrieval is related to the query used in the initial retrieval. In particular, the query

terms in the initial retrieval is a subset of those in the second round of retrieval, and these terms will be processed twice in this two rounds of retrieval process. Moreover, the results of these two rounds of retrieval might have a great overlap. Thus, it would be interesting to study how to leverage the results of the first round of retrieval to reduce the computational cost. But how to leverage them? This is not a simple problem without significant challenges.

The idea of exploiting the results of initial retrieval to improve the efficiency of the second round retrieval was discussed in previous study [25], but was not found useful. In the next subsection, we re-visit this basic idea and propose an incremental approach that is shown to be both efficient and effective based on the experimental results.

6.2.3 The proposed incremental approach

The basic idea to improve efficiency of pseudo-relevant relevant feedback is that the initial retrieval process should be treated as part of the query processing for the second round retrieval. Instead of processing the expanded query in the second round from the scratch, we should be able to resume the query processing results of the initial query, and continue the processing for the expanded query terms. But how to resume the results?

One possible strategy is to resume the last ranking-related stage, i.e., REFINE, in the query processing results of the initial retrieval. Recall that REFINE stage is designed to process only the accumulators of top K ranked documents. Thus, if we resume the status from the REFINE stage in the second round retrieval, it is equivalent to re-ranking those top K ranked documents using the expanded query. This strategy is illustrated in the middle part of Figure 6.3. Since the number of accumulators used in REFINE stage is very small, this re-ranking method would be quite efficient. However, it would suffer significant loss in terms of the effectiveness because one of the major benefits of feedback methods is to find relevant documents that were not among top

ranked results for the initial retrieval and the re-ranking strategy seems to disable this nice benefit. Clearly, this is not an optimal solution. Can we do better?

Intuitively, if more document accumulators can be included in re-ranking process, the retrieval effectiveness could be improved. On the extreme case, when all the documents are considered for the re-ranking, the cost would be the same as submitting a new query. Thus, the main challenge here is how to select a set of documents to be re-ranked so that we can increase the efficiency without sacrificing the effectiveness.

Recall that the pruning technique consists of four stages: OR, AND, REFINE and IGNORE. The number of active accumulators becomes smaller as the system switches from one stage to the other. As discussed earlier, resuming from the REFINE stage is equivalent to re-ranking only top k documents, which hurts the effectiveness. On the contrary, resuming from the OR stage would not hurt the effectiveness. However, since the number of expanded terms is much larger than the number of original query terms, we might not be able to reduce the number of accumulators significantly. Thus, we propose to resume from the AND stage.

The main idea of our incremental approach is shown in the lower part in Figure 6.3. REFINE is the last ranking-related stage in the initial retrieval. Thus, in order to resume from the AND stage of the initial retrieval, we have to first rewind the query processing results from the end of REFINE stage back to the end of the AND stage. This new stage is referred to as RECOVERY stage in our system. The RECOVERY stage has two tasks: (1) re-enable the accumulators that were disabled at the REFINE stage; and (2) turn back the inverted list pointers to the positions where they were at the end of AND mode. Our experimental results show that this stage takes a very short ignorable time. After the RECOVERY stage, we will switch to the new AND stage to update the accumulators based on the expanded terms, and then continue to the other two stages as usual.

6.2.4 Discussions

The proposed incremental approach can improve the efficiency because of the following two reasons.

First, query processing results of the original query terms can provide useful information for effective pruning in the second round retrieval. Specifically, accumulator trimming is used in the AND stage to dynamically reduce the number of active accumulators, and a threshold is used to decide whether an accumulator should be kept active or not. The threshold is to estimate a lower bound of the relevance scores for the top K ranked documents. This threshold is set to $-inf$ at the beginning of the retrieval process, and will be updated at the later stage of the retrieval when more information is gathered. Since initial value of the threshold is rather small, little pruning is applied at the early stage of the retrieval process. If the system could be informed with the range of the threshold, more pruning can be done, which would lead to shorter processing time. Since resuming the process of the initial retrieval can provide a much larger initial value for the pruning threshold, the proposed approach can reduce the query processing time.

Second, the efficiency is closely related to the number of accumulators that need to be processed. Long queries usually lead to a huge number of accumulators, which significantly hurt the efficiency. However, when resuming the query processing results of the initial query, we are able to start with a much smaller set of accumulators. Note that this strategy is not ranking safe when the expanded query is significantly different from the initial query (i.e., when λ in Equation 6.1 is very small). However, as shown in next section, the optimal value of λ is often large and this strategy does not affect the effectiveness significantly.

6.3 Experiments

6.3.1 Experiment design

In this study, we use tree standard TREC data sets which were created for TREC Terabyte tracks in 2004 to 2006. These three sets are denoted as TB04, TB05

and TB06 in this section. All the data sets use Gov2 collection as the document set, and the collection consists of 25.2 million web pages crawled from the .gov domain. In additional to that, we also use another data set which were created for TREC web tracks from 2009 to 2012 which contains 200 queries. We denote it as CW09-12. Different from the previous TB data sets, CW09-12 is tested on Clueweb category B collection which contains 50 millions English web pages. The same for other part of the thesis, all experiments were conducted on a single machine with dual AMD Lisbon Opteron 4122 2.2GHz processors, 32GB DDR3-1333 memory and four 2TB SATA2 disks.

The basic retrieval model used in our experiments is the Dirichlet Prior smoothing method [119], where the parameter μ was set empirically to 2500. This method is labeled as **NoFB**. We use the relevance model [64] as the pseudo-relevance feedback method. This model is chosen because of the following two reasons. First, it is a state-of-the-art pseudo-relevance feedback method that has been shown to be both effective and robust. This method is labeled as **FB**.

The implementation of our basic retrieval system (i.e., **NoFB**) is described in previous section. Based on the basic retrieval system, we implemented two pseudo-relevance feedback systems: (1) **FB-BL:** traditional method of implementing pseudo-relevance feedback methods, i.e., the expanded query is processed independently to the original query; (2) **FB-Incremental:** our proposed approach that exploits the query processing results of the original query to speed up the processing of the expanded query.

6.3.2 Experimental results

The first two sets of experiments were conducted with the ad hoc queries used in TREC Terabyte tracks [2]. We use title only field to formulate queries. The third set of experiments was conducted with the efficiency queries used in TREC Terabyte tracks. One data set has 50K queries while the other has 100K queries.

Table 6.1: Efficiency comparison using TREC Terabyte ad hoc queries (i.e., average query processing time (ms) to retrieve 1K documents)

Data sets	NoFB	FB-BL	FB-Incremental
TB04	105	5,522	3,023
TB05	77	4,502	2,462
TB06	71	4,542	2,082
CW09-12	103	6,823	2,123

6.3.2.1 Efficiency and effectiveness

Firstly, let us explore whether the new methods can improve the efficiency. In particular, we set the number of feedback documents to 10, the number of expansion terms to 20 and the expansion weight λ is set to 0.6. These parameters are chosen because they can lead to near-optimal performance in terms of the effectiveness. Table 6.1 shows the average processing time (ms) for each query when 1000 documents are retrieved. The trends on the three collections are similar. Both FB systems are slower than the NoFB system. For three terabyte query sets, it takes around 5 seconds for FB-BL to process a query, and takes around 2.5 seconds for FB-Incremental to do so. Clearly, FB-Incremental can achieve a speed up of 2 compared with FB-BL. For Clueweb data set, FB-BL takes average 6.8 seconds for a query while FB-Incremental takes average 2.1 seconds which is more than 3 times faster than the baseline. All the results show that the proposed incremental approach can improve the efficiency. Note that we also evaluate the efficiency of the Indri toolkit, a widely used open source IR system. It takes around 1 minute for the Indri toolkit to process a query on the same collection. This suggests that the baseline system we implemented, i.e., FB-BL, is very efficient.

As discussed earlier, the speed up of FB-Incremental is achieved for two reasons. First, FB-Incremental leverages the query processing results of the initial retrieval, which avoids processing the original query terms twice. Second, the accumulator list inherited from the initial retrieval by FB-Incremental is much smaller. Table 6.2 shows the comparison of the accumulator list size between the two methods. Our incremental

Table 6.2: Comparison of the average size of accumulator lists per query

	FB-BL	FB-Incremental
TB04	2,368K	370K
TB05	1,805K	252K
TB06	1,901K	240K
CW09-12	3,857K	364K

Table 6.3: Effectiveness comparison using TREC Terabyte ad hoc queries (MAP@1000)

Data sets	NoFB	FB-BL	FB-Incremental
TB04	0.246	0.256	0.255
TB05	0.309	0.334	0.334
TB06	0.275	0.285	0.284
CW09-12	0.161	0.157	0.158

method creates only about 1/7 (1/10 for clueweb collection) accumulators as those of baseline method. As a result, the new method avoids a lot of unnecessary index access and computing for those accumulators it does not include. Note that our baseline system, i.e., FB-BL, is a very strong baseline since it applied the accumulator trimming techniques which can reduce unnecessary accumulators during the process.

We also conduct experiments to examine whether the proposed approach would hurt the performance. Since our incremental approach leverages the accumulators used in the AND stage of the initial retrieval, it is possible that we may miss some relevant documents because they were not relevant to the original query and thus were not included in its accumulator lists. As a result, it is possible that the incremental approach might hurt the effectiveness, but it only happens when the expanded query is very different from the original query. First we look at the results of three terabyte collections. Recall that the optimal value of λ is 0.6 which indeed indicates that the original query and expanded query are similar. Table 6.3 shows the results measured with MAP@1000. To ensure the correct implementation of our system, we compare the baseline performance with the ones reported in the previous study [89] and find that

Table 6.4: Impact of the expansion weight (i.e., λ) on the efficiency (the average execution time (ms) per query)

Data sets	Methods	λ				
		0.5	0.6	0.7	0.8	0.9
TB04	FB-BL	6,125	5,522	4,435	2,723	1,597
	FB-Incremental	3,798	3,023	2,130	1,221	826
TB05	FB-BL	5,146	4,502	3,577	2,466	1,323
	FB-Incremental	3,071	2,462	1,762	1,113	845
TB06	FB-BL	5,281	4,542	3,571	2,412	1,114
	FB-Incremental	2,664	2,082	1,406	778	560
CW09-12	FB-BL	8,579	6,823	5,229	3,844	2,202
	FB-Incremental	2,579	2,136	1,657	1,166	974

the results are similar, which confirms our implementation of $NoFB$ is correct. Moreover, we find that both FB methods can improve the retrieval accuracy about 4% to 8%, which is also similar to the performance improvement of the feedback method implemented in Indri. One interesting observation is that the effectiveness of the two FB systems are similar, which indicates that our proposed approach improve the efficiency without sacrificing the effectiveness. The same conclusion can be got from the web track collection. The only difference is that pseudo-relevance feedback cannot improve retrieval accuracy over the original retrieval. We think it is due to the characteristics of the queries. In spite of this, the trend is same that the two FB systems can output results with almost the same accuracy.

6.3.2.2 Impact of parameter values

There are multiple parameters in the pseudo-relevance feedback methods such as the number of expansion terms, the number of feedback documents and the expansion weight (i.e., λ in Equation (6.1)). Since we use impact based document summary for term select, the impact of the number of feedback documents on efficiency is not very significant. Thus, we only focus on the other two parameters.

As discussed earlier, the expansion weight is an important parameter and we should examine the impact of its value on both efficiency and effectiveness. The results

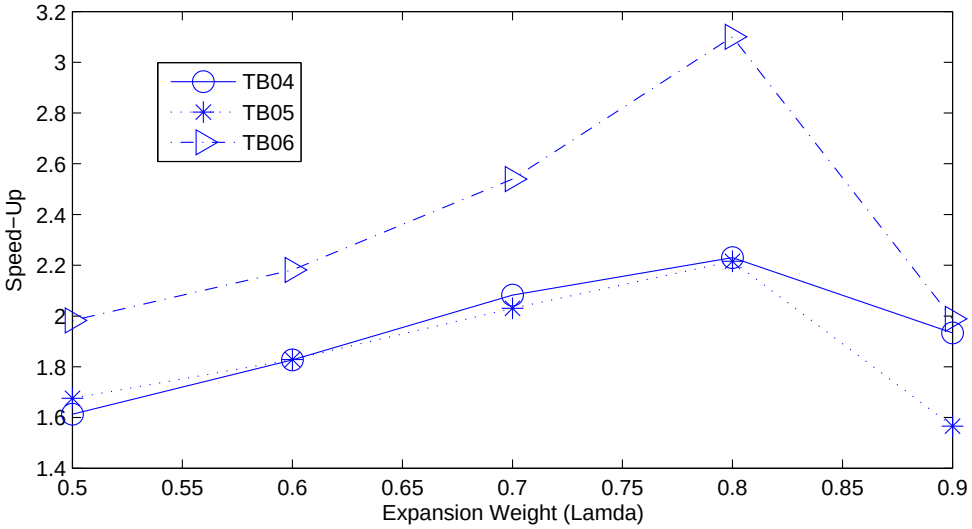
Table 6.5: Impact of the expansion weight (i.e., λ) on the effectiveness (MAP@1000)

Data sets	Methods	λ				
		0.5	0.6	0.7	0.8	0.9
TB04	FB-BL	0.254	0.256	0.257	0.255	0.252
	FB-Incremental	0.252	0.255	0.256	0.255	0.252
TB05	FB-BL	0.333	0.334	0.332	0.328	0.321
	FB-Incremental	0.333	0.334	0.332	0.327	0.321
TB06	FB-BL	0.284	0.285	0.285	0.284	0.281
	FB-incremental	0.282	0.284	0.283	0.282	0.280
CW09-12	FB-BL	0.153	0.157	0.162	0.163	0.164
	FB-incremental	0.152	0.158	0.161	0.163	0.164

Table 6.6: Average query execute time (ms) based on different expansion weights (top 10 documents for each query)

	NoFB	FB Methods	λ				
			0.5	0.6	0.7	0.8	0.9
TB04	69	FB-BL	4,731	4,016	2,786	1,756	933
		FB-Incremental	1,295	915	578	437	403
TB05	53	FB-BL	3730	2,923	1,999	1,131	547
		FB-Incremental	983	709	451	341	310
TB06	46	FB-BL	3909	3,060	2,167	1,317	573
		FB-Incremental	921	669	447	319	284
CW09-12	67	FB-BL	5,567	4,313	3,222	1,903	884
		FB-Incremental	841	692	550	471	433

Figure 6.4: The speed-up rate on different expansion weight λ (top 1000 documents returned)

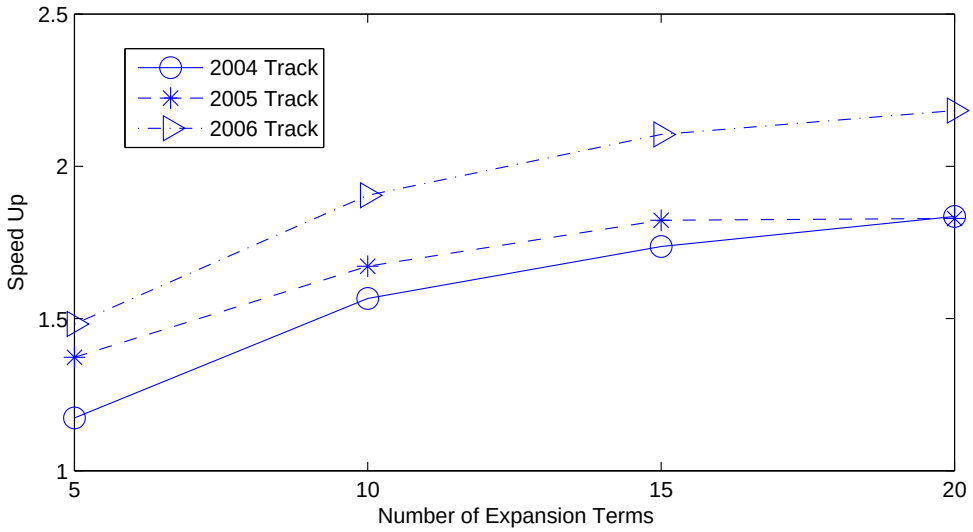


are shown in Table 6.4 and Table 6.5. We see that the optimal value of λ is around 0.6, which means that we should put more weights to the original query terms. According to Equation (6.1), the higher value of λ means that we put more trust on the original query and less weights to the expansion terms. Given the characteristics of SAAT pruning technique, postings of terms with smaller weights are more likely to be pruned. As a result, we can observe a clear trend that both methods can improve the efficiency more when the value λ is larger.

Figure 6.4 shows how the speed-up rate of $FB-Incremental$ compared with FB changes with the expansion weight. The higher expansion weight we choose, the more the original query determines the final ranking list and the more time the incremental method can save. However when the expansion weight is too high (e.g. 0.9), SAAT pruning technique is rather efficient and leave small room for further improvement. As a result, the speed-up rate at high expansion weight decreases.

We further test the impact of number of expansion terms on efficiency. Table 6.5 shows the speed up rate of $FB - Incremental$ compared with $FB - BL$. It shows that our new method benefits more when the system adds more expansion terms into

Figure 6.5: The speed-up rate on different number of expansion terms

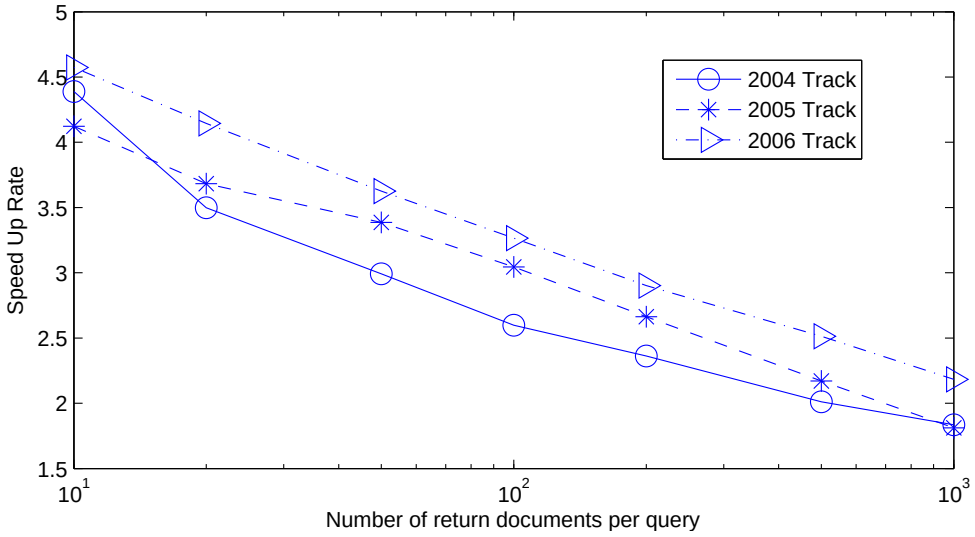


the original query. The main reason of this trend is that our method efficiently controls the grows of accumulator list size when the query becomes longer.

Another important factor that could affect the retrieval speed is the number of documents retrieved for each query. Figure 6.6 shows how the number of retrieved documents affect the speed up of $FB{-}Incremental$ over $FB{-}BL$. The results indicate that the new method can achieve more speed-up when the system returns fewer results to users. We believe it is due to the reason that returning fewer documents brings higher cut-off threshold. And in our method, the high cut-off threshold which is got from the initial run provides a strong and efficient guidance on the accumulator selections in the resumed retrieval. As a result, the pool of candidate documents/accumulators is kept in a very small size and final result is generated soon. In opposite, the baseline method which does the second round retrieval independently cannot get the benefit from the high cut-off threshold and it still generates large number of accumulators in which most of them are unnecessary.

We also report the efficiency when retrieving only 10 documents for each query in Table 6.6. It is clear that when retrieving fewer documents, FB-Incremental can achieve a higher speed up, i.e., around four. This indicates that this method could be

Figure 6.6: The speed-up rate on different number of retrieved documents per query



very useful for Web search domain since search users only need to look at 10 results per page. Finally, Figure 6.7 shows impact of the expansion weight on the speed-up when only 10 documents are retrieved. The trend is very similar to Figure 6.4.

6.3.2.3 Scalability

To further test the scalability of the incremental method and study how original query length affects the pseudo feedback system speed, we test both methods on 50k 2005 efficiency queries and 100k 2006 efficiency queries. For each query, 5 expansion terms are added and it only returns top 10 documents. The results show that the incremental method can stably improve the efficiency at a rate more than 2 at different original query length. For short queries, the baseline is fast and it leaves less space for further improvement. For long queries, the processing time is dominated by the original query terms and the optimizations in pseudo feedback part will not affect the total time too much. As a result, it is reasonable to observe a speed up rate summit at medium length query (i.e., 3-4 terms).

Another interesting discussion is how this incremental approach can work with different retrieval function other than language modeling approach (i.e. Dirichlet Prior

Figure 6.7: The speed-up rate on different expansion weight λ (top 10 documents returned)

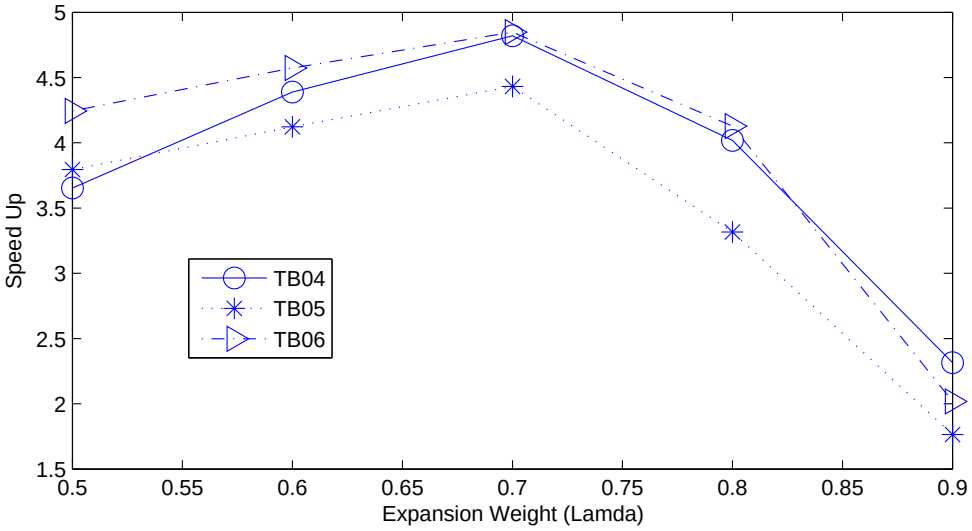


Table 6.7: Average query processing time (ms) on different query length (2005 50K queries)

	Avg	1	2	3	4	5	¿5
NoFB	35	4	10	26	55	91	175
FB-BL	297	38	94	302	526	749	1,102
FB-Incremental	124	18	43	97	204	316	540
Speed-up	2.4	2.1	2.2	3.1	2.6	2.4	2.0

Table 6.8: Average query processing time (ms) on different query length (2006 100K queries)

	Avg	1	2	3	4	5	¿ 5
NoFB	103	3	12	39	81	132	275
FB-BL	739	39	99	357	690	1,030	1,642
FB-Incremental	300	26	43	110	248	441	797
Speed-up	2.5	1.5	2.3	3.2	2.8	2.3	2.0

Table 6.9: Comparison of effectiveness and efficiency of BM25 retrieval method

	TB04	TB05	TB06
MAP@1000			
FB-BL	0.259	0.339	0.302
FB-Incremental	0.258	0.338	0.301
Efficiency (ms)			
FB-BL	6,296	5,649	5,071
FB-Incremental	3,211	2,744	2,114

smoothing method). To resolve this concern, we conduct similar experiments on Okapi BM25 [82] retrieval method. For each query, we select 20 terms from top 10 documents from the initial run with $\lambda = 0.8$ and return 1000 documents for the second run. The results are reported in table 6.9. It is obvious that the results on BM25 retrieval method is similar to Dirichlet Prior smoothing method: on one hand, the incremental approach can provide comparable results as the baseline and it does not affect effectiveness. On the other hand, it reduces query processing time for more than two times. It proves that this incremental approach is salable across different retrieval methods.

6.4 Conclusion and Future Work

Long query processing can be slow due to more postings to be evaluated and less effective dynamic pruning. To speed up SAAT pseudo-relevance feedback as one special case of long query processing, we propose an incremental approach which uses the results from the initial round retrieval to leverage the query processing in the second round. The method is proved to be useful and comparing to strong baseline it can increase query processing speed by 2 to 4 times without hurting effectiveness.

One direction of the future work is to apply similar strategy for DAAT query processing. However we expect it might not be as significant as SAAT: First DAAT's query processing time increases slower than SAAT as query becomes longer. Second and most important, DAAT does not use accumulated lists to store intermediate results which makes it harder for the second round retrieval to get benefit from the results of

initial round retrieval.

Another interesting direction may be expanding this incremental method to general long query processing. We may first identify several key terms of the original query and process such key terms as the initial round. We then use the results of the initial round retrieval to speed up the process of the rest terms. We leave the exploration as the future work.

So far we have successfully proposed two new methods which help to improve query process efficiency in various situations. However they are still not enough. In general, how can we further improve efficiency? If we can scarify effectiveness of the results to exchange shorter query process time in extreme situations, how should we look for good strategies? In the following chapter, we will more details about it.

Chapter 7

OPTIMIZING EFFICIENCY/EFFECTIVENESS TRADE-OFF

7.1 Introduction

In the previous chapters, we developed an analytical model. Based on this model, we propose a new retrieval method called document prioritization. Compared with dynamic pruning methods, it can significantly increase efficiency when k is large with insignificant effectiveness loss. We also use a new incremental approach to improve the speed of pseudo-relevance feedback as a partly solution for long query processing.

However, an IR systems efficiency is not only about average query processing time; more importantly, it depends on whether every query can finish in time. Previous studies [23, 32, 87] have shown that query latency directly affects users' search satisfaction. Users usually expect to see results shortly after they submit their queries. If a query's response time is too long (i.e. more than 1 second), users may lose their patience and move to another faster search engine, even if the slower one can provide more accurate results. Hence, maintaining certain efficiency is a "must-do" for a general search engine, and in some situations we can even sacrifice effectiveness for efficiency if necessary. In general, efficiency and effectiveness are at odds with each other. A highly efficient system may suffer from effectiveness loss because it relies on overskipping documents or an oversimplified mechanism. On the other hand, a highly effective system may be slowed by its complicated ranking functions and additional features. Therefore, maintaining a balance between efficiency and effectiveness and making smart tradeoffs between them is very important for designing an IR system.

Many IR systems apply fixed efficiency/effectiveness strategies on all queries. The typical design can be found on WAND [30], which sets a higher cut-off threshold

and skips more documents, reducing query processing at the risk of losing highly scored relevant documents. These fixed systems may have some problems. Imagine two queries: query A is difficult and requires highly complex computations, while query B is easy and requires low computational complexity. If a system uses a safe or less aggressive document pruning strategy, both queries can have high effectiveness results, but query As result may be delayed. However, if a system uses an aggressive document pruning strategy, both queries can be answered in time, but query B may perform worse in terms of effectiveness. Obviously, the best strategy is to process query A aggressively and process query B safely so that both queries can be answered with maximal effectiveness while meeting certain efficiency requirements. A similar idea is proposed by Tonellotto [77], who tried to predict query processing time and applied an aggressive pruning strategy to queries predicted to have a long query processing time. However, Tonellottos work still does not address core questions: How can we adjust strategy so that every query processing time falls within certain time constraints? How can we make smart tradeoffs to maximize effectiveness while maintaining the efficiency requirement?

In this chapter, we attempt to answer such questions by proposing a new framework. First, we provide a theoretical analysis to help find good trade-off strategies. Based on our theoretical foundation, we then propose several efficiency/effectiveness trade methods and compare them with experimental results to check whether the results meet our theory.

7.2 Theory

Top-k query processing under strict time constraints is different from that without time constraints. For some queries, we do not have time to go through all the candidate documents, which causes effectiveness to suffer if some relevant documents are therefore not evaluated. The results of time-constrained top-k query processing can somehow be judged by their similarity to the results from those in which all candidate documents are evaluated. Usually, the closer the result is to the exhaustive approach,

the more effective it is. In the ideal case, a time-constrained method should at least evaluate the k documents from the exhaustive approachs results to ensure exactly the same output. To help identify good efficiency and effectiveness trade-off strategies in the time constraint environment, we provide the following theories:

Theorem 1 (High score document keeping property) *A good time-constrained top-k query processing strategy should not miss the documents with the highest evaluation scores.*

Documents with the highest evaluation scores are most likely to be relevant. Skipping such documents or not having enough time to evaluate them risks a significant loss of effectiveness.

Theorem 2 (law score document avoid property) *A good time-constrained top-k query processing strategy should avoid evaluating low-score documents.*

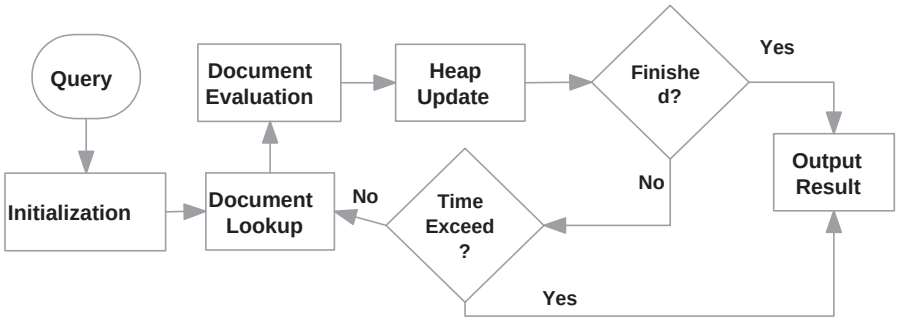
The strict time constraint limits the number of documents that can be evaluated before the deadline. Hence, if an IR system evaluates too many low-score documents, it may not have time to evaluate high-score documents, thus hurting its effectiveness.

Theorem 3 (fast document evaluation property) *A good time-constrained top-k query processing strategy should evaluate documents quickly.*

Fast documents evaluation can help a system to evaluate more documents. The more documents evaluated before the deadline, the less the IR system risks missing relevant documents.

Imagine that we are in a house on fire. The fire is so large that we cannot extinguish it. To reduce loss, we should try to save assets before the fire destroys everything. Obviously, we should pick valuable things and ignore those worth less. We should also do it quickly so that we can pick up more things. Time-constrained top-k query processing operates in a similar way. A good top-k query processing method

Figure 7.1: The architecture of Query Early Termination strategy



should be simple in order to evaluate more documents. It should avoid evaluating low-score documents and keep high-score documents. We will use this theory to analyze the methods in the next section.

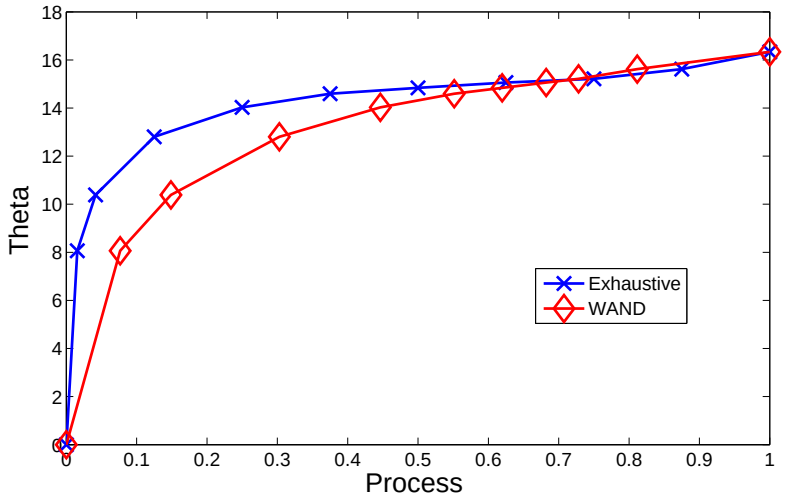
7.3 Top-k Query Processing at Time Constrained Environment

In the previous section, we have shown the theorems that help to identify good top-k query processing strategies for time-constrained environments. In this section, we will propose five different methods and use the theorems to analyze them.

7.3.1 Force termination of query processing

To make sure that every query's processing time meets certain time constraints, the native thinking is that we can process documents with a ranking-safe dynamic pruning framework (e.g. WAND) until the deadline expires. We then force the query processing to terminate and output the results. This designs implementation is shown in figure 7.1. In contrast to a typical query processing system without time limitations, we add a stage to check whether it exceeds the time deadline. The strategy is quite simple but very useful for controlling the processing time for every query. As output results from the top-k heap take almost no time, the top-k query processing time with this strategy can be guaranteed to meet the time constraint strictly. But how effective is this method?

Figure 7.2: cut-off threshold curve during query processing



First, as figure 7.2 shows, the cut-off threshold of traditional rank-safe WAND query processing is set to zero at the beginning and gradually increases later. Hence, at the beginning, by using the original dynamic pruning framework, no document skipping occurs, and every document, including those with low scores, is evaluated, which obviously miss the "low score document avoidance" property. Most importantly, forcing a termination improves efficiency by cutting off the documents at the tails of the posting lists. For the DAAT docID-sorted inverted index , a document position does not represent the documents score or relevance. Rather, high-score documents are more likely to be distributed throughout the posting list: head, tail, or middle. Simply cutting off documents at the tail results in a high risk of missing high-score documents and obviously misses the "high score document keeping" property. Therefore, the forced termination technique should not provide very good results with regard to effectiveness.

7.3.2 Adjusting cut-off threshold rate F through efficiency prediction

Similar to the idea of [30,77], we can also control queries' top-k processing time by adjusting the cut-off threshold rate F in WAND. As we know, F controls how

aggressively the IR system prunes documents. When $F = 1$, the cut-off threshold equals the score of the kth document in the top-rank lists, and the results will be rank-safe. If we set $F > 1$, the cut-off threshold will be larger than the score of the kth document, and hence more documents will be pruned. Based on this feature, we develop separate strategies for easy and hard queries. The design is shown in algorithm 5 where Q is a query, K is the number of returned documents, TC is the time constraint for each query, PT is the predicted query processing time, and F' is a parameter representing the cut-off threshold rate to be set for queries predicted to exceed the time constraint. The main idea is to predict a query's processing time and compare the predicted result with the time constraint. If the predicted time falls within the time constraint, the query should be finished on time, and the original rank-safe framework will be applied. However, if the predicted time is longer, a more aggressive strategy (i.e. $F = F' > 1$) will be applied to reduce the query processing time.

Algorithm 5 Binary selection of cut-off threshold rate based on query processing prediction

Function{GetCutoffThresholdRate}$\{Q, K, TC\}$
$PT = TimePrediction(Q, K)$
if $PT > TC$ **then**
  return F'
else
  return 1
end if
EndFunction

By pruning more documents, this method can reduce the time consumption of top-k query processes that may exceed the processing deadline of the original rank-safe approach. However, in contrast to the previously mentioned forced termination method, this strategy cannot guarantee that every query will meet the time constraint, for two reasons. First, the efficiency prediction is not 100% percent accurate. For example, a hard query may be falsely predicted to finish on time; applying a conservative strategy to it will generate overdue results. Second, the parameter F' does not fit all overdue queries. For some extremely hard queries requiring very long query processing

times, a moderate threshold rate is still not aggressive enough to force their query processing times into the range. Alternatively, if we set F' to be very large, some easier queries requiring a query processing time slightly longer than the deadline may lose considerable effectiveness to the too-aggressive pruning strategy. This dilemma is shown more clearly in the experimental section.

This methods effectiveness depends largely on the parameter F'. The higher F' is, the more likely the queries are to meet the time constraints, but the less effective the results may be. Additionally, we would like to analyze the method with the rules we provided in the previous section. Figure 7.4 and figure 7.3 show how the K-th document score (τ) and cut-off threshold change during query processing for different F'; we generate the results using the average of the TREC 2006 terabyte collection. The two figures yield some interesting observations. First, even if we apply large F', a buildup process occurs for the cut-off threshold, and at the beginning, low-score documents are not pruned, thus not meeting the "low score document avoid" property. However as F' becomes larger, the cut-off threshold grows faster and the number of low-score document evaluations becomes smaller. In other words, we expect the low-score document evaluations to influence the term of effectiveness less than during the previous forced termination approach. Second, although increasing F' can surely increase both efficiency and the cut-off threshold, the score of the k-th document obviously decreases. This means that as F' increases in size, some high-score documents that should be ranked in the top-k are skipped, and their absence from the top list leads to a lower score for the k-th documents. In fact, for some queries, the absent documents even include top high-score documents containing all the query terms and should be ranked at the very top of the results if F' is large enough. Obviously, the larger F' is, the more likely it is to miss the "high score documents keeping" property.

Another approach is to adjust the cut-off threshold rate F through linear function. Applying the binary strategy we discussed before may still be arbitrary and does not distinguish the very hard queries that are processed much longer than the deadline with those easier ones that require just a little more time than the constraint. To

Figure 7.3: Average Cut-off threshold change during query processing for different F'

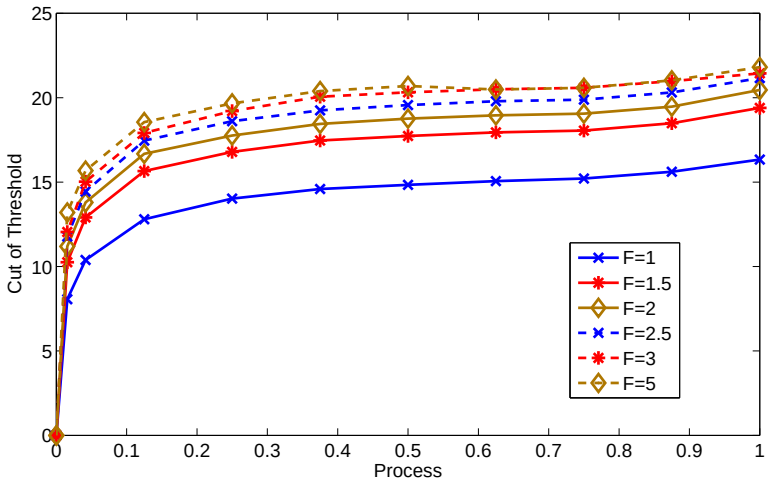
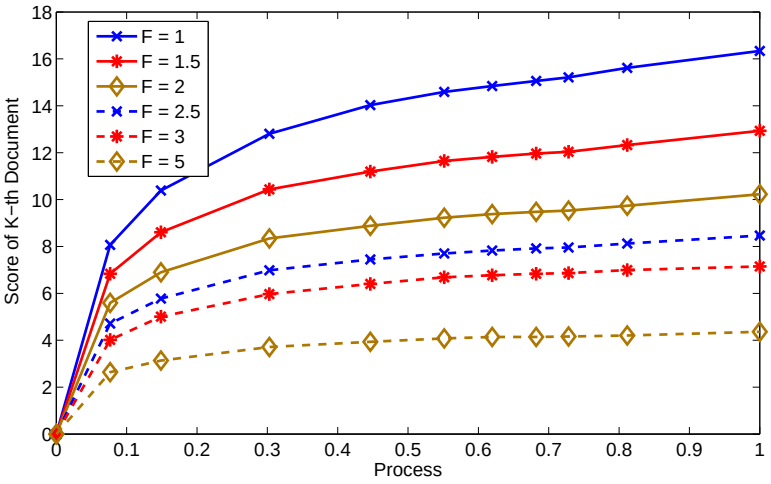


Figure 7.4: Average of the K-th document score change during query processing for different F'



solve this problem, we propose a new method, shown in algorithm 6. The main idea is to use the difference between the prediction time (PT) and time constraint (TC) to determine the cut-off threshold F. The larger the gap ($PT - TC$), the larger F is, and the more aggressive strategy will be applied to prune more documents. α is a parameter that connects the time difference and F value. This method may perform better when query processing times vary. However, it does not overcome all difficulties. In fact, either adjusting the cut-off threshold binarily or linearly creates a dilemma: queries that use conservative parameters (i.e., small F' or α) suffer a slow start and miss the "low score document avoid" property. However, queries that use aggressive parameters miss high-score documents and miss the "high score document keeping" property. As a result, we do not expect these two methods to perform the best in terms of effectiveness.

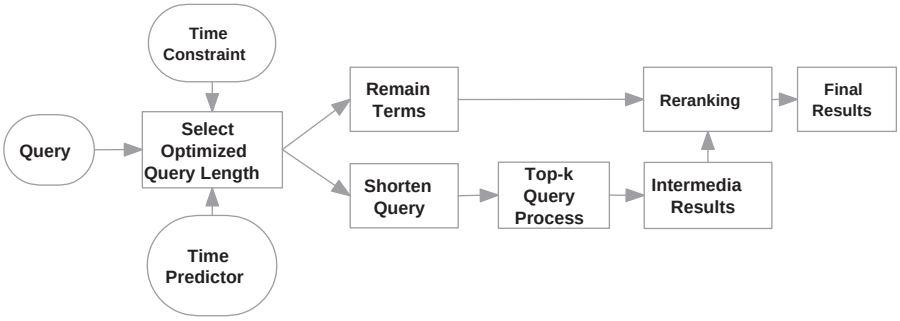
Algorithm 6 Linearly selection of cut-off threshold rate based on query processing time prediction

Function{GetCutoffThresholdRate}$\{Q, K, TC\}$
$PT = TimePrediction(Q, K)$
if $PT > TC$ **then**
  return $\alpha * (PT - TC) + 1$
else
  return 1
end if
EndFunction

7.3.3 Simplifying queries

In the previous chapter, we have shown that query-processing time is related to query length (i.e., the number of unique query terms). A longer query not only means evaluating more documents and postings but also indicates more computational burden in candidate document lookup. Consequently, if we can replace the original long query with a shorter query, we can greatly reduce the top-k query processing time. What is more, because a long querys result is usually dominated by several main high IDF terms, removing the low IDF terms should not significantly influence the

Figure 7.5: The architecture of Query simplifying query processing



result [105–107, 123]. Following this idea, we propose the query simplification method and the architecture shown in figure 7.5. First, we use the analytical model discussed in the previous chapter to predict the query processing time at different query lengths for a given query Q. Based on the prediction results and time constraints, we then select a subquery to generate an intermediate result through typical dynamic pruning query processing (e.g., WAND). Finally, we re-rank the intermediate result using the remaining lower-impact terms to generate the final result.

Algorithm 7 shows the optimized subquery selection process. Q is the original query, K is the number of returned documents, TC is the time constraint, $L(Q)$ is the original query length, and Q_i is the ith term of the sorted query terms. The query terms are sorted by their impacts (i.e., the maximum score of their postings). At the beginning, the subquery is empty, and we then try to add query terms to it, one by one, from highest-impact term to lowest-impact term, until the subquery's predicted processing time exceeds the time constraint. Obviously, the length of the subquery is closely related to the time constraint. When the time constraint is very tight, the subquery includes just a few high-impact terms. On the contrary, if the time constraint is relaxed , the subquery will include more terms; in the most extreme case, the subquery is identical to the original query.

Algorithm 8 shows how we use the remaining lower-impact query terms to re-rank the intermediate results, where D and S are the document IDs and scores of the intermediate results, D_j and S_j are the document ID and score of the jth document,

Algorithm 7 Select optimized sub-query based on query processing time prediction

Function{GetSubQuery}$\{Q, K, TC\}$
$sort(Q)$ % Sort by maximum score
$SQ = Q_1$
for $i = 2$ to $L(Q)$ **do**
  $PT = TimePrediction(\{SQ, Q_i\}, K)$
  if $PT > TC$ **then**
    $BREAK$
  else
    $SQ = \{SQ, Q_i\}$
  end if
end for
return SQ
EndFunction

QR is the set of remaining terms, and $L(D)$ is the number of remaining terms. The algorithm includes three functions: $MoveToDoc(i, D_j)$ simply moves the pointer of the ith term's posting list to a document whose ID is at least D_j; $CurDoc(i)$ simply returns the current document of the ith term, and $Grade(i, D_j)$ returns the evaluation score (e.g., BM25) of document D_j from the ith term. First, we sort the candidate documents based on their document IDs for the convenience of matching the posting lists of remaining terms. We then go through the documents from the lowest document ID to the highest. If it also contains a remaining term, we update its score by adding the additional evaluation score from the remaining term. Finally we sort the documents based on their scores from the highest to lowest and output the results as the final results.

By applying the similar strategy as [49,50] to the intermediate results re-ranking process, we can skip most of the postings without decompressing or evaluating them. Consequently, the time consumed by re-ranking is quite small and negligible. Hence, if the time prediction is accurate, the query should finish on time. However, this strategy is still not as reliable as the forced termination method because accurately predicting the query processing time is impossible [70, 110].

Query simplification greatly reduces the complexity and time for document

Algorithm 8 Re-rank intermediate results with remain query terms

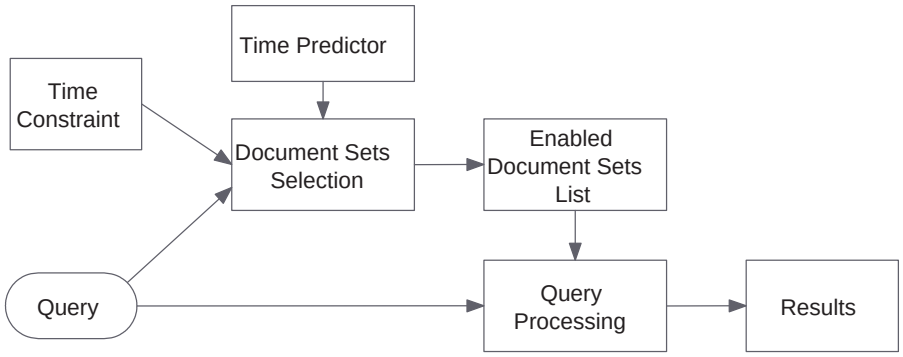
```
Function{ReRankIntermediateResults}{D, S, QR}
sort(D, S) % sort based on document IDs
for i = 1 to L(QR) do
  for j = 1 to L(D) do
    MoveToDoc(i, D_j)
    if CurDoc(i) == D_j then
      S_j+ = Grade(i, D_j)
    end if
  end for
end for
sort(D, S) % sort based on score.
return (D, S)
EndFunction
```

lookup and evaluation. Hence, it meets the "fast document evaluating" requirement very well. However as query simplification applies the original dynamic document pruning framework to intermediate result generation in a manner similar to that of the forced termination method, it still suffers from the slow start as the cut-off threshold increases, and it may somehow miss the "low score document avoid" property. Moreover, for some queries in which query terms are equally important, cutting off terms may shift the original meanings and lead the system to miss the high-score relevant documents, hurting effectiveness and missing the "high score documents keeping" property. Therefore, we don't expect that query simplification can output good results for every query, although it might be a good solution for some (e.g., long queries).

7.3.4 Document prioritization

The previous four methods do have weaknesses. They all waste time by evaluating low-score documents, and they are all at risk of losing high-score documents. Is there any way to evaluate only high-score documents? Following the previous chapters analysis, all candidate documents can be classified into document sets based on the query terms they contain. Dynamic document pruning techniques (e.g., WAND) are implemented by gradually disabling the document sets as the score of the k-th document increases in order from the document set with the lowest maximum score

Figure 7.6: The architecture of Document Prioritization query processing



to the document set with the highest maximum score. Generally, the document sets with high maximum scores are more likely to contain high-score, relevant documents, while most of the documents in low maximum score documents sets are low-score and nonrelevant. To reduce query processing times to meet constraints, can we evaluate only the high score document sets? Based on this hypothesis, we propose the document prioritization method which borrows some ideas from [111].

Figure 7.6 shows the architecture of the document prioritization method. First we use the analytical model and the time constraint to select the document sets with the highest maximum scores. The system then evaluates only the documents in the given sets and skips all others. By applying a pivoted-based document lookup strategy similar to WAND, this method can prune documents efficiently. Algorithm 9 shows the document set selection process, in which DS represents all the documents generated by given query Q, TC is the time constraint, PT is predicted executing time, and ES represents the selected document sets. As we know, candidate documents can be divided into document sets based on the query terms they contain. For example, a query AB with two query terms will yield three document sets:(1) documents containing all both A and B, (2) documents containing only A, and (3) documents containing only B. Obviously, for query Q with $L(Q)$ unique query terms, there will be $2^L(Q)$ document sets. We present all the document sets as DS. We then sort DS and try to involve document sets (i.e., d) one after another in the order from those with the

highest maximum scores to those with the lowest maximum scores until the prediction time exceeds (PT) the time constraint (TC). Finally, we output the involved document sets ES and use them for top-k query processing.

Algorithm 9 Select document sets based on query processing time prediction

```
Function{GetDocumentSets}{Q, TC}
DS = DocSet(Q) % Get all document sets of Q
sort(DS) % Sort Document Sets by their maximum score
ES = % Initial enabled documents set as empty
for d in DS do
  PT = TimePrediction({ES, d})
  if PT > TC then
    BREAK
  else
    ES = {ES, d}
  end if
end for
return ES
EndFunction
```

No matter what the time constraint is, the document sets with the highest maximum score are always selected and evaluated. As a result, the documents with the highest scores are very likely to be processed. The document prioritization method has the "high score document keeping" property. What is more, the document sets with low scores are likely to be discarded under tight time constraints, and the system will not spend too much time on low-score document evaluation, thus having the "low score document avoid" property as well.

7.3.5 Summary

In the previous discussion, we proposed five methods to help a top-k query processing system meet certain time constraints. The five methods have their own advantages and disadvantages. Overall, however, document prioritization best meets the rules, and we expect it to perform best. It is also worth mentioning that all methods except forced termination dynamically adjust their query processing strategies based on efficiency prediction results. Because current efficiency prediction is not

accurate [70, 110] the actual query processing time of the four methods is likely to exceed the deadline and cannot meet the time constraint strictly. To resolve this problem, we can mix the four methods with forced termination strategy: when they exceed the deadline, we force terminate the query processing and output the results. We show more discussion and results in the next section.

7.4 Experiments

In the previous sections, we proposed five different methods to reduce top-k query processing time to meet certain time constraints. In this section, we test them with data sets to check whether they meet our hypothesis.

7.4.1 Experiment design

To test how the three efficiency control methods work, we conduct experiments on the TREC Gov2 collection [2], which consists of 25.2 million web pages from the .gov domain. We use the queries from the TREC official terabyte track 2004-2006 [37], denoting them as *TB04*, *TB05*, and *TB06*. Each query set contains 50 queries, and for each query we return 1000 documents unless stated otherwise.

As in other chapters, all experiments were conducted on a single machine with dual AMD Lisbon Opteron 4122 2.2GHz processors and 32GB DDR3-1333 memory. And we used okapi BM25 [82] as the retrieval function to rank documents. The query processing system is implemented similar to Virtual IR Lab which is described in previous chapter.

We denote the force termination method as *CUT*, the binary adjusting cut-off threshold rate as *RATE-B* and its linear version as *RATE-L*, the query simplifying method as *QS*, and document prioritization as *DP*.

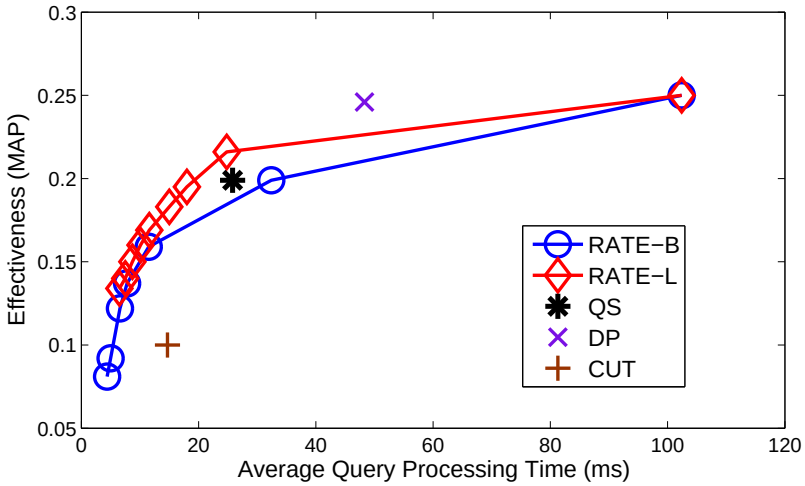
7.4.2 Performance at loose time constraints

In some situations, we would like query processing to follow some loose time constraints. Instead of forcing every query to finish within strict time deadlines, we try to reduce query-processing time so that most query processing times and the average

query processing time fall within certain range. We test the five methods and compare their performance through three statistics: the effectiveness, which is measured by MAP(mean average precision), the average query processing time (AT (ms)), and the percentage of late queries (LR %). Considering that the computational time may have small drafting during the experiments, we define late queries as the queries that exceed the constraint by more than 20%. For example, if the deadline is 15ms, any query whose query processing time is longer than 18ms will be considered as late. Ideally, we expect a method to produce high effectiveness results with low average query processing time and few late queries. However, as we discussed in the previous section, effectiveness and efficiency are usually at odds with each other. Table 7.1, table 7.2, and table 7.3 show the performance comparison between the methods on *TB04*, *TB05*, and *TB06* collections. We tested four time constraints (15ms, 30ms, 50ms, and 100ms) to simulate situations ranging from very tight to weak time constraints. For each time constraint, the winner is not obvious because each performance is evaluated by three features. However, all the samples demonstrate the trend that efficiency usually conflicts with effectiveness. In particular, if we use a more aggressive parameter, we may expect higher efficiency and a lower late query rate but more effectiveness loss.

To more easily compare the differences, we can draw a trend curve from the samples. Figure 7.7 shows a comparison of those methods when the time constraint is 15ms for the TB04 collection. Similar figures can be drawn for different time constraints and collections. In this figure, the x-axis is the average query processing time in microseconds and the y-axis is effectiveness as measured in MAP@1000 [19,41]. The points are samples from different methods with different parameters. The curve of *RATE-L* is above the curve of *RATE-B*, which confirms our hypothesis that adjusting the cut-off threshold rate through linear function may lead to a better tradeoff between the curves. Although query simplification has a disadvantage (i.e., a high risk of losing high-score documents), its advantage (a simpler mechanism) helps it to a performance comparable to that of *RATE-B* and *RATE-L*. The point above the two curves, representing the document prioritization method, shows that the document prioritization

Figure 7.7: The comparison of difference methods when time constraint is 15ms for TB04 collection



method can provide higher effectiveness results at the same level of efficiency. This advantage results from its better compliance with the three rules. Overall, the results confirm our hypothesis in the previous section.

7.4.3 Performance at strict time constraints

The purpose of efficiency control is merely to reduce average query processing, but more importantly, to make sure every query can finish on time to maximize user satisfaction. Unfortunately, four of the five methods, (*RATE-B*, *RATE-L*, *QS*, and *DP*), cannot guarantee that every query processing meets the time constraint. The four methods all dynamically adjust their strategies based on efficiency prediction. Because efficiency prediction is not accurate, the methods may apply a "not aggressive enough" strategy and exceed the time deadline. The experimental results shown in table 7.1 also confirm the problem that none of the four methods has zero late queries under the strict time constraints (e.g., 15ms). The fifth method, *CUT*, can guarantee query processing finish on time; however it hurt effectiveness a log. To solve this problem, we combine *CUT* with aforementioned four methods. Specifically, we terminate their query processing when they hit the deadline. Thus, every query can be finished in time,

Table 7.1: Performance Comparison on *TB04* collection in loose time constraints. Methods are evaluated by MAP(Mean Average Precision),AT(Average Processing Time:ms) and LR(Late Rate %)

	Deadline = 15*ms*			*Deadline* = 30*ms*			*Deadline* = 50*ms*			*Deadline* = 100*ms*			*Deadline* = 200*ms*		
	MAP	AT	LR	MAP	AT	LR	MAP	AT	LR	MAP	AT	LR	MAP	AT	LR
Terminate Query Processing when it hits the deadline															
CUT	0.100	14.7	0%	0.151	27.6	0%	0.200	41.2	0%	0.240	64.4	0%	0.246	87.8	0%
Adjust cut-off threshold rate through binary decision															
F'=1	0.250	102.4	86%	0.250	100.7	66%	0.250	101.0	54%	0.250	100.5	26%	0.250	100.3	12%
F'=1.5	0.199	32.4	46%	0.207	34.4	26%	0.207	35.2	18%	0.238	53.1	8%	0.247	71.1	2%
F'=2	0.159	11.5	18%	0.180	14.4	0%	0.181	15.0	0%	0.222	35.9	2%	0.240	58.6	2%
F'=2.5	0.137	7.8	2%	0.164	11.0	0%	0.164	11.9	0%	0.214	33.9	2%	0.239	58.6	2%
F'=3	0.122	6.6	2%	0.155	9.9	0%	0.155	10.6	0%	0.210	33.7	2%	0.239	57.2	2%
F'=5	0.092	5.0	2%	0.135	8.4	0%	0.135	9.2	0%	0.204	33.1	2%	0.239	58.1	2%
F'=10	0.081	4.4	2%	0.134	7.9	0%	0.134	8.9	0%	0.201	32.9	2%	0.239	56.9	2%
Adjust cut-off threshold rate through linear function															
α=0.002	0.244	48.1	82%	0.245	50.5	54%	0.247	55.1	42%	0.250	66.2	16%	0.251	85.7	6%
α=0.005	0.216	24.8	56%	0.224	27.4	26%	0.227	33.3	16%	0.237	47.5	2%	0.249	70.2	4%
α=0.008	0.195	18.0	46%	0.208	21.7	20%	0.219	26.9	4%	0.236	41.0	2%	0.249	64.8	2%
α=0.01	0.183	15.0	34%	0.197	18.7	14%	0.216	24.4	2%	0.228	37.9	2%	0.246	65.0	2%
α=0.015	0.169	11.6	20%	0.176	14.6	0%	0.202	20.7	0%	0.224	36.4	2%	0.242	61.3	2%
α=0.02	0.160	10.1	12%	0.166	12.0	0%	0.189	18.5	0%	0.223	36.3	2%	0.241	60.0	2%
α=0.03	0.150	8.7	10%	0.155	10.1	0%	0.170	16.2	0%	0.222	34.9	2%	0.239	58.9	2%
α=0.05	0.140	7.5	6%	0.144	8.9	0%	0.158	13.4	0%	0.218	34.2	2%	0.239	57.4	2%
α=0.1	0.134	6.5	4%	0.134	8.0	0%	0.142	10.1	0%	0.215	33.2	2%	0.239	56.6	2%
Query Approximation															
QS	0.181	37.9	46%	0.224	44.1	34%	0.227	46.5	18%	0.245	60.5	8%	0.250	78.9	2%
Document Prioritization															
DP	0.246	48.3	54%	0.246	50.1	40%	0.250	51.1	28%	0.247	62.8	14%	0.247	76.3	2%

Table 7.2: Performance for different efficiency control methods on *TB05* collection

	Deadline = 15*ms*			*Deadline* = 30*ms*			*Deadline* = 50*ms*			*Deadline* = 100*ms*			*Deadline* = 200*ms*		
	MAP	AT	LR	MAP	AT	LR	MAP	AT	LR	MAP	AT	LR	MAP	AT	LR
Terminate Query Processing when it hits the deadline															
CUT	0.093	14.8	0%	0.163	28.1	0%	0.208	42.9	0%	0.291	67.0	0%	0.312	81.7	0%
Adjust cut-off threshold rate through binary decision															
F'=1	0.324	81.1	88%	0.324	81.5	70%	0.324	81.4	50%	0.324	81.1	18%	0.324	81.5	4%
F'=1.5	0.275	21.7	44%	0.280	24.6	24%	0.297	29.1	8%	0.309	41.3	2%	0.322	62.6	0%
F'=2	0.214	10.2	12%	0.233	13.6	4%	0.250	19.2	2%	0.286	35.6	2%	0.316	61.0	0%
F'=2.5	0.172	7.3	6%	0.193	11.0	4%	0.222	17.0	2%	0.274	34.5	2%	0.311	61.0	0%
F'=3	0.158	6.3	4%	0.181	10.2	4%	0.212	16.4	2%	0.270	34.3	2%	0.310	60.7	0%
F'=5	0.136	4.8	2%	0.164	8.8	4%	0.196	15.2	2%	0.263	33.3	2%	0.307	60.4	0%
F'=10	0.117	4.1	2%	0.148	8.3	4%	0.185	14.8	2%	0.255	33.2	2%	0.306	60.6	0%
Adjust cut-off threshold rate through linear function															
α=0.002	0.316	40.9	78%	0.319	47.7	60%	0.321	51.7	36%	0.322	55.4	8%	0.323	65.6	0%
α=0.005	0.283	26.3	64%	0.293	29.9	36%	0.300	36.0	16%	0.312	49.3	4%	0.319	63.8	0%
α=0.008	0.263	19.2	50%	0.273	23.6	18%	0.281	30.4	8%	0.303	42.1	2%	0.315	63.4	0%
α=0.01	0.249	16.7	40%	0.265	21.1	14%	0.273	27.9	6%	0.295	40.8	2%	0.315	63.0	0%
α=0.015	0.220	12.3	26%	0.243	17.2	8%	0.263	23.9	2%	0.289	39.2	2%	0.314	62.7	0%
α=0.02	0.204	10.5	18%	0.229	15.0	4%	0.251	21.8	2%	0.284	38.2	2%	0.314	61.6	0%
α=0.03	0.191	8.7	14%	0.202	12.7	4%	0.237	19.0	2%	0.277	36.4	2%	0.313	61.5	0%
α=0.05	0.162	7.1	8%	0.184	11.0	4%	0.216	17.1	2%	0.270	34.5	2%	0.314	61.9	0%
α=0.1	0.141	5.8	2%	0.149	9.1	4%	0.192	15.8	2%	0.252	33.6	2%	0.303	61.1	0%
Query Approximation															
QS	0.242	30.9	48%	0.289	37.3	30%	0.310	41.7	20%	0.316	51.7	4%	0.322	68.1	0%
Document Prioritization															
DP	0.308	40.8	58%	0.308	43.3	40%	0.309	46.4	22%	0.322	57.6	10%	0.323	67.1	2%

and the quality of a method is evaluated only by the effectiveness of force-terminated query processing results.

The difficulty in this mixture of method is that, if a method is too aggressive and finishes query processing far before the deadline, it may naturally miss many relevant documents. On the other hand, if a method is not aggressive enough and requires time beyond the deadline, its effectiveness may still suffer because large numbers of documents at the tails of posting lists are not evaluated. A good method is one that selects proper strategy and collects as many relevant documents as possible.

Table 7.4 compares the effectiveness of different methods for strict time constraints. Because they all meet the efficiency requirement, we compare only their effectiveness, which is measured by MAP@1000. For all collections, forcing termination alone *CUT* performs the worst. We tuned the parameters of *RATE-L* and *RATE-B* (e.g., F and α) and report the best performance in the table. Although the optimized parameters are used, they still don't show the best performance. Compared with adjusting the cut-off threshold, query simplification *QS* performs better in TB04 and TB05 collections, while document prioritization shows the best results on all collections.

To explain the results, we measure the three statistics for each method during query processing: total number of evaluated documents, average BM25 score of evaluated documents, and processing percentage when it is force terminated. We estimate processing percentage by dividing the number of current processed and skipped postings by total number of postings. Table 7.5 shows the average of the three statistics when we apply the strict time constraint at 15 ms. First, we look only at *CUT*, the force termination approach. It proceeds slowly and only finishes under 20% of the process when timed out. Thus, we expect more than 80% of high-score documents to be missed. Moreover, the average score of the evaluated documents is quite low, suggesting that most of the evaluated documents are low-score and nonrelevant. These two observations do indicate that force termination misses the "high score document keeping" and "low score document avoid" properties. Although forced termination

alone does evaluate more documents than other methods because fewer documents are skipped, it still outputs the worst effective results. Compared with forced termination alone, *RATE-L + CUT* and *RATE-B + CUT* show higher average document scores and finish percentages because the higher cut-off threshold helps avoid low document evaluations. Consequently, the two methods perform better together than forced termination alone *CUT*.

Document prioritization *DP + CUT* demonstrates even more advantages in these statistics. First, the average evaluation score is significantly higher than that produced by other methods, supporting our hypothesis that document prioritization has a strong ability to avoid low-score documents and keep high-score documents. Moreover, document prioritization also shows a higher average process percentage, and fewer documents at the tails are cut off during forced termination. These advantages let document prioritization process more high-score documents under the strict time constraint. Consequently, although document prioritization evaluates few documents on average because it results in more skipping, it outputs significantly better results compared with the other four methods under strict time constraints.

Compared with the previous four methods, query simplification *QS* is very different. It may reduce the number of process query terms in exchange for increased speed. Table 7.5 shows that *QS* can usually evaluate more documents in the given time compared with other methods. *QS* runs fast, which greatly reduces the cost of forced termination. The average document score is just a little lower than *RATE-B + CUT* and *RATE-L + CUT*. Considering that it has few terms, this number still indicates the good quality of its evaluated documents. As a result, it performs better than *Rate-B* and *Rate-L* on TB04 and TB05 collections. However, for queries with phrases, deleting query terms may change the meaning of original queries and lead to poor effectiveness. As a result, in TB06 it performs more poorly than the baseline methods. Despite *QS*, we observe that this method has some potential for long queries. The comparison for long queries is reported as Table 7.6. Because the number of long queries in the three collections is small, we merged all long queries (i.e. query length

≥ 5) in TB04, TB05, and TB06 and report their average. It is obvious that *CUT*, *RATE-B + CUT*, and *RATE-L + CUT* perform much worse than *DP + CUT* and *QS + CUT*, especially when the time constraint is very tight. Between the two winners, *QS + CUT* performs a little better than *DP + CUT* because of its strong ability to speed up query processing. We expect the method to perform better if there are more longer queries.

To make our claim more convincing, we also test the five methods on a larger collection (i.e. TREC clueweb09 category B collection [1]) which contains 50 millions documents and its size is almost as twice large as the Gov2 collection which we tested before. As more documents need to be processed, the expected average query processing time can be longer and we expect the influence of time constraints will become more obvious. We test the five methods through the official 200 queries in TREC web track 2009 to 2012. We force each methods to stop at different time constraints and compare their performance by using the results effectiveness as MAP (Mean Average Precision) 1000. The results are summarized in table 7.7. Comparing to the similar comparison in table 7.4, the advantages of document prioritization (DP) and query simplification (QS) become more obvious. The large effectiveness improvement is not only shown in very tight time constraints (e.g. 15ms and 30ms), but also in loosen time constraints such as 50ms, 100ms and 200ms as well. This observation also confirms our hypnosis that forcing termination at tight time constraints may bring more influence in effectiveness for queries who requires longer query processing time and in this situation it is more important to select a wise effectiveness and efficiency trade off method. Since commercial search engines may use much larger data collections and require longer query processing time, we expect our analysis and proposed methods can make better impacts in real applications.

[1] http://http://www.lemurproject.org/clueweb09.php/

7.5 Conclusion

Only reducing average query processing time is not enough. To maximize user satisfaction, an IR system should also try to make sure that every query's processing time meets certain time constraints. Unfortunately, previous work did not pay enough attention to the problem. To solve it, we provide three rules for selecting good top-k query processing methods in a time-constrained environment. In addition, we proposed five methods; the experimental results confirm our theoretical analysis, and the method that follows the rules performs better than other methods under the same time constraints.

Table 7.3: Performance for different efficiency control methods on *TB06* collection

	Deadline = 15*ms*			*Deadline* = 30*ms*			*Deadline* = 50*ms*			*Deadline* = 100*ms*			*Deadline* = 200*ms*		
	MAP	AT	LR	MAP	AT	LR	MAP	AT	LR	MAP	AT	LR	MAP	AT	LR
Terminate Query Processing when it hits the deadline															
CUT	0.086	14.5	0%	0.165	28.3	0%	0.225	42.4	0%	0.270	59.6	0%	0.283	70.9	0%
Adjust cut-off threshold rate through binary decision															
F'=1	0.285	63.7	92%	0.285	63.8	64%	0.285	63.8	36%	0.285	63.8	14%	0.285	63.9	2%
F'=1.5	0.264	20.9	44%	0.267	25.0	22%	0.271	29.1	6%	0.279	42.3	4%	0.282	55.5	0%
F'=2	0.218	10.4	18%	0.221	15.3	6%	0.232	20.8	2%	0.267	37.5	4%	0.279	53.9	0%
F'=2.5	0.171	7.1	2%	0.180	12.4	4%	0.198	18.6	2%	0.257	36.6	4%	0.278	53.5	0%
F'=3	0.153	6.0	2%	0.166	11.5	6%	0.186	17.7	2%	0.255	36.1	4%	0.278	53.5	0%
F'=5	0.113	4.4	2%	0.140	10.3	4%	0.165	16.7	2%	0.252	35.7	4%	0.277	53.4	0%
F'=10	0.093	3.7	2%	0.137	9.8	6%	0.163	16.3	2%	0.245	35.3	4%	0.277	53.4	0%
Adjust cut-off threshold rate through linear function															
α=0.002	0.279	39.1	88%	0.279	41.4	54%	0.279	44.4	22%	0.281	50.6	6%	0.285	62.9	0%
α=0.005	0.265	26.0	72%	0.267	29.2	30%	0.273	34.3	8%	0.278	44.3	6%	0.282	58.8	0%
α=0.008	0.252	21.7	62%	0.257	25.1	20%	0.265	30.5	4%	0.273	40.6	4%	0.282	57.9	0%
α=0.01	0.245	19.3	54%	0.250	23.5	16%	0.258	28.8	2%	0.271	39.8	4%	0.282	57.6	0%
α=0.015	0.233	15.2	34%	0.239	21.2	12%	0.249	26.5	2%	0.268	38.9	4%	0.282	56.8	0%
α=0.02	0.203	12.9	26%	0.229	18.4	12%	0.240	24.6	2%	0.260	38.2	4%	0.282	56.4	0%
α=0.03	0.181	10.0	18%	0.204	16.2	10%	0.229	21.9	2%	0.257	37.5	4%	0.281	55.8	0%
α=0.05	0.157	8.0	12%	0.172	13.3	8%	0.204	19.7	2%	0.253	36.5	4%	0.281	55.3	0%
α=0.1	0.142	6.2	4%	0.159	11.2	6%	0.188	18.5	2%	0.247	35.8	4%	0.279	53.7	0%
Query Approximation															
QS	0.199	25.8	42%	0.250	31.8	24%	0.271	37.0	10%	0.281	50.5	8%	0.283	60.1	0%
Document Prioritization															
DP	0.283	28.1	52%	0.284	32.8	24%	0.286	36.7	10%	0.285	48.7	8%	0.285	59.5	0%

Table 7.4: Performance Comparison on strict time constraints

		15ms	30ms	50ms	100ms	200ms	no constraint
TB04	CUT	0.100	0.151	0.200	0.240	0.246	0.250
	Rate-B + CUT	0.143	0.192	0.202	0.232	0.249	0.250
	Rate-L + CUT	0.148	0.193	0.207	0.231	0.250	0.250
	QS + CUT	0.161	0.209	0.216	0.243	0.246	0.250
	DP + CUT	0.195	0.223	0.239	0.241	0.249	0.250
TB05	CUT	0.093	0.163	0.208	0.291	0.312	0.324
	Rate-B + CUT	0.186	0.244	0.269	0.306	0.319	0.324
	Rate-L + CUT	0.188	0.247	0.273	0.308	0.318	0.324
	QS + CUT	0.196	0.252	0.273	0.307	0.319	0.324
	DP + CUT	0.197	0.261	0.270	0.313	0.319	0.324
TB06	CUT	0.086	0.165	0.225	0.270	0.283	0.285
	Rate-B + CUT	0.191	0.229	0.264	0.277	0.282	0.285
	Rate-L + CUT	0.197	0.234	0.259	0.274	0.282	0.285
	QS + CUT	0.179	0.223	0.260	0.277	0.283	0.285
	DP + CUT	0.233	0.250	0.276	0.279	0.285	0.285

Table 7.5: Statistic Comparison when process is forced terminated at 15 ms

		# of Evals	Avg. score	Process
TB04	CUT	29,818	5.89	18%
	Rate-B + CUT	13,212	7.62	45%
	Rate-L + CUT	13,489	7.73	49%
	QS + CUT	30,349	6.59	59%
	DP + CUT	7,257	11.9	59%
TB05	CUT	33,227	5.60	18%
	Rate-B + CUT	13,313	7.31	53%
	Rate-L + CUT	13,680	7.31	55%
	QS + CUT	28,739	7.07	64%
	DP + CUT	10,374	11.0	62%
TB06	CUT	32,033	5.89	16%
	Rate-B + CUT	13,994	7.70	54%
	Rate-L + CUT	15,436	7.69	53%
	QS + CUT	33,876	6.83	68%
	DP + CUT	7,149	12.1	72%

Table 7.6: Performance Comparison on strict time constraints for long queries (≥ 5)

	15ms	30ms	50ms	100ms	200ms
CUT	0.012	0.037	0.091	0.124	0.151
RATE-B + CUT	0.030	0.053	0.066	0.077	0.125
RATE-L + CUT	0.026	0.042	0.053	0.076	0.137
DP + CUT	0.070	0.118	0.121	0.130	0.151
QS + CUT	0.072	0.111	0.146	0.142	0.145

Table 7.7: Performance Comparison on strict time constraints on larger collection

	15ms	30ms	50ms	100ms	200ms	500ms	no constraint
CUT	0.049	0.084	0.117	0.143	0.173	**0.185**	0.188
RATE-B + CUT	0.075	0.105	0.123	0.142	0.154	0.182	0.188
RATE-L + CUT	0.074	0.098	0.116	0.132	0.152	0.181	0.188
DP + CUT	**0.098**	**0.130**	**0.149**	**0.166**	**0.178**	**0.185**	0.188
QS + CUT	0.093	0.117	0.131	0.151	0.172	0.183	0.188

Chapter 8

CONCLUSION AND FUTURE WORK

Comparing to effectiveness, efficiency and usability are also important features of IR systems. If an IR system cannot satisfy certain efficiency requirements, users may lose patience and switch to another IR system. What is more, efficiency is also important for server health and energy saving. Maintaining reasonable query processing efficiency is one of the necessary requirements of running a commercial information retrieval system. Although a lot of efforts have been done to improve the speed of query processing, they do not solve the whole puzzle. In some situation such as withing a large number of returned documents, long queries, the query processing speed can be still slow and better methods are needed.

In addition to query processing efficiency, we also study another kind of IR efficiency: how fast users may implement different retrieval functions. Although existing IR toolkits, such as Indri [3] and Lucene [7], provide users the possibility to implement retrieval methods through their API, they are complicated and may not be suitable for education and research.

To solve these critical problems, first we introduce a novel toolkit called Virtual IR Lab. Virtual IR Lab applies a simple, but efficient and flexible architecture. By applying dynamic code generation and multiple optimization, it can help users to initiate various retrieval functions in a convenient and efficient way. Its friendly user interface and interesting features, such as a leading board, make it a good fit for both teaching and research. In addition to this, Virtual IR Lab also performs much faster than existed IR toolkits such as Indri [3] in the part of query processing. It is a good example that how we can improve query processing efficiency by applying simple but optimized architectures.

Parallel to optimizing in implementation, we also try to improve query processing efficiency by designing better query processing strategies. Focused on DAAT [50,97] query processing, we study why queries are slow. Specifically, we identified some important features and built an analytical model to monitor how these features can affect top-k query-processing time. The model proved to be more accurate and more salable than previous models.

Inspired by this novel model, we study some important problems of IR efficiency. First of all, existing dynamic pruning methods such as BMW and live block can reduce query processing greatly when k (i.e., number of returned documents) is small. However as k becomes large, such dynamic pruning methods are less efficient. To solve this problem, we designed a new document-prioritization method. The novel method prioritizes documents before grading them. By applying an efficient, decision-tree-based mechanism, the method can greatly reduce top-k query processing time with minimal effectiveness loss and it can be more than two times faster than strong baseline when k is large.

Second, query processing time increases exponentially as the query becomes longer. This weakness hinders the usage of some advanced techniques, such as pseudo-relevance feedback. To solve this problem, we proposed a novel incremental approach in which we used the results of an initial round of pseudo-relevance feedback to leverage the query processing of the second-round retrieval. By using the new method, we reduced the query-processing time of SAAT [18] mode with pseudo-relevance feedback by 2-4 times.

Designing this method is not enough. In order to meet efficiency requirements, we explored methods that sacrifice effectiveness in exchange for even shorter query-processing times. We provide three theorems to identify good methods, which can satisfy efficiency with small effectiveness loss. We then propose five different time-constrained top-k query-processing methods and use the theorems to analyze them. The experimental results match the analysis, and the two time-constrained methods, "document prioritization" and "query simplification," are the winners. We can use the

two methods to further improve efficiency with minimized effectiveness loss.

There are several interesting directions for future work: First, we can develop methods to improve efficiency without or with ignorable effectiveness loss for TAAT, DAAT and SAAT query processing. There are multiple direction we can do for it. One way is to follow the idea of dynamic pruning to skip more low score documents. In order to do so, we need additional features to better distinguish high score documents from low score documents. Another way is trying to reduce the costs of document evaluating. The Tie-breaking method [102, 109] shows a comparable effectiveness with simplified retrieval strategies and we may be able to reduce query processing time by applying these methods. In addition to improve query processing speed in single machine, we can borrow the power of parallel computing to improve query processing speed as well.

Second, we can explore methods for time-constrained query processing those can further reduce effectiveness loss. In particular, we can try methods in SAAT query processing. Since SAAT query processing is based on impact-sorted indices, the most impacted postings are usually processed first. Therefore, we expect SAAT query processing methods can perform well in time constrained environment.

Third, we can extend the features and functionality of the Virtual IR Lab. One important thing is to improve its ability of building indices from large collections. We may achieve this goal by optimizing low level I/O code or by applying parallel computing framework such as Hadoop [6]. In addition to that, we may consider add more features for Virtual IR Lab (e.g. term proximity, average term frequency in each documents and etc.) and it will help it support even more retrieval functions. Finally, we can incorporate distributed computing techniques to speed up query processing of Virtual IR Lab as well.

BIBLIOGRAPHY

[1] The clueweb09 dataset. http://lemurproject.org/clueweb09.php/.

[2] Gov2 test collection. http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm.

[3] Indri - language modeling meets inference networks. http://www.lemurproject.org/indri/.

[4] Ivory: A hadoop toolkit for web-scale information retrieval. http://lintool.github.io/Ivory/.

[5] Trec 2004 robust track guidelines. http://trec.nist.gov/data/robust/04.guidelines.html.

[6] Welcome to apacha hadoop. https://hadoop.apache.org/.

[7] Welcome to apache lucene. https://lucene.apache.org/.

[8] Welcome to the terrier ir platform. http://terrier.org/.

[9] James Allan. Relevance feedback with too much data. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '95, pages 337–343, 1995.

[10] Ismail S. Altingovde, Rifat Ozcan, and Özgür Ulusoy. Static index pruning in web search engines: Combining term and document popularities with query views. *ACM Trans. Inf. Syst.*, 30(1).

[11] Gianni Amati and Cornelis Joost Van Rijsbergen. Probabilistic models of information retrieval based on measuring the divergence from randomness. *ACM Trans. Inf. Syst.*, 20(4), October 2002.

[12] V. N. Anh and A. Moffat. Compressed inverted files with reduced decoding overheads. In *Proceedings of SIGIR'98*, 1998.

[13] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. In *Proceedings of SIGIR'01*, 2001.

[14] Vo Ngoc Anh and Alistair Moffat. Impact transformation: efffective and efficient web retrieval. In *Proceedings of SIGIR'02*, 2002.

[15] Vo Ngoc Anh and Alistair Moffat. Index compression using fixed binary codewords. In *Proceedings of the 15th Australasian Database Conference - Volume 27*, ADC '04, 2004.

[16] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1), January 2005.

[17] Vo Ngoc Anh and Alistair Moffat. Simplified similarity scoring using term ranks. In *Proceedings of SIGIR'05*, 2005.

[18] Vo Ngoc Anh and Alistair Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of SIGIR'06*, 2006.

[19] Jaime Arguello, Fernando Diaz, Jamie Callan, and Ben Carterette. A methodology for evaluating aggregated search results. In *Advances in information retrieval*, pages 141–152. Springer, 2011.

[20] Nima Asadi and Jimmy Lin. Fast candidate generation for two-phase document ranking: Postings list intersection with bloom filters. In *Proceedings of CIKM'12*, 2012.

[21] Nima Asadi and Jimmy Lin. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In *Proceedings of SIGIR'13*, 2013.

[22] Ricardo Baeza-yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silverstri. The impact of caching on search engines. In *Proceedings of SIGIR'07*, 2007.

[23] Luiz Andre Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: the google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[24] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, 2006.

[25] Bodo Billerbeck and Justin Zobel. Efficient query expansion with auxiliary data structures. *Information Systems*, 31(7):573–584, 2006.

[26] Carolina Bonacic, Carlos Garcia, Mauricio Marin, Manuel Prieto-Matias, and Francisco Tirado. Building efficient multi-threaded search nodes. In *Proceedings of CIKM'10*, 2010.

[27] Peter Brass. *Advanced data structures*. Cambridge University Press Cambridge, 2008.

[28] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2003.

[29] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of CIKM'03*, 2003.

[30] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, pages 426–434, 2003.

[31] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, 1994.

[32] Jake D. Brutlag, Hilary Hutchinson, and Maria Stone. User preference and search engine latency. In *JSM Proceedings, Qualtiy and Productivity Research Section.*, 2008.

[33] C. Buckley. Automatic query expansion using SMART: Trec-3. In D. Harman, editor, *Overview of the Third Text Retrieval Conference (TREC-3)*, pages 69–80, 1995. NIST Special Publication 500-225.

[34] Chris Buckley and Stephen Robertson. Relevance feedback track overview: Trec 2008. In *Proceedings of TREC'08*, 2008.

[35] C. Burckley and A. Lewit. Optimizations of inverted vector searches. In *Proceedings of SIGIR'85*, 1985.

[36] Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin. Earlybird: Real-time search at twitter. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, 2012.

[37] S. Buttcher and C. L. A. Clarke. The trec 2006 terabyte track. In *Proceedings of TREC'06*, 2006.

[38] Stefan Büttcher and Charles L. A. Clarke. A document-centric approach to static index pruning in text retrieval systems. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM '06, 2006.

[39] Guihong Cao, Jian-Yun Nie, Jianfeng Gao, and Stephen Robertson. Selecting good expansion terms for pseudo-relevance feedback. In *Proceedings of SIGIR'08*, 2008.

[40] David Carmel, Doron Cohen, Ronald Fagin, Eitan Farchi, Michael Herscovici, Yoelle S. Maarek, and Aya Soffer. Static index pruning for information retrieval systems. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, 2001.

[41] Ben Carterette. System effectiveness, user models, and user utility: a conceptual framework for investigation. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 903–912. ACM, 2011.

[42] Marc-Allen Cartright, James Allan, Victor Lavrenko, and Andrew McGregor. Fast query expansion using approximations of relevance models. In *Proceedings of the CIKM'10*, 2010.

[43] Kaushik Chakrabarti, Surajit Chaudhuri, and Venkatesh Ganti. Interval-based pruning for top-k processing over comparessed lists. In *Proceedings of ICDE'11*, 2011.

[44] Junghoo Cho and Sourashis Roy. Impact of search engines on page popularity. In *Proceedings of the 13th international conference on World Wide Web*, pages 20–29, 2004.

[45] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, 1977.

[46] Kevyn Collins-Thompson and Jamie Callan. Estimation and use of uncertainty in pseudo-relevance feedback. In *Proceedings of SIGIR'07*, 2007.

[47] W Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: Information retrieval in practice*, volume 283. 2010.

[48] Joshua V. Dillon and Kevyn Collins-Thompson. A unified optimization framework for robust pseudo-relevance feedback algorithms. In *Proceedings of CIKM'10*, 2010.

[49] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. A candidate filtering mechanism for fast top-k query processing on modern cpus. In *Proceedings of SIGIR'13*, 2013.

[50] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of SIGIR'11*, 2011.

[51] Caroline M Eastman and Bernard J Jansen. Coverage, relevance, and ranking: The impact of query operators on web search engine results. *ACM Transactions on Information Systems (TOIS)*, 21(4):383–411, 2003.

[52] Ronald Fagin. Combining fuzzy information: an overview. *ACM SIGMOD Record*, 31(2), 2002.

[53] Hui Fang, Tao Tao, and ChengXiang Zhai. A formal study of information retrieval heuristics. In *Proceedings of SIGIR-04*, 2004.

[54] Hui Fang, Hao Wu, Peilin Yang, and ChengXiang Zhai. Virlab: A web-based virtual lab for learning and studying information retrieval models. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, pages 1249–1250, 2014.

[55] Hui Fang and ChengXiang Zhai. Web search relevance feedback. In *Encyclopedia of Database Systems*, pages 3493–3497. 2009.

[56] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *Proceedings of WWW'09*, 2009.

[57] Elke Greifender and Mark-Shane Scale. Facebook as a social search engine and the implications for libraries in the twenty-first century. *Library Hi Tech*, 26(4):540–556, 2008.

[58] Ben HE and Iadh Ounis. A study of parameter tuning for term frequency normalization. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, pages 10–16, 2003.

[59] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), October 2008.

[60] Bernard J Jansen and Amanda Spink. How are we searching the world wide web? a comparison of nine search engine transaction logs. *Information Processing & Management*, 42(1):248–263, 2006.

[61] John Lafferty and Chengxiang Zhai. Probabilistic relevance models based on document and query generation. In *Language modeling for information retrieval*, pages 1–10. 2003.

[62] Amy N Langville and Carl D Meyer. *Google's PageRank and beyond: The science of search engine rankings*. 2011.

[63] Victor Lavrenko and James Allan. Real-time query expansion in relevance models. Technical Report IR-473, University of Massachusetts Amherst, 2006.

[64] Victor Lavrenko and Bruce Croft. Relevance-based language models. In *Proceedings of SIGIR'01*, pages 120–127, Sept 2001.

[65] Kyung-Soon Lee, W. Bruce Croft, and James Allan. A cluster-based resampling method for pseudo-relevance feedback. In *Proceedings of SIGIR'08*, 2008.

[66] Yong Kyu Lee, Seong-Joon Yoo, Kyoungro Yoon, and P. Bruce Berra. Index structures for structured documents. In *Proceedings of the First ACM International Conference on Digital Libraries*, DL '96, 1996.

[67] Yuanhua Lv and ChengXiang Zhai. Positional relevance model for pseudo-relevance feedback. In *Proceedings of SIGIR'10*, 2010.

[68] Yuanhua Lv, ChengXiang Zhai, and Wan Chen. A boosting approach to improving pseudo-relevance feedback. In *Proceedings of SIGIR'11*, 2011.

[69] Craig Macdonald, Rodrygo L. T. Santos, and Iadh Ounis. The whens and hows of learning to rank for web search. *Information Retrieval*, 2012.

[70] Craig Macdonald, Nivola Tonellotto, and Iadh Ounis. Learning to predict response times for online query scheduling. In *Proceedings of SIGIR'12*, 2012.

[71] A. MacFarlane, J.A. McCann, and S.E. Robertson. Parallel search using partitioned inverted files. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 209–220, 2000.

[72] A. MacFarlane, S. E. Robertson, and J. A. McCann. Parallel computing in information retrieval - an updated review. *Journal of Documentation*, 53:274315, January 1997.

[73] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[74] W. D. Maurer and T. G. Lewis. Hash table methods. *ACM Comput. Surv.*, pages 5–19, 1975.

[75] Jun Miao, Jimmy Huang, and Zheng Ye. Proximity-based rocchio's model for pseudo relevance. In *Proceedings of SIGIR'12*, 2012.

[76] Allstair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, 1996.

[77] Craig Macdonald Nicola Tonellotto and Iadh Ounis. Efficient and effective retrieval using selective pruning. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, 2013.

[78] Alexandros Ntoulas and Junghoo Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, 2007.

[79] Iadh Ounis, Craig Macdonald, Jimmy Lin, and Ian Soboroff. Overview of the trec 2011 microblog track. In *Proceedings of TREC'11*, 2011.

[80] M. F. Porter. Readings in information retrieval. chapter An Algorithm for Suffix Stripping, pages 313–316. 1997.

[81] S. Robertson and K. Sparck Jones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27:129–146, 1976.

[82] S. E. Robertson, S. Walker, S. Jones, M. M.Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *Proceedings of TREC-3*, 1995.

[83] J. Rocchio. Relevance feedback in information retrieval. In *The SMART Retrieval System: Experiments in Automatic Document Processing*, pages 313–323. Prentice-Hall Inc., 1971.

[84] Cristian Rossi, Edleno Silva de Moura, Andre Luiz Carvalho, and Altigran Soares da Silva. Fast document-at-a-time query processing using two-tier indexes. In *Proceedings of SIGIR'13*, 2013.

[85] G. Salton and C. Buckley. Improving retrieval performance by relevance feedback. *Journal of the American Society for Information Science*, 44(4):288–297, 1990.

[86] Gerard Salton, editor. *The SMART Retrieval System - Experiments in Automatic Document Processing*. Prentice Hall, Englewood, Cliffs, New Jersey, 1971.

[87] Eric Schurman and Jake Brutlag. Performance related changes and their user impact. In *Velocity - Web performance and operations conference*, 2009.

[88] Amit Singhal. Modern information retrieval: A brief overview. 2001.

[89] Trevor Strohman. *Efficient processing of complex features for information retrieval*. PhD thesis, University of Massachusetts Amherst, 2007.

[90] Trevor Strohman and Bruce W. Croft. Efficient document retrieval in main memory. In *Proceedings of SIGIR'07*, 2007.

[91] Trevor Strohman, Howard Turtle, and Bruce W. Croft. Optimization strategies for complex queries. In *Proceedings of SIGIR'05*, 2005.

[92] Daisuke Takuma and Hiroki Yanagisawa. Faster upper bounding of intersection sizes. In *Proceedings of SIGIR'13*, 2013.

[93] Bin Tan, Atulya Velivelli, Hui Fang, and ChengXiang Zhai. Term feedback for information retrieval with language models. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, 2007.

[94] Tao Tao and ChengXiang Zhai. Regularized estimation of mixture models for robust pseudo-relevance feedback. In *Proceedings of SIGIR'06*, 2006.

[95] Shirish Tatikonda, B. Barla Cambazoglu, and Flavio P. Junqueira. Posting list intersection on multicore architectures. In *Proceedings of SIGIR'11*, 2011.

[96] Anthony Tomasic, Héctor García-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, 1994.

[97] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing & Management*, 31(1):831–850, 1995.

[98] Lidan Wang, Jimmy Lin, and Donald Metzler. Learning to efficiently rank. In *Proceedings of SIGIR'10*, 2010.

[99] Lidan Wang, Jimmy Lin, and Donald Metzler. A cascade ranking model for efficient ranked retrieval. In *Proceedings of SIGIR'11*, 2011.

[100] Xuanhui Wang, Hui Fang, and ChengXiang Zhai. A study of methods for negative relevance feedback. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, pages 219–226, 2008.

[101] Xuanhui Wang, Hui Fang, and ChengXiang Zhai. A study of methods for negative relevance feedback. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, pages 219–226, 2008.

[102] Yue Wang, Hao Wu, and Hui Fang. An exploration of tie-breaking for microblog retrieval. In *Advances in Information Retrieval*, pages 713–719. 2014.

[103] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

[104] H Wu, H Fang, SJ Stanhope, et al. Exploiting online discussions to discover unrecognized drug side effects. *Methods Inf Med*, 52(2):152–9, 2013.

[105] Hao Wu and Hui Fang. An exploration of new ranking strategies for medical record tracks. In *TREC*, 2011.

[106] Hao Wu and Hui Fang. An exploration of query term deletion. *Carterette et al.[2]*, 2011.

[107] Hao Wu and Hui Fang. Concept detection and using concept in ad-hoc of microblog search. Technical report, DTIC Document, 2012.

[108] Hao Wu and Hui Fang. Relation based term weighting regularization. In *Advances in Information Retrieval*, pages 109–120. 2012.

[109] Hao Wu and Hui Fang. Tie breaker: A novel way of combining retrieval signals. In *Proceedings of the 2013 Conference on the Theory of Information Retrieval*, ICTIR '13, 2013.

[110] Hao Wu and Hui Fang. Analytical performance modeling for top-k query processing. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, pages 1619–1628, 2014.

[111] Hao Wu and Hui Fang. Document prioritization for scalable query processing. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, 2014.

[112] Hao Wu, Hui Fang, and Steven J. Stanhope. An early warning system for unrecognized drug side effects discovery. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, pages 437–440, 2012.

[113] Jiafu Xu, Zhijian Wang, and Chengxiang Zhai. *Object-oriented Programming Language*. Nanjing University Press, January 1993. in Chinese.

[114] Yang Xu, Gareth J. F. Jones, and Bin Wang. Query dependent pseudo-relevance feedback based on wikipedia. In *Proceedings of SIGIR'09*, 2009.

[115] Hao Yan, Shuai Ding, and Torsten Suel. Compressing term positions in web indexes. In *Proceedings of SIGIR'09*, 2009.

[116] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 401–410, 2009.

[117] J. Yang and J. Leskovec. Temporal variation in online media. In *Proceedings of WSDM'11*, 2011.

[118] Peilin Yang, Hongning Wang, Hui Fang, and Deng Cai. Opinions matter: a general approach to user profile modeling for contextual suggestion. *Information Retrieval Journal*, 18(6):586–610, 2015.

[119] Chengxiang Zhai and John Lafferty. The dual role of smoothing in the language modeling approach. In *Proceedings of the Workshop on Language Modeling and Information Retrieval*, 2001.

[120] Chengxiang Zhai and John Lafferty. Model-based feedback in the KL-divergence retrieval model. In *Tenth International Conference on Information and Knowledge Management (CIKM 2001)*, pages 403–410, 2001.

[121] Chengxiang Zhai and John Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, 2001.

[122] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list cachign in search engines. In *Proceedings of WWW'08*, 2008.

[123] Wei Zheng and Hui Fang. A study of pattern-based suptopic discovery and integration in the web track. In *TREC*, 2011.

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

[125] Justin Zobel and Alistair Moffat. Exploring the similarity space. *SIGIR Forum*, 32(1), April 1998.