Implementation of a Search Engine with Rank-Safe Query Optimization Algorithms

THESIS

Submitted in Partial Fulfillment

of the Requirements for the

Degree of

MASTER OF SCIENCE (Computer Science)

at the

POLYTECHNIC INSTITUTE OF NEW YORK UNIVERSITY

by

Roman Khmelichek

June 2011

Approved:

_____
Advisor Signature

_____
Date

_____
Department Head Signature

_____
Date

Copy No. _____

Vita

Roman Khmelichek was born in Feodosiya, Ukraine on June 8, 1988 and moved to New York City with his family in December 1996. Prior to enrolling at Polytechnic Institute of NYU (Polytechnic University at the time), he attended Brooklyn Technical High School, majoring in Computer Science; it was there that he first discovered his passion for programming, initially making games in Java such as a clone of Puzzle Bubble, and a souped-up version of Snake. At NYU-Poly, he was part of the Honors Program, which allowed him to earn both a BS in Computer Engineering and an MS in Computer Science in an accelerated time frame. While at NYU-Poly, he was the designer, programmer, and administrator of the Honors website. He also participated in the undergraduate research program under Professor Joel Wein, where he and his peers collaborated on a project which resulted in a paper published in SIGCSE 2009, entitled *Virtualized Games for Teaching About Distributed Systems*.

The research for this thesis was conducted from Fall 2009 to Spring 2011 under the guidance of Professor Torsten Suel, a faculty member in the Department of Computer Science and Engineering at Polytechnic Institute of NYU.

For my mother, Marina Khmelichek, for her support during the course of this work

and

In memory of my father, Aleksandr Khmelichek

Acknowledgements

I would like to thank my thesis advisor, Professor Torsten Suel. I was first introduced to his passion and knowledge while taking his Search Engines course, and this motivated me to pursue a thesis topic in the search/information retrieval field. It was a wise choice, as Professor Suel provided both a challenging topic and generous guidance that allowed me to complete this work.

Abstract

Implementation of a Search Engine with Rank-Safe Query Optimization Algorithms

by

Roman Khmelichek

Advisor: Torsten Suel, Ph.D.

Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science (Computer Science)

June 2011

In this thesis, we discuss the design and implementation of an information retrieval toolkit capable of efficiently handling large document collections. We show how our indexing pipeline is able to take advantage of machines with large memories to speed up indexing, examine the on-disk data structures and the index layout to achieve fast querying speeds, and explore other additional features of our framework. We conduct experiments on the 426 GB TREC Gov2 dataset to show the indexing and querying performance of our toolkit, and discuss the result effectiveness on TREC ad-hoc topics.

Our information retrieval toolkit uses the Okapi BM25 ranking measure based on the $TF \cdot IDF$ principle. Search engines often incorporate complex ranking functions, but they are expensive to calculate over the entire document collection. Instead, the search engine could get the top-m highest scoring documents using a simple function, such as BM25, for a large value of m, and then apply the complex ranking function over the top-m to get the final top-k results for presentation to the user. In such a use case, the recall of the top-m results must be high so as to not disqualify relevant documents that could achieve high scores under the complex ranking function. Disjunctive (OR-type) queries achieve the best recall, but are also much slower to evaluate than conjunctive (AND-type) queries. We implement and discuss *rank-safe* query optimization algorithms to speed up disjunctive queries without any degradation in result quality, compare their performance on a TREC query log, and show how they scale to different query lengths. Our experiments for top-k retrieval include values of k that range from 10 to 10,000 to cover various information-retrieval use cases.

Contents

List of Figures

List of Tables

1 Introduction

Information retrieval is a field that deals with searching a large corpus of documents to find a small set of the most relevant documents and the information contained within. The Internet, with its abundance of information has caused users to turn to search engines to find the content they are looking for. Thus, search engines have the challenging task of providing relevant results over billion page document collections to millions of queries issued per day, with query response times of well under a second. While Internet-scale search engines present the most extreme challenges, our work is just as applicable to smaller search and information retrieval systems, which often still have to deal with a large amount of content and a heavy query load.

In this thesis, Chapter 2 starts off with a general discussion of information retrieval and search engines, defines common terminology used throughout this work, and briefly looks at other open-source search engine toolkits. We then discuss the design and implementation of the Polytechnic Information Retrieval Toolkit (which we refer to as PolyIRTK, as a shorthand, from now on), in Chapter 3. In Chapter 4, we shift our focus to rank-safe query optimization algorithms, where we use PolyIRTK as a common framework for experimenting with different query algorithms. For readers familiar with information retrieval who are interested only in the chapter on rank-safe query optimization, it may prove useful to first peruse the common terminology we define in Chapter 2 before venturing to the chapter on query algorithms. We define our testing methodology and system environment in Chapter 5 and explore the performance of our toolkit in Chapter 6. Finally, we suggest future work and improvements to our toolkit in Chapter 7.

1.1 Information Retrieval Toolkit Design

The purpose of PolyIRTK is to be able to efficiently index and search large web document collections, while being extensible and configurable. We achieve this by

learning from previous efforts, carefully designing our data structures, and utilizing the most recent index compression algorithms. Our toolkit includes a configuration file, allowing users to tweak memory consumption, select compression algorithms, and modify other index and query time details.

We leave the preliminary discussion of information retrieval for Chapter 2 and the details of our implementation for Chapter 3, where we discuss our indexing pipeline, examine the on-disk data structures and the index layout to achieve fast querying speeds, and other additional features of our toolkit. Our experiments, the results of which are discussed in Section 6.1, are conducted on the 426 GB TREC Gov2 dataset to show the indexing and querying performance of our toolkit as well as the result effectiveness on TREC ad-hoc topics.

All code for this project is available under the New BSD License in an online repository[1], along with an issue tracker, documentation, and future direction of this project.

1.2 Query Optimization Algorithms

Search engines often incorporate complex ranking functions, but they are expensive to calculate over the entire document collection. Instead, the search engine could get the top-m highest scoring documents using a simple function, for a large value of m, and then apply the complex ranking function over the top-m to get the final top-k results for presentation to the user. In such a use case, the recall of the top-m results must be high so as to not disqualify relevant documents that could achieve high scores under the complex ranking function. Disjunctive (OR-type) queries, where any of the terms can appear in a candidate document, achieve the best recall for top-m evaluation, but are also much slower to evaluate than conjunctive (AND-type) queries, where all of the query terms must appear in a candidate document. In Chapter 4, we discuss *rank-safe* query optimization algorithms to speed up disjunctive queries without any degradation in result quality. In Section 6.2, we compare their performance on a

[1] http://code.google.com/p/poly-ir-toolkit/

TREC query log, as well as show how they scale to different query lengths. Our experiments for top-k retrieval include values of k that range from 10 to 10,000 to evaluate use of PolyIRTK as a standalone information retrieval system, as well as the case of feeding the results into another, complex ranking function.

2 Preliminaries

This chapter discusses information retrieval background and introduces common terminology that will be encountered in later sections.

2.1 Index Data Structures

All the major search engines use an *inverted index* structure to answer queries because its design allows a query system to quickly find all occurrences of a term across the entire document collection. The *vocabulary* of a document corpus is the set of all unique words that the system chooses to recognize as valid (say, any strings separated by whitespace). Each term in the vocabulary maps to an *inverted list*, which is a collection of document identifiers, stored in sorted order, representing the documents in which that term appears at least once. The inverted index is composed of the set of all inverted lists. This setup is enough to answer a simple query by looking up the inverted list for each query term and merging the lists of document identifiers for all the terms. As a final step for presentation to the user, the document identifiers would be converted into meaningful strings, such as document URLs. However, this simplistic view ignores many of the issues that search engines encounter, such as building inverted indices for terabyte sized document collections, ranking the documents by relevancy to the query, and of course answering the query very quickly — within a fraction of a second.

Typically, document identifiers are integers that are assigned during the index construction stage; we will refer to them as *docIDs*. They can be assigned sequentially as documents are picked for inclusion into the index. However, other assignment schemes are also used; for example, in Section 4.1, we discuss a query algorithm that relies on docIDs being assigned according to the rank of a document by its PageRank score [1] (a link-based ranking), and in Section 3.5, we discuss the benefits of assignment of docIDs by their sorted document URLs. PolyIRTK assigns docIDs sequentially

during indexing in order to compactly store the docIDs, but has options to reassign docIDs after the index is built.

A complete index has at least three structures:

- The **inverted index** is the primary data structure for answering queries, which holds sorted lists of docIDs for each term in the document collection. It resides on disk, and is usually partially or fully cached in main memory for performance reasons.
- The **lexicon** or the vocabulary is the data structure used to look up the start of an inverted list in the index, given a term. It can also hold additional information such as the number of documents containing a term, for ranking purposes (say, how rare a term is) and traversal of the inverted lists. The lexicon can be loaded into main memory upon startup into a hash table, binary search tree, or into a sorted array (which can be binary searched). If disk seeks to look up terms are acceptable, it can reside on disk, such as in a B-tree, or it can be partially cached in memory for the most commonly queried terms.
- The **document map** is a mapping of the numeric docIDs to the associated documents' information, such as the URL, length in words, language, crawl date, etc. Often, it is acceptable to load the document map into main memory, since information such as document URLs tends to compress very well, especially if documents from the same host are stored close together.

2.2 Document Ranking

An inverted list is not just a series of docIDs, but also includes additional information useful for the final ranking of documents. Indices typically include the term frequencies for each document, for use with a $TF \cdot IDF$ based ranking function. The $TF \cdot IDF$ ranking measure is the partial ranking of the document for each query

keyword, which is a combination of the term frequency (TF) and the inverse document frequency (IDF), usually multiplied together; the summation of the $TF \cdot IDF$ score for each query keyword would result in the final score of a document. The TF component increases for documents in which a term appears more often, but it is usually normalized by the document length, so that short documents are not unfairly penalized. The IDF component is used to discount the contribution of a common query term appearing in a document, and likewise, gives more weight to rare terms; its growth is usually inhibited by the use of a $log()$ function so that extremely rare keywords are not given excessive weight. An example of a $TF \cdot IDF$ based ranking function is BM25 (see Subsection 3.4.1), which is utilized by PolyIRTK for all its document ranking.

A traditional information retrieval ranking formula is the cosine similarity measure, based on the vector space model, which views the query and documents as vectors having dimensions equal to the size of the whole corpus vocabulary. The vectors themselves are composed of the term frequencies. The cosine measure can then be seen as the cosine of the angle between the document and query vectors, resulting in a value of 1 when the vectors are perfectly aligned (angle is 0°) and 0 when the vectors have no terms in common (angle is 90°). The cosine measure can thus be defined as below, using the formula for Euclidean distance between two vectors.

$$cos\theta = \frac{w_d \cdot w_q}{|w_d|\,|w_q|}$$

Here, w_q and w_d are the query and document vectors, both of which are of length n, the size of the corpus vocabulary. The numerator takes the dot product of the two vectors, but does not distinguish between rare and common keywords and the denominator normalizes the score by the magnitude of both vectors. However, the cosine measure usually incorporates the IDF for the term weights and normalizes the TF component, resulting in a formula such as the one below [2], where the numerator

is a summation over the set of query terms Q that exist in a document D_d.

$$cosine\_score(Q, D_d) = \frac{\sum_{t \in Q \cap D_d} \log(1 + \frac{N}{f_t}) \cdot (1 + \log(f_{d,t}))}{\sqrt{\sum_{t=1}^{n} w_{q,t}^2} \cdot \sqrt{\sum_{t=1}^{n} w_{d,t}^2}}$$

In the above expression, f_t is the number of documents containing the query term t, N is the total number of documents in the collection, and $f_{d,t}$ is the frequency of the query term t in document d. As before, the denominator computes the magnitude of both vectors, and it serves as the length normalization. Also notice that the numerator now uses an IDF component ($\log(1+\frac{N}{f_t})$) and a TF component ($1+\log(f_{d,t})$), instead of treating both the document and query vectors equally.

Very common terms are called *stop words*; their inverted lists are very long, thus they are expensive to process, while bringing little relevancy information. A search engine could choose to completely skip indexing them, but at the risk of missing phrases and names that contain stop words. The costs of processing queries containing stop words can be significantly reduced with the query optimization algorithms we discuss later. Another technique search engines use is called *stemming*, which converts terms into their root form, by removing the suffix. As an example, the word *search* would be the stemmed form of the words *search*, *searches*, *searched*, and *searching*, when using the Snowball English stemmer. Stemming happens at both query and indexing time, and is a form of query expansion. Stemming algorithms for many languages are available from the Snowball project[1]. PolyIRTK does not implement stemming, but it has the capability to ignore stop words at query time.

[1] http://snowball.tartarus.org/

2.3 Index Compression

The inverted lists store *postings*, which are tuples of the form <docID, frequency, <positions>>. An index that includes the positions is called a *word-level* index, and it stores a list of word offsets for each docID within a term's inverted list. Such an index is significantly larger than one containing only docIDs and frequencies, but positions can be used as a filter (e.g. disqualifying documents whose query terms are not within n words of each other) or as an addition to a $TF \cdot IDF$ ranking function (e.g. including a measure of the proximity of the terms within a document).

Given the requirement to store docIDs, frequencies, and possibly positions, an uncompressed inverted index would likely be larger than the whole compressed (say, with gzip) document corpus. With the large sizes of web collections, index compression is used to reduce the storage requirements. The benefits are decreased transfer time from disk to main memory, the ability to cache more of the index in main memory, and decreased pressure on the available memory bandwidth. Of course, there are decoding overheads and due to the way lists are compressed, there must be ways to seek within a list without decompressing the undesired parts.

Typically, an inverted list will have its docIDs sorted in increasing order, so only the differences between subsequent docIDs need to be stored; these are called *d-gaps*. The same thing can be done with term positions within a single document, which are called *p-gaps*. The d-gaps and p-gaps allow for better compression since the resulting integers are smaller. Research into index compression algorithms aims to make decompression fast, while still minimizing the index size. Next, we will discuss several notable compression algorithms.

An older, well-known compression algorithm is Golomb coding [2]. Under this coding method, a parameter b is chosen to divide an integer n into its quotient $q = \lfloor n \,/\, b \rfloor$ and remainder $r = n \% b$. The quotient q is stored using a unary code (q ones followed by a zero) and the remainder is stored as a binary number using either $\lfloor log_2(b) \rfloor$ or

$\lceil log_2(b) \rceil$ bits, depending on the magnitude of r. An implementation of Golomb coding that restricts b to powers of two is called Rice coding; it is a slightly less optimal code, but allows for a more efficient implementation due to the $log_2(b)$ fixed-size binary bits and efficient multiplication by two through bit shifting (although, we note that multiplication performance is comparable to bit shifting on many modern processors). While Golomb and Rice coding provide very good compression, newer compression techniques offer significantly faster decoding speeds; this is mainly due to the branching during the decoding of the unary part for Golomb codes. We note that Rice coding can also be implemented in a way that shares similarities with PForDelta (described below), which the authors call Turbo-Rice coding [3], resulting in a significant speedup over the traditional Rice implementation. By placing the unary and binary portions of a group of integers (multiples of 32) in separate arrays, the authors were able to process the unary portion 8-bits at a time by doing a switch on all 256 possible cases, and adjusting the corresponding fixed-width binary portions (by adding a factor of b) appropriately for each case.

Rice and Golomb coding use a parameter b, which can be local (chosen for an inverted list) or global (chosen for the entire index). The parameter b can be chosen globally by the formula $0.69 \cdot \frac{N \cdot n}{f}$ [2], where f is the total number of <docID, term> pairs in the index, N is the total number of documents in the collection, and n is the number of unique terms in the collection. This is an approximation of the optimal value of b if the docIDs in the index follow a geometric distribution, describing a Bernoulli process with a success probability of $\frac{f}{N \cdot n}$ (a document picked at random contains some randomly chosen term from the collection). A similar idea can be applied to get b locally for each inverted list, so that lists for frequent terms use a lower value for b, calculated the same way as before, but now using $n = 1$ and $f = f_t$, where f_t is the frequency of a particular term in the collection.

Unfortunately, the Bernoulli model does not take into account docID clustering within a list (e.g. groups of documents assigned nearby docIDs that contain the same terms).

Taking advantage of clustering is especially useful if documents are assigned docIDs in sorted URL order, as discussed in Section 3.5. Since the parameter b of Golomb coding is fixed for a group of integers, any clustering within that group is not being exploited. The Skewed Bernoulli model [2] is an optimization for Golomb coding, to take advantage of clustering within a list. This model picks the value of b for each list as the median of all the d-gaps. Since a skewed docID distribution (where docIDs are clustered, resulting in many small d-gaps) would have a median value smaller than a random distribution of docIDs, this choice of b results in better compression. Within the Bernoulli model, this can be viewed as boosting the probability of encountering small d-gaps, and so, coding them with a smaller binary portion.

The approach taken in PolyIRTK (which does not include Golomb coding, but rather a Rice coding option) is to store the value of $log_2(b)$ for each chunk (composed of 128 integers), which is calculated as the closest power of two to the average of the integers (d-gaps, frequencies, or p-gaps) contained in the chunk. This might be better chosen as the median, to take advantage of docID clustering (especially in the case of docID reassignment), however, we only utilized Rice coding for positions in our experiments, with PForDelta coding (discussed below) being used for docIDs, since it provides significantly faster performance at only a slight increase in index size.

Elias Gamma coding takes an integer n and codes it in two parts, the first being the value $\lfloor log_2(n) \rfloor + 1$ in unary and the second part being the number $n - 2^{\lfloor log_2(n) \rfloor}$ using $\lfloor log_2(n) \rfloor$ bits; thus, the first part serves to tell how many binary bits are used to code the number (excluding the most significant bit, which is always one). Elias Delta coding is similar to Gamma coding, but additionally applies Gamma coding to the value of the unary part. These codings provide good compression for smaller numbers (such as frequencies). Unfortunately, decompression is relatively slow due to branches in decoding the unary part, and the binary part being different in length for each integer.

Variable Byte coding has been widely used in information retrieval systems because its

decoding speed is fast and the implementation is simple; however, compression suffers because it operates at the byte level — due to this coarseness, it cannot effectively take advantage of docID clustering. Variable Byte coding takes an unsigned integer and transforms it into a series of bytes; each byte has a value from 0 to 127, with the most significant bit reserved as a flag, where a 0 indicates the end of the byte sequence. Given a series of encoded bytes, an unsigned integer can be decoded by $\sum_{i=0}^{j} 128^i \cdot B_i$, where B_i represents the byte value (disregarding the flag bit), and j is the index of the byte with a flag of 0. As can be seen, Variable Byte coding still requires at least one branch per integer, but this is typically less than the Golomb and Gamma codes.

PolyIRTK uses new compression methods that are significantly faster than the more traditional methods, including Variable Byte coding. These coding techniques decompress blocks of integers at a time and reduce the number of branch instructions executed, thus keeping the CPU pipeline full; they provide good compression and are very fast to decode. One such coding technique is Simple9 (S9) [3], which uses 9 cases to pack as many integers as possible into a 4-byte word. S9 would store which case is used using 4 status bits, followed by either of the 9 cases, which are 28 1-bit numbers, 14 2-bit numbers, 9 3-bit numbers, 7 4-bit numbers, 5 5-bit numbers, 4 7-bit numbers, 3 9-bit numbers, 2 14 bit numbers, or 1 28-bit number. During decoding, the correct case to unpack the integers can be selected by doing a switch statement on the status bits. This means only one branch is required to decode several integers in most cases. An improved compression method based on S9, is the Simple16 (S16) coding method [3]. S16 includes more cases (a total of 16, thus fully making use of the 4 status bits) for compressing groups of small integers — which is especially useful for encoding small gaps produced by clustering of docIDs.

Another coding method, PForDelta [3], compresses blocks of n integers, where n is a multiple of 32. It chooses a fixed-size b such that most of the n integers are smaller than 2^b. The integers that are larger than 2^b are stored in a separate *exceptions* array,

with each exception integer taking up a fixed number of bits (implementations could either use 32-bits or choose 8, 16, or 32-bits, depending on the value of the largest exception). PForDelta then codes the n integers (with the values for the exceptions specially handled) into an array using $b \cdot n$ bits (the reason for compressing blocks of a multiple of 32 is so that the $b \cdot n$ bits would fit exactly on a word boundary, without wasting space). Within this array of n integers, each of fixed-size b bits, the slots that correspond to the exceptions store an offset to the next exception slot. The exceptions can then be identified by following the chain of offsets, forming a linked list (the offset of the first exception can be stored along with the value of b as part of the metadata). Note that since the offsets must be stored within b bits, additional exception slots must sometimes be inserted so that the exceptions are never more than 2^b slots apart. The compressed size and decoding speed of PForDelta depends on the number of exceptions that need to be decoded, which can be tweaked by relaxing the value of b.

The PForDelta implementation described above could also be modified to take even better advantage of docID clustering [4] by coding the exception offsets in a separate array (compressed with S16) and storing the lower bits of the exception integers within the array of size $b \cdot n$ bits. The upper bits of the exception integers are stored in another array, also compressed with S16. This modification to PForDelta solves a problem with the previously described version, where exceptions more than 2^b slots apart would need additional exceptions forced in-between. Since clustering is likely to produce many small d-gaps, this version allows the algorithm to make use of small values of b more effectively.

2.4 Answering Queries

Answering queries has three major costs: fetching the inverted lists corresponding to the query terms from disk (or cache), decompressing the inverted lists, and ranking documents (which could involve decompressing the frequency and possibly positions

of a docID). The query processor depends on an efficient layout of the index to quickly return the top matching documents. The PolyIRTK index is composed of fixed-size *blocks*, and is the basic unit for inverted list caching. Furthermore, each block is composed of a *block header* and a series of *chunks*, all of which contain n postings, except the last chunk of an inverted list, which contains $list\_size \,\%\, n$ postings.

Query optimization techniques often achieve speedups through inverted *list skipping*. List skipping is essentially done by building a mini index on top of the main index, so that parts of the compressed list can be skipped without encountering the decoding overheads of the parts we wish to skip. For background on list skipping, see [5], where the authors embed skip information within an inverted list. The PolyIRTK block header implements list skipping functionality by providing a secondary docID index for the chunks, so they can be easily skipped without decompression. Another query optimization technique is *early termination*, which allows the query processor to stop further processing of an inverted list.

Finally, in our descriptions of query algorithms, we will often refer to *list pointers*. This is an in-memory data structure that keeps state about the current position in an inverted list and is primarily used to move forward to the next docID in a list. It abstracts away decompression and list skipping, so that the interface user only specifies which docID to advance to.

2.5 Other Search Toolkits

Middleton and Baeza-Yates [6] looked at the performance and effectiveness of many open-source information retrieval toolkits available in 2007. In this section, we briefly describe several notable open-source search engines, in use either in academia or industry. All of them use an inverted index as their underlying data structure for answering queries, as it was shown in [6] that search engines using a relational database significantly lagged in performance compared to those using an inverted index.

Indri[2] is a C/C++ based search engine developed jointly between the University of Massachusetts and Carnegie Mellon University. It has many features, including the ability to distribute both indexing and querying over a cluster of machines, international language support, and in particular, it has an extensive query language. It makes use of Variable Byte coding for index compression.

Lucene[3] is a widely deployed and full-featured search engine library, written completely in Java, supported by the Apache Software Foundation. A vector space ranking function is used in Lucene, based on the cosine similarity measure, with modifications to support user boosting of certain documents or terms within certain document fields. Lucene uses Variable Byte coding for index compression, but we note that work is ongoing to integrate some of the new index compression algorithms described in Section 2.3.

Sphinx Search Server[4] is a full-featured search engine written in C++, developed primarily by Sphinx Technologies. The primary differentiator of Sphinx is its ability to closely integrate with the MySQL database to provide full text search for database records, and its inclusion of an optional SQL-like querying language. Nonetheless, it is also able to index data that is input with a custom XML format. Sphinx uses Variable Byte coding for index compression.

Terrier[5] is a Java based search engine developed at the University of Glasgow. It allows for several different indexing modes, including MapReduce indexing. Index compression in Terrier is limited to more traditional algorithms such as Golomb, Elias Gamma, and Delta coding. Terrier has UTF support and so is able to search across document collections with languages other than English. Terrier supports a few different document ranking methods, including BM25 and weighing schemes taking into account term proximity.

[2] http://www.lemurproject.org/indri/
[3] http://lucene.apache.org/
[4] http://sphinxsearch.com/
[5] http://terrier.org/

Finally, Zettair[6] is a search engine developed at RMIT University, written in the C language. Zettair is capable of scaling to very large document collections, with a linear growth in time with respect to the document collection [6]. It was also notably significantly faster than the other search engine benchmarked in [6] for the indexing stage, and competitive in querying speed. Zettair had the best precision among the compared search engines in [6] (with Indri also coming close). It uses Variable Byte coding as its primary index compression method.

Lucene (in conjunction with Solr, a search server built on top of Lucene, both of which have been merged into a single project) and Sphinx are generally the more full-featured search engines, with out-of-the-box support for features like incremental indexing, sharding an index over a cluster of machines, deleting documents from an index, faceted search, and geo-spatial search, among others. Indri, Terrier, and Zettair are more research-oriented search engines, with Indri being the more full-featured of the group. Most of the search engines discussed here use Variable Byte coding as their index compression method, with the exception of Terrier, which uses Golomb, Gamma, and Delta codings. PolyIRTK, on the other hand, includes support for the latest index compression techniques, with the ability to separately specify coding policies for each type of index data (docIDs, frequencies, positions, etc.). We also allow the user to specify block sizes and to mix block-based codings (PForDelta and Turbo-Rice, which operate on a multiple of 32 integers at a time) with other codings. This capability is useful when there are not enough integers to fill a complete block, since padding the block-based codings with zeros would lead to a larger index size due to the majority of inverted lists being smaller than the block size. Finally, PolyIRTK includes several query optimization algorithms based on the popular and effective BM25 ranking function.

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

3 Design and Implementation of PolyIRTK

Like most efficient information retrieval systems, we use the well studied inverted index data structure to answer queries. In this chapter, we describe the indexing pipeline used to create the inverted index, as well as the intermediate and final data structures created. In later sections, we present the BM25 ranking function used by PolyIRTK and show how unoptimized query evaluation works for conjunctive and disjunctive queries. We also discuss additional features of PolyIRTK, such as reassigning docIDs and creating index list layers, which we use in Chapter 4 to speed up query evaluation.

3.1 Indexing the Document Collection

PolyIRTK is designed to efficiently construct an index on a single machine, using only one CPU core. The most efficient way is to use a single-pass index construction method [7]. The advantage of this index construction method is that it does not require the vocabulary of the whole document collection to remain in main memory throughout indexing; this allows it to easily scale from very resource and memory constrained environments to systems with very large memories available, where the additional memory is used to speed up indexing.

The PolyIRTK indexing system can be relatively easily scaled to more than one machine or multiple CPU cores (assuming the cores are not starved due to disk I/O), simply by partitioning the document collection among all the machines/cores and letting each one generate an index. The indices can then either be merged together or the querying system can then ask each machine for the results from each index and merge them to get the final result set. However, this would require a modification where each index, generated by a separate instance of the PolyIRTK process, would need to know its docID offset, so that the docIDs of an index can be converted to absolute docIDs of the whole collection. The document maps (discussed

in Subsection 3.2.3) would also need to be merged together for the final index. We leave the modifications required for parallel indexing capability for future work.

We also note that, as of recently, a popular way to generate inverted indices is through MapReduce [8] or implementations like Hadoop[1]. The benefit of MapReduce is that it allows an easy way to parallelize index construction over many machines; however, it is unlikely to outperform single-pass index construction, due to its general-purpose nature, and not a logical choice when there is only one machine available.

3.1.1 Parser

Our description of the indexing pipeline starts with the parser component, an important part of an information retrieval system due to its effect on the result quality and the indexing performance, since a large percentage of CPU time is spent in the parsing stage. In this thesis, we do not focus on how to achieve the best result quality, so the parser is a simple component, optimized for speed, that ignores HTML markup and JavaScript. The parser tokenizes on non-alphanumeric characters and keeps track of which HTML tag tokens are currently being processed; thus, some context information can be extracted for each token (i.e. whether a token appears inside the h tag), although the context is not utilized at this time. PolyIRTK accepts compressed document bundles in either the TREC or WARC formats, which are commonly used for research datasets (such as Gov2 and ClueWeb, respectively). PolyIRTK memory maps the document bundles (to avoid copying from kernel to user space) one at a time and decompresses them into a buffer designed to dynamically expand to accommodate bundles of all sizes. The parser then tokenizes each document in the bundle and builds structures consisting of a token, token length, docID, and token position, the pointers to which are passed to the posting collection component. Upon finishing parsing a document, the parser sends the docID, document length, URL, and TREC identifier to the document map builder component.

[1] http://hadoop.apache.org/

3.1.2 Posting Collection

Tuples of the form <term, docID, position> are extracted by the parser from the document collection and passed on to our posting collection stage. Depending on how PolyIRTK is configured, the term positions may be discarded. The tuples coming from the parser are compressed one at a time using the Variable Byte coding method. Variable Byte coding is fast for compression and decompression and it allows us to buffer significantly more <docID, position> tuples in main memory as opposed to storing fixed-size four byte integers. We also take the differences between subsequent docIDs and positions (d-gaps and p-gaps), since they are always stored in increasing order, to make Variable Byte coding more effective.

In order to map terms to their corresponding compressed lists of docIDs and positions (let's call the space for these lists, the *term_blocks*), we use a move-to-front chained hash table [9]. Upon doing a *find* or *insert* operation on a move-to-front hash table, the *term_block* pointer is moved to the front of the linked list of its assigned hash bucket. Since the vocabulary of a document collection is skewed towards frequent terms, and we have to make a lookup for every token handed to us by the parser, this optimization delivers a significant performance boost. The majority of lookups will either not need to walk the linked list of *term_block* pointers hashed to the same bucket, or will be found towards the front of the linked list. With the move-to-front optimization, the hash table load factor is not as important to performance, so it allows us to keep the hash table a constant size. This has the benefit of avoiding having to resize and rehash all the elements.

A memory pool is used by the posting collection component, the size of which is configurable by the user based on how much memory the machine can spare for indexing. PolyIRTK allocates a big block of memory at startup, and then uses this memory pool to hand out small, fixed-size chunks of memory to the *term_blocks*, which use them to compress <docID, position> tuples. Requesting chunks from the

memory pool is a very cheap operation, since they are assigned sequentially. Each *term_block* is composed of a linked list of pointers to the fixed-size memory chunks. The memory pool approach avoids the memory fragmentation that would result from many small dynamic allocations.

There is a tradeoff between the chunk size and memory pool utilization. Since the majority of the corpus vocabulary consists of obscure terms that only need a few bytes to store <docID, position> tuples, a large chunk size will wind up wasting a lot of the memory pool. On the other hand, a very small chunk size will result in many pointers to chunks for common terms. These pointers take up space in the *term_block* linked lists and cause the traversal of the compressed tuples to be slower (when writing the index to disk). PolyIRTK uses a chunk size of 128 bytes by default, however this value is configurable; the only limitation is that it must be a factor of the memory pool size.

To summarize, the following steps are taken by PolyIRTK when the parser hands the posting collection component a <term, docID, position> tuple:

- If there exists a *term_block* pointer for the current term in the move-to-front hash table, its pointer is moved to the front of the hash bucket list; otherwise, a new *term_block* is initialized and its pointer is inserted at the front of the hash bucket list.

- The docID and position are compressed into the last assigned chunk from the memory pool for the current *term_block*. If the chunk does not have enough space to hold either the compressed docID or position, a new chunk is requested from the memory pool; its pointer is saved in the *term_block* for our current term and the leftover bytes of the compressed docID or position are copied into the new chunk. Note that we are able to compress across memory chunks so that there is no wasted space. In the event that the memory memory pool runs out of available chunks, the current index run is written to disk and the move-to-

front hash table and the memory pool are reset, which we discuss in more detail below.

Once the memory pool runs out of space, the move-to-front hash table is traversed and each *term_block* pointer is saved to an array, which is then sorted lexicographically. We then proceed to extract the compressed <docID, position> tuples from each *term_block* and convert them into postings; we decompress the docIDs and positions, collapse the same docIDs to determine their frequency value, and collect all the positions for each docID. We collect groups of 128 docIDs, frequencies, and up to $(128 \cdot max\_number\_positions\_per\_doc\_id)$ positions in separate arrays for each term and pass them onto the index builder stage. The index builder is responsible for compressing the groups of postings to build the index, as well as building the lexicon, which records terms and the start of their inverted lists as an offset into the index. The index builder buffers part of the index and lexicon in main memory before flushing to disk.

The indexing process can proceed faster when more <docID, position> tuples can be mapped to terms in main memory; there will be less index runs produced and less indices to merge later. The index runs that PolyIRTK produces during the indexing phase are full indices that can be used for answering queries. We look at the indexing time in relation to the size of the memory pool in Subsection 6.1.1.

3.2 On-Disk Data Structures

Throughout the indexing process (after the memory pool is exhausted), PolyIRTK writes out postings to disk as full indices which can be used for querying. In this section, we describe the data structures that are written out to disk during the indexing process.

3.2.1 Inverted Index

The inverted index is designed to be able to efficiently search millions of documents for the appearance of keywords. PolyIRTK represents each document by a unique docID, and stores the frequency with which a term occurred within a document. Optionally, PolyIRTK is capable of producing a word-level index that contains the term positions in each document. The frequencies and positions are used to filter and rank the list of results by their relevance to the query. The docIDs, frequencies, and positions are 32-bit unsigned integers which are compressed using user specified coding policies, enabling us to take advantage of the differences between their characteristics to choose the best compression algorithm. In this section, we discuss our inverted index format and the options available for the coding policies.

3.2.1.1 Compression

PolyIRTK supports many state of the art compression algorithms due to the use of open-source implementations developed at the Web Exploration and Search Technology Lab of Polytechnic Institute of NYU. We discussed the details of each compression algorithm in Section 2.3. We also note that PolyIRTK can be easily extended to include new compression algorithms by writing classes that implement the *compress* and *decompress* methods that operate on arrays of integers; a new algorithm would then need to be assigned a name in order to be specified in the configuration file.

For the purpose of defining the coding policies used by PolyIRTK, the compression algorithms can be divided into two classes, based on whether they operate on blocks of b integers at a time. Typically, $b = 128$, the maximum number of postings allowed in a chunk (see Subsection 3.2.1.2), however, it can be different for the positions array. The two blockwise compression algorithms that PolyIRTK supports are Turbo-Rice coding and PForDelta coding. The non-blockwise codings supported by PolyIRTK are Rice coding, S9 and S16 codings, Variable Byte coding, and Null coding (note that Null coding simply does not perform any compression).

To compress an array of n integers with a blockwise coding method, the array must be padded with zeros so that it is a multiple of b. This increases the index size, as most of the inverted lists are small (and less than b postings). The non-blockwise coding methods do not have this problem. A coding policy allows the user to select the primary compression algorithm, which can be any coding method supported by PolyIRTK. However, if it is a blockwise algorithm, the user must also specify a secondary non-blockwise compression method and the minimum number of integers min required to be present to still use the blockwise algorithm. If there are less than min integers to compress with a blockwise coder, the secondary non-blockwise coder will be used instead.

3.2.1.2 Layout

It is crucial to performance that the index is laid out in a manner that allows for efficient query processing. The PolyIRTK inverted index is laid out in blocks and chunks. A block is a fixed-size of 64 KB, while a chunk is at most 128 postings, with both values configurable at compile time. An inverted list cannot share a chunk with other lists, but blocks could be shared with many lists. Each block starts with a header, which stores an integer indicating the total number of chunks contained within, and for each chunk, its size as well as its last docID. The block header is also optionally compressed with the coding policy specified in the PolyIRTK configuration file. During index building, when a chunk cannot fit into the remaining free space in a block, that space is filled with zeros, and a new block is created for the chunk. Table 3.1 shows the on-disk format of a single block.

Each chunk contains 128 docIDs, followed by 128 frequencies, optionally followed by up to $(128 \cdot max\_number\_positions\_per\_doc\_id)$ positions. These components are stored separately within a chunk rather than interleaved so that the query processor can decompress each component individually and make better use of the CPU caches. Positions, when included, add significantly to the index size. The reason for limiting

Field Name	Field Type	Field Size (bytes)	Description
`block header size`	`uint`	4	Size of the block header in bytes, including the `num chunks` field (currently unused).
`num chunks`	`uint`	4	Total number of chunks contained within the block (from all lists).
`block header`	`uint[]`	varies	The size (measured in 4-byte words), followed by the last docID of every chunk contained in the block (compressed).
`chunk data`	`uint[]`	varies	Array of docIDs, followed by array of frequencies, optionally followed by an array of positions, for every chunk contained in the block (compressed).
`padding`	`uint[]`	varies	NULL bytes follow until block is exactly 65536 bytes in length.

TABLE 3.1: The index on-disk format for a single 64 KB fixed-size block.

the number of positions to *max_number_positions_per_doc_id* is the fixed block size; if a particular group of documents have an abnormally large number of occurrences of certain words, it is possible that a chunk will be formed that is bigger than than that of our fixed-size block. The *max_number_positions_per_doc_id* is a compile time constant, which we set to 64; this limit does not apply to the frequency values, which always store the total number of occurrences of a term. In practice, this should not be an issue since web documents are generally short, and postings that encounter this limit are most likely stopwords or insignificant web markup tokens.

3.2.1.3 Caching

Index caching results in query throughput gains by avoiding the retrieval of inverted list data from disk. A study by Zhang et al. [3] showed that a search engine could answer 90% of all queries without going to disk by caching only 30% of the inverted index. To improve performance when querying with the index on-disk, PolyIRTK

implements a least recently used (LRU) block eviction policy. The fixed-size index block layout used by PolyIRTK allows for a simple unit of index data to cache.

The number n of index blocks to cache in main memory is configurable. The cache is implemented by initializing an array of size $64\,\mathrm{KB} \cdot n$. A linked list, with each node holding a block number and its position in the cache array, is used to maintain the order of the blocks by their access times, with the front of the list signifying the least recently used node. The block number refers to a block's sequential position on disk, which can be read by seeking to byte offset $65536 \cdot block\_num$ in the inverted index file. The lexicon, discussed in the next section, knows the initial block number of each inverted list.

A hash table maps block numbers to pointers to the linked list nodes, allowing us to quickly check whether a block is present in the cache and, if so, retrieve its position in the cache array. When there is a hit for a certain block number, its pointer is moved to the back of the linked list to maintain LRU order. When there is a miss to the cache, a new block must be read from disk. However, firstly, if the cache is full, a block needs to be evicted. The LRU block number, found in the node at the front of the linked list, is erased from the hash table. Then, the LRU node has its block number updated and is moved to the back of the linked list. A new hash table mapping is defined, and the data in the cache array is replaced with that of the block from disk. PolyIRTK places appropriate mutex locks during the updates of these data structures, so that potentially several query threads could share the cache.

PolyIRTK uses the POSIX Asynchronous I/O (AIO) API to request blocks from disk. Typically, the index reader would request several blocks in advance, but it can continue to process an inverted list when it has just one block. With AIO, we can request several blocks in advance to read from the index, but return to list processing immediately, while blocks are loaded by another thread in the background. The blocks which are read into the cache are locked, so that they cannot be evicted until they have been fully read or are otherwise unnecessary to the index reader.

The PolyIRTK cache interface consists of an abstract base class that defines functions used internally by all the query processing algorithms. These functions, given a range or a single block number, queue up blocks, free blocks, and return pointers to the block data. By defining the implementation of these functions through inheritance, it is possible to create different caching schemes. In addition to the LRU cache implementation discussed above, PolyIRTK makes use of two other caching schemes. One of these cache implementations is to optimize merge performance, since merging several lists (as discussed in Section 3.3) always requires reading the inverted indices from beginning to end (in alphanumeric order of the terms). This means PolyIRTK can always cache the next several (2 MB) blocks from each index it is merging, and on the first miss, flush the cache, and load the next set of blocks. The other cache implementation allows PolyIRTK to memory map the entire index. In this case, functions designed to queue up the next blocks to load are no-ops and we just directly return the block from the memory-mapped file when requested.

3.2.2 Lexicon

The lexicon file contains the vocabulary of the document collection, in alphanumeric order. For each term, it stores the block and chunk numbers which determine the offsets into the index file for the inverted lists. The lexicon also records the total number of chunks in an inverted list, which is used to keep track of the end of the list. The last critical piece of information necessary to store is the total number of documents in an inverted list, which is a component of the BM25 ranking function, discussed in Subsection 3.4.1. Table 3.2 shows the complete structure of an entry in the PolyIRTK lexicon.

Chapter 4 discusses query optimization algorithms that require an index with multiple layers, where a term can have more than one postings list. For this reason, the lexicon is able to store information for multiple inverted list layers for each term. These query algorithms also require upper bounds on the list layers, and so the lexicon stores the

Field Name	Field Type	Field Size (bytes)	Description
num layers	int	4	Number of layers the list is composed of.
term len	int	4	Length of the term str without NULL terminator.
term str	char[]	term len	Term data (no NULL terminator).
num docs	int[]	4 · num layers	Number of documents contained in each layer.
num chunks	int[]	4 · num layers	Number of chunks contained in each layer.
num chunks last block	int[]	4 · num layers	Number of chunks contained in the last block of each layer.
num blocks	int[]	4 · num layers	Number of blocks each layer spans across.
block number	int[]	4 · num layers	Initial block number of each layer.
chunk number	int[]	4 · num layers	Initial chunk number in the first block of each layer.
score threshold	float[]	4 · num layers	Upper bound on the score of each layer (max partial document score).
external index offset	uint[]	4 · num layers	Start of the external index list for each layer (an offset into the external index file, measured in 4-byte words).

TABLE 3.2: The lexicon on-disk format for a single term entry.

maximum partial document score contained within each layer. This additional data is generated by the index builder during the index layering process, discussed in Section 3.6.

PolyIRTK loads the lexicon into a hash table in main memory at startup of query processing. This is the same move-to-front hash table used during the indexing stage and helps to keep access fast for the most frequently queried terms. Currently, PolyIRTK

does not compress the lexicon file and has to load it completely into main memory; while this was done for access speed and is acceptable for our experiments, we discuss improvements in Chapter 7 that would allow PolyIRTK to be used on machines with limited amounts of memory.

In related work, Strohman and Croft [10] describe a unique way to keep the in-memory lexicon small. Their inverted index is split up into approximately 32 KB sized blocks, with no list spanning multiple blocks; a term that has more than 32 KB of inverted list data gets its own block (which we assume has no size limits). An *abbreviated* vocabulary table, compressed with 15-of-16 front coding [2], is used to look up the block of a particular term. The abbreviated vocabulary table is so called because many of the terms contained within are actually partial stems, that point to a block, where an additional partial vocabulary is held. The partial vocabulary is sequentially scanned for the exact term that is being looked up and its list offset within the current index block is found. Frequent terms with large lists get their own blocks, so they can be found directly in the abbreviated vocabulary table, while short lists have to share 32 KB blocks. Thus, only rare terms need to have the partial vocabulary, located at the beginning of the block, sequentially scanned. The authors state that this scheme allows the in-memory abbreviated vocabulary to take up only 2 MB for the Gov2 collection. This way of storing the lexicon is conceptually similar to a B+ tree with only one level, but since part of the lexicon is stored in the inverted index file, it allows the lexicon to be cached along with the inverted index, and requires fewer disk seeks to lookup the start of a list.

3.2.3 Document Map

The document map is responsible for translating the docIDs assigned by PolyIRTK to pertinent information about a document; it is shared between all indices generated for a document collection. PolyIRTK stores the document length in words, the URL, and the TREC identifier (for evaluating TREC experiments). After parsing a document to

obtain this data, it appends a new entry to the document map to an in-memory buffer, which is periodically flushed to two files, the *basic* and the *extended* document maps. The basic document map contains the document lengths and offsets into the extended document map for each document in the collection. Since this file is relatively small, it is loaded into main memory for fast access to the document lengths, which are necessary for evaluation of the BM25 ranking function (see Subsection 3.4.1).

The extended document map is much larger, and we therefore keep it on disk; it stores document URLs and TREC identifiers in an uncompressed format. The extended document map is only accessed for the final top-k results, using the offsets found in the basic document map. Since accessing this file in a random fashion would be a performance bottleneck, during our performance experiments, we do not make lookups into the extended document map; however, it is used for interactive command line querying and in our TREC effectiveness experiments.

3.2.4 External Index

The external index is designed to hold additional index information that is not in the main inverted index file. Currently, PolyIRTK generates it during the index layering process described in Section 3.6, and it is used to store the maximum partial document scores for each chunk and block as 4-byte floating point numbers. When a query algorithm that utilizes the score information is used, the external index is memory mapped and traversed in sync with the main inverted index during query processing. The external index file is uncompressed, and ranges in size from 0.6 to 1 GB for the complete Gov2 document corpus, depending on how the index layers were generated.

3.2.5 Index Meta File

The index meta file is a simple text file composed of key and value pairs that store index configuration information and index statistics. The statistical information in-

cludes the number of bytes written individually for the docIDs, frequencies, positions, and the block header, the total number of documents in the collection, and the average document length (used by the BM25 ranking function). PolyIRTK uses some of this data to validate index operations, such as checking that the merged index contains the sum of all documents in the indices it was merged from; these sanity checks are helpful for debugging purposes.

Important index configuration information includes the coding policies that were used for the docIDs, frequencies, positions, and block header. These coding policies are used by the PolyIRTK index reader to properly decode the inverted lists. An index whose docIDs have been reassigned (see Section 3.5) stores a flag in the meta file so that the document map can be properly adjusted to the new docID mappings. Additionally, configuration information on index layers is used to determine which query algorithms a particular index can support.

3.3 Merging Indices

The number of indices that PolyIRTK will need to merge is dependent on the amount of memory configured to be used during indexing, since allocating more memory for posting collection allows PolyIRTK to write out larger index runs. The merge degree is the number of indices at a time that PolyIRTK will merge and it is configurable, with a default value of 16. The maximum number of indices that can be merged at once is dependent on the amount of memory the machine has available, since PolyIRTK buffers the next several megabytes of both the lexicon and inverted index of each run being merged. We do not need to load the entire lexicon or index into main memory since the lists are merged in alphanumeric order of the terms. A lower merge degree could result in more passes having to be made over the index postings to get the final merged index; the merge degree and the initial number of indices determines the number of passes by the relationship $num\_passes = ceil(log_{merge\_degree}(num\_indices))$.

PolyIRTK uses a min-heap, holding one element from each index to be merged, to

extract the next lowest posting (first by term, then by docID), to be placed into the new index; this posting then gets replaced in the heap by the next posting from the same list. The merger collects chunks of 128 postings and passes them off to the index builder, which takes care of index compression and building the lexicon. This process continues until the heap becomes empty, at which point all the postings have been extracted from the indices being merged. The merger assumes that each index has a unique subset of the docIDs, so that positions for a particular docID are not split across indices (we enforce this during the indexing phase). PolyIRTK also has configuration options to compress the final merged index using different coding policies from those of the indices it is being merged from. This is possible because all the indices keep track of their coding policies in their meta files, allowing the merger to make use of the index reader and index builder, which abstract away the index decoding and encoding, respectively.

3.4 Query Processing

PolyIRTK supports modes for interactive querying via command line and for batch query processing, either for benchmarking efficiency or running TREC effectiveness experiments. The batch queries are read into an array from a file or standard input. If the user wants to run efficiency benchmarks, the query log is executed twice; the first pass warms up the caches (either the LRU cache or memory maps the index), and the second pass times the run. During batch benchmarking mode, the docIDs are not resolved to URLs through the document map and query results are not printed to the screen; at the end, only summary query statistics are presented. If the user runs batch queries and sets the output to TREC mode, the query results are printed to the screen in a format suitable for passing to the `trec_eval` program for evaluating result effectiveness; the docIDs are translated to their TREC identifiers using the document map. Finally, the interactive querying mode outputs results with the top-k docIDs, document scores, and document URLs for each query.

The query processor neither takes into consideration the order of the query terms, nor the number of appearances of a term, since it is using the BM25 ranking function, discussed in the following subsection. The query terms are sorted in ascending order of their associated inverted list lengths and duplicate terms are removed. There are two basic query types, conjunctive (AND-type) and disjunctive (OR-type); both output the top-k relevant results using the BM25 ranking formula. Conjunctive queries are typically significantly faster than disjunctive queries because the AND operator applies a Boolean filter that requires all query terms to exist within a document, prior to ranking it. Disjunctive queries are more flexible since they allow any of the query terms to appear in a document for it to be considered for the final top-k results.

Upon looking up a query term in the lexicon, a list pointer is returned, which is a data structure that keeps state of the traversal of the inverted list. The *NextGEQ()* function operates on a list pointer and handles the bulk of the work in traversing an inverted list. It takes a docID as a parameter, and returns the next docID (starting from the last docID it returned) in an inverted list that is greater than or equal to the docID parameter. When all the docIDs have been consumed from a list, *NextGEQ()* returns a sentinel value indicating the end of the list.

Since each index block knows the last docIDs and sizes of the chunks contained within, *NextGEQ()* can check when the next requested docID is not contained within a chunk without actually taking the effort to decompress it. This leads to very efficient list skipping in some query algorithms. *NextGEQ()* only decompresses the docIDs of a chunk and does not operate on the frequencies or positions; other functions are able to retrieve this data for a particular docID when it is necessary (such as when a docID is in the intersection of all lists in a conjunctive query). List skipping mechanisms have been explored in [11].

3.4.1 BM25 Ranking Function

The document ranking function used throughout PolyIRTK is Okapi BM25, which computes the full score of a document D_d over the set of query terms Q, using the formula defined below.

$$bm25\_score(Q, D_d) = \sum_{t \in Q \cap D_d} idf(t) \cdot \frac{f_{d,t} \cdot (k_1 + 1)}{f_{d,t} + k_1 \cdot (1 - b + b \cdot \frac{len(d)}{avg\_len})}$$

This function computes the sum of the partial scores of the query terms (terms not existent within document D_d do not contribute). The function $f_{d,t}$ returns the frequency of the term t in the document being scored. The $len(d)$ function returns the length, in words, of the document being scored, while avg_len is the average length of all documents in the collection. The constants k_1 and b are parameters set to 2.0 and 0.75, respectively. The $idf(t)$ function, defined below, is the inverse document frequency (IDF), which determines the importance of a query term through how widely it appears in the document collection; rare terms are weighted more heavily.

$$idf(t) = \log(1 + \frac{N - f_t + 0.5}{f_t + 0.5})$$

The constant N represents the total number of documents in the collection, and f_t represents the total number of documents containing the term t. In the standard BM25 function, a document score can be negative due to the IDF component, which affects very frequent terms (those appearing in over half of the documents in the collection). Here, we modify the IDF by adding one within the $log()$ component, so that it is always positive. Having positive partial scores was necessary for some optimized query algorithms discussed in the next chapter, which assume that the partial document scores always increase as more term scores are resolved. This change to the IDF should not cause any harm to retrieval effectiveness.

PolyIRTK is not limited to using only the BM25 ranking function; with some modifications to the source, any desired function can be plugged in during list traversal. For the optimized querying algorithms discussed in Chapter 4, any function that is based on the sum of positive list scores should work.

3.4.2 Unoptimized Query Algorithms

In this subsection we describe unoptimized query processing, while optimized query processing will be discussed in detail in the next chapter. There are two basic algorithms for query processing, Document at a Time (DAAT) and Term at a Time (TAAT), both of which can be applied to both conjunctive and disjunctive queries.

3.4.2.1 TAAT Traversal

The TAAT query algorithm traverses each inverted list one at a time. For this reason, it needs to keep track of each document's partial score in an accumulator structure, which can get quite big. For the 25 million page Gov2 dataset, the accumulator structure would amount to 100 MB; this is assuming that an accumulator is a four byte floating point value, and the implementation is an array, indexed by docID. Access to the accumulator array to update a document's score is constant time. Even though it is so large, there should be few cache misses because the array is accessed in order of increasing docID, and not randomly.

Another implementation of the accumulator structure is a hash table that stores <docID, score> tuples. While this approach might require less memory, it is also more computationally expensive due to hashing. Furthermore, the access to the array of the underlying hash table would be random (and in the case of a chained hash table, would require walking linked lists) and so it would suffer from many cache misses, unlike the array implementation.

Conjunctive queries can be efficiently supported by processing the inverted lists in increasing order of their lengths. The shortest inverted list would establish the initial

accumulator structure used to make lookups into the longer lists. The size of the accumulator list can be further reduced upon processing each additional inverted list, as more docIDs are filtered out. In this case, the accumulator structure would be implemented as an array, although it would be smaller since it would not need to be indexed by docID. With both the accumulator list and inverted lists stored in docID order, list skipping is possible, which often avoids decompressing entire chunks.

At the end of list traversal, the top-k accumulators must be selected; however, a full sort of the accumulators is undesirable. An effective solution is to use a min-heap to keep track of the top-k <docID, score> pairs, while iterating over the final set of accumulators. Inserting into the heap (and maintaining the min-heap property) only needs to be done for an accumulator if its score is greater than the score at the top of the heap; once the minimum score threshold increases, the majority of accumulators would not be able to make it into the top-k heap.

TAAT processing has several disadvantages when compared to the DAAT algorithm discussed in the next subsection. The accumulator structure has a potentially large memory footprint and TAAT processing has poor support for utilizing term positions, whether for document filtering or ranking. Due to these issues, PolyIRTK implements DAAT algorithms for basic conjunctive and disjunctive query processing. However, one of the optimized query processing techniques discussed in the next chapter is based upon TAAT processing.

3.4.2.2 DAAT Traversal

The DAAT query algorithm loops sequentially over all docIDs in the collection, fully scoring a document for each term that appears in it; thus, the algorithm traverses all inverted lists together by attempting to move to the next sequential docID in all of the lists.

A min-heap of size k is used to maintain the top-k docIDs and their full scores throughout query processing. It is necessary to insert a docID into the heap only

if its score is larger than the score at the top of the heap. Maintaining the heap property (by bubbling docIDs up and down by score) is a $O(log(k))$ operation, but k is constant, and tends to be small; since the heap is implemented as an array, it can often fit into the CPU cache. Even for extremely large values of k, it is possible to use B-heaps to keep cache performance very good [12]. After all lists have been traversed, the heap is sorted by score to get the final top-k results ordered by their relevancy to the query.

Like TAAT query processing, the DAAT algorithm can take advantage of conjunctive queries by sorting the query terms in ascending order of their list sizes, so that the shortest list can drive docID lookups into the longer lists. Both DAAT and TAAT are able to terminate a docID lookup into all longer lists when a docID is non-existent in a shorter list. This gives PolyIRTK the ability to skip portions of the longer lists, often without decompression.

Unlike TAAT, since a docID is fully considered at each step, only the top-k docIDs need to be kept in main memory throughout query processing. Additionally, a document's term position lists can be easily used, either in the ranking function, or with a document filtering step that requires the terms to be within a certain distance of each other. For example, PolyIRTK implements a DAAT-AND processing mode that retrieves the top-m documents with the BM25 scoring function only and then combines it with the proximity score of a document [13] to get the final top-k results, where $m >> k$.

3.5 Reassignment of DocIDs

During initial indexing of the document collection, PolyIRTK assigns docIDs sequentially. However, in [4] Yan et al. showed that assigning docIDs in URL sorted order of the documents leads to a substantially smaller index size and faster querying speeds, especially while utilizing PForDelta compression for docIDs. The reason is that documents that share part of their URL prefix tend to be related (i.e. they are in the

same path on a host). Related documents are likely to share many of the same terms, leading to smaller d-gaps because terms are clustered around similar groups of documents. Taking advantage of docID clustering leads to better compression and list skipping behavior.

PolyIRTK is able to take an existing index and generate a new index, with docIDs reassigned based on a file mapping existing docIDs to new docIDs; the new docIDs must be within the same range as the existing docIDs. PolyIRTK includes a utility to generate a mapping file for reassignment by document URLs. This utility scans the document collection in the same order it was indexed and extracts the document URLs, which it writes to a temporary file along with the original docIDs. The temporary file is sorted by the document URLs and each row is assigned a new docID based on the sorted position of a document's URL. Finally, the temporary file is used to generate a mapping file, consisting of only the original and reassigned docIDs.

The docID reassignment procedure is designed to be I/O-efficient, so it can deal with very large inverted lists without running out of memory or becoming bottlenecked by the disk. It takes as input an original index and the mapping file. The mapping file is read and the reassigned docIDs are stored in an array indexed by the original docIDs. PolyIRTK reads postings from the original index into a buffer whose size is defined by the configuration file; if the index is word-level, positions are read into a separate buffer. While copying postings to the buffer, docIDs are reassigned using the docID mapping array that was previously created.

There exist two cases, listed below, when PolyIRTK must sort the buffered postings by the reassigned docIDs, and pass them to an index builder, responsible for writing the postings to an index.

1. All postings for an inverted list have been consumed.

2. The postings buffer becomes full.

In case 2. above, PolyIRTK not only hands off the postings to an index builder, it also flushes the current index to disk and creates a new index to collect additional postings.

The final step of the docID reassignment process is to merge all intermediate indices into a fully reassigned index. This is done by reusing the index merging components discussed in Section 3.3.

An issue that remains is the document map, which needs to be updated to reflect the newly reassigned docIDs. However, due to our design of having a separate *extended* document map, it is sufficient to only update the *basic* document map in main memory by swapping the document info between the original and reassigned docIDs. Thus, prior to starting query processing, PolyIRTK must read the mapping file used to generate the reassigned index. Since the *basic* document map is loaded into main memory as an array indexed by the docID, PolyIRTK is able to use the mapping file to swap the document lengths and offsets (for the *extended* document map) between the original and reassigned docIDs. Any lookups into the *extended* document map would then return data for the correct document.

3.6 Index Layers

Inverted lists can be split up into several groups of postings sorted by docID and we refer to these groups as list layers. PolyIRTK supports creating and querying an index with lists consisting of multiple layers; these layered indices are used to support the Multi-Layer query optimization algorithm discussed in Section 4.2. Each layer can be opened and traversed like a standard inverted list, by specifying the term and layer number. A layer contains postings whose partial scores are strictly lower than those of the preceding layer; i.e. PolyIRTK would not group same scoring postings across different layers. This allows the layers to have unique, decreasing upper bounds which PolyIRTK stores within the lexicon.

A layered index is created by sorting the original list postings by their partial BM25 scores, determining the number of postings per layer using one of the layering strategies described below, and sorting the postings by their docIDs within each respective layer; the index builder creates layers from the docID-sorted postings. For simplicity of the layer implementation, PolyIRTK assumes that any single inverted list can fit completely into main memory, although it is also possible to use I/O-efficient merge and sort techniques if inverted lists larger than available memory are expected to be processed.

In the configuration file, the user can specify an arbitrary number of layers to create using a layer strategy from one of those listed below.

Percentage: The user defines the percentage of postings (out of the total for the whole list) each layer should contain. The last layer will receive all remaining postings.

Exponential: PolyIRTK splits a list such that each subsequent layer contains exponentially more postings, with respect to the number of layers that the user defined. We implement a method which Anh and Moffat [14] used to separate the terms in a single document into exponentially increasing buckets, but applied it to all the postings in an inverted list. This method calculates a base, $B = num\_docs^{1/num\_layers}$, and then uses it to determine the number of postings in each layer i: $(B-1) \cdot B^i$, where $0 <= i < num\_layers$.

The user can also define the minimum and maximum number of postings each layer can contain (with 0 being no limit), which would override the number determined by one of the layering strategies. Setting lower bounds is especially useful for forcing short lists into one layer; our minimum layer size is defined to be 32,768 postings. PolyIRTK will also make sure that each successive layer is larger than the preceding layer; if there are not enough postings remaining to make a subsequent layer larger, the postings from both layers will be merged into one final layer.

A problem with the exponential layering strategy described above is that it generates

very short lists for the initial layers. For this reason, we created two modifications to the base exponential strategy, designed to increase the sizes of the first few layers. The first modification, *Exponential Fib*, fixes the first two layers at a minimum of 32,768 postings (assuming the list is long enough). It then determines subsequent layer sizes using a Fibonacci-like pattern based off the first two layers. Once the exponential bucket sizes become bigger than the Fibonacci-like sequence, we switch to using the exponential bucket sizes.

The second modification, *Exponential Pow2*, selects an integer x such that $L[0] \cdot 2^x >= 32768$, where $L[0]$ is the first layer size calculated by the original exponential method. We determine $x = \left\lceil log_2(\frac{32768}{L[0]} \right\rceil$ and then multiply each original layer size by 2^{x-i}, where i is the increasing layer number and $x - i >= 0$. This strategy makes subsequent layers larger by a decreasing power of two, since they are already exponentially increasing by the base B. As a side effect of this modification, we get less overall layers than specified by the *num_layers* parameter in the original formula.

Table 3.3 shows several inverted list examples and the resulting layers with both exponential layer strategy modifications.

Term	List Size	Strategy	Resulting Layer Sizes (millions)
gov	23.08 million	Fib	0.03, 0.03, 0.05, 0.08, 0.13, 0.29, 2.44, 20.02
		Pow2	0.06, 0.25, 1.04, 4.33, 17.41
one	5.27 million	Fib	0.03, 0.03, 0.05, 0.08, 0.13, 0.21, 0.65, 4.08
		Pow2	0.04, 0.16, 0.58, 2.01, 2.48
world	1.22 million	Fib	0.03, 0.03, 0.05, 0.08, 0.13, 0.21, 0.68
		Pow2	0.03, 0.11, 0.32, 0.75
man	0.36 million	Fib	0.03, 0.03, 0.05, 0.08, 0.16
		Pow2	0.05, 0.31

TABLE 3.3: Exponential layer strategy examples for several different sized inverted list terms, with the *num_layers* parameter set to 8 in all cases. Note that the Pow2 strategy has a maximum of 5 layers with this parameter. The list and layer sizes are shown in millions of postings.

3.7 Debug Tools

PolyIRTK comes with two debugging tools, invoked from the command line with the `cat` and `diff` options. These tools are similar to their Unix counterparts, but are designed to operate on inverted indices (although `cat` does not concatenate indices). They proved to be helpful during development of the indexing, merging, and layering stages, as well as debugging the integration with various index compression algorithms.

These tools can operate either on whole indices or only on a particular term's inverted list. The `cat` tool outputs the postings of an inverted list in a human readable form as tuples of <term, docID, frequency, <positions>>. The `diff` tool outputs the differences between the inverted lists of two indices. There are three types of differences that `diff` is able to detect and output the mismatches for, listed below.

1. One of the indices does not include a posting.
2. One of the indices is missing a position within a posting.
3. The indices do not have a matching frequency value for a particular posting.

4 Query Optimization Algorithms

Compared to an exhaustive algorithm, an optimized query evaluation algorithm could provide four different guarantees about the top-k results; they could either be *set-safe*, *rank-safe*, *score-safe*, or *unsafe* [15]. Unsafe algorithms do not make any guarantees about the top-k results. Set-safe algorithms only guarantee that the top-k results will be the same set as produced by an exhaustive algorithm. Rank-safe algorithms guarantee that the results will be the same set and in the same order as produced by an exhaustive algorithm. Finally, score-safe algorithms are the strongest in terms of their guarantees; they produce the same result set, and in addition, the documents in the result set will have the same scores as those produced by an exhaustive algorithm. In the implementations provided by PolyIRTK, we focus only on the rank-safe class of query optimization algorithms (which also includes the score-safe algorithms).

Web search engines often apply complex, machine learned ranking functions to generate the top results displayed to the user. Since these functions are expensive to calculate and a search engine must respond within a fraction of a second, it is not practical to apply such a ranking function to the complete web collection, which could be billions of documents. It is therefore necessary to first use a simpler ranking function that could be quickly evaluated over a large set of documents. The results from the simple ranking function could then be fed to the complex ranking function to present the final result set to the user.

The type of query semantics used for the initial retrieval of the top-k documents plays a major role in both recall of the relevant documents and performance of the algorithm. As we show in Subsection 6.1.4, disjunctive OR-type queries (any of the query terms can appear within candidate documents) provide better recall at larger values of k than conjunctive AND-type queries (all of the query terms must appear in a candidate document). However, due to their nature, disjunctive queries are significantly slower than conjunctive queries, where inverted list skipping keeps processing

costs low. For these reasons, our rank-safe query algorithm implementations focus on optimizing disjunctive query performance.

For this work, we focus on using the BM25 ranking function, discussed in Subsection 3.4.1, which is widely used in information retrieval, both in academia and in popular search engine packages on the web. BM25 is also available as an implementation[1] for the Lucene search engine. In Section 6.2, we show query latency experiments for top-k retrieval with $k = 10$, $k = 100$, $k = 1{,}000$, and $k = 10{,}000$; this should cover cases whether PolyIRTK is intended to be used as a standalone IR system, or where the top-k results will be used as input to a more complex ranking function.

4.1 Related Work

The major search engines must be able to answer millions of queries per day, each with a sub-second response time, over indices consisting of billions of web documents; thus it is not surprising that there have been numerous works that attempt to reduce the computational resources of answering queries. Many of these works fall into the categories of static pruning, dynamic pruning, and index tiering. A static pruning algorithm modifies the index offline — before query processing begins. Generally, the algorithm works by pruning low scoring documents from the index based on a metric of query-independent document quality (e.g. PageRank, spam score) or by removing low scoring document-term occurrences (individual postings from an inverted list). Tiering segregates the index offline into different tiers — this could be done based on document, term, or both (e.g. taking into account query-independent document scores, partial document-term scores, query term popularity, etc.). Then, at query time, the algorithm will determine which tiers need to be traversed to acceptably answer a query. Finally, a dynamic pruning algorithm attempts to improve query performance by skipping portions of the index or ignoring the scoring of certain documents at query run time. Dynamic pruning and tiering algorithms are generally

[1] http://nlp.uned.es/~jperezi/Lucene-BM25/

more flexible than static pruning — these algorithms can be tuned to search fewer tiers or skip/ignore more documents while providing acceptable result quality, without re-indexing. During heavy query loads, if necessary to keep response time low, they can also be made more aggressive at the cost of result quality. In this section, we provide an overview of both safe and unsafe query optimization algorithms related to static pruning, dynamic pruning, and index tiering.

Persin et al. [16] sorted inverted list postings by document term frequencies to support unsafe query optimization with a dynamic pruning algorithm. Their TAAT algorithm reduced the number of accumulators created and processed, thereby saving memory and CPU time by scoring less documents. The algorithm processed inverted lists in order of decreasing weight (IDF); it maintained two thresholds, computed prior to processing each term, that controlled whether to add the partial score of the current document to its corresponding accumulator (if the accumulator already existed), and whether to insert a new accumulator for the current document with its partial score. These thresholds were based on the current maximum partial score in an inverted list, with the parameters for the thresholds empirically determined (the authors noted that they are not particularly sensitive in regards to the resulting performance). The thresholds were chosen by a tradeoff between query processing speed and result quality. Note that a frequency-sorted index can still be compressed as well as a docID-sorted index. The idea is to group docIDs by their frequency and take d-gaps of all the docIDs in every frequency grouping. Even better compression is possible by storing the high frequencies ($> n$) at the beginning of the list as a single group (this is equivalent to a docID-sorted index for frequencies $> n$). This scheme works well since frequency values are skewed towards lower values.

Ntoulas et al. [17] discuss an algorithm that makes use of static pruning and index tiering to avoid degradation of the top-k result quality. They split an index into two-tiers where the first tier is a statically pruned index and the second tier is the full index. Then, they devise a correctness indicator function that answers whether the

top-k results returned only from the first tier are the same results, and in the same order, as the top-k from an exhaustive evaluation over the second tier. When the correctness indicator function returns that there is a possibility of non rank-safe results from the pruned index, the query is fully evaluated over the second tier. The index was pruned by eliminating unpopular keywords and low scoring documents (based on both the document's query dependent and independent scores). The correctness indicator function was efficiently computed only from the first tier.

Long and Suel [18] look at early termination during query processing, by sorting inverted lists by query independent document scores. Specifically, inverted lists were organized in descending order of the associated documents' PageRank [1] values (by assigning docIDs in order of descending PageRank); it was assumed that the scoring function is equally influenced by the document's PageRank as well as the cosine measure [2] between the query and document. Sorting by PageRank allows for an easy way to intersect lists, since documents would be in the same relative order for all lists, as opposed to sorting lists by a query dependent scoring function, where a high scoring document for some term could be low scoring for a different term in the query. The authors limited testing to queries with only two keywords because their ranking function relied on the query containing two terms.

Long and Suel devised both rank-safe and unsafe query optimization techniques using their PageRank sorted index layout. Since the focus of our work is on rank-safe query algorithms, we will briefly describe the reliable technique used in [18], which yielded performance increases over the baseline. The authors split each inverted list into two tiers; the first tier is the *fancy list*, inspired by the paper on the early Google architecture [19], while the second tier is all the other documents. The fancy lists contained a percentage (typically several thousand documents) of the top scoring documents based on the cosine measure. Within each tier, the documents were sorted by their descending PageRank score. The reliable query processing technique would work like the following:

- Maintain two structures, D_0 and D_1, which hold docIDs that occur only in the inverted list for term 0 and 1, respectively; these are analogous to accumulator structures.
- Process the fancy lists first; documents occurring in both lists are scored immediately, otherwise, their docIDs are put into the D_0 and D_1 structures.
- Intersect the D_0 and D_1 structures with the longer lists (the second tier).
- Periodically, purge docIDs from D_0 and D_1 that cannot make it into the top-k.
- Terminate when D_0 and D_1 are empty, and no new document can make it into the top-k.

In order to prune the D_0 and D_1 structures, as well as stop further processing of documents in the longer lists, the score upper-bound on a document has to be determined. The PageRank score of any document in D_0 and D_1 is known, so all that remains is the partial score from the cosine measure which we know is upper bounded by the largest document score in the long lists. The upper-bound score of any new document can be determined from the last accessed document's PageRank (since they are always decreasing) as well as the sum of the highest term scores in the longer lists.

4.1.1 Fagin's Early Termination Algorithms

Fagin [20] discusses three top-k algorithms with proven bounds for early termination: Fagin's Algorithm (FA), Threshold Algorithm (TA), and No Random Access Algorithm (NRA). These algorithms assume that all lists are sorted according to their partial scores, in non-increasing order, and we will discuss them in the context of web search (they are also widely used in databases).

The FA algorithm traverses all lists in parallel until it finds k documents common to all lists. From that point on, the algorithm makes random accesses for all the remaining documents which have been previously encountered in only some of the

lists, and finally returns the top-k documents. Maintaining a list of the unresolved documents means FA must keep docID buffers of unbounded size. FA provides a rank-safe guarantee on the result set.

In TA, the required docID buffers are independent of the size of the lists because it only keeps track of the top-k documents at all times. TA accesses all lists in parallel and when it encounters a docID in one list, it immediately does a random lookup in all the other lists and fully evaluates the document score. It keeps the document if it ranks within the current top-k. If d_i is the last docID seen under sorted access in each list i, the threshold value, τ, is $\tau = \sum d_i$. Further processing of inverted lists can stop as soon as we have k documents whose scores are at least equal to τ. TA would never stop processing later than FA, but could require more random lookups. Like FA, TA also provides a rank-safe guarantee on the result set, although it is possible to modify it into an unsafe algorithm for additional performance.

The NRA algorithm makes no random accesses into the lists; they are always traversed in sorted order. This algorithm is guaranteed to retrieve the top-k documents, but is only set-safe because it is possible to determine the top-k documents with only partial information about the document scores. Unlike TA, which discards docIDs right away by doing random accesses to resolve them, NRA must keep an in-memory buffer of non-discarded docIDs and their partial scores. For each unique docID encountered, NRA keeps a lower bound on the final document score by using a score of zero for each list that this docID has not yet been found in. Likewise, an upper bound on the final score is maintained by using the most recent partial document score for each list that this docID has not yet been encountered in. The most recent partial score is the best we can hope for (since lists store docIDs in decreasing partial score order) at the moment for a docID that has not yet been found within a list. The algorithm makes sorted accesses in parallel to all lists and maintains the most recent score from each list so that it can compute the upper and lower bounds. When NRA encounters a docID in one of the lists being traversed, its upper and lower bounds on the final

score are tightened by utilizing the actual partial score. The current top-k list is maintained by keeping only the docIDs with the largest lower-bound scores, with ties broken by keeping the docID with the highest upper-bound. A docID is viable (can possibly make it into the top-k) only if its upper bound on the score is better than the worst lower bound out of the top-k documents. The early termination condition for NRA is met when both of the following are true:

- There are k distinct docIDs in the top-k.
- There are no viable docIDs left outside of the top-k; that is, the upper bound on the score for any viable docID is smaller or equal than the kth worst lower bound on the score in the top-k.

The problem with these algorithms is the requirement of sorting inverted lists by their partial scores. Doing so would result in bad compression because d-gaps could no longer be taken. Additionally, FA and TA require random lookups, while NRA is costly in data structures involved and often stops too late to gain significant performance benefits. Random lookups could be efficiently supported for docID-sorted indices through list skipping, but it is not clear how to do it efficiently in a score-sorted index. In order to cope with these problems, inverted lists can be split into several layers, each containing a unique subset of the postings within a particular score range; the layers would be arranged from highest to lowest score range. The maximum partial document score within each layer would be known, serving as an upper bound for all documents within the layer. Each layer would also be docID-sorted for good compression and allow random accesses through list skipping. These techniques are similar to those being employed in the algorithm discussed in Section 4.2.

Techniques to improve on TA and NRA have also been studied. Theobald et al. [21] explore unsafe algorithms that try to avoid the conservative lower and upper bounds of the NRA algorithm (which they call TA-sorted) by predicting whether a document is viable to make it into the top-k with a certain probability, based on its partial

score; documents that are unlikely to make it are pruned from further consideration. Bast et al. [22] discuss how to better schedule list accesses for TA-style algorithms in order to drive upper and lower bounds for faster early termination; they achieve this by preferring to do sorted accesses on particular lists and resolving certain docIDs with a random access. Their goal is to minimize access costs (sum of random and sorted accesses) by utilizing the distribution of docIDs in each list and correlations between query keywords.

4.2 Multi-Layer Query Algorithm

In this section, we describe an algorithm that uses multiple inverted list layers for query optimization. It is based on Anh and Moffat's impact-sorted query algorithm [14] with the optimizations applied by Strohman and Croft [10] that improved query throughput by 69%. Anh and Moffat also discuss a modification of their impact-sorted algorithm that increases processing speed at the cost of retrieval effectiveness. This unsafe algorithm is based on the early termination of the AND phase (see Subsection 4.2.2), where a parameter is added that specifies the percentage of the remaining postings to process.

The main difference of our algorithm is the adaptation of the BM25 ranking measure instead of document impacts. To some extent, this algorithm resembles the NRA algorithm discussed in Subsection 4.1.1. Like NRA, the Multi-Layer query algorithm keeps an in-memory unbounded buffer of docIDs and their scores (accumulators), and prunes docIDs from consideration when they are no longer viable. The Multi-Layer algorithm is rank-safe; it is not score-safe because it is able to terminate early based only on partial scores of documents.

4.2.1 Background on Impact-Sorted Indices

The algorithms described in [10, 14] used document impacts [23] as scores, and so it is prudent to first discuss the motivation for them. In a standard docID-sorted index,

d-gaps are taken to produce a series of smaller integers that are more compressible. In order to apply some early termination techniques, docIDs need to be sorted by score, but d-gaps cannot be taken since the docIDs in a list are no longer monotonically increasing. To address this issue, Persin et al. [16] looked at storing inverted lists in frequency-sorted order, and they found techniques to effectively compress the index. However, sorting by frequencies means only approximately sorting by score because ranking functions like BM25 and cosine similarity additionally normalize the score by document length, with the result being that query processing was still unsafe. This problem is solved by document impacts, which are integer scores that take into account document length normalization and term frequencies.

The impacts are calculated at index construction time; within each document, terms are sorted by decreasing frequency and based on this ordering, impacts are assigned values between 1 and k. The value of k is decided before index construction, with a typical value being 8. An exponentially growing number of terms are assigned to the lower valued impacts, with common stop words assigned impact scores of 1. Note that the impact score includes the TF component with document length normalization, since they are assigned relative to the total number of terms in a document. Furthermore, the exponentially increasing bucket sizes implicitly scale the TF component, thus avoiding giving infrequent terms excessive weight within a document.

Impact-sorted indices no longer require the storage of term frequencies. Instead, each list is organized into a set of k segments, with each latter segment implying that all postings within are lower scoring than the prior segment; furthermore, all postings within a segment have the same partial score (a value in the range of 1 and k). A small number of segments allows docIDs to be densely packed in monotonically increasing order within each segment, making d-gaps smaller, and allowing for better compression. The query terms are also assigned impact scores (values 1 through k) and an inner product is taken between the query and document vectors to get the full

document score; this means that the partial score of any document ranges from 1 to k^2. The term impact score acts as the IDF component (determined at query time), while the segment score acts as the TF component, and combined you have a familiar $TF \cdot IDF$ calculation that is utilized by many scoring formulas like BM25 and the cosine measure.

4.2.2 Query Processing

The Multi-Layer query algorithm we describe here is an adaptation of the impact-sorted query algorithm, using the BM25 ranking measure. This allows us to compare the Multi-Layer query algorithm directly to the others we explore, and it is more applicable to popular search engine toolkits which already make use of the BM25 ranking formula. Additionally, while assigning partial document impacts between 1 and k allows for a fast implementation, it does so at a loss of some retrieval effectiveness. Like the impact-sorted algorithm, the Multi-Layer algorithm achieves speedups by dynamically pruning accumulators, which reduces memory usage and allows for better skipping within lists; it can also terminate early, skipping processing of some layers altogether.

In the Multi-Layer algorithm, inverted lists are split up into several layers (we experiment with a few possibilities) by BM25 score. Each layer contains all the postings within a distinct score range, sorted by their docIDs, so that existing compression techniques can be applied. The layers are laid out on disk in order of decreasing score ranges. The query algorithm opens a pointer to each layer and sorts the pointers by the maximum partial score of each layer (these scores are stored in the lexicon). The layers are then processed in order of maximum potential score contribution.

There are two values that will be used during the description of the algorithm, which we define using the terminology in [10]. The *remainder score*, $\rho(L_i)$, of an inverted list, L_i, is the maximum partial score of the next layer to be processed. Since our list layers are laid out in decreasing order of their maximum partial scores, $\rho(L_i)$ will

continually decrease throughout list processing. It is best viewed as an upper bound on the partial score of a list, that improves through further processing. The sum of the remainder scores over all lists computes an upper bound on the score of any new document that has not yet been seen. The *threshold score*, τ, is the k-th highest score in the accumulator array, or 0 if less than k accumulators have been initialized. It is a lower bound on the score of the k-th document in our final top-k, which also improves throughout processing.

The steps outlined below are taken for each layer.

- **Process Layer**

  Processing a layer involves updating document scores and adding new documents to the accumulator array. Initially, each layer is processed in OR-mode, where all newly encountered docIDs are added to the accumulator array. After processing a layer in OR-mode, the accumulator array is sorted by docIDs. Note that this can be accomplished in $O(n)$ time by allocating a new accumulator array and merging the two sorted portions (i.e. the existing accumulators and the newly inserted accumulators) since we traverse a layer in a docID ordered manner.

  The switch to AND-mode processing occurs when the condition $\sum_{i=0}^{n-1} \rho(L_i) < \tau$, over all n inverted lists, becomes true. Since no more new docIDs can make it into the top-k, only docIDs already included in the accumulator array are updated. Since the accumulator array is docID-sorted, it can be used to drive the lookups of docIDs in the current layer. The result is a significant performance win through the use of list skipping.

- **Prune Accumulator Array**

  After processing a layer in either OR-mode or AND-mode, the accumulator array can be pruned by identifying accumulators that no longer have a chance to make it into the top-k. Each accumulator knows its current score and the set

of lists from which the score is composed from. To determine if an accumulator can be pruned, the condition $S + \sum \rho(L_i) < \tau$, must be true; here, S is the score of the accumulator and L_i is in the set of lists not included as part of the accumulator score. More accumulators can be pruned as the upper bound on the maximum accumulator score decreases, and the threshold value τ increases.

Pruning the accumulator array reduces memory consumption and the number of accumulators that need to be sorted after processing a layer. In AND-mode, list skipping becomes more efficient as the accumulator array grows much smaller than the remaining list layers.

- **Check Early Termination Condition**

  Query processing can terminate when the final order of the top-k documents can be determined. This is satisfied by the following two conditions:

  - The top-k documents must be known.

    This condition is met when every accumulator with a current score less than the threshold cannot have an upper-bound score (determined in the same way as during the accumulator pruning process) that is greater than the threshold. We can check this condition while pruning accumulators and if it is true, we can check the following condition.

  - The order of the top-k documents cannot be changed through processing of additional layers.

    This condition can be verified by sorting the accumulators by their current scores in ascending order and comparing each accumulator with its higher scoring neighbor. If the lower ranking accumulator has an upper-bound score greater than the current score of its neighbor accumulator, the algorithm cannot terminate processing.

Finally, the algorithm sorts the accumulator array by score, and returns the top-k results.

4.2.3 DAAT Processing

It is also possible to do DAAT processing using the same layered index layout. This is accomplished by treating each layer as if it were an independent list and then running any DAAT algorithm that merges all the layers. Anh and Moffat [14] tried this approach and found performance to be acceptable, but slower than their optimized algorithm. We also tried this approach for comparison purposes, but in addition experimented with the MaxScore algorithm for DAAT processing, described in the upcoming Section 4.4. Unlike Anh and Moffat, we used a standard binary heap, and did not implement a length-biased merge, which yielded additional performance improvements in their experiments. These additional experiments with a DAAT approach to querying multi-layer indices appear in Subsection 6.2.1 — the algorithms are labeled as *Multi-Layer DAAT* and *Multi-Layer MaxScore* to differentiate them from the optimized *Multi-Layer TAAT* algorithm.

4.2.4 Implementation

There are a few implementation details that are worth discussing. This is a TAAT algorithm and it requires accumulators to be maintained. Each accumulator is a tuple of the docID, current score, and a term set (bitmap of the inverted lists that are taken into account in the current score). The accumulators are sorted by docID in an array for good cache performance and to take advantage of the skipping optimization.

Amongst the accumulators, it was necessary to keep track of the k-th largest score since we required the threshold score after processing each layer. The simplest way is to use a min-heap (so the lowest scoring <docID, score> tuple is on top); however, when an accumulator is updated (and its score is already in the heap), if we simply add it to the heap again, we'll be artificially increasing the threshold score, leading to possible incorrect results. The solution is to reinitialize the heap and insert every accumulator (whether updated or not) into the heap again, which takes $O(n \cdot log(k))$, where n is the number of accumulators and k is the number of scores we keep in the

heap. We experimented with another method to find the threshold — the quick-select algorithm which runs in $O(n)$ time. Quick-select requires additional passes over the accumulator array after processing each layer.

Finally, the best performing method we found was to use a small hash table (open addressed with linear probing) that tracks docIDs that are currently in the top-k min-heap. After updating the score of an accumulator, we would check the hash table to see whether the docID was in the heap. If it was in the heap, we would linearly search the heap until we found the matching <docID, score> tuple, update its score, and do a bubble down operation on this updated tuple to maintain the min-heap property. On the other hand, if the hash table indicated that the accumulator was not in the top-k, we would insert it into the heap only if its score was greater than the threshold. The hash table also had to be updated when removing a tuple from the top-k heap by marking the docID deleted. The worst case running time of this method is $O(n \cdot (k + log(k)))$, assuming hash table lookups and deletions remain constant time and every accumulator must have its score updated. However, this is rarely the case, as most of the accumulators do not make it into the top-k, and so they only require a constant time hash table lookup. We found that the performance was better than the other techniques considered, even for large values of k.

4.3 WAND Query Algorithm

The WAND (Weak AND) algorithm of Broder et al. [24] is a score-safe DAAT query algorithm that works with traditional docID-sorted inverted lists. This technique achieves performance improvements through list skipping and selectively scoring docIDs.

The WAND algorithm requires the maximum partial document score of each list to be known, which serves as the upper bound on the score of that list. Assuming the list pointers are stored in an array, sorted by their current docID, let L_i be the list pointer in its sorted position i, and let $L_{i,did}$ and $L_{i,ub}$ be the current docID and

upper-bound score of the list pointer, respectively.

WAND works by sorting the list pointers by their current docID in non-descending order after each list pointer moves forward. Prior to choosing a list pointer to move forward, a *pivot* is selected at position j, which is the first list pointer that meets the condition $\sum_{i=0}^{j} L_{i,ub} >= \tau$, where τ is the threshold score (the k-th lowest scoring document). The list pointer chosen as the pivot has the lowest docID that can possibly make it into the top-k; thus, list pointers at positions less than j can be safely advanced to at least the docID found at the pivot $L_{j,did}$. In our implementation, we chose to advance list pointer L_0, the shortest list. WAND fully scores a document only when all list pointers at positions less than or equal to j (the pivot) have the same docID, which is simply checked as $L_{0,did} == L_{j,did}$, since they are in docID-sorted order. If a pivot cannot be chosen, processing will terminate, possibly without fully traversing all lists. Termination will happen when at least one list has been fully traversed and the remaining lists cannot produce a document with an upper bound on its score that is greater than the threshold.

The major cost of the WAND algorithm, relative to the unoptimized DAAT-OR algorithm, comes from the constant sorting of the list pointers; however, as can be seen in the experimental results in Subsection 6.2.2, the cost is more than made up through much better list skipping and fewer document score evaluations.

4.4 MaxScore Query Algorithm

The MaxScore algorithm of Turtle and Flood [25] is a DAAT query algorithm that is similar to WAND in that it is also score-safe and works with traditional docID-sorted inverted lists. Note that there is also a TAAT version of the MaxScore algorithm, which we did not experiment with.

Like WAND, the MaxScore algorithm assumes that the maximum partial document score for each list is known, which serves as the upper bound on the list score. In

MaxScore, the n list pointers are sorted only once, prior to the start of list traversal, in descending order of their upper-bound scores. Let L_i be the list pointer in its sorted position i, and let $L_{i,did}$ and $L_{i,ub}$ be the current docID and upper-bound score of the list pointer L_i, respectively. We compute a *total_upperbound* array, which is the inclusive prefix sum over the list upper-bounds, calculated by $total\_upperbound[i] = \sum_{j=i}^{n-1} L_{j,ub}$. The important observation is that $total\_upperbound[i]$ will return the upper bound on the score of a new document that is composed of only partial scores from list pointers at positions greater than or equal to i.

MaxScore selects the list pointer with the smallest current docID *next_doc_id*, that has a total upper bound on its score greater than the threshold τ (the k-th largest document score so far). This requires a linear search of the array of list pointers, in decreasing order of their upper-bounds; the search can terminate when $\tau > total\_upperbound[i]$. When true, this condition indicates that the list pointers beyond the i-th position do not have sufficiently high upper-bounds to produce a docID that can make it into the top-k without help from the lists at positions less than or equal to i. Thus, all list pointers at positions greater than i can be safely advanced to at least *next_doc_id*.

MaxScore then computes the sum of the actual partial scores of the list pointers currently at docID *next_doc_id* to get the exact document score. However, MaxScore also includes an additional performance improvement that allows it to terminate the scoring of a docID. During the evaluation of a docID, we know the current actual partial score from the lists we scored so far, and the remaining total upper bound on the lists from which we have not yet evaluated the docID. Thus, while traversing the list pointers array to compute the score of *next_doc_id*, we check whether the threshold is greater than the sum of the current actual partial score plus the remaining upper-bound; if true, we can safely stop further scoring of the document, since it cannot make it into the top-k.

Lastly, MaxScore can terminate processing when $\tau > total\_upperbound[0]$. However,

this early termination optimization is only likely to help in a handful of queries, since it requires at least one list has been fully processed so that the *total_upperbound* array can decrease in value. As an example, this optimization can speedup up evaluation of a two term query, with both a very rare and common keyword; once the short list for the rare term is processed, and if the range of docIDs in the short list is smaller than the range of docIDs for the common term, evaluation can end early for the large list. However, most of the performance increase in this case will come from the list skipping introduced by the rare term driving lookups in the common list, once the threshold value increases sufficiently.

Compared to the unoptimized DAAT-OR algorithm, which uses a heap to select the lowest docID after each list move (a $O(log(n))$ operation), the major cost of MaxScore is the linear $O(n)$ search for selecting the lowest docID that can make it into the top-k. However, the cost of selecting a list pointer decreases as the threshold grows, since the search can terminate early. As can be seen in the experimental results in Subsection 6.2.2, MaxScore outperforms DAAT-OR significantly because of list skipping and fewer score computations, and even outperforms WAND, which has the larger overhead of sorting the list pointers.

4.4.1 MaxScore Optimizations

Strohman and Croft [26] describe a scheme that implements MaxScore with a candidate list of top documents for each inverted list. The standard DAAT MaxScore algorithm can be seen as maintaining candidate lists consisting of only a single top document, representing the upper bound on each list score. Like the standard MaxScore algorithm, MaxScore with candidate lists of top documents is also score-safe, but the authors achieve even better performance than standard MaxScore. By establishing a lower-bound score on the candidate list documents first (lower-bound because lookups are not made for candidate list documents occurring only in some of the lists), the MaxScore algorithm can start off with a higher threshold value and

better upper bounds on the partial scores of each list, allowing it to be better at list skipping and terminating full evaluations of docIDs, as described previously.

Strohman [15] discusses another optimization to standard MaxScore that, in addition to list skipping by docID, requires the index to have the capability to skip ahead to a posting having a minimum arbitrary score. The changes to the MaxScore algorithm to take advantage of the score skipping capability are minor. Similar to the candidate lists optimization described previously, embedding fine-grained upper-bound scores into the index allows MaxScore to take advantage of more accurate upper bounds on the partial scores of documents to enable better list skipping. This optimization is particularly attractive because it does not require an additional index layer (candidate list). This gives the MaxScore with score skipping algorithm an advantage of neither having to tune the size of the candidate lists nor requiring an additional computational step to evaluate the candidate lists.

5 Experimental Setup

All of our experiments were conducted on a system named *Pangolin*; it is a two socket machine with quad core Intel Xeon E5520 CPUs with Hyper-Threading enabled, each running at 2.26 GHz (with a turbo frequency of 2.53 GHz — that is, each core is capable of being automatically boosted to a higher frequency during light usage or idling of the other cores). Each core has a 256 KB L2 cache and there is an 8 MB shared L3 cache. Pangolin has a total of 72 GB of RAM, but it is a NUMA architecture [27], where each processor can directly access 36 GB of RAM; accessing the remote RAM would incur a penalty for reads and writes for the corresponding processor. In our experiments, the parts of the index used by the query log were loaded into main memory (through memory mapping). Memory usage of our process was well under 36 GB, as our non-positional indices ranged in size from 9.5 to 12 GB (including the layered indices). An in-memory index circumvents the slow random access times due to disk seeking, and allows us to ignore the effects that various caching algorithms would produce, in order to only focus on the performance of the query algorithms. We tried to run experiments during times of little load since CPU usage and memory bandwidth would affect our timed runs.

The dataset we chose to run our experiments on was the Gov2 corpus, also used in the TREC Terabyte Track[1]. It is a corpus of 25 million documents, with an uncompressed size of 426 GB. It is distributed as 80 GB of gzipped bundles, all of which were used during the indexing step.

The query log used in these experiments was from the 2006 TREC Efficiency Task. This query log was first pre-processed by converting any characters not in the ASCII character set (those above decimal value 128), to their closest ASCII equivalents (e.g. *ó* to just *o*, or *ñ* to just *n*). Afterwards, any non-alphanumeric characters were converted into spaces, since we do not index words with non-alphanumeric characters;

[1] http://trec.nist.gov/data/terabyte.html

this resulted in an average query length of 4.168 terms. Using the full 100,000 query log, we extracted queries with lengths of only one, two, three, all the way up to ten terms. These will be used to see how the various query algorithms perform on different query lengths. The average query latency experiments used the first 10,000 queries from the full query log. Neither stemming nor a stop word list was utilized.

The PolyIRTK software was compiled with gcc 4.4, using the -O3 optimization flag, and assertions were removed. The index was memory mapped and the query log was run twice; the first time around, for warm-up purposes, the utilized portions of the index are mapped to the virtual pages in our process, and the second time around, with the inverted lists in main memory, we would time each query. We ran the queries in batch mode, where all queries were read into an array in main memory on startup. Document map lookups (to convert docIDs to document URLs) were not made, as they would introduce random disk lookups that would normally not happen in a production search engine.

All of the indices used in the query latency experiments were built without positions. The fixed block size used was 64 KB, composed of chunks of a maximum size of 128 postings. For the final merged index, we did not compress the block header, so as to not inhibit list skipping. The docIDs were compressed with PForDelta (the first version described in Section 2.3), with a block size of 128 integers, but if we had less than 96 postings in one chunk, we would compress the chunk with Variable Byte coding instead. The frequencies were compressed with S16 coding. In experiments comparing the indexing time and index size of a word-level index, positions were compressed with Rice coding.

6 Experimental Results

6.1 Information Retrieval Toolkit Performance

In this section, we empirically evaluate the performance of the major operations supported by PolyIRTK.

6.1.1 Indexing and Merging Performance

Table 6.1 shows the sizes of the various parts of the inverted index and other important on-disk data structures created for the final merged index. The size of the positions is taken from a separate word-level index; they were compressed with Rice coding, as it compresses the positions better than other algorithms (although it is more expensive to decode).

Collection Size (gzipped)	80.70 GB
Num Docs	25,205,179
Lexicon Size	1.83 GB
Num Unique Terms	37,728,619
Basic Document Map Size	288.45 MB
Extended Document Map Size	3.01 GB
Inverted Index Total Size	9.54 GB
DocIDs	6.10 GB
Frequencies	2.78 GB
Block Headers	662.05 MB
Block Wasted Space	13.88 MB
Positions	24.54 GB

TABLE 6.1: This table shows the index statistics for the Gov2 document collection. The docIDs were compressed with PForDelta with a block size of 128; however, if there were less than 96 docIDs in one chunk, they would be compressed with Variable Byte coding instead. The frequencies were compressed with S16 coding, while the block headers were left uncompressed. The positions were compressed with Rice coding. The block wasted space is due to our 64 KB fixed block size.

In Table 6.2, we look at how indexing and merging performance scales with the size of the memory pool used during indexing. Since the number of intermediate indices generated depends on the amount of memory dedicated to buffering compressed postings, larger memory pools produce shorter indexing and merging times. In these experiments, the vocabulary hash table size was configured in relation to the size of the memory pool because it is not reasonable to assume that a user indexing with a 128 MB memory pool can afford a 1 GB vocabulary table (it is actually likely to result in worse performance due to its sparseness, thereby creating a lot of cache misses). Since the size of the vocabulary does not grow linearly with the size of the collection, to configure the vocabulary hash table sizes, we used an initial indexing step for each memory pool size to create one index run. We then examined the number of unique terms in that index run and used it configure the hash table size. Generally, a good hash function will help to make lookups linear over a load factor from 0 up to around 0.7, so we sized the hash table to have a maximum load factor of 0.7 at its peak; the move-to-front optimization should also help to make the load factor a minor issue in general. The vocabulary hash table sizes in Table 6.2 are stated in MBs; our implementation uses an array of pointers and we assumed a pointer is 8 bytes (64-bit architecture).

Indexing time decreased by 31% when going from a 128 MB to 1 GB memory pool, and merging time decreased by 49%. However, there were diminishing returns in indexing speed when moving up to a 4 GB memory pool, with only a 3% decrease in time. Nevertheless, the merge time still decreased by 53% due to fewer overall indices. The merge time is dependent on the number of passes required over the collection (determined by the merge degree) and the total number of indices to merge, so it is unsurprising that larger memory pools lead to faster merge times.

Building a word-level index with a 4 GB memory pool takes roughly the same amount of time, but merge time is increased significantly. This is a result of handling more than three times as much data (with two merge passes over the postings), with posi-

Memory Pool Size	Vocab Size (MB)	Index Time (hr:min)	Num Indices	Indices Size (GB)	Merge Time (hr:min)
128 MB	6	5:57	405	12.08	1:14
1024 MB	30	4:06	36	10.28	0:38
4096 MB	100	3:59	8	9.83	0:18
4096 MB (positions)	100	3:58	17	35.00	1:13

TABLE 6.2: This table shows the indexing and merging performance with respect to the size of the memory pool used during indexing. The move-to-front hash table was sized relative to the size of the memory pool because the longer we accumulate in-memory postings, the higher the load factor on the hash table. The merge degree was set to 16 in all cases. Since the final index size will be the same as shown in Table 6.1, here we show the number of intermediate indices produced (before merging) and the total size of the intermediate indices.

tions decoded and encoded with the slower Rice coding method.

6.1.2 Unoptimized Querying Performance

Table 6.3 shows the unoptimized querying performance for the disjunctive (DAAT-OR) and conjunctive (DAAT-AND) algorithms with regards to different cache sizes, including memory mapping the index. We also show the average amount of data read from disk and cache in Table 6.4, to analyze the effect of caching.

The cache size is specified in number of blocks in the PolyIRTK configuration file; the minimum cache size is determined by the maximum number of unique terms allowed in a query, multiplied by the number of blocks we read ahead into a list. The cache is designed this way so that blocks that were read ahead would not get evicted early, before they have been processed. Our read ahead was configured to 2 MB (32 blocks) and we assumed query lengths would not exceed 32 terms, making the minimum cache size 64 MB. Note that PolyIRTK will not read ahead blocks beyond the current inverted list, since it knows the total number of blocks per list.

The cache configurations tested were 64 MB (1,024 blocks), 512 MB (8,192 blocks),

3 GB (49,152 blocks), and Memory Mapped. Aside from fully caching the index, memory mapping has an advantage in not having to manage LRU cache data structures. To generate the data shown in Tables 6.3 and 6.4, we used the first 99,000 queries from the full 100,000 TREC query log to warm up the index cache; we then timed the last 1,000 queries to get the average data read and the average latencies. From Table 6.3, we see that query latencies for both the DAAT-OR and the DAAT-AND algorithms improve only slightly with the increasing cache configurations. These small improvements are most likely due to the OS caching most of the inverted index file during our warm-up run (our machine has 72 GB of RAM) and running only one query processor instance at a time. The cache sizes would likely have a more significant impact when running simultaneous query streams and especially when the index size is much larger relative to the machine's available memory.

From Table 6.4, we see that with a 3 GB cache size (roughly 30% of the index), most queries can be answered from the cache. We note that with LRU block caching, even though we cache at the block level and not complete lists, most blocks of a single list will be evicted together.

	Cache Configuration			
Query Algorithm	**64 MB**	**512 MB**	**3,072 MB**	**Memory Mapped**
DAAT-OR	532.68	532.11	531.67	529.16
DAAT-OR (Reassigned)	455.41	454.29	453.10	452.26
DAAT-AND	22.45	20.17	17.82	17.24
DAAT-AND (Reassigned)	13.14	10.45	9.25	8.31

TABLE 6.3: This table shows the average query latencies, in milliseconds, for the unoptimized DAAT-AND and DAAT-OR algorithms over increasing cache sizes. We also include the latencies for queries run over an index having docIDs reassigned by document URLs.

6.1.3 DocID Reassigned Index Querying Performance

As Table 6.5 shows, reassigning docIDs by document URLs yields significant savings in index size, a 22% decrease when compared to the index size listed in Table 6.1. A

Query Algorithm		Cache Configuration 64 MB	512 MB	3,072 MB	Memory Mapped
DAAT-OR	**D**	15.82	5.70	0.69	0.00
	C	1.51	11.63	16.63	17.32
DAAT-OR (Reassigned)	**D**	11.90	3.66	0.26	0.00
	C	1.90	10.15	13.54	13.80
DAAT-AND	**D**	15.43	5.71	0.69	0.00
	C	1.46	11.18	16.20	16.74
DAAT-AND (Reassigned)	**D**	11.60	3.64	0.26	0.00
	C	1.82	9.78	13.15	13.19

TABLE 6.4: This table shows the average data read, in MBs, from disk and cache (with each row marked with the letter *D* or *C*, respectively) over increasing cache sizes. These numbers are based on the amount of 64 KB blocks fetched by the query processor.

smaller index keeps down the costs of caching and improves query performance. We show DAAT-OR and DAAT-AND query latencies in Table 6.3, as well as the amount of list data fetched from disk and the cache in Table 6.4. Reassignment of docIDs yields a 15% latency improvement in DAAT-OR and a 52% latency improvement in DAAT-AND, for the memory-mapped querying performance. DAAT-AND shows the biggest improvement because the Boolean AND filter takes advantage of the better list skipping in the reassigned index. These numbers could be further improved by utilizing the PForDelta code optimized for clustering of docIDs, which we discussed in Section 2.3.

Reassigned Inverted Index Size	7.42 GB
DocIDs	4.47 GB
Frequencies	2.29 GB
Block Headers	661.79 MB
Block Wasted Space	9.32 MB

TABLE 6.5: This table shows the size of a docID reassigned index by URL. We do not show the index position sizes since reassignment would affect them only minimally.

6.1.4 Query Effectiveness

In this subsection we show query effectiveness results produced by the unoptimized conjunctive (DAAT-AND) and disjunctive (DAAT-OR) algorithms, using the TREC 2006 ad-hoc topics. To show that PolyIRTK is capable of reasonably answering queries, we will compare the result precision to another efficient system.

PolyIRTK was configured to output the results in a format suitable for running the `trec_eval` utility, developed by the TREC community. The `trec_eval` program uses a corresponding result set for each query in the ad-hoc topics, where documents are marked as relevant, somewhat relevant, or irrelevant. For the TREC 2006 ad-hoc queries, 5,893 documents were marked as definitely or possibly relevant and 26,091 documents marked as not relevant. It then outputs query statistics after comparing it to our results. We look at the standard information retrieval result evaluation metrics: *precision*, *recall*, and *mean average precision* (MAP).

Precision is the fraction of relevant documents retrieved out of the total number of retrieved documents, while recall is the fraction of relevant documents retrieved out of the total number of relevant documents in the collection. Typically, it makes sense to talk about precision and recall at k retrieved documents, because of the two extremes of precision and recall; when returning no documents, precision is 100% and recall is 0%, while when retrieving every document, precision moves towards 0% and recall is 100%. The MAP measure is the average of the precision measured at different values of k, so it includes both aspects of precision and recall.

PolyIRTK returned a maximum of 1,000 results per query and obtained a MAP of 0.2132 and precision@20 of 0.4380 for disjunctive query semantics. This number is in the ballpark for other systems, albeit, slightly lower than the Gallago system described in [15], which obtained a MAP of 0.2592 and P@20 of 0.4590. This can likely be attributed to our simple HTML parser and lack of stemming support, which were not the focus of this work. We also did not use a stop list at index or query

time.

Table 6.6 compares the precision and recall of both conjunctive (AND-type) and disjunctive (OR-type) queries when retrieving up to 10,000 results per query. It makes sense to look at precision at k for smaller values and recall at k for larger values of k. Perhaps not intuitively, the conjunctive queries perform slightly better than disjunctive queries under the precision metric. Since we measured precision at small values of k, this is likely due to the conjunctive semantics filtering out documents containing only some of the query terms which happened to be irrelevant (but they were still highly ranked under disjunctive semantics). On the other hand, looking at recall@10,000, disjunctive queries do better by returning more relevant documents overall. The conjunctive semantics cause PolyIRTK to completely miss some relevant documents.

This is important because search engines generally would not rank documents only by a simple measure like $TF \cdot IDF$. A search engine would use many features of a document, including PageRank and term proximity, with the exact function likely being determined by a machine-learning algorithm combining many other features. Since this function would be expensive to evaluate on all documents, it makes sense to retrieve the top-m documents for large values of m, with a simple ranking function, and then further rank these documents with a thorough ranking function to get the final top-k, suitable for presentation to the user. For such a task, good recall for the initial top-m would be important, and so disjunctive queries would be preferable given the data in Table 6.6. Speeding up disjunctive queries while maintaining the same result quality was the focus of Chapter 4, and we present performance results in the next section.

6.2 Rank-Safe Algorithms Querying Performance

In this section, we compare the performance of the rank-safe query optimization algorithms we have implemented. These algorithms use the disjunctive query semantics

	Top-10,000 OR	Top-10,000 AND
Relevant Results	5,893	5,893
Results Returned	478,064	142,081
Relevant Results Returned	4,697	3,958
Precision @ 10	0.4620	0.4640
Precision @ 100	0.2808	0.3070
MAP	0.2132	0.2392
Recall @ 1,000	0.5759	0.6034
Recall @ 10,000	0.7970	0.6716

TABLE 6.6: This table shows the query effectiveness results for conjunctive and disjunctive query semantics.

and their effectiveness results are the same as presented in Table 6.6.

6.2.1 Splitting Lists Into Layers

The Multi-Layer algorithm's performance is closely tied to the selection of the number of list layers and their sizes. Table 6.7 shows how performance (average latency and list data accessed) varies for different layer generation strategies. Along with the strategy used, we show the maximum number of layers that result from each strategy; depending on list length, there could be fewer layers. For comparison, we also show the performance of DAAT approaches to querying a layered index; we adapted the unoptimized DAAT algorithm and the MaxScore algorithm to query over multiple layers by treating each layer as an independent list. We chose MaxScore over WAND because in our experiments, in Subsection 6.2.2, MaxScore performed better than WAND in not only the average cases, but also across increasing number of query terms. The performance across increasing number of query terms is important since, in the case of layered DAAT processing, we could be merging $n \cdot q$ lists, where n is the maximum number of list layers and q is the number of query terms.

These experiments were conducted in the same manner described in Chapter 5, but we instead selected 1,000 queries randomly from the full 100,000 TREC query log.

The Multi-Layer algorithm that appears in Subsection 6.2.2, where we compare all of the query algorithms, is the optimized TAAT version that uses the layer parameters from the best method we determine here, which is the *Exponential Pow2* strategy with a maximum of 5 layers.

We have previously discussed the ways in which PolyIRTK generates layers in Section 3.6, including the two exponential strategies, *Fib* and *Pow2*. The *Equal* and *Percentage* strategies presented in Table 6.7 are both specified as fixed percentages of postings for each layer. The *Equal* strategy simply assigns an equal percentage of the postings to each layer. The *Percentage* strategy was specified to recursively divide each layer into two, containing 25% and 75% of the postings from the layer being divided; we would then assign the resulting percentages to layers in increasing order. For example, for two layers, the percentages assigned would be 25% followed by 75%. Then, for four layers, the 25% and 75% ratio would be applied to each of the previous two layers to get 6.25% (25% · 25%), 18.75% (25% · 75%), 18.75% (75% · 25%), and 56.25% (75% · 75%).

Generally, we expect that having larger successive layers will produce better results, since AND-mode processing would allow list skipping in the larger layers. This is supported by Table 6.7, where the *Equal* layer strategy performs the worst out of all the other strategies tested. Having more layers overall allows for additional opportunities to switch to AND-mode processing, however there are also overheads involved with more layers, such as pruning accumulators and worse compression (although that is largely influenced by how the layers are sized as well).

Since we memory mapped the index, the average list data accessed shown in Table 6.7 is not divided into cache and disk portions. We show the average list data accessed because the optimized Multi-Layer TAAT strategy should be the most effective algorithm at early termination, since it can stop before processing every layer. From our discussion in Section 4.4, we know that MaxScore is also able to early terminate list processing in some rare circumstances, but generally requests all of the index blocks,

and does skipping at the chunk-level. As a side note, PolyIRTK has the capability to skip entire blocks through an in-memory block-level index, however, we found that skipping at the block level is very rare, so average query performance was actually slightly worse due to minor overheads of using the block-level index. Table 6.7 confirms that Multi-Layer TAAT and Multi-Layer MaxScore are able to early terminate. From the average MBs accessed per query, we see that Multi-Layer MaxScore does slightly better than Multi-Layer DAAT, and Multi-Layer TAAT improves on it further. Based on these numbers, we believe that the majority of the performance improvement obtained by the Multi-Layer TAAT algorithm is obtained through list skipping (AND-mode processing) rather than early termination.

We also see that the list data accessed grows as the number of layers increases. This is primarily due to worse compression, as d-gaps become larger with documents split across more layers. Another minor reason is that adjacent list layers share blocks. The query processor fetches these blocks more than once (only in terms of accounting — they are already in the cache), which slightly increases the list data accessed.

Multi-Layer DAAT unsurprisingly does not perform very well; it is mainly included for comparison to the Multi-Layer MaxScore algorithm. Multi-Layer MaxScore should give tighter upper bounds on each list layer, but performs poorly for many layers since each layer is treated like a separate inverted list; MaxScore has to linearly search the list pointers array after each list movement forward, so for the *Equal* and *Percentage* strategies with 8 layers, it performs even worse than the Multi-Layer DAAT algorithm. For strategies producing fewer layers, Multi-Layer MaxScore improves significantly on the unoptimized DAAT algorithm, but does worse than Multi-Layer TAAT, and cannot improve on the single layer MaxScore performance we show in Subsection 6.2.2.

It appears that the *Exponential Fib* strategy performs worse than the *Exponential Pow2* strategy. The reason is likely that in the *Exponential Fib* strategy, the initial layers still do not grow big enough, resulting in more layers being processed before the switch to AND-mode can take place. While we chose the *Exponential Pow2* strategy

with 5 layers to compare to the other algorithms in Subsection 6.2.2, we note that the *Equal*, *Percentage*, and *Exponential Fib* strategies with 2 layers performed very similarly and had some of the lowest list data accessed numbers, likely due to better compression.

6.2.2 Comparison of Query Algorithms

Table 6.8 and Figure 6.1 show the top-k retrieval latencies for all of the query algorithms. For top-10 retrieval, the DAAT-OR algorithm is 30 times slower than DAAT-AND, but the optimized querying algorithms bring that down to 2 to 5 times slower, depending on the algorithm.

The DAAT-AND and DAAT-OR algorithms are generally unaffected as the number of retrieved results increases from 10 to 10,000 since the only extra cost involved is maintaining a larger min-heap; since heap operations are $O(log_2(k))$, it is a minor cost to scale up k. The optimized, rank-safe algorithms, however, degrade in performance since they rely on having a tight threshold score to skip postings. For example, top-10 retrieval is 92% faster for MaxScore than unoptimized DAAT-OR, but it is only 59% faster for top-10,000 retrieval. The Multi-Layer TAAT algorithm is slightly faster than WAND for all top-k retrieval values until k = 10,000; we suspect that the Multi-Layer TAAT algorithm would benefit from having larger initial layer sizes when retrieving the top-k results for large values of k, since it would need to process more postings before being able to switch to AND-mode layer processing.

Figures 6.2, 6.3, 6.4 and 6.5 show how the algorithms perform over different query lengths. Generally, the more query terms there are, the worse the query latencies become. The only exception is DAAT-AND queries whose average latencies largely remain constant across increasing number of terms and increasing values of k; latencies could even benefit from having more query terms since better skipping is possible. The query latencies for DAAT-OR and DAAT-AND do not vary much as we increase the number of top results retrieved, and thus we only include them in

Layer Strategy — Max Num Layers	Multi-Layer TAAT	Multi-Layer DAAT	Multi-Layer MaxScore
Equal — 2	227.28 ms 17.09 MB	720.50 ms 17.14 MB	265.52 ms 17.12 MB
Equal — 4	239.60 ms 18.49 MB	1011.82 ms 18.57 MB	623.94 ms 18.57 MB
Equal — 8	310.34 ms 20.64 MB	1459.92 ms 20.77 MB	1790.25 ms 20.76 MB
Percentage — 2	165.55 ms 16.86 MB	692.17 ms 16.89 MB	217.27 ms 16.88 MB
Percentage — 4	168.96 ms 18.15 MB	984.17 ms 18.20 MB	515.45 ms 18.19 MB
Percentage — 8	215.68 ms 20.06 MB	1415.51 ms 20.13 MB	1462.34 ms 20.12 MB
Exponential Fib — 2 (k = 2)	163.45 ms 16.34 MB	589.40 ms 16.36 MB	208.47 ms 16.34 MB
Exponential Fib — 4 (k = 4)	181.93 ms 16.90 MB	779.70 ms 16.92 MB	495.79 ms 16.91 MB
Exponential Fib — 8 (k = 8)	210.39 ms 18.18 MB	1120.41 ms 18.22 MB	1105.02 ms 18.21 MB
Exponential Pow2 — 2 (k = 2)	166.37 ms 16.36 MB	590.55 ms 16.38 MB	208.25 ms 16.36 MB
Exponential Pow2 — 3 (k = 4)	157.54 ms 16.82 MB	688.83 ms 16.85 MB	276.85 ms 16.84 MB
Exponential Pow2 — 5 (k = 8)	157.07 ms 17.55 MB	879.17 ms 17.59 MB	479.97 ms 17.58 MB

TABLE 6.7: Multi-Layer query performance across different layer generation strategies and maximum number of layers for the optimized TAAT, unoptimized DAAT, and DAAT MaxScore approaches. The average list data accessed is based on how many blocks (64 KB each) were requested, on average, to answer a query.

	Top-k			
Query Algorithm	**10**	**100**	**1,000**	**10,000**
DAAT-AND	18.66	19.69	19.85	19.54
MaxScore	42.85	64.93	108.70	232.61
Multi-Layer TAAT	63.54	95.18	145.94	327.21
WAND	91.19	133.21	191.22	304.03
DAAT-OR	519.29	541.65	521.04	539.17

TABLE 6.8: This table shows the average top-k query latencies, in milliseconds, of all implemented algorithms in PolyIRTK over increasing values of k.

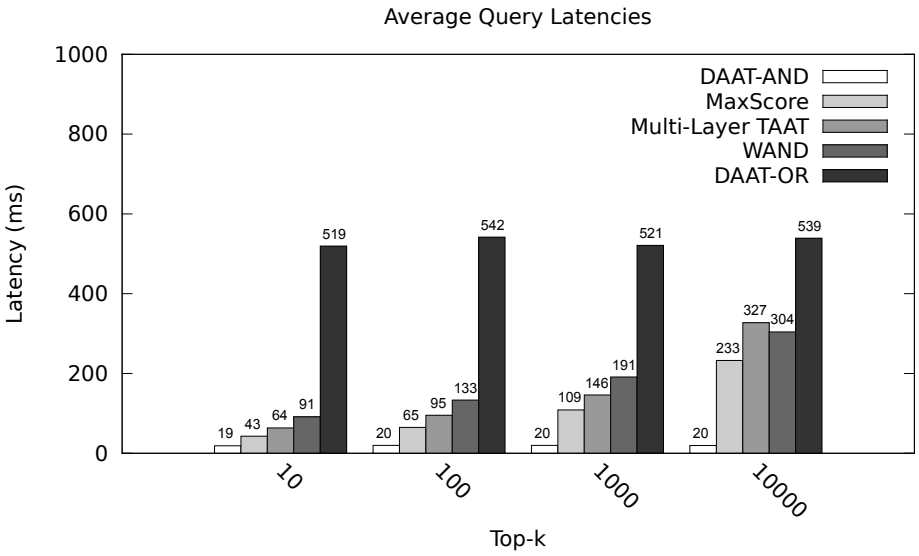

FIGURE 6.1: This figure shows the average top-k query latencies of all implemented algorithms in PolyIRTK over increasing values of k.

Figure 6.2. Figures 6.3, 6.4 and 6.5 only include the optimized query algorithms for improved readability, which have noticeably different performance characteristics under increasing values of k.

The WAND algorithm's performance with respect to longer query lengths degrades quicker than MaxScore (while they are initially almost equal in latencies for queries with one to three terms). This is likely due to having to sort list pointers ($O(n \cdot log(n))$), while MaxScore linearly scans the pointers ($O(n)$), and can even terminate the scan early. Similarly, the Multi-Layer TAAT algorithm, which has accumulator

management overheads upon processing each additional layer, is outperformed by MaxScore. As we increase the number of top results retrieved, we see that MaxScore scales to longer queries better than the other rank-safe algorithms. In general, MaxScore appears to have the least overheads, and thus it has the lowest average query latencies in our experiments.

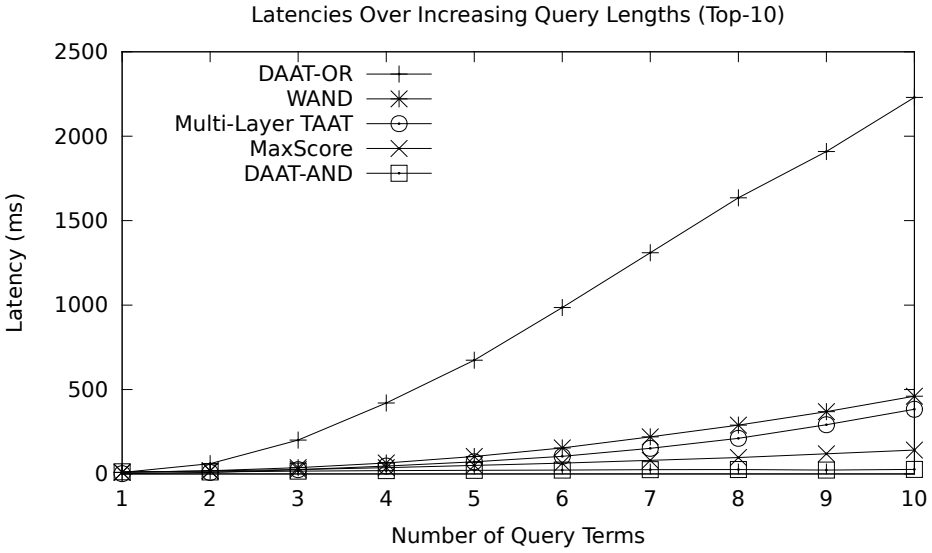

FIGURE 6.2: This figure shows how all the implemented query algorithms scale with respect to the number of terms in a query for top-10 retrieval.

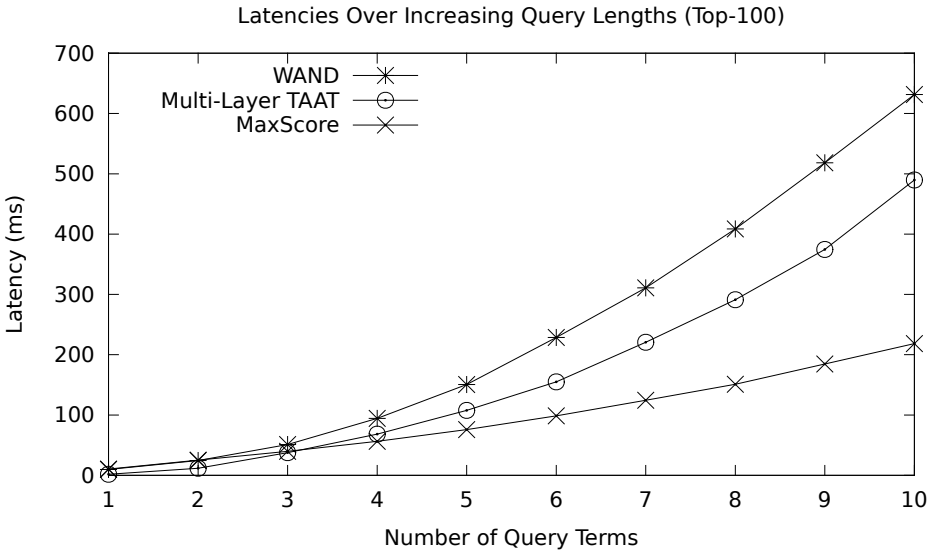

FIGURE 6.3: This figure shows how the optimized query algorithms scale with respect to the number of terms in a query for top-100 retrieval.

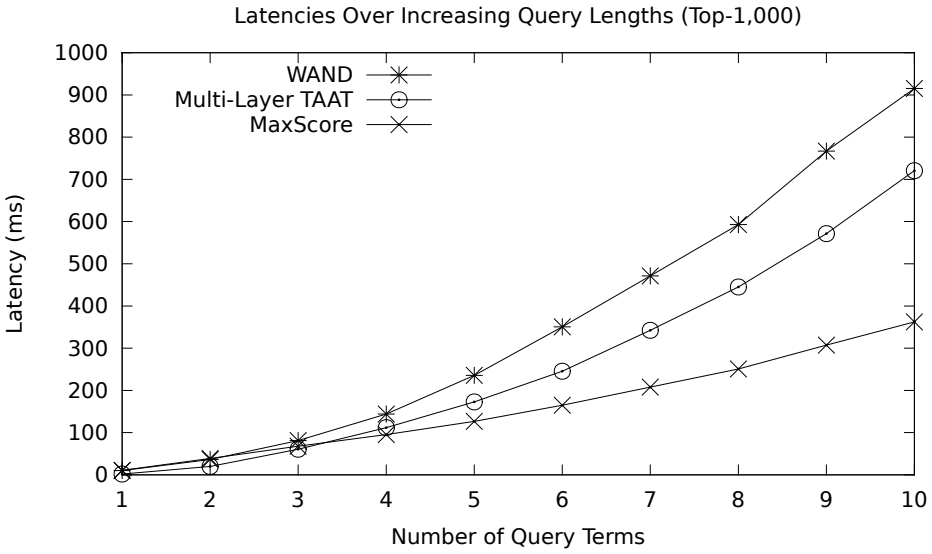

FIGURE 6.4: This figure shows how the optimized query algorithms scale with respect to the number of terms in a query for top-1,000 retrieval.

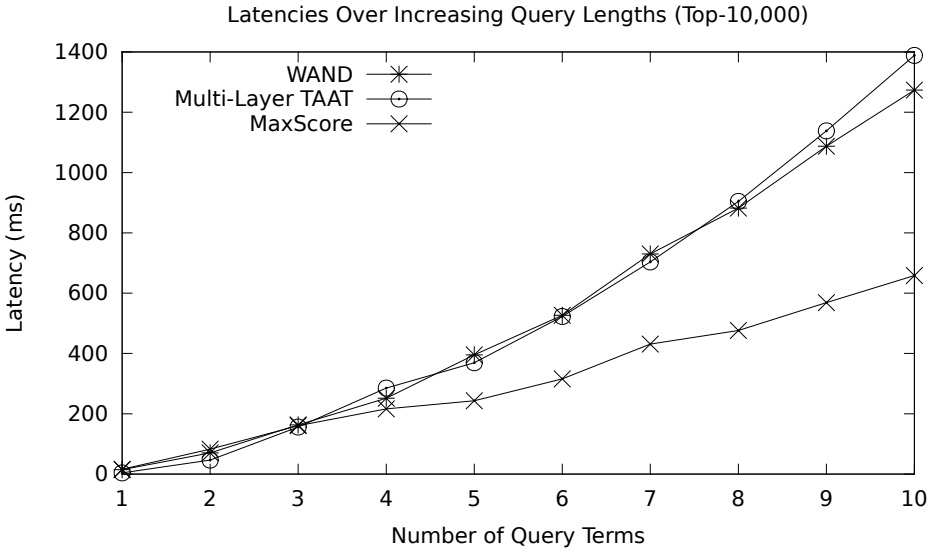

FIGURE 6.5: This figure shows how the optimized query algorithms scale with respect to the number of terms in a query for top-10,000 retrieval.

7 Conclusion and Future Work

We have described the implementation of an information retrieval program called PolyIRTK that is able to efficiently index and query large web collections. Additionally, we have evaluated several top-k query optimization algorithms that return rank-safe results, and compared their performance on disjunctive queries.

There are a variety of features that would make PolyIRTK more useful and flexible, which we leave for future work; here, we list several important ones. During the development of PolyIRTK, we primarily focused on indexing English text. However, Unicode support is nowadays essential for real applications, and new research datasets also include international pages. For this reason, the PolyIRTK parser should be able to handle UTF-8 (a very common character encoding standard) text and allow for user specified tokenization rules, which could include characters beyond the standard ASCII spaces and newlines. Of course, this would also require changes in how the lexicon stores the corpus vocabulary.

While PolyIRTK was designed for general web collections, it can be extended to query more structured data by allowing searches only on particular document fields. It would also be useful to mix conjunctive and disjunctive queries, support more Boolean filtering options, such as a NOT operator and filtering on position distance between terms. These features would no doubt be useful to developers who need full text search in their own applications, as well as advanced users of a search engine.

Another important feature, index updating — deleting and adding documents, is useful for continuous index maintenance; it allows for document updates to quickly propagate over to the search results, which is desirable for many applications. Typically, document updates are buffered in main memory and at some point flushed to disk. The lists of newly inserted and deleted documents can be treated as separate inverted index slices, so they can be traversed in parallel by the query processor,

therefore including all index updates. The index slice for deletions is necessary since deletes have to be done lazily. It is also sufficient for the deleted index slice to only store docIDs since it is a list of documents that are to be ignored from the main index. Index merges, which apply to both the slices with document additions and deletions, take care of optimizing the number of index slices needed to be searched by the query processor, and serve to actually remove the deleted docIDs.

Inevitably, at some point the index would either be too big to fit on a single machine or the machine would not have the computational resources to answer queries in the desired time. So, for very large document collections, it is essential to be able to shard the index across several machines. Typically, the sharding is done on a document level. Such a change would require the addition of networking logic, as well as a query integrator component, designed to merge results from several nodes.

To make more efficient use of todays multi-core machines, it is desirable to run multiple queries on a single machine. Even on a single core, running multiple queries could be beneficial, as one query stalls for I/O, the other query could continue processing. While this may increase individual latencies, it should help overall query throughput. PolyIRTK should not have much trouble running concurrent queries, however, it needs logic for accepting them and spawning multiple threads.

A final improvement to PolyIRTK that we mention here, is to make it more suitable as an information retrieval toolkit providing full text search for low memory devices, such as embedded systems and smartphones. The PolyIRTK indexing pipeline is already very configurable and memory efficient, although the index itself might also be built on a different machine and transferred to the phone, for static text collections, such as help files. However, querying could use improvement in the memory consumption of a few data structures. The lexicon and the basic document map are currently loaded into main memory for performance and simplicity, however, this is not necessarily required. The lexicon could be indexed by a B-tree, with both the internal nodes and leaves compressed with front coding. The B-tree can remain on disk or have its nodes

cached in memory for fewer disk seeks to look up a term. The basic document map is read into main memory due to the necessity of quick access to the document lengths for the BM25 scoring function. However, since inverted lists are always accessed sequentially in ascending order of docIDs, only part of the document map needs to be cached in main memory, with occasional loads from disk for the next sequence of docIDs. Having an efficient full text search solution is especially important on devices like smartphones, where extra CPU cycles would result in a lower battery life.

Testing all the optimized query techniques within a common framework allowed a direct way to compare the merits of each algorithm. In addition to query latency and list data accessed, it would be useful to measure query throughput by running concurrent queries, to see how each algorithm fares. Since we measured the list data accessed in blocks of 64 KB, it would also be useful to measure data access on a more granular level, so the algorithm skipping and early termination performance could be better characterized. Finally, there are some additional variations of the rank-safe algorithms which we did not explore, that would be interesting to compare, such as using chunk-level upper bounds on the scores to improve list skipping.

Bibliography

[1] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. 1999.

[2] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[3] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proceeding of the 17th International World Wide Web Conference*, pages 387–396, 2008.

[4] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International World Wide Web Conference*, pages 401–410, 2009.

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

[6] C. Middleton and R. Baeza-Yates. A comparison of open source search engines. 2007.

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

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] J. Zobel, S. Heinz, and H. E. Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277, 2001.

[10] T. Strohman and W.B. Croft. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 175–182, 2007.

[11] A.N. Vo and A. Moffat. Compressed inverted files with reduced decoding overheads. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 290–297, 1998.

[12] P.H. Kamp. You're doing it wrong. *Communications of the ACM*, 53(7):55–59, 2010.

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

[14] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 372–379, 2006.

[15] T. Strohman. *Efficient processing of complex features for information retrieval*. PhD thesis, University of Massachusetts Amherst, 2008.

[16] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[17] A. Ntoulas and J. 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*, pages 191–198, 2007.

[18] X. Long and T. Suel. Optimized query execution in large search engines with

global page ordering. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 129–140, 2003.

[19] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Proceedings of the Seventh World Wide Web Conference*, 1998.

[20] R. Fagin. Combining fuzzy information: an overview. *ACM SIGMOD Record*, 31(2):109–118, 2002.

[21] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 648–659, 2004.

[22] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. IO-Top-k: Index-access optimized top-k query processing. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 475–486, 2006.

[23] V.N. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 226–233, 2005.

[24] A.Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, pages 426–434, 2003.

[25] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[26] T. Strohman, H. Turtle, and W.B. Croft. Optimization strategies for complex queries. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 219–225, 2005.

[27] U. Drepper. What every programmer should know about memory. 2007.