EFFICIENT PROCESSING OF COMPLEX FEATURES FOR INFORMATION RETRIEVAL

A Dissertation Presented

by

TREVOR STROHMAN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 16, 2007

Computer Science

© Copyright by Trevor Strohman 2007

All Rights Reserved

EFFICIENT PROCESSING OF COMPLEX FEATURES FOR INFORMATION RETRIEVAL

A Dissertation Presented

by

TREVOR STROHMAN

Approved as to style and content by:

W. Bruce Croft, Chair

James Allan, Member

Emery Berger, Member

Csaba Andras Moritz, Member

Andrew Barto, Department Chair
Computer Science

To Rollin Strohman.

ACKNOWLEDGMENTS

Writing acknowledgments is just a little bit absurd. Acknowledgments read like a disclaimer, as if except for the few people mentioned here I deserve complete credit for finishing this work. Unfortunately there is no way to give credit to everyone who deserves it. Even thanking my teachers individually is impossible; I estimate that I have had more than a hundred of them in my academic career. To be honest, I would need to add all of the unofficial teachers: my friends, classmates, and authors of books and articles I have read. An attempt to be too exhaustive would only succeed in leaving out some important influential person who is all too deserving. Therefore, I have chosen to to aim for brevity with the disclaimer that literally hundreds of people deserve mention here.

My advisor, W. Bruce Croft, takes the kind of calculated risks that I assume one must take to have the kind of academic success that he has had. He took one of those risks on me, a practically-minded applicant who wanted to study databases at a school with no database program. In just four years, he has guided me from complete ignorance of Information Retrieval, through conference publications, a dissertation, and now a new textbook. I have never forgotten that Bruce took a chance in accepting me, and I have worked hard to make his bet pay off.

My dissertation committee members have been gracious enough to take time out of their busy schedules to read, comment on, and watch me defend this dissertation. I appreciate their time and greatly respect their contributions to science.

A large part of my first two years in graduate school was spent writing code for the Indri retrieval system. While the code writing was generally a solitary activity, Indri

is part of the Lemur project, which is controlled by the Lemur steering committee. My experience with that committee and the feedback of many of its members, especially including Howard Turtle, Jamie Callan, David Fisher, Kevyn Collins-Thompson and Paul Ogilvie, gave me a much needed additional perspective on Information Retrieval. Implementing a retrieval system with the help of these people kept me excited in those first years while the students around me researched theoretical issues that I struggled to comprehend. I appreciate Bruce for giving me this experience and the committee members for their incredible support.

It still amazes me that doctoral education in the sciences can be free. Of course it isn't actually free: my education and living expenses were funded by agencies and people who believe that advancing scientific knowledge is a good investment. The family of J.L. Moore offered me a fellowship for three years that made these years far more bearable for me and my family. The NSF (grant CNS-045018) and ARDA (grant CCF-0205575) provided direct funding for my research, although the opinions, findings and recommendations in this work are mine and not necessarily those of these sponsors. I thank the people at those agencies and I hope my research is somewhere near what they had hoped for. Finally, two companies, Monster and Lexalytics, decided to hire me as a consultant in my final year. Consulting helped me immensely, not just for the financial benefits, but for the transition experience into the industrial world.

The faculty, students and staff in the Computer Science department, and especially the CIIR, have been a welcome benefit to being a student here. I found none of the kind of competitiveness and tension here that can be a part of academic communities, which I credit to everyone in the department. While all the members of the lab have been extremely helpful to me, I want to single out André Gauthier, Fernando Diaz and Mark Smucker especially for their helpful conversations and insights. Don Metzler deserves his own distinguished mention for how helpful he has been as a co-researcher,

both on the Indri project, TREC, and joint work on binning. His talents have been an excellent fit for mine, and I will count myself lucky if I find a research partner again with whom I can work so well.

My parents, sister and in-laws have been so supportive of my graduate progress, even though it meant constant cross-country trips to see their new grandchildren. I hope that those grandchildren will soon live a little closer. My father in particular has been patient with my meandering academic plan, while instrumental in demonstrating the kind of work ethic it takes to succeed. My mother spent plenty of hours in my childhood making sure I had the kind of opportunities that would help me succeed academically. Soon our family reunions will contain five Dr. Strohmans. I blame my parents for causing this name confusion.

Finally, my wife Anne-Marie and children Evan and Natalie have put in their own efforts to see this dissertation to completion. They have patiently waited for me as I was away at conferences or spending late nights at work. Evan always has the kind of excitement that gives me extra energy even when work hasn't gone well. Natalie is a new joy and I hope that this degree will indirectly give her opportunities she wouldn't have had otherwise. And finally, to Anne-Marie: thanks for taking this graduate school adventure with me. You have supported me and encouraged me through constant self-doubt, and we have worked together to understand this odd place. Two kids and almost five years later, we are both almost finished and I am ready for the next adventure with you.

ABSTRACT

EFFICIENT PROCESSING OF COMPLEX FEATURES FOR INFORMATION RETRIEVAL

December 16, 2007

TREVOR STROHMAN

B.S., CALIFORNIA POLYTECHNIC STATE UNIVERSITY

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor W. Bruce Croft

Text search systems research has primarily focused on simple occurrences of query terms within documents to compute document relevance scores. However, recent research shows that additional document features are crucial for improving retrieval effectiveness.

We develop a series of techniques for efficiently processing queries with feature-based models. Our TupleFlow framework, an extension of MapReduce, provides a basis for custom binned indexes, which efficiently store feature data. Our work in binning probabilities shows how to effectively map language model probabilities into the space of small positive integers, which helps improve speeds without reducing query effectiveness. We also show new efficient query processing results for both document-sorted and score-sorted indexes. All of our work is evaluated using the largest available research dataset.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS v

ABSTRACT viii

LIST OF TABLES xiv

LIST OF FIGURES xvi

CHAPTER

1. INTRODUCTION 1

1.1 Overview 1
1.2 Our Approach 3
1.3 Contributions 5
1.4 Layout 6

2. BACKGROUND 8

2.1 The inverted index 8
2.2 Ranking documents 10

2.2.1 Ranking with Impacts 14

2.3 Search Tasks 15
2.4 Using Additional Features 16

2.4.1 Indri query language 17
2.4.2 Query examples 19
2.4.3 Probabilistic Interpretation 20
2.4.4 Bag of words 21
2.4.5 Adding Term Proximity 22
2.4.6 Document Mixture Models 23
2.4.7 Additional Document Features 24
2.4.8 The UMass 2005 Navigational Formulation 26

2.4.9 Query Expansion 27

2.4.9.1 Stemming 27
2.4.9.2 Synonyms 29
2.4.9.3 Pseudo-relevance feedback 30

2.5 Evaluating Effectiveness and Efficiency 31

2.5.1 Test Collections 31
2.5.2 Effectiveness 32

2.5.2.1 Precision 32
2.5.2.2 Success 33
2.5.2.3 Recall 33
2.5.2.4 Average Precision 34
2.5.2.5 Normalized Discounted Cumulative Gain 36

2.5.3 Efficiency 38

2.5.3.1 Experiments 41

2.6 Basic query evaluation 44

2.6.1 Score-ordered evaluation 45
2.6.2 Precomputed Phrase Lists 48

2.7 Optimization Types 49

3. TUPLEFLOW 51

3.1 Introduction 51
3.2 Example 52

3.2.1 A Traditional Approach 52
3.2.2 Using TupleFlow 54

3.3 Related Work 64
3.4 Model of Computation 67
3.5 Step Implementation 69
3.6 Execution 70
3.7 Code Generation 72

3.7.1 Hash functions 72
3.7.2 Comparators 74
3.7.3 Order Compatibility 75
3.7.4 Compression 76

3.8 Built-in Steps . . . 78
3.9 Storing Streams . . . 79
3.10 Checkpointing . . . 80
3.11 Sample Tasks . . . 82

3.11.1 Building an Index . . . 82

3.12 Rapid Experimentation . . . 82
3.13 Experiments . . . 84

3.13.1 Word Count . . . 84

3.13.1.1 Balance . . . 85
3.13.1.2 Compression . . . 87

3.13.2 Anchor Text Combination . . . 88
3.13.3 Indexing . . . 91

3.14 Weaknesses . . . 91
3.15 Summary . . . 92

4. BINNED PROBABILITIES . . . 94

4.1 Introduction . . . 94
4.2 Method . . . 95
4.3 Exploration . . . 97
4.4 Evaluation . . . 99
4.5 Results . . . 101
4.6 Summary . . . 101

5. SCORE-SORTED INDEX OPTIMIZATION . . . 103

5.1 Algorithm . . . 105

5.1.1 AND Processing . . . 108
5.1.2 Trimming Accumulators . . . 109
5.1.3 Ignoring Postings . . . 110

5.2 Implementation . . . 112

5.2.1 Indexing . . . 112

5.3 Choosing Skip Lengths . . . 113
5.4 Evaluation . . . 116

5.4.1 Analysis . . . 121

5.4.1.1 Multiple Cores 122

5.5 Related Work 123
5.6 Summary 125

6. DOCUMENT-SORTED INDEX OPTIMIZATION 127

6.1 Introduction 127
6.2 Algorithm 128

6.2.1 Traditional 128
6.2.2 Max-Score 130
6.2.3 Score Skipping 134

6.3 Index Construction 136

6.3.1 Inverted lists 136
6.3.2 Other structures 138
6.3.3 Construction with TupleFlow 139

6.4 Evaluation 139

6.4.1 Effectiveness 140
6.4.2 Efficiency 143

6.5 Related Work 144
6.6 Conclusion 147

7. NAVIGATIONAL SEARCH WITH COMPLEX FEATURES 149

7.1 Introduction 149
7.2 Model 150

7.2.1 Conversion 150
7.2.2 Feature Combinations 154
7.2.3 Index Construction 154

7.3 Evaluation 156

7.3.1 Effectiveness 156
7.3.2 Efficiency 158

7.4 Conclusion 161

8. EXTENSIONS **163**

8.1 Introduction 163
8.2 Update 163

8.2.1 Update background 163
8.2.2 Application Notes 167

8.3 Distribution 169

8.3.1 Background 169
8.3.2 Application Notes 170
8.3.3 Improvements 171

8.4 Query caching 172

9. CONCLUSION **174**

9.1 The Broad View 174
9.2 Contributions 176

BIBLIOGRAPHY **178**

LIST OF TABLES

Table **Page**

2.1 Definitions of some important efficiency metrics 38

4.1 Mean Average Precision over varying number of integer bins 100

4.2 Mean Average Precision over varying saturation parameter values. Bold values are significant improvements over $s = 1.0$ (t-test, $p < 0.05$). 100

4.3 Fraction of postings that are over the saturation probability for many saturation levels. 100

4.4 WT10G index sizes for various numbers of bins 101

5.1 Effectiveness on TREC Ad Hoc Queries 117

5.2 TREC 2005 Efficiency Queries, average query execution times, in milliseconds. Throughput is measured in queries per second. 118

5.3 TREC 2006 Efficiency Queries, average query execution times, in milliseconds. Throughput is measured in queries per second. 119

5.4 Speedup when using multiple cores. Throughput is measured in queries per second, while speedup is measured relative to each algorithm's performance on a single processor. 120

5.5 Efficiency at varying skip lengths, TREC 2005 Efficiency Queries 121

6.1 Effectiveness results from the GOV2 collection. 140

6.2 TREC 2005 Efficiency Queries, average query execution times, in milliseconds. Throughput is measured in queries per second. 141

6.3 TREC 2006 Efficiency Queries, average query execution times, in milliseconds. Throughput is measured in queries per second. 142

7.1 Parameter settings used in the UMass TREC 2005 experiments. 151

7.2 Parameter settings for features in our experiments. 153

7.3 GOV Collection, 2002 Navigational Queries 156

7.4 GOV Collection, 2003 Navigational Queries 156

7.5 GOV2 Collection, 2005 Named Page Queries 157

7.6 GOV2 Collection, 2006 Named Page Queries 157

7.7 Throughput results for 2005 TREC Efficiency queries. 159

7.8 Throughput results for 2006 TREC Efficiency queries. 159

LIST OF FIGURES

Figure **Page**

2.1 Complex query operators from Indri 18

2.2 Discounting factors used in NDCG. Note that the influence of a document at rank 20 is 1/8th of that of the first document. 36

2.3 A simple document-at-a-time retrieval algorithm. 43

2.4 A simple term-at-a-time retrieval algorithm. 44

3.1 A simple Python word count program. 53

3.2 The command line for building the `WordCount` type. The arguments shown here are: file name of the generated code, package name of the resulting class, class name of the result, `word` specification, `count` specification, and two `order` specifications. 54

3.3 Source code of CountsMaker.java 56

3.4 Some `WordCount` tuples, after processing by `WordCounter`. Notice that there are two lines for `the`. 56

3.5 `WordCount` tuples, after sorting in `+word` order. 56

3.6 Source code of CountsReducer.java 58

3.7 Stage description for `counting`. 59

3.8 Stage description for `reduce`. 61

3.9 Stage description for `filenames`. 62

3.10 Description of connections between the filenames, counting and reduce stages. 62

3.11 A simple TupleFlow execution graph. 67

3.12 A TupleFlow execution graph with a replicated stage. 68

3.13 The on-disk representation of a stream with 4 inputs and 2 outputs. 79

3.14 On-disk representation of a pass-through stream with 3 inputs and 3 outputs. 79

3.15 A TupleFlow computation graph for building a traditional, positions-based text index. Small boxes are steps, large boxes are stages, and gray boxes indicate stages that can be replicated. 81

3.16 A short Pig script to find all unique queries in a query log that appear more than ten times. 82

3.17 Total time and parsing time to count the words in the GOV collection for many levels of parallelism. 83

3.18 Processes running over time for many levels of parallelism. 84

3.19 An example of the tuples emitted by the WordCount parsing stage. 87

3.20 A stage diagram of the TupleFlow anchor text combination process. 88

3.21 DocumentURL tuples parsed from the GOV collection. 89

3.22 ExtractedLink tuples parsed from the GOV collection. 89

3.23 Speed of anchor text combination on the GOV collection for many levels of parallelism. 90

4.1 Distribution of binned values for four different terms. This data comes from the TREC45 collection, using 64 bins and saturation = 0.001. 98

5.1 Relative number of accumulators used during the query evaluation process. The gray filled area represents the usage pattern in Anh and Moffat. The thick solid line represents the decreased accumulator usage of our approach. 105

5.2 The expected cost of processing bins of varying lengths with varying skip sizes. The expected cost is expressed as a fraction of the cost of processing the data without skipping. Expectations are based on the 50,000 TREC 2005 Efficiency Track query set. 115

5.3 Distribution of query lengths (before removing stopwords) across collections. 117

6.1 A picture of the TupleFlow execution graph that builds document-sorted indexes. 137

7.1 A sample query formed using the UMass TREC 2005 Named Page formulation. 151

7.2 Simplified diagram of the TupleFlow job for constructing navigational indexes. 155

CHAPTER 1

INTRODUCTION

1.1 Overview

Nine years ago, Brin and Page published "The anatomy of a large-scale hypertextual Web search engine" [17]. Since then, it has been cited more than 3500 times. We would expect this paper to be popular, since it was the basis of an extremely successful and popular company. However, it is also interesting because it presents a fresh look at how to build a text search engine for web text. The paper includes contributions in web crawling, indexing, distribution and retrieval, and shows how all of these pieces have to work together to create a search engine for the web. The paper is interesting because of how all of these contributions work in concert.

The world has changed in the intervening nine years. The web has exploded in size, and web search has moved from a curiosity to a tool that common people rely on every day. The field of information retrieval has grown in response, and with both academics and industry employees working to make sense of the huge amount of information that we now face. We have noticed two important trends that we will focus on in this dissertation.

First, the scale of text search, both in terms of the quantity of text indexed and in the query load experienced, has grown at an amazing rate. Fifteen years ago, some of the largest text search systems were pay-for-use legal and news databases. The pay-for-use model, and the limited connectivity of most end-users, created a natural limit to the number of queries issued in a day. These databases were also domain-specific, which created a limit on the amount of data in a particular system. Contrast this

with web search, which is freely delivered to general users. These users have come to the Internet in droves, and have brought along massive quantities of user-generated content. It is this Internet reality that sets the new benchmark for search engine scale.

Second, this change in the user base of search has changed the search problem. Instead of legal experts typing long, precise queries, the new breed of user types short, vague queries, often just two or three words long. Their information needs are different too. A paralegal or intelligence analyst is recall-focused, meaning that they care about finding as many interesting documents as possible. The new search environment has brought about a kind of need: the navigational search. Users use the search engine to find specific new web pages, and are often only interested in a single page. Increasingly, users just want to re-find pages they have already seen [95], as the search engine starts to supplant the web browser's bookmark functionality. We see evidence of this change in popular web search interfaces, which have very few buttons or tools, small text boxes, and display just the top 10 results to each query.

These two trends have caused a shift in approach, especially among commercial vendors. Because of the massive scale of the problem, all major search engines use a distributed architecture for indexing and retrieval [17, 15]. In particular, the problem of indexing has spawned specialized distributed computing frameworks, like MapReduce and Dryad [42, 51]. The huge number of queries and the seek time gulf between memory and disk has made memory an attractive way to store indexes [89].

The huge increase in data and the changing needs of users has also caused a change in the algorithms for ranking pages. The number of results that users are willing to consider has dropped, so that now users do not even bother to scroll to see results [53, 3]. By contrast, the web now contains well over 10 billion pages [50]. This means that the fraction of the collection that users are willing to consider has decreased

dramatically. Metzler notes that this shift calls for additional discriminating power in ranking functions [66].

One way to improve document ranking is to extract more information from each document. Typical standard ranking methods rely on counts of individual terms in each document and in the document collection to determine document rankings. More complex rankings are possible, for instance, by treating word occurrences in a document title differently than the body text, or by detecting phrase occurrences. Previous systems have used these more complex formulations in the past, but often in an ad hoc manner. Recent trends focus on finding document relevance predictors, called *features*, and combining them into a ranking function using statistical machine learning methods [24]. Advances in feature combination methods are leading to ranking functions that use ever-increasing numbers of document features in more complicated combinations than ever before.

Because of this shift in the problem of search, we hypothesized that a holistic view of search systems, like Brin and Page a decade ago, might yield interesting results. As we will show in the rest of this dissertation, our hypothesis was correct. This work responds to the new search environment with a set of research contributions that are tightly interdependent. Our goal was to create an efficient search system for shorter queries that could make use of new detailed feature representations. Chapter 7 addresses this problem directly, but to do this it builds on contributions in distributed systems, efficient query processing, and feature binning. While this dissertation contains a set of contributions that can be used individually, we believe it is most interesting when viewed as a whole system.

1.2 Our Approach

Previous systems, including our Indri system [93], performed feature-based ranking by using complicated query processing code. In this dissertation, we transfer as

much ranking logic as possible to the indexer. Instead of storing individual document features like term counts or document lengths in the index, we store partial document scores. It is the job of the indexer to calculate these partial scores and store them in the index. This approach was taken in the SMART system, although it has fallen out of vogue recently [21]. While this approach reduces some flexibility at query time, it opens up a large number of opportunities for query optimization, as we will see in Chapters 5 and 6. Unlike most previous systems, our systems store partial scores as small positive integers. This allows us to compress indexes very tightly and improves retrieval speed.

As we mentioned before, trends in information retrieval are for ranking functions with ever increasing complexity. Therefore, storing partial scores in an inverted list requires a flexible indexing system that can take advantage of new feature extraction advances quickly, while also distributing load across an arbitrary number of computers. In this dissertation, we present TupleFlow, the basis of our indexing system, which helps us achieve flexibility and scalability.

Another issue is how to combine features together into small positive integers. In particular, statistical language models generate very small probability values that are much smaller than one. These language models also incorporate smoothing, which as we will see is difficult to integrate into an integer-based retrieval system. Our work on binning probabilities shows how language model features can be integrated effectively into the index.

We also show how these indexes can be used to retrieve documents very efficiently, using new query optimization techniques. At the end of this dissertation, we present a case study of using a previously developed feature-based ranking function with our indexing and retrieval system, and showing how it achieves high efficiency and effectiveness.

All experiments in this dissertation were performed using the Galago retrieval system[1], which is being made available as an open source toolkit.

1.3 Contributions

- **Score-ordered query optimization.** We add skipping and trimming to the best known algorithm for score-ordered query evaluation, and show a query throughput increase of almost 70%. This algorithm is the fastest *rank-safe* query evaluation method known to us for *ad hoc* queries.

- **Document-ordered query optimization.** We show how score skipping can give some of the speed advantages of score-ordered indexes to document-ordered indexes. Score skipping improves throughput by over 80% on a set of web queries, with almost a 400% throughput increase for two word queries.

- **TupleFlow.** We present TupleFlow, a system for distributed computation on a grid of machines. TupleFlow extends MapReduce with comparators, compression, graph-based scheduling and arbitrary data flow, allowing for complex distributed task execution, but while still allowing for single-threaded tasks. TupleFlow is particularly suited for index building and text processing.

- **Binning Probabilities.** We show language model binning, which allows language model probabilities to be used in retrieval systems that require positive integers. We show how to solve the problem of smoothing, and how a saturation parameter can improve overall effectiveness.

- **Case study of named page retrieval.** We show how features used in a real web ranking function can be stored as integers to produce an effective and efficient search engine.

[1] http://www.galagosearch.org

1.4 Layout

This dissertation begins with a discussion of background material in Chapter 2. This background material includes information about the basic principles of probabilistic information retrieval, including how to rank documents effectively in response to a query. We discuss methods for evaluating the efficiency and effectiveness of an information retrieval system, and also discuss how traditional inverted indexes are built. Those who are already familiar with information retrieval can safely skip this chapter.

Chapter 3 presents TupleFlow, a framework for distributed computation. TupleFlow is primarily inspired by MapReduce, but combines aspects of many parallel and distributed computing frameworks. We highlight some of the major features of TupleFlow, including job graph scheduling, checkpointing, sorting, hashing, and tuple compression.

We use TupleFlow to build the indexes in Chapter 5, which presents an improvement to the score-sorted indexes presented by Anh and Moffat. Using a slight modification on their impact-ordered indexes, we are able to modify the query evaluation algorithm to achieve a substantial improvement in throughput.

The score-sorted optimization chapter motivates a look at what other kinds of weights, other than document-centric impacts, might be stored in binned indexes. We investigate binned probabilities in Chapter 4, and show how language modeling probabilities can be translated into positive integers. In particular, we show how to deal with the problem of translating term smoothing into the integer domain.

We present document-ordered query optimization in Chapter 6, this time using the binned probabilities from Chapter 4. As in Chapter 5, we use TupleFlow to build these indexes. Our optimization is called *score skipping*, and we show how it improves substantially on Turtle and Flood's max-score algorithm [100].

We combine all of our contributions in Chapter 7, which is an exploration of query processing for named page finding on the Web. Doing named page finding effectively requires a combination of many features, and we use the flexibility of TupleFlow to create these feature-based indexes. We generate probabilities which are binned using our work in Chapter 4, and then stored in both score-sorted and document-sorted indexes, as in Chapters 5 and 6. We evaluate our results based on both efficiency and effectiveness.

There are some important problems in indexing and retrieval that this dissertation does not address experimentally. However, we discuss how our work might be extended or integrated with other work in Chapter 8. In particular, we discuss snippet generation, crawling, query caching and index update.

We conclude in Chapter 9 with a discussion of how all of these contributions fit together.

CHAPTER 2
BACKGROUND

This chapter reviews some of the foundations of information retrieval necessary to understand the rest of the dissertation work. The chapter begins with a discussion of index structures, then moves to basic query evaluation techniques. After that, we discuss ranking functions with a special focus on the language modeling approach. This brings us into a discussion of how document features can be combined in effective query formulations, with a special focus on how this is done in the Indri query language. Finally, we discuss the basics of information retrieval evaluation in both efficiency and effectiveness.

2.1 The inverted index

The standard data structure for searching text document collections is the inverted index [108]. Other data structures, most notably signature files, have been proposed in the past. However, in the last decade, compressed inverted indexes have been shown to be superior to all other indexing methods in terms of efficient retrieval speed [109]. In this section, we describe the basic structure of an inverted index, and basic query processing strategies for evaluating queries using this data structure.

An inverted list, in its simplest form, is a mapping from a single word to a set of documents that contain that word. An inverted index is formed from a set of inverted lists. Usually the inverted index contains one inverted list for each unique word in the collection of documents indexed, although sometimes certain very frequent terms (called *stopwords*) are not indexed.

Consider the following four “documents”:

1. Cats, dogs, dogs.

2. Dogs, cats, sheep.

3. Whales, sheep, goats.

4. Fish, whales, whales.

The inverted index for this document collection looks like this:

cats	dogs	fish	goats	sheep	whales
1	1	4	3	2	3
2	2			3	4

Using these lists, it is simple to find the documents that contain the word ‘dogs’; the inverted list for ‘dogs’ tells us that documents 1 and 2 contain this word. To find the documents that contain both ‘sheep’ and ‘dogs’, we look at the lists for both words and take the intersection, to find that document 2 contains both words.

If we want to find documents about dogs, it seems likely that a document that contains the word ‘dogs’ twice is more likely to have the information we want than a document that contains ‘dogs’ only once; therefore we would prefer the first document over the second one. The simple inverted list structure cannot make this kind of distinction. Therefore, it is very common to include term counts in the lists:

cats	dogs	fish	goats	sheep	whales
(1,1)	(1,2)	(4,1)	(3,1)	(2,1)	(3,1)
(2,1)	(2,1)			(3,1)	(4,2)

In this example, each inverted list entry contains a document number and a count of the times that term appears in the document. We see that ‘dogs’ appears twice in the first document, and ‘whales’ appears twice in the last document.

In a further refinement, we can add position information to the inverted lists:

cats	dogs	fish	goats	sheep	whales
(1,1) : 1	(1,2) : 2, 3	(4,1) : 1	(3,1) : 2	(2,1) : 3	(3,1) : 1
(2,1) : 2	(2,1) : 1			(3,1) : 2	(4,2) : 2, 3

Now, we know that the word 'dogs' appears twice in the first document, and that it occurs at the second and third word positions in that document. This organization allows us to search for phrases in the corpus, such as "dogs cats" (which appears only the second document).

In practice, inverted indexes without word counts are rarely used. With compression, an inverted index with just word counts can be stored in approximately 5% of the total text collection size, while counts and positions can be stored in 20% of the text collection size [105]. This difference in space also translates into a performance gain; indexes without word positions can be built faster and can be queried faster than those without positions. However, recent work shows that word location information is essential for achieving high effectiveness, even in queries that do not explicitly request phrases [68].

Since the inverted index is the core of all modern retrieval systems, efficient construction techniques have been the subject of a significant amount of research. For more information about construction techniques, Witten, Moffat and Bell [105] distill the best of research up until the mid 1990s.

2.2 Ranking documents

Information retrieval systems return a ranked list of documents based on a query. A perfect ranking would rank documents in order of relevance; the most documents would come first, and the irrelevant documents would be last. However, perfect document ranking appears to require detailed models of human cognition that are simply not tractable at this time. Instead, retrieval systems compute the probability that a document is relevant based on features correlated with relevance.

The language modeling approach assumes that queries and documents are generated from some probability distribution of text [81]. Under this assumption, the probability $P(R_Q|D)$ that a document D is relevant to a query Q is rank-equivalent to the probability $P(Q|D)$ that the query Q was generated by the same distribution as the document D. The problem of ranking documents then becomes the problem of estimating $P(Q|D)$.

Under the traditional *bag of words* assumption, we assume that there is no need to model term dependence. Of course, dependence does matter, and this will be revisited later in this thesis. However, for the time being, ignoring dependence gives us a simple approximation for $P(Q|D)$ as a product of probabilities:

$$P(Q|D) = \prod_{w \in Q} P(w|D) \tag{2.1}$$

Here, the term $P(w|Q)$ is estimated for each term w in the query Q. A simple estimate for this quantity is the number of times the word w appears in the document D, divided by its length:

$$P(w|D) = \frac{c(w; D)}{|D|} \tag{2.2}$$

Here, $c(w; D)$ is the count of the number of times w appears in D, and $|D|$ is the length of document D. Intuitively, we can think of this as the probability of choosing w when picking words randomly from D.

This estimate is unappealing for two reasons. First, if a word w appears in the query Q but not in the document D, $P(Q|D) = 0$. According to the model, then, a document that does not contain all of the query terms cannot ever be relevant to the query. Furthermore, a document that contains some of the query terms is not any more likely to be relevant than a document that contains no query terms. This is too drastic an assumption. Second, all words are treated identically—in a query like 'maltese falcon', a document that contains the word 'maltese' once and the word

‘falcon’ twice is ranked the same as a document of similar length that contains ‘falcon’ once and ‘maltese’ twice. It seems like the word ‘maltese’ is more informative in this query, so the latter document should be preferred.

Both of these problems are solved by smoothing. We suppose that the document D is just a sample from some larger text distribution. We don’t know anything about this larger text distribution other than that it is natural language text. Our best sample of natural language text is the entire text of our document collection, giving us a collection probability of:

$$P(w|C) = \frac{c(w;C)}{|C|} \tag{2.3}$$

We can then generate a smoothed estimate for $P(w|D)$ using this additional model of text. The simplest way to combine these models is to use a linear combination of both, with parameter $0 \leq \lambda \leq 1$:

$$P(w|D) = (1-\lambda)\frac{c(w;D)}{|D|} + \lambda\frac{c(w;C)}{|C|} \tag{2.4}$$

Another possibility is to assume that the document model for D might be anything, but that document models are generated by a model of natural language text. Since the model we are using for text is the multinomial, the natural generating distribution is the Dirichlet distribution. If we take the maximum likelihood estimate of $P(w|D)$ given that D is generated by a Dirichlet distribution with a parameter vector set to the collection distribution, we get:

$$P(w|D) = \frac{c(w;D) + \mu c(w;C)/|C|}{|D| + \mu} \tag{2.5}$$

An intuitive way to look at this equation is that D is modeled by the $|D|$ words we see in the document, plus μ additional words drawn at random from the collection.

In practice, these probabilities can be quite small, so we take the log of each term in order to avoid loss of precision:

$$\log P(Q|D) = \sum_{w \in Q} \log P(w|D) \tag{2.6}$$

Other models exist other than language models, but all models in common use today can be decomposed into similar parts. Each document is given a score based on its probability of relevance. The document score is a sum of term scores, one for each term in the query. The term score incorporates the count of the term in the document, some statistics about that term from the collection, and the length of the document.

From the point of view of a retrieval system, the score for a document is a sum of term scores:

$$f_{(Q,C)}(D) = \sum_{w \in D} f_{(Q,C,w)}(c(w;D), |D|) \tag{2.7}$$

As can be seen, the only document-dependent data necessary to compute the score is the count of w in D ($c(w;D)$), and the length of D. The first quantity is readily available in the inverted index described previously. Typically the document lengths can be stored in an array.

In many cases, this equation can be separated into parts dependent on term counts and a term dependent on document length:

$$f_{(Q,C)}(D) = d_{(Q,C)}(|D|) \sum_{w \in D} f_{(Q,C,w)}(c(w;D)) \tag{2.8}$$

It should be noted that so far, we have made the assumption that the position of each term in the document is not important. Later in this dissertation we will consider scoring documents based in part on the distance between query terms in the document.

2.2.1 Ranking with Impacts

It is important to realize that while these expressions are complex, the input data is noisy and imprecise.

For instance, suppose our query is “dog”. Suppose we use a collection that contains the word “dog” exactly twice: once in document A, which is 100 words long, and once in a document B, which is 101 words long. Every popular retrieval model will consider document A more likely to be relevant than document B, since both contain the same number of occurrences of “dog,” but B is longer than A. In fact, Fang et al. [47] consider this property to be an axiom of useful retrieval models.

Just because retrieval models make a distinction between A and B doesn't mean that there is much evidence for this distinction. With just one word difference in document length, just one word as a query, and no information about the remaining contents of A and B, it would be foolish to claim that A is a much better document than B. We do not have enough evidence to make that kind of statement.

A more honest way to represent our knowledge about relevance might be to assign documents a single number representing our belief that “dog” is an important concept in the document. Retrieval models like the RPI model [48] and the Multiple-Bernoulli model [69] have taken this approach. Anh and Moffat use integers, called *impacts*, to represent belief levels [7, 8, 9, 10]. In their work, they find that just eight integer values are necessary to achieve retrieval effectiveness levels that are indistinguishable from high precision floating point probabilities for document ranking.

From a human perspective, perhaps this is not a very surprising result. If a user types just the word “dog” as a query, and we are using just the document length and term frequency in documents in order to rank them, it is unreasonable to think we can do much relevance distinction among the documents. But from a modeling perspective, this raises important issues—can we really deploy systems that rank documents using just eight impact levels? Can that really be enough to give us high

performance when ranking billions of documents? We claim that it isn't enough, but that Anh and Moffat have shown that term counts and document lengths do not contain enough information to rank documents effectively. In the next few pages, we will discuss ways that we can infuse the retrieval process with additional information in order to increase our belief confidence.

Using these integer impact values is important for two other reasons. First, using integer score values leads to a particularly elegant and efficient retrieval system architecture which we will adopt for this dissertation work. Second, since impacts are generated at index time, they can be created from any information we choose. This allows us to infuse extra document information, over and above term counts, into these document impacts in order to improve retrieval effectiveness.

While Anh and Moffat use the term *impact* to describe integer term weights, the term *document-centric impact* refers to a particular kind of weight that they have developed [10]. Because of this potential confusion, we use different terminology to be clear about meaning. We use the word *binning* to descrbe the process of transforming a real-valued weight into an integer value. We also present indexes that are sorted by these binned values, which we call *score-sorted*.

2.3 Search Tasks

In the Introduction, we mentioned that information retrieval has undergone a shift in the needs of its primary users. Two decades ago, search tasks were often highly recall-focused, and this was encouraged by the Text REtrieval Conference (TREC) tasks at the time. The TREC task corresponding to recall-focused retrieval is known as *ad hoc* search. This task involves retrieving the top thousand documents for each query, and evaluating quality using Mean Average Precision (MAP).

More recently, the web has brought about a different kind of search task. Broadly, this is called *navigational search*, although it has a number of subcomponents, like

named page search and *home page* search. Navigational search involves finding a particular document or web page instead of many results on a particular topic. In *named page* search, the focus is a particular page that we assume the user has seen before. In *home page* search, the user is looking for the primary web page for some entity, like a company or a person.

We focus on both *ad hoc* and navigational search in this dissertation, although there are many more kinds of text search, including blog search, expert finding, and question and answer search. While these are not the focus of this dissertation, some of the evidence combination techniques used in our work could be applicable to these other areas.

2.4 Using Additional Features

So far, our discussion of ranking has dealt only with counting word occurrences, which is a simple but effective ranking strategy. However, higher quality rankings are possible by using additional document features in the ranking process.

Many different kinds of features are possible. Some deal with the whole document, like the number of hypertext links that point to the document, or the number of misspelled words in the document. Others deal with where particular words appear, as we might expect that words that appear in a document's title are more representative of the true document content.

While the core of information retrieval research has focused on rather simple feature sets, more complicated formulations have been tried. For years, researchers have tried to use noun phrases or word proximity to improve retrieval effectiveness [4, 23], although it seems that larger collections were necessary to see consistent gains [68]. Weighting different parts of a document differently has also been tried. The advent of both large web collections and new machine learning techniques has added interest in massive feature combinations for ranking [24].

In the next few pages, we will discuss some methods for combining document features. Since we will describe feature combination with the Indri query language, our discussion starts by explaining how the Indri query language works. We then move on to discussion of two specific feature combination methods: one for combining evidence from document fields, and another for using word proximity information. Both of these methods have been shown to be useful for navigational search, which we will consider in Chapter 7.

2.4.1 Indri query language

The Indri query language, while relatively new, is based on over a decade of experience with the successful Inquery retrieval system [30]. The operators were carefully designed to handle the widest possible range of textual features that might be useful in the retrieval process. For those features that cannot be computed within the system, the `#prior` operator allows for external sources of feature information to be used during ranking (e.g. PageRank, inlink count).

Both Indri and Inquery are based on the inference network model, originally developed by Turtle and Croft [99]. This model is based on the intuition that there may be many features of a document that indicate it is relevant—documents about 'dogs' may contain words like 'beagle' or 'puppy,' or phrases like 'basset hound'. A word like 'ringworm' may support the hypothesis that a document is relevant, but is probably not enough to indicate relevance by itself. The inference network model gives a strong theoretical basis for combining these different sources of evidence about relevance.

Indri expands on the operator set of Inquery by including an array of new operations based on document fields. Research has shown that certain sections of documents (such as the title field of a web page) are known to be more important than others in determine the relevance of the document.

Construct	Name	Description
`#odN` $(q_1, ..., q_n)$	Ordered window	A match occurs if the q_i's appear in order with no more than N words between adjacent terms
`#uwN` $(q_1, ..., q_n)$	Unordered window	A match occurs the q_i's appear in any order within a window of N words
`#any` (*field*)	Any	A match occurs if any field called *field* is found
`#syn` $(q_1, ..., q_n)$	Synonym	A match occurs if any of the q_i's appear.
`#wsyn` $(w_1 q_1, ..., w_n q_n)$	Weighted synonym	Similar to `#syn`, but occurrences of certain words can be weighted more highly than others.
term.*field*	Field restriction	A match occurs if *term* is found in a field named *field*
`#combine` $(q_1, ..., q_n)$	Combine	Combines inference from the q_i's; similar to `#and` from [67]
`#weight` $(w_1 q_1, ..., w_n q_n)$	Weight	Combines inference from the q_i's, using the w_i's as weights; similar to `#wand` from [67].
`#greater` (*field* n)	Greater	Evaluates to true if the document contains a field *field* with a numeric value greater than n
`#less` (*field* n)	Less	Evaluates to true if the document contains a field *field* with a numeric value less than n
`#equal` (*field* n)	Equal	Evaluates to true if the document contains a field *field* with a numeric value equal to n
`#prior` (*name*)	Document prior	Assigns a user-specified prior probability of relevance to each document

Figure 2.1. Complex query operators from Indri

2.4.2 Query examples

The following query is a translation of "david fisher lemur" into the Indri query language, using the proximity and field weighting methods we will discuss later:

```
#weight( 0.8 #combine( #wsum( 1.0 david.(inlink)
                              1.0 david.(title)
                              3.0 david.(mainbody)
                              1.0 david.(heading) )
                       #wsum( 1.0 fisher.(inlink)
                              1.0 fisher.(title)
                              3.0 fisher.(mainbody)
                              1.0 fisher.(heading) )
                       #wsum( 1.0 lemur.(inlink)
                              1.0 lemur.(title)
                              3.0 lemur.(mainbody)
                              1.0 lemur.(heading) )
         0.1 #combine( #wsum( 1.0 #1( david fisher ).(inlink)
                              1.0 #1( david fisher ).(title)
                              3.0 #1( david fisher ).(mainbody)
                              1.0 #1( david fisher ).(heading) )
                       #wsum( 1.0 #1( david fisher lemur ).(inlink)
                              1.0 #1( david fisher lemur ).(title)
                              3.0 #1( david fisher lemur ).(mainbody)
                              1.0 #1( david fisher lemur ).(heading) )
                       #wsum( 1.0 #1( fisher lemur ).(inlink)
                              1.0 #1( fisher lemur ).(title)
                              3.0 #1( fisher lemur ).(mainbody)
                              1.0 #1( fisher lemur ).(heading) )
         0.1 #combine( #wsum( 1.0 #uw8( david fisher ).(inlink)
                              1.0 #uw8( david fisher ).(title)
                              3.0 #uw8( david fisher ).(mainbody)
                              1.0 #uw8( david fisher ).(heading) )
                       #wsum( 1.0 #uw12( david fisher lemur ).(inlink)
                              1.0 #uw12( david fisher lemur ).(title)
                              3.0 #uw12( david fisher lemur ).(mainbody)
                              1.0 #uw12( david fisher lemur ).(heading) )
                       #wsum( 1.0 #uw8( fisher lemur ).(inlink)
                              1.0 #uw8( fisher lemur ).(title)
                              3.0 #uw8( fisher lemur ).(mainbody)
                              1.0 #uw8( fisher lemur ).(heading) )
```

The query terms appear by themselves at the top of the query, but occurrences of each term are weighted differently depending on where the term occurs in the document. Notice that the inlink text (that is, text used to describe the document in

links to this page) is treated separately, as well as the title of the document, heading text (from tags like `h1` or `h4`) and the main body of the document. Proximity expressions occur later in the query, first in exact phrases, such as `#1( david fisher )`, which matches only the words David and Fisher occurring in order. This restriction is relaxed at the bottom of the query with expressions like `#uw8( david fisher )`, which matches the words David and Fisher occurring within 8 words of each other, in any order.

In a different example, suppose that an extraction system is trying to verify that George W. Bush is the 43rd President of the United States. This Indri query searches for sentences that contain a number, the phrase "george w bush", and the word "president" within 10 words of each other:

```
#combine[sentence]( #uw10( #3( george w bush )
                           #any:number
                           president ) )
```

Finally, suppose that the user is looking for documents in a desktop search application. The user has issued the query "income taxes". Since many documents might match this query, the desktop search implicitly prefers documents that were modified recently, or that have been accessed frequently:

```
#combine( income taxes
          #1(income taxes)
          #prior(modificationtime)
          #prior(accessfrequency) )
```

2.4.3 Probabilistic Interpretation

In Indri, each query operator has a probabilistic interpretation, and corresponds directly to a mathematical function. Therefore, an Indri query represents not the

user's information need, but instead defines a function which acts on documents and produces document scores.

A single word, appearing by itself, is scored using the Dirichlet smoothed language model described earlier:

$$\log P(w|D) = \log \frac{c(w;D) + \mu c(w;C)/|C|}{|D| + \mu} \tag{2.9}$$

Phrase occurrences or field occurrences use this same formula. Just as with a single term, we can calculate $c(p;D)$ and $c(p;C)$ for a phrase p.

Probabilities can be combined together using a combination operator. The simplest combination operator is called `#combine`, which has this interpretation:

$$\#\texttt{combine(Q)} = \frac{1}{|Q|} \sum_{w \in Q} \log P(w|D) \tag{2.10}$$

More complicated combinations are possible. The `#weight` operator extends the `#combine` operator by allowing some operands to be weighted more than others:

$$\#\texttt{weight}(\lambda_1 w_1 ... \lambda_n w_n) = \frac{1}{\sum_1^n \lambda_i} \sum_1^n \lambda_i \log P(w_i|D) \tag{2.11}$$

The weighted sum operator (`#wsum`) allows for a linear probability mixture instead:

$$\#\texttt{wsum}(\lambda_1 w_1 ... \lambda_n w_n) = \log \sum_1^n \frac{\lambda_i}{\sum_1^n \lambda_i} \log P(w_i|D) \tag{2.12}$$

2.4.4 Bag of words

Even though plenty of research has shown that additional document features are necessary for effective retrieval, it is still popular to ignore all information except single term occurrences. As we have discussed, this is called the *bag of words* approach, and corresponds to the unigram language model.

In Indri, this approach can be achieved by simply typing the query terms, like this: `white house`.

2.4.5 Adding Term Proximity

To illustrate term proximity features, we consider Metzler and Croft's Markov Random Field experiments. Metzler and Croft [68] achieved significant effectiveness gains by adding term proximity features to their query formulation. They noticed that queries were often composed of multiword phrases, or words that would need to appear close together in order to indicate relevance. In our example query, `white house`, it is clear that finding the exact phrase "white house" is a much stronger indicator of relevance than just finding "white" and "house" scattered within a document.

While the Markov Random Field model that the authors propose is capable of integrating any arbitrary feature set, the authors consider three types of features:

- Single term features – these are the standard unigram language model features we have used so far
- Exact phrase features – these features involve a set of words appearing in sequence in candidate documents
- Unordered window features – these features require words to be close together, but not necessarily in an exact sequence order.

For the query `white house`, the authors recommend the following Indri query:

```
#weight(0.8 #combine(white house)
        0.1 #1(white house)
        0.1 #uw8(white house))
```

This query not only includes the single words as indicators of relevance (in the first `#combine`), but also the exact phrase (in the `#1` operator), and occurrences of both words within 8 words of each other (`#uw8`).

2.4.6 Document Mixture Models

Traditional retrieval models consider each word in a document as an equal, independent source of topical evidence. However, we know from experience that certain sections of a document are better summaries of a document's subject matter than others. For instance, the title of a document can be seen as a very short summary of the document's contents.

Suppose we have two possible results for our query "white house." Based on all of the models we have seen so far, document A and document B seem equally likely to be relevant. However, suppose that "white house" appears in the title of document A, but just in the body text of document B. Since "white house" appears in the title of document A, we may assume that the content of document A has something to do with the White House. However, document B may merely mention "white house" in passing, so it may not actually be central to the topic of the document. This evidence would lead us to prefer document A over document B.

While not all documents have explicit titles, web pages almost always do. Web pages also have other important tags that point us to important document information. These include heading tags like `h1` or `h2`, alternate font style tags like `strong` or `em`, image alternate text, incoming link text, and `META` keywords. All of these tags may indicate topical content that is less likely to contain words that are used in passing. A highly effective model should take into account these additional sources of evidence.

Many approaches have been published on this problem. Two popular works on this subject include Robertson's BM25F [84] and Ogilvie's work on mixing document representations [77]. We focus on the Ogilvie approach since it incorporates the same language modeling probabilities used in the rest of this work.

Ogilvie et al. consider a HTML document to be a set of different representations of the same document. The title is one representation of the document, while the inlink

text is another, and the body text is another. To create a single representation of the document, sub-document models are constructed for each of these representations, then the representations are mixed into a single larger document model. Documents are then ranked by using this mixed document model, as shown below:

$$\begin{aligned} P(w|D) \;=\; & \lambda_{\text{title}} P(w|D_{\text{title}}) + \\ & \lambda_{\text{body}} P(w|D_{\text{body}}) + \\ & \lambda_{\text{inlink}} P(w|D_{\text{inlink}}) + \\ & \lambda_{\text{meta}} P(w|D_{\text{meta}}) + \\ & \lambda_{\text{heading}} P(w|D_{\text{heading}}) \end{aligned}$$

The mixing parameters in this model (the λ terms) allow the more effective document representations to be emphasized during ranking. These parameters, also called weights, can be trained using existing queries, documents, and relevance judgments.

This mixing approach has been shown to outperform simple, single document representations. Chapter 7 uses this model, but without `META` text.

2.4.7 Additional Document Features

The features we have discussed so far are all query-dependent in some way, in that at least one of the query terms is a part of computing the feature. While these query-dependent features are important, many structured document retrieval systems can be improved by the use of query-independent evidence. This is especially true in web search.

The most studied query-independent ranking evidence comes from link structure. The basic assumption behind these methods is that a hypertext link represents an implicit vote for the relevance of the page that is linked to. Links are generally added to pages by people, and presumably people would not waste time linking to

non-informative pages. Therefore, a simple way to incorporate this information is to prefer pages with more incoming links. Many, many more complicated formulations exist. The most popular two are Page et al.'s PageRank, and Kleinberg's HITS algorithm [78, 54].

Another important feature is URL depth. Each web page has a unique address, called a uniform resource locator, or URL. Web sites are generally structured so that the main entry point has a very short URL, like `http://www.umass.edu`. From that page, more specific information can be found, like `http://www.cs.umass.edu`. In general, shorter URLs point to general pages, while longer URLs point to specific pages. Therefore, for all pages that are a good match for a query, if we prefer shorter URLs we will find more general pages. For instance, a search for "ibm" finds 719 million matches, with over 7 million of those from the `ibm.com` domain. Using URL depth as a feature helps us prefer the main `http://www.ibm.com` page as the best answer over the millions of other possible matches.

More recent work from Joachims [53] and Agichtein [3] focus on implicit user-supplied evidence. The simplest of these forms of evidence comes from user clicks—presumably if a user clicks on a query search result, the user thinks the result may be relevant. As Joachims found, this evidence source is quite noisy, for many reasons—first, the user is preconditioned to believe the search engine, and is therefore likely to click on the first result whether it looks relevant or not. Second, the user may click on a result that looks relevant based on its title or summary, but later find that the result is not relevant at all. Still, Agichtein finds that proper use of this data can improve query results.

In addition, Agichtein [3] studies more subtle user features, such as dwell time (the amount of time the user spends looking at a document), or domain deviation (the deviation from the mean amount of time spent looking at documents from any

particular domain). While some of Agichtein's features are query-dependent, many are query independent.

Finally, authors have developed document quality models, such as the one developed by Zhou et al. [107]. The authors note that since natural language follows certain grammatical rules and structures, a document with an unusual term distribution is likely ungrammatical, and therefore a poor retrieval candidate. By using this feature along with another, simpler feature (information-to-noise ratio), they find it possible to achieve small improvements in query effectiveness.

In combination, these query-independent features can be seen as indicators of the quality or usefulness of a particular document. These features can be used in combination with query-dependent features for high-quality ranking.

2.4.8 The UMass 2005 Navigational Formulation

We have already discussed Metzler's Markov Random Field term dependence model and Ogilvie's method of combining document representations. These two models can be combined with the query-independent features from the previous section into one web query model which is highly effective for navigational web search. We call this the UMass 2005 formulation, since it was used by Metzler, Strohman and Croft for their UMass TREC 2005 submission [72].

We repeat a query example used earlier, with slight alterations, to show the UMass 2005 formulation to the query "david fisher lemur":

```
#combine(
0.1 #weight( 0.4 #prior(pagerank) 0.6 #prior(inlinks) )
1.0 #weight( 0.8 #combine( #wsum( 1.0 david.(inlink)
                                  1.0 david.(title)
                                  3.0 david.(mainbody)
                                  1.0 david.(heading) )
                           #wsum( 1.0 fisher.(inlink)
                                  1.0 fisher.(title)
                                  3.0 fisher.(mainbody)
                                  1.0 fisher.(heading) )
                           #wsum( 1.0 lemur.(inlink)
```

```
                                1.0 lemur.(title)
                                3.0 lemur.(mainbody)
                                1.0 lemur.(heading) )
            0.1 #combine( #wsum( 1.0 #1( david fisher ).(inlink)
                                1.0 #1( david fisher ).(title)
                                3.0 #1( david fisher ).(mainbody)
                                1.0 #1( david fisher ).(heading) )
                          #wsum( 1.0 #1( david fisher lemur ).(inlink)
                                1.0 #1( david fisher lemur ).(title)
                                3.0 #1( david fisher lemur ).(mainbody)
                                1.0 #1( david fisher lemur ).(heading) )
                          #wsum( 1.0 #1( fisher lemur ).(inlink)
                                1.0 #1( fisher lemur ).(title)
                                3.0 #1( fisher lemur ).(mainbody)
                                1.0 #1( fisher lemur ).(heading) )
) )
```

This query includes the Ogilvie et al. approach of mixing language models (title, mainbody, heading and inlink) as well as Metzler's use of phrase and proximity operators from the Markov Random Field dependence model. The `#prior` operators incorporate some of the query-independent features discussed in the previous section.

The UMass 2005 Navigational Formulation is highly effective, and produced the best navigational results at TREC in 2006.

2.4.9 Query Expansion

So far, we have restricted our attention to either query-independent features, or occurrences of query terms in documents. By contrast, *query expansion* techniques attempt to improve the query processing results by using words related to those in the query. Documents that contain these related terms should be considered more likely to be relevant than those that do not.

2.4.9.1 Stemming

The simplest form of query expansion is called *stemming*. This method comes from the observation that many related words have the same prefix stem; for example, documents about running may contain words like 'run', 'ran', 'running', or 'runs'.

These words are all related, and all describe the action of running, but are used in different contexts. If a query contains the word 'run', it makes sense to also look for documents that contain 'ran', 'running' and 'runs.' Stemming is performed automatically using an algorithm called a *stemmer*. Such an algorithm assigns natural language words to *stem classes*, which are groups of words that are presumed to represent the same concept. A stemmer can make two kinds of errors—it can fail to group two related words, or it can group two words together that should not have been grouped. An aggressive stemmer makes a small number of stem classes, and therefore risks making the latter type of error. A conservative stemmer (or no stemmer at all) makes a large number of stem classes, and risks making the former type of error.

The simplest kind of English stemming is provided by a suffix-s stemmer. This stemmer only conflates plural words with their singular versions—for instance, 'plates' and 'plate' would be considered the same word. This type of stemming is very conservative, and therefore works well in practice. The Krovetz and Porter stemmers are less conservative [82, 56]. These stemmers also consider more complicated word endings, like -ing or -ed. The Porter stemmer is an algorithmic stemmer, so it determines classes of similar words just on the basis of suffix patterns. This leads to a very simple algorithm, but one that can be fooled by special cases; for instance, the Porter stemmer conflates 'marine' and 'marinate,' although these words have very different meanings. The Krovetz attempts to solve this problem by using a supplemental dictionary that can catch these special cases. This makes the Krovetz stemmer more conservative and less likely to make errors.

The stemmers shown so far must be hand-crafted by an expert with extensive linguistic knowledge, and this process can be time consuming. Another option is statistical stemming, which uses word co-occurrence data to determine if two terms are related. For instance, even though 'marine' and 'marinate' have common prefixes, they are unlikely to appear in the same document. However, 'marinate' and 'mari-

nated' are likely to occur in the same document. A statistical stemmer can use this kind of data to determine stem classes without expert intervention [5].

In our experiments, we use the Porter2 stemmer, which is based on the original Porter stemmer [82]. It is an algorithmic stemmer without an additional dictionary.

2.4.9.2 Synonyms

Even the most aggressive stemmers only conflate words with similar prefixes. However, many similar words have no obvious letter pattern in common. A simple example is the relation between words in different languages—'white' in English expresses the same concept as 'blanco' in Spanish. Other pairs are open to interpretation, or may be query independent. For instance, "president" is the same concept as "commander-in-chief" with reference to the President of the United States, but not when considering the president of a company. Also, "president" and "prime minister" express the same "head of state" concept, but are certainly not interchangeable when talking about a particular person.

This makes expansion by synonyms a very difficult process. A very aggressive system that uses a long synonym list is likely to improve some queries impressively, but may hurt the performance of other queries. A system that uses a small list will rarely hurt query performance, but will rarely help either.

There are two possibilities that can help here. One is to include synonyms in a query, but to prefer matches from the actual user query term. The Indri `#wsyn` operator has this property. By taking this approach, an excellent match to the original query will almost certainly be ranked first, but if no good match exists, synonym matches can appear later. A second option is to use the context provided by the whole query to determine which words to use. For instance, we know that expanding with "prime minister" is a good idea for the query "president of canada", but not a good idea for "prime minister tony blair". Doing this perfectly requires a system that

has excellent conceptual language understanding, but in the next section, we consider an approximation that works well in practice.

2.4.9.3 Pseudo-relevance feedback

Instead of trying to make an information retrieval system that can think like a human, perhaps a better idea would be to encourage users to interact with the system to improve query results. Interactive retrieval systems use user *feedback* to find high quality documents. Generally, the user chooses a few documents from the collection that seem relevant to the query, and the system attempts to find documents that are both similar to the query and the documents that were marked relevant. Typically, words are extracted from the known relevant documents and added to the query in order to bias the query results to look like the known relevant documents. This process is called *relevance feedback*.

A simple extension of this process is to remove the user entirely. We can assume that the results of the original query are fairly good, and so the top n documents are likely to be relevant. We can then use these documents as a source of excellent words to feed into the query.

As an example, suppose the user types the query "band-aid." We assume that the user is actually interested in adhesive bandages in general, not just those made by Johnson and Johnson under the trade name Band-Aid. If we run the query "band-aid" with a good information retrieval system, we will find the Band-Aid web page, which contains the text "adhesive bandages." If we run the query again, this time using the words "adhesive bandages" too, we may find products by other manufacturers too.

Pseudo-relevance feedback is generally not be as precise as synonym expansion. Pseudo-relevance feedback may add popular phrases like "privacy policy" or "contact us" to the query, which aren't likely to help query effectiveness, while synonym methods aren't likely to make this kind of error. However, pseudo-relevance feedback

has the major advantage that it is context-sensitive. The added words come from actual documents that match the original query well. This additional context helps pseudo-relevance feedback systems avoid the problems we saw with synonyms and lack of context.

In this thesis, we will focus on query expansion techniques that do not involve pseudo-relevance feedback. Pseudo-relevance feedback is a powerful technique, but one that is fairly expensive. Because the added words depend on the whole query, feedback terms are difficult to cache well. Additionally, traditional pseudo-relevance feedback techniques require two retrievals, which can more than double query evaluation time. This is changing, since efficient pseudo-relevance feedback is the subject of some recent research [57, 97].

2.5 Evaluating Effectiveness and Efficiency

The information retrieval community has developed a set of standard means for evaluating the effectiveness and efficiency of information retrieval systems. We outline here some of the effectiveness measures and efficiency tests we will use, including descriptions of the test collections we will use for evaluation. More information on evaluation procedures can be found in an information retrieval textbook, such as [64].

2.5.1 Test Collections

Our primary test collection for efficiency and effectiveness will be the TREC GOV2 collection. The GOV2 collection is a snapshot of the .gov Internet domain crawled in early 2004. There are 25 million pages in this collection, and its uncompressed size is 426GB of text. This collection is four times larger than any other public test collection, which makes it appropriate for us to use in testing system scalability.

This collection of documents has been used for three years in TREC, and in that time a large query set has been developed for evaluating effectiveness and efficiency

with this collection. For efficiency testing, approximately 100,000 government-related queries are available. There are also 453 judged navigational topics and 150 judged ad hoc topics. This collection is used in Chapters 5, 6 and 7.

For additional tests, we make use of the TREC GOV collection. The GOV collection contains 1.2 million pages from the .gov domain, and consists of about 20GB of text. This collection has 450 judged navigational topics for additional testing, and its size is convenient for smaller experiments. This collection is used in Chapters 3 and 7.

In our effectiveness experiments in Chapter 4, we use the TREC12, TREC45 and WT10G collections. The TREC12 and TREC45 collections are small 2GB collections of newswire text, while WT10G is 10GB of web text, not restricted to any particular domain. The newswire collections in particular have extensive *ad hoc* relevance judgments and are the standard for *ad hoc* effectiveness testing.

2.5.2 Effectiveness

We will consider five search effectiveness measures in our evaluation. We briefly describe them here, although more detail on the first four can be found in an information retrieval textbook.

Our focus in evaluation will be on the top of the ranked list, which is best measured by the precision, success and NDCG measures. However, we will also measure mean average precision because of its ubiquity in the information retrieval literature.

2.5.2.1 Precision

The precision metric measures the amount of relevant results found in a ranked list, as a percentage of the total number of results found. For instance, in a list of ten results that contains six relevant documents, the precision is 0.6.

The length of the ranked list is an important factor in precision calculations, since precision values after ten results are almost always higher than at twenty results.

This is a natural consequence of the way information retrieval systems work: good systems rank relevant documents at the top of the ranked list. Therefore, precision is always reported along with the length of the list considered. In common language, our example result is a "precision at 10" value, which is abbreviated P@10.

Note that if a ranked list of ten results contains a single relevant document, its P@10 is 0.1. This is true whether the relevant document is at the first rank or the tenth rank, even though humans prefer that the relevant documents appear at the very top of the list. Therefore, while precision is a simple measure, other measures may be more useful.

2.5.2.2 Success

Success is a synonym for P@1: it is equal to 1 if the first document returned is relevant, or zero otherwise. The measure is not very interesting for a single query, but is an intuitive measure of performance when averaged over a large set of queries. It is primarily used when there is a single relevant document. For instance, the query "google" is one for which there is one obvious correct answer (the Google homepage[1]). The TREC conference also defines a Success@k measure, which is 1 if a relevant document has been found by rank k and 0 otherwise.

2.5.2.3 Recall

Topical queries may have many important correct answers. For example, a paralegal may need to find all federal legal decisions that dealt with antitrust law. In this query, precision is important, but it is more important to measure how well the system performed at finding all correct answers. Recall measures this property—the recall is the number of relevant documents found out of the total number of relevant documents that exist.

[1] http://www.google.com

Like precision, we always state the number of documents retrieved, such as "recall at 10." Generally in recall experiments, we consider ranked lists that are much longer than precision experiments, so "recall at 100" or "recall at 1000" is not uncommon.

While recall is a useful measure for research tasks, it also measures a system's document finding capability. For instance, suppose that a system has found 60% of the relevant documents for a query by rank 1000. This may indicate that the other 40% of documents simply cannot be found by the current system, since simple tweaks to the ranking algorithm are not likely to find these other documents.

2.5.2.4 Average Precision

Average precision is an attempt to combine recall and precision into one measure. Unlike recall or precision, the order of results in the list does matter. It is typically measured at 1000 documents, and this depth is assumed unless reported otherwise. The measure is abbreviated as MAP (mean average precision, an average of average precision values from many different queries).

To compute average precision, we take the average of the precision value at the point each relevant document was retrieved. For example, suppose we have the following ranked list:

Here, a shaded box represents a relevant document, and a white box represents a non-relevant document. The list's highest rank is on the left, and its lowest rank is on the right.

We consider the precision at each shaded box:

Rank	Precision
1	1 / 1 = 1.00
5	2 / 5 = 0.20
7	3 / 7 = 0.43
8	4 / 8 = 0.5
10	5 / 10 = 0.5

The average precision is the arithmetic mean of these precision values:

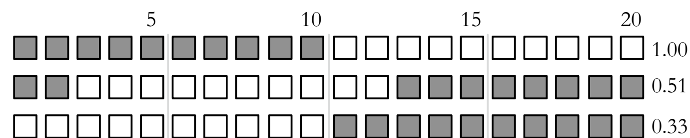
$$(1.0 + 0.2 + 0.43 + 0.5 + 0.5)/5 = 0.53$$

Note that this assumes that all relevant documents were retrieved. If there were other relevant documents that were not retrieved, we assume that they were retrieved at an infinite rank where precision is equal to zero. So, if there were 6 relevant documents in total, we have:

$$(1.0 + 0.2 + 0.43 + 0.5 + 0.5 + 0.0)/6 = 0.44$$

This measure is convenient for many reasons. First, it combines recall and precision into a single measure. Second, the order of the results matters—note that if all the relevant documents in the example had occurred at the end of the list, the P@10 value still would have been 0.5, but the average precision would drop to 0.354.

However, mean average precision is not ideal for measuring web retrieval. Consider the following three rankings of web documents, and their corresponding average precision values:



Power	Rank	Discount $(1/\log(1+i))$
e^0	1	1
e^1	2.7	0.5
e^2	7.4	0.25
e^3	20.1	0.125

Figure 2.2. Discounting factors used in NDCG. Note that the influence of a document at rank 20 is 1/8th of that of the first document.

The first result is excellent, since it retrieves all the relevant results first. The second result is not great, but it does manage to put two good results at the top of the list. The third result finds all the relevant documents last.

Notice that in a traditional web search engine, only ten results display on the first page, and often only seven of those will show in a user's browser without scrolling. To the user, the first result is excellent and the second result is still good, since both have good results right at the top of the list. However, the third result is awful. Even though there are excellent results at ranks 11 to 20, the user will never see them, since they are on the second page. After seeing a full first page of horrible results, the user is not likely to check the next page to see if the ranking improves. It is more likely that the user will modify her query, or perhaps give up.

Even though the first result is great, the second result is pretty good, and the third result is terrible, we find that average precision rates the second result closer to the third result than the first result. This does not match user behavior in web search. To more closely match this behavior, we consider a final metric.

2.5.2.5 Normalized Discounted Cumulative Gain

The Normalized Discounted Cumulative Gain (NDCG) measure has two different definitions. The original measure, defined in Järvelin and Kekäläinen [52], is:

$$N \sum_i r(i) / \log(i)$$

An alternate version, used in this thesis and in many recent papers (like [101]) is:

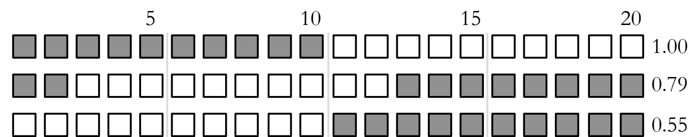
$$N \sum_i (2^{r(i)} - 1) / \log(1 + i)$$

This is a sum over result ranks, where i represents the rank, and $r(i)$ represents the relevance value at that rank. For the metrics we have considered so far, we have only considered two classes of relevance: relevant and non-relevant. However, NDCG can use more than two levels of relevance, with four levels of relevance being common. In the two-class case, non-relevant documents have a relevance value of 0, while relevant documents have a relevance value of 1. In the multiclass case, relevance values start at 0 for non-relevant documents, but continue up through various different shades of relevance.

The value of N is chosen for each query such that the highest possible value of NDCG is 1. The lowest possible value is always 0.

Notice that the influence of documents that are low in the ranked list is curtailed by the log factor in the denominator. See Table 2.2 for a sampling of the discount factors at different ranks of the ranked list.

Returning to our previous example, we compute the NDCG value of three different ranked lists:



The first ranking is still perfect. The second ranking is recognized for having two relevant documents at the top of the ranking, and is now closer to the perfect ranking (0.21 difference) than the poor third ranking (0.24 difference).

Metric name	Description
Elapsed indexing time	Measures the amount of time necessary to build a document index on a particular system.
Indexing processor time	Measures the CPU seconds used in building a document index. This is similar to elapsed time, but does not count time waiting for I/O or speed gains from parallelism.
Query throughput	Number of queries processed per second.
Query latency	The amount of time a user must wait after issuing a query before receiving a response, measured in milliseconds. This can be measured using the mean, but is often more instructive when used with the median or a percentile bound.
Indexing temporary space	Amount of temporary disk space used while creating an index.
Index size	Amount of storage necessary to store the index files.

Table 2.1. Definitions of some important efficiency metrics

2.5.3 Efficiency

Compared to effectiveness, the efficiency of a search system seems like it should be easier to quantify. Most of what we care about can be measured automatically with a timer instead of with costly relevance judgments. However, like effectiveness, it is important to determine exactly what aspects of efficiency we want to measure.

The most commonly quoted efficiency metric is query throughput, measured in queries processed per second. Throughput numbers are only comparable for the same collection and queries processed on the same hardware, although rough comparisons can be made between runs on similar hardware. As a single-number metric of efficiency, throughput is good because it is intuitive, and mirrors the common problems we want to solve with efficiency numbers. A real system user will want to use throughput numbers for capacity planning, to help determine if more hardware is necessary to handle a particular query load. Since it is simple to measure the number of queries

per second currently being issued to a service, it is easy to determine if a system's query throughput is adequate to handle the needs of an existing service.

The trouble with using throughput alone is that it does not capture latency. Latency measures the elapsed time the system takes between when the user issues a query and when the system delivers its response. Previous work suggests that users consider any operation that takes less than about 150 milliseconds to be instantaneous [40]. Above that level, users react very negatively to the delay they perceive. Mayer [65] reports that Google experimented with as many as 30 results on the first page of results instead of the standard 10, but found that users hated waiting for the extra results. Mayer remarks that in terms of user satisfaction, this is "actually the worst experiment we've done to date." The lesson here is that latency is a very important predictor of user satisfaction.

This brings us back to throughput, because latency and throughput are not orthogonal: generally we can improve throughput by increasing latency, and reducing latency leads to poorer throughput. To see why this is so, think of the difference between having a personal chef and ordering food at a restaurant. The personal chef prepares your food with the lowest possible latency, since she has no other demands on her time and focuses completely on preparing your food. Unfortunately, the personal chef is low throughput, since her focus only on you leads to idle time when she is not completely occupied. The restaurant is a high throughput operation with lots of chefs working on many different orders simultaneously. Having many orders and many chefs leads to certain economies of scale, for instance when a single chef prepares many identical orders at the same time. Note that the chef is able to process these orders simultaneously precisely because some latency has been added to some orders: instead of starting to cook immediately upon receiving an order, the chef may decide to wait a few minutes to see if anyone else orders the same thing. The result is that the chefs are able to cook food with high throughput but at some cost in latency.

Query processing works the same way. It is possible to build a system that handles just one query at a time, devoting all resources to the current query, just like the personal chef devotes all her time to a single customer. This kind of system is low throughput, because only one query is processed at a time, which leads to idle resources. The radical opposite approach is to process queries in large batches. The system can then reorder the incoming queries so that queries that use common subexpressions are evaluated at the same time, saving valuable execution time. However, interactive users will hate waiting for their query batch to complete.

Like recall and precision in effectiveness, low latency and high throughput are both desirable properties of a retrieval system, but they are in conflict with each other, since the highest throughput levels come at the expense of high latency. One approach is to hold one metric fixed while optimizing the other. For instance, DeCandia et al. [43] note that Amazon has a strong focus on latency. Their goal is to achieve maximum throughput while bounding on maximum latency (or, more accurately, the latency required by all but the worst 0.1% of operations), as they find this correlates better with user experience than a bound on median latency.

Query throughput and latency are the most visible system efficiency metrics, but we should also consider the costs of indexing. For instance, given enough time and space, it is possible to cache every possible query of a particular length. A system that did this would have excellent query throughput and query latency, but with enormous storage and indexing costs. Therefore, we also need to measure the size of the index structures and the time necessary to create them. Because indexing is often a distributed process, we need to know both the total amount of processor time used during indexing and the elapsed time. Since the process of inversion often requires temporary storage, it is interesting to measure the amount of temporary storage used.

2.5.3.1 Experiments

There are two classes of experiments for measuring efficiency. We will call one class *direct* experimentation, which means that the experimenter attempts to create a real-world system, then directly measures its attributes using a clock or filesystem statistics. The other alternative is *simulation*, where the operation of the system is simulated in software.

The primary advantage of simulation is repeatability. It is unlikely that two experimenters will have access to the same hardware, but it is likely that they can use the same simulation tools. If all experimentation happens in the same simulation framework, results are directly comparable across experiments. In addition, simulation can model how a system would perform on very expensive hardware, even if the experimenter does not have much money.

There are also problems with simulations. First, a simulation experiment runs on a model of reality, and as such is only as good as its model, and modern computer systems are notoriously difficult to model. Considering processors alone, a faithful model needs to consider all the ways that execution can stall, from pipeline conflicts to cache misses (at two or three levels, while also considering associativity). Accurate models are difficult to create, and extreme accuracy comes with an execution speed penalty. Therefore, experimenters are forced to consider tradeoffs between accuracy and the cost of building and running simulators.

By contrast, direct experimentation is straightforward. The experimenter can run software directly on a particular kind of hardware and simply time the results. Unlike simulation results, we can be assured that the timings generated by the experiment are realistic for at least one hardware configuration and environment. Unfortunately, it is not necessarily easy to describe the environment that was used. Webber and Moffat [102] detail many ways that repeatability can be elusive in direct experiments,

including mysteriously slow disks and performance problems that come from unpredictable disk block allocation.

One way to increase the generality of both simulation and direct experiments is to publish operation counts (like the total number of disk accesses or CPU additions) in addition to actual runtime. Seltzer et al. [88] use this kind of benchmarking for hardware, and show how it allows for easier inference about a new software package might perform on a piece of tested hardware. This technique can be just as reliable in making inferences about how an algorithm might perform on a new hardware configuration.

Disks tend to be the worst offenders when it comes to repeatability. Since disk reads are transparently cached by the operating system, a disk read can be slow one minute and fast the next. File fragmentation, general block placement, and bad block reallocation make it difficult to make predictions about how file accesses will behave. When running experiments on disk-based algorithms, it is best to use a long test, to reduce the noise from disk caching and placement effects. The standard TPC database benchmark [83] includes an unmeasured warmup period to allow the system to set up caches and internal structures appropriately before steady-state throughput is measured. In search engine evaluation, warmup periods are not typically mentioned, but long query logs are used to reduce the effect of the experiment start state on execution time. In this dissertation, we test with long query logs for this purpose, but we also focus on memory-based execution which tends to have more repeatable performance.

The goal of our efficiency tests is to establish the speed of the system under a difficult sustained workload. To do this, we turn to the GOV2 dataset, including the 150,000 queries in the TREC 2005 and TREC 2006 efficiency query logs.

The TPC benchmarks, while designed for different application areas (databases and web services), still provide a useful outline for performing throughput evalua-

```
procedure DocumentAtATimeRetrieval(Q)
    L ← Array()
    R ← PriorityQueue()
    for all terms w_i in Q do
        l_i ← InvertedList(w_i)
        L.add( l_i )
    end for
    for all documents D in the collection do
        for all inverted lists l_i in L do
            s_D ← s_D + f_(Q,C,w_i)(c(w_i; D))          ▷ Update the document score
        end for
        s_D ← s_D · d_(Q,C)(|D|)          ▷ Multiply by a document-dependent factor
        R.add( s_D, D )
    end for
    return the top n results from R
end procedure
```

Figure 2.3. A simple document-at-a-time retrieval algorithm.

tion [83]. The traditional evaluation strategy is to measure system throughput over a one hour period under a specified workload. Measuring an hour of activity should ensure that outliers are not found. However, in order to represent the typical performance of systems, a warm-up period is allowed before measurement starts. Systems typically get faster after the initial seconds of operation, as the operating system caches appropriate files, code is loaded into memory, and the branch predictor and caches of the processor adjust to the typical code execution paths of the system. While measuring start-up performance may be important for certain end-user systems, we focus on the scenario where the system is presumed to run as a server process for at least hours at a time.

For repeatability, it is customary in the information retrieval literature to report average throughput over a predefined set of queries instead of over an interval of time. In this dissertation we will follow this convention.

```
procedure TermAtATimeRetrieval(Q)
    A ← HashTable()
    for all terms w_i in Q do
        l_i ← InvertedList(w_i)
        for all documents D in l_i do
            s_{w_i,D} ← A[D] + f_{(Q,C,w_i)}(c(w_i; D))
        end for
    end for
    R ← PriorityQueue()
    for all accumulators A[D] in A do
        s_D ← A[D] · d_{(Q,C)}(|D|)            ▷ Normalize the accumulator value
        R.add( s_D, D )
    end for
    return the top n results from R
end procedure
```

Figure 2.4. A simple term-at-a-time retrieval algorithm.

2.6 Basic query evaluation

There are two basic ways to evaluate a query given an inverted index: document-at-a-time and term-at-a-time (see pseudocode implementations of each in Figures 2.3 and 2.4). Clearly to evaluate the retrieval function $f_{(Q,C)}$ over all documents, the retrieval system must loop over a set of documents, and loop over the terms of the query. In document-at-a-time evaluation, the outer loop iterates through documents, while the inner loop iterates over terms. In term-at-a-time evaluation, these loops are reversed—the inner loop iterates over documents, while the outer loop iterates over terms.

The document-at-a-time has a few advantages over the term-at-a-time method. First, the document-at-a-time method produces complete document scores early in its execution. This allows the algorithm to quickly display partial results, if desired. As we will see later, these complete document scores can be important tools in improving query efficiency. Second, although the basic algorithm fetches all inverted lists first, the document-at-a-time method can incrementally fetch inverted list data from disk.

When this is done, the document-at-a-time has a memory advantage, since it does not have to maintain an accumulator table.

However, the term-at-a-time algorithm is often preferred for efficient systems implementation. It does not need to jump between inverted lists during evaluation—this may save branch prediction misses as well as expensive disk seeks. The inner loop iterates over documents, which means this tight inner loop is executed for a long time. This makes this inner loop code simpler and easier to optimize, both for the programmer and the compiler/processor. Finally, some very efficient query processing strategies have been developed for term-at-a-time evaluation that allow some inverted list information to be skipped.

2.6.1 Score-ordered evaluation

The term-at-a-time and document-at-a-time evaluation both assume a document-ordered index. However, Anh and Moffat claim that an score-sorted index lends itself to an even faster evaluation strategy [7].

Suppose we have an index that contains impacts instead of term counts. In a document-ordered index, we might have an inverted list that looks like this:

(1,1) (2,3) (3,2) (7,1) (8,1) (10,4)

In these pairs, the document is the first element of the pair and the impact level is the second element. Notice that the pairs are ordered by document number. Another possible ordering would be:

(10,4) (2,3), (3,2) (1,1) (7,1) (8,1)

This list contains the same pairs as the last list, but they are now ordered by impact. Since many documents may share the same impact level, we order by document number within each impact level. We can make this index smaller in general by storing the impacts before the documents, like this:

[4,1] (10) [3,1] (2) [2,1] (3) [1,3] (1, 7, 8)

The pairs in brackets represent impact levels and a count of the number of documents to come. After each impact pair, a list of document numbers follows in parentheses. In this example, the list is not actually smaller than the previous impact-sorted representation, but in practice this organization offers substantial space savings.

This organization also allows new scoring possibilities. Even though the traditional term-at-a-time evaluation strategy is still possible in this organization, notice that these new inverted lists store the best documents at the beginning. In the case of a query that contains just one term, the inverted list is already in order—no extensive query processing is necessary to rank the documents. For queries with more than one term, some processing is required, but reading the entire list may not be necessary.

Anh and Moffat's recent work presents the current best refinement of score-ordered query evaluation procedure [11]. Their query evaluation approach has four stages, called OR, AND, REFINE and IGNORE.

In the OR stage, evaluation begins by processing all of the data in the inverted lists, starting with the largest impact values of the inverted lists and moving toward smaller values. Each time a new document is seen in one of the inverted lists, a new accumulator is created. At some point during evaluation, it may be possible to prove that accumulators exist for every document that could possibly enter the top n. At this point, evaluation switches to AND mode.

Here we briefly explain how such a proof could be carried out. We suppose that the impact values for a document for each of the query terms are simply added to obtain a final document score. Suppose there are 2 terms in the query, and there are already n accumulators that contain values larger than 2. We then know that it would be impossible for a document that had an impact value of 1 for both query terms to enter the top n of the ranked list (since we already know of n documents that are better). Since we score documents by impact level, we can stop generating

new accumulators at this point, since we can be sure we have already seen all the good documents. However, just because we have n accumulators with values larger than 2 does not mean that these accumulators contain final document scores. We still need to complete query processing for the documents that have accumulators, however, we can skip all of the other documents. This gives us a substantial speed savings.

The next step is AND processing mode, where query processing continues, but we ignore all information about documents that do not currently have an accumulator. As this process continues, we will continue to gain information about the maximum possible score for each accumulator. At some point, we hope to gain enough information such that we know exactly what the top n documents are, but we do not know their order. We do this using the same logic as used in the AND mode transition.

REFINE mode proceeds very quickly, because we can ignore all but a handful of documents. However, we can further optimize this process. At some point, we may determine that the rank order of the documents has been fixed (as in Buckley [22]) and query processing can complete. This final stage is called IGNORE, because all remaining inverted list information can be ignored.

While this processing strategy contains many of the elements of Turtle and Flood's max-score optimization [100], It is able to prune more efficiently because the documents are stored in order by impact. This ensures that the best documents are seen first.

Long and Suel explore query processing with a document-ordered system where documents are ordered by PageRank value [61], and also separated in two pieces, as in Strohman et al [94]. The evaluation is non-traditional by information retrieval standards in that it does not use any of the traditional information metrics. However, this appears to be the closest approximation to the work we show in Chapter 7.

2.6.2 Precomputed Phrase Lists

As we have seen in Metzler's Markov Random Field dependence model, term proximity information can have a significant impact on retrieval effectiveness. However, these phrases can be expensive to compute at query time. A simple alternative is to precompute these phrases instead.

The most comprehensive work on precomputed lists comes from a series of papers by Bahle, Williams and Zobel, summarized by a journal article by Williams et al. [104]. They find that it is possible to precompute all two-word phrases in space roughly twice the size of the original index (with positions). Furthermore, they suggest a structure called a nextword index. In a nextword list, each term posting contains information about not just the location of a word, but the identity of the next word. For example, a list for the word 'white' would contain all occurrences of the word 'white' in the collection, and for each occurrence of 'white,' the word that follows it. This list alone would be enough to evaluate the phrase 'white house'; simply scan the list looking for all the occurrences of 'white' that are followed by 'house'.

The authors found that the nextword index was particularly useful for frequent words like 'the'. In their recommended configuration, nextword indexes are created only on very common words, so to evaluate a query like "walk the dog," we look up the list for 'walk' and the nextword list for 'the'. The nextword index allows efficient processing of "the dog," which can then be combined quickly with results for "walk." The authors find that only rarely do phrase queries end with a very common word like 'the,' so rarely does an entire common word list need to be scanned to evaluate a phrase query.

The authors find that indexing every possible two-word phrase leads to highly efficient processing of phrase queries (over 95% faster than the baseline). They also find that a combination of nextword indexes and popular phrase indexing leads to a small index that processes phrase queries 75% faster than without them.

In Chapter 7, some experiments include two-word phrase indexes. No position-based or nextword indexes are used in this dissertation.

2.7 Optimization Types

In this dissertation, we refer to a variety of types of query optimizations. We can classify these optimizations into categories based on how they effect the top k results returned to the user.

Unoptimized retrieval systems compare the user's query to every document in the collection as dictated by the retrieval model, and the score of each document is computed as accurately as possible. The documents are then sorted by retrieval score and the top k highest-scoring documents are returned to the user, in order.

Note that two documents may have the same score, so document scoring does not induce a total order on the set of documents. The relative ordering of a pair of documents with the same score is undefined. This means that the precise set of k documents to return is also not completely defined. Suppose that the kth document in the ranked list shares the same score as the $(k+1)^{\text{th}}$ document; either document could legitimately be considered in the top k. The extreme case occurs when all documents share the same score; in this case, the top k could legitimately consist of any size k subset of the collection documents in any order.

Therefore, for any query Q, collection C, and retrieval model M, the model induces a set of scored documents $R = M(C, Q, k)$. From R, we can derive R_k, which is the set of equivalent, ordered result sets of size k. We consider that the retrieval system has given us a perfect ranked list r_k as long as $r_k \in R_k$.

Unsafe algorithms produce a set of documents with no guaranteed set of properties. Specifically, the documents returned have no provable resemblance to any document set in R_k. However, most unsafe algorithms in common use today have been shown to produce results that, on average, produce results that are close to

the effectiveness of unoptimized systems when evaluated with standard information retrieval metrics. Most of the information retrieval optimizations known today are unsafe.

Set safe algorithms produce a set of documents that is guaranteed to be in R_k. However, the ordering of the documents within that set is not guaranteed to be correct. Buckley [22] described an optimization of this type.

Rank safe algorithms produce an ordered set of documents that is guaranteed to be in R_k. However, the document scores produced by this type of algorithm are not guaranteed to be identical to those produced by an unoptimized system. The algorithm recently proposed by Anh and Moffat [11] is *rank safe*.

Score safe algorithms produce an ordered set of documents that is guaranteed to be in R_k, with document scores identical to those produced by an unoptimized system. The algorithm proposed by Strohman et al. [94] is *score safe*.

Note that *score safe* systems are both *rank safe* and *set safe*, while *rank safe* systems are also *set safe* (but not necessarily *score safe*).

Algorithms with guaranteed safety levels are inherently attractive, since they decouple optimization research from retrieval model research. The guarantees listed at varying levels of safety allow us to make certain guarantees about system performance under common information retrieval metrics. For instance, all *set safe* systems are guaranteed to equal some retrieval in R_k in Precision and Recall at k. All *rank safe* systems are guaranteed to equal the retrieval performance of an unoptimized system under most common metrics we are aware of, including Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (NDCG).

All optimizations presented in this dissertation are at least *rank safe*, and almost all are also *score safe*.

CHAPTER 3

TUPLEFLOW

3.1 Introduction

The focus of this dissertation is on fast and effective document retrieval, but efficient retrieval starts with a carefully designed index. Existing indexing systems are already carefully crafted for speed and scale, but our work goes further. In later chapters, we will be developing new kinds of index structures that are custom tuned for particular query types. This kind of custom tuning requires a level of flexibility in the indexing system that we have not seen before in the literature.

In this chapter we present TupleFlow[1], a distributed computing framework which performs all of the indexing results in this dissertation. TupleFlow achieves the following goals:

Flexibility: TupleFlow allows users to describe complex computations in parameter files that can be changed at runtime without modifying code. Users assemble computations from reusable building blocks that can be easily shared between users.

Scalability: TupleFlow understands the dependencies between parts of a computation, and uses this information to schedule parts of the computation on many processors. TupleFlow can use multiple processors on a single machine or multiple machines that share a filesystem. We will later show how TupleFlow

[1] The author pronounces TupleFlow as "too-pul flow," although "tuh-pul flow" is a reasonable alternative.

can efficiently scale a computation when more processing resources are applied to a problem.

Disk Abstraction: TupleFlow hides all disk and network accesses from the programmer by using a streaming data model. The more difficult tasks of disk-based sorting and data distribution are either handled by included building blocks or hidden in the mechanics of the system.

Low Abstraction Penalty: TupleFlow automatically generates glue code, including custom hashing, sorting and serialization functions for each data type used in a computation. We show how this code generation, combined with TupleFlow's detailed knowledge of the data used in a computation, allows TupleFlow to automatically compress data to a level similar to a manually tuned file format.

3.2 Example

3.2.1 A Traditional Approach

Before diving into the specifics of the system, we will start with a simple example computation.

Many kinds of text computation systems are based on an accurate statistical model of word occurrence in text. In information retrieval systems, we use word occurrence data to build features like IDF (Inverse Document Frequency) which help balance the importance of query terms. In statistical machine translation, word occurrence data is used as a smoothing tool to help the system prefer translations that are more like real natural language.

Typically we create a word occurrence model by counting the number of times each unique word appears in a set of documents. The following Python program shows a very simple approach to solving this problem:

```
words = {}

for filename in files:
    for line in file(filename):
        for word in line.split():
            words[word] = 1 + words.get(word, 0)

print words
```

Figure 3.1. A simple Python word count program.

This small program begins by creating a hash table called `words`. We assume that there is a list of filenames stored in an array called `files`. The outer loop processes one filename at a time. The next loop processes each line of the file `filename`. The inner loop splits a line into words. For each execution of the inner loop, the program looks up `word` in the hash table, and adds one to its word count. The last line of the program prints the contents of the hash table, which contains a list of unique words in the set of documents and the number of times each word occurs.

The brevity of this program should indicate that this is a fairly simple problem, as many text processing tasks are. The trouble is that this program is not particularly flexible, and it is not scalable.

This program currently makes the assumption that there are only a few files to process. If there are millions, perhaps it makes sense to process files using a pool of threads. Moving to the multithreaded model then requires that we place a lock around the words hash table, which would lead to high lock contention among the threads. Another option would be to keep one hash table in each thread, and then merge the tables when execution ends. This approach would allows us to scale to multiple machines.

Another scalability problem is that it assumes the words hash table fits in memory. Unfortunately real text files contain a limitless number of unique words, including

misspellings, part numbers, URLs, and names of all kinds (people, products and companies). Given enough data to process, the hash table will not fit in memory. Fixing this problem requires changing the entire computational approach of the program.

Finally, the program is not particularly flexible. What if the document data came from a database instead of a list of text files? What if we want to use a more advanced parser to split the document into word tokens? What if we would like to remove very frequent words like “the” or “and” to speed up the program? Each one of these requires a code change.

Next, we show how to write this word counting program using TupleFlow. The resulting program is certainly more complicated, but the scale and flexibility problems are solved.

3.2.2 Using TupleFlow

TupleFlow works by connecting computational building blocks, called *Steps*, together. The name TupleFlow is fitting because data tuples flow between these steps. To define our TupleFlow computation, we will first define a tuple type, then create some Steps, and then specify how the steps should be connected together.

```
java -cp galago.jar galago.tupleflow.TemplateTypeBuilder \
     src/mytype/WordCount.java  \
     mytype                     \
     WordCount                  \
     String:word                \
     long:count                 \
     order:+word                \
     order:
```

Figure 3.2. The command line for building the `WordCount` type. The arguments shown here are: file name of the generated code, package name of the resulting class, class name of the result, `word` specification, `count` specification, and two `order` specifications.

First, we define a tuple datatype for this computation. Since we are interested in word counts, we will call this type `WordCount`. A `WordCount` has two elements: A `word`, which is a `String` type, and a `count`, which is a `long` type. We will be interested in sorting tuples of this type in ascending order by `word`. In TupleFlow, we represent that order like this: `+word`.

We use the `TemplateTypeBuilder` to create a type from this specification. The full command line is shown in Figure 3.2. This command processes the specification of the tuple we give it and generates a Java source code file that represents this type. This particular tuple specification generates a 649 line source file, which includes code for sorting `WordCounts`, computing hash functions on them, reading them from disk, and writing them to disk. As we will see later, the read and write methods change depending on the order the tuples are in, since the order of the tuples makes a difference in how the data is compressed.

Next, we need to write some steps that actually perform the word counting process. We split this particular computation into two steps, one called `CountsMaker` (Figure 3.3), and one called `CountsReducer` (Figure 3.6)[2].

The first step is a class called `CountsMaker`, which is shown in Figure 3.3. `CountsMaker` has a single method, called `process`, which takes a `Document` object as a parameter. The `Document` class contains an array called `terms` which contains all of the word tokens in the document. For each word in the `terms` array, a new `WordCount` object is made containing `word` and the number 1 as a count.

An example of how this works on actual text is shown in Figure 3.4. The text "the cat in the hat" has been processed through a `CountsMaker` object. The result is a list of words and counts. Notice that the word `the` appears twice in the text.

[2]In a MapReduce system, `CountsMaker` would be considered a Map step, while `CountsMaker` would be a Reduce step. As we will see later, TupleFlow is not restricted to just these two kinds of steps.

```
public void class CountsMaker
          extends StandardStage<Document, WordCount> {
    public void process( Document document ) {
        for( String word : document.terms ) {
            processor.process( new WordCount( word, 1 ) );
        }
    }
}
```

Figure 3.3. Source code of CountsMaker.java

Word	Count
the	1
cat	1
in	1
the	1
hat	1

Figure 3.4. Some `WordCount` tuples, after processing by `WordCounter`. Notice that there are two lines for `the`.

Word	Count
cat	1
hat	1
in	1
the	1
the	1

Figure 3.5. `WordCount` tuples, after sorting in `+word` order.

Instead of a single tuple of output with a count of 2, there are two rows containing `the` in the output, each with a count of 1. To create the output we actually want, which contains one tuple for each unique word and its count in the text, we need to add up all of the output tuples that contain the same word.

The `CountsReducer` class (Figure 3.6) performs the tuple addition. Like `CountsMaker`, this class contains a `process` method, but it accepts a `WordCount` tuple as a parameter. At the top of the class, notice the `order="+word"` declaration. This declaration tells TupleFlow that the WordCount tuples are expected in ascending order by `word`, as in Figure 3.5. If the input is sorted this way, all of the tuples for a particular word appear next to each other in the list, as in the case of the word `the`. The `process` method detects any duplicate lines and uses the `count` variable to make a complete tally of word occurrences. The `flush` method creates a single `WordCount` tuple for each unique word, and the `close` method ensures that `flush` is called one last time once the input data is complete.

Our strategy is now clear. We convert documents into a stream of `WordCount` tuples, then we sort those tuples by `word`, then we sum up the adjacent tuples to form final word counts.

We use TupleFlow parameter files to connect these pieces together. TupleFlow parameter files are similar to Makefiles or Ant build files used to specify how source code projects are built. The TupleFlow parameter file specifies a set of stages, which are like Makefile targets, and dependencies between those stages. It is the job of TupleFlow to schedule these stages so that no dependence orders are violated, but also so that the maximum amount of parallel computation is achieved.

The first stage description we will consider is for the initial word counting stage, and it is specified in Figure 3.7. The `connections` block at the top of the stage description tells us that this stage processes `FileName` tuples, and outputs `WordCount` tuples in `+word` order. The actual computation is described in the `steps` section.

```
@InputClass(className="galago.types.WordCount", order="+word")
@OutputClass(className="galago.types.WordCount", order="+word")
public void class CountsReducer
          extends StandardStage<WordCount, WordCount> {
    WordCount last = null;
    long count = 0;

    public void process( WordCount wc ) {
        if ( last != null && last.word.equals(wc.word) ) {
            count += wc.count;
        } else {
            flush();
            last = wc;
            count = wc.count;
        }
    }

    public void flush() {
        if ( last != null ) {
            WordCount wc = new WordCount( last.word, count );
            processor.process( wc );
        }
        last = null;
        count = 0;
    }

    public void close() {
        flush();
    }
}
```

Figure 3.6. Source code of CountsReducer.java

```
<stage id="counting">
    <connections>
        <input class="galago.types.FileName"
               order="+filename"
               id="filenames" />
        <output class="galago.types.WordCount"
               order="+word"
               id="counts" />
    </connections>

    <steps>
        <input id="filenames" />
        <step class="galago.parse.UniversalParser" />
        <step class="galago.parse.TagTokenizer" />
        <step class="CountsMaker" />
        <step class="galago.tupleflow.Sorter">
            <class>galago.types.WordCount</class>
            <order>+word</order>
        </step>
        <step class="CountsReducer"/>
        <output id="counts"/>
    </steps>
</stage>
```

Figure 3.7. Stage description for `counting`.

Input starts with a list of `filenames`. These filenames are passed to the `UniversalParser`, which opens the files and extracts the text. These documents are passed to the `TagTokenizer`, which splits the text into word tokens. The next step is the class we built in Figure 3.3, called `CountsMaker`. This object generates a list of `WordCount` tuples, each with a count of 1. Those tuples are passed to a `Sorter`, which sorts the tuples in ascending order by `word`. A `CountsReducer` class is then used to remove duplicate words and add the counts together. Finally, the list of word counts is written to a stream called `counts`.

Ignoring for the moment that we have not described how the filenames are generated, this stage description already shows how we have solved the problem of the hash table that is too large for memory. Our `CountsMaker` and `CountsReducer` use only a fixed amount of memory which does not depend on the number of documents processed or the size of the vocabulary. For large collections of documents, the sorting stage will need to use disk files to sort the counts list, but this functionality is included in the library so we do not need to write it. The result is a program that can handle large collection sizes without running out of memory, assuming enough temporary disk space is available.

We also have solved the problem of flexibility. The logic for counting words has been split apart from the logic of parsing and word manipulation. If we want to extract a stream of words in a different way, we can just change the `UniversalParser` line in the parameter file to some other word source.

What we have not solved yet is the problem of parallelism. This stage, by itself, describes how to convert a series of files into a list of counts using a single processor. We would like to be able to do this on many processors at once. This is where two other stages come in.

First, we will look at the more interesting stage, `reduce` (Figure 3.8). The `reduce` stage takes a list of `WordCount` tuples, processes them through `CountReducer`,

```
<stage id="reduce">
    <connections>
        <input class="galago.types.WordCount"
               order="+word"
               id="counts"/>
        <output class="galago.types.WordCount"
               order="+word"
               id="reducedCounts"/>
    </connections>

    <steps>
        <input id="counts" />
        <step class="CountReducer" />
        <output id="reducedCounts" />
    </steps>
</stage>
```

Figure 3.8. Stage description for `reduce`.

then writes them to a `reducedCounts` stream. This stage seems redundant until we consider running many of the counting stages in parallel. Each one of the counting stages will produce a list of sorted `WordCount` tuples. TupleFlow can combine all of those sorted lists together into a single sorted list, which will have duplicate words in it. This stage reads that combined list, adds word count tuples as necessary, and writes out the final output as a reducedCounts stream.

The filenames needed by the counting stage are provided by the `filenames` stage (Figure 3.9). This stage uses the `FileSource` class to create one `FileName` tuple for each file found in the `/work/corpus` directory. If we had wanted just a single thread of execution, the `FileSource` step could have been inserted directly into the counting stage. However, by putting the filename generation in a different stage, we allow TupleFlow to transparently distribute filenames to many different copies of the counting stage.

The connections block completes the TupleFlow computation specification (Figure 3.10). Each connection describes a dependency between two or more TupleFlow

```
<stage id="filenames" >
    <connections>
        <output class="galago.types.FileName"
                order="+filename"
                id="filenames" />
    </connections>

    <steps>
        <step class="galago.tupleflow.FileSource" >
            <directory>/work/corpus</file>
        </step>
        <output id="filenames"/>
    </steps>
</stage>
```

Figure 3.9. Stage description for `filenames`.

```
<connections>
    <connection class="galago.types.FileName"
                order="+filename"
                hash="+filename">
        <input stage="filenames"
               endpoint="filenames" />
        <output stage="counting"
               endpoint="filenames"
               assignment="each" />
    </connection>

    <connection class="galago.types.WordCount"
                order="+word">
        <input stage="counting"
               endpoint="counts" />
        <output stage="reduce"
               endpoint="counts"
               assignment="combined" />
    </connection>
</connections>
```

Figure 3.10. Description of connections between the filenames, counting and reduce stages.

stages. More importantly, in TupleFlow each dependency is a data dependency, because it means that one stage is generating data that is used by another stage. That is why these are called connections, because they specify data transfer points between stages. In our implementation of TupleFlow, these actually correspond to files in a filesystem, but there is nothing about the job specification that requires that to be true.

In the first connection, we see that the filenames stage generates a named stream of `FileName` tuples called `filenames`. These tuples need to be passed to the counting stage, to the input endpoint also called `filenames`. Notice the attribute `assignment` which is set to `each`. This means that if there are multiple copies of the counting stage, the filenames should be partitioned out evenly among them. The text `hash="+filename"` tells TupleFlow to use a hash function based on the `filename` field to perform the distribution.

In the second connection, we see that `WordCount` tuples need to be transported from the `counts` stream in the `counting` stage into the `counts` stream in the `reduce` stage. This time the `assignment` attribute is set to `combined`, which means that even if there are many different counts streams coming from many copies of the counting stage, TupleFlow should combine those streams into a single sorted stream, then pass that single stream to the reduce stage. TupleFlow can also do a partial reduction in the counting stage, although that has been omitted for this example.

Based on the assignment attribute in the connection descriptions, TupleFlow can decide when it is safe to make copies of a stage, and when only one copy can be used. The connection dependencies also tell TupleFlow when different kinds of stages can be scheduled concurrently.

Now we have solved the flexibility problem and both of the scalability problems of the initial program. While it is true that this second solution is considerably longer than the original solution, the TupleFlow solution did not require a huge

amount of code. More importantly, if we were to try this exercise without TupleFlow, our resulting program would have used more advanced programming techniques like threads, mutual exclusion and out-of-memory processing. Instead, we wrote relatively simple configuration logic which is then typechecked for correctness by TupleFlow.

3.3 Related Work

Distributed computation is one of the oldest problems in computer science, and the literature of proposed solutions to the distributed computing problem is huge.

MapReduce [42] is the closest ancestor to TupleFlow, and was the primary inspiration for its development. Like TupleFlow, MapReduce distributes computation of data-driven tasks across clusters of machines. Each task has two parts; a *map* stage which pre-processes incoming data and splits it into key/value pairs, and a *reduce* stage, which combines the key/value pair data into some final output representation. MapReduce uses hashing on the keys of each key/value pair to distribute pairs across reduce instances. The framework includes automatic fault tolerance and load balancing tools that are essential for very large scale use. The result has been in use at Google since 2003 and has been effective at distributing computation across thousands of processors. The Hadoop project [39] contains a conceptual copy of Google's MapReduce implementation.

MapReduce can be thought of as a subset of TupleFlow where each computation graph is composed of only two steps, tuples have only two elements, and there is only one tuple stream. TupleFlow extends this framework by allowing arbitrary execution graphs with an unlimited number of steps and tuples with unlimited cardinality.

Dryad [51] is a refinement of MapReduce which was developed concurrently and independent of TupleFlow. Dryad also extends the MapReduce idea to support a directed acyclic graph of computation steps. However, unlike TupleFlow, it does not support automatic tuple type generation, sorting, hashing and compression. In ad-

dition, it shares with MapReduce the idea that computational stages are monolithic, and each runs in its own thread, although some communicate using shared memory. The TupleFlow step and stage model encourages the development of very small interchangeable components, since all elements of a stage communicate within a process boundary. TupleFlow further encourages reconfiguration through its use of parameter files to assemble computation graphs.

TupleFlow relies on a shared filesystem to transport data between execution stages, but does not supply one. However, many suitable shared filesystems exist in the literature. The Network File System (NFS) [85] is the *de facto* standard for shared filesystems, but is not scalable enough to handle the kind of load that TupleFlow can produce. Distributed filesystems, like Lustre [87], Google File System [49], and Hadoop DFS [39], store a single filesystem across many different nodes, allowing the bandwidth and capacity of the filesystem to scale with the number of nodes used. We used both NFS and Lustre in our experiments.

The concurrent programming language Erlang [12] is based on lightweight processes that communicate with messages. Its model is similar to the TupleFlow model where single-threaded steps communicate through tuple messages. Also like TupleFlow, Erlang processes can communicate even if they are placed on different nodes. However, the communication model of Erlang allows any process to communicate with any other, meaning that all processes must exist simultaneously. By contrast, by forcing computations to use a dependencies in a directed acyclic graph, TupleFlow can execute an entire computation with only one process running at a time. This can be particularly important because of the large amount of memory text processing can require.

The idea of using a cluster of machines to solve large scale problems was pioneered by the Berkeley NOW project [13, 6]. These original explorations included a

special focus on high speed sorting and data distribution, which TupleFlow, Dryad and MapReduce build on.

Scientific computing applications on large scale multiprocessors and clusters typically use a message passing framework for communication like MPI [1]. These programs are often spatial simulations where the state of a particular space is mapped to a number of different nodes. The model is updated in time steps by passing messages quickly between the nodes. This kind of task requires high-bandwidth, low latency messaging. By contrast, messaging in TupleFlow can be equally high-bandwidth, but with huge amounts of latency. As mentioned above, the TupleFlow model also only parts of the execution graph to be active at one time, while MPI tasks generally run simultaneously.

The name TupleFlow comes from dataflow, as in dataflow architecture microprocessors. The TRIPS project [86] is a modern example of a hardware dataflow architecture. Instructions are presented to the processor as miniature dependency graphs, which allows the processor to schedule functional units more efficiently. TupleFlow uses this computational model but at a massive scale and with more datatype complexity.

Splitting tasks into computational steps can also improve the ease of developing scalable servers. Welsh et al. [103] demonstrated the Staged Event Driven Architecture (SEDA) which achieved high server performance by performing batches of similar tasks simultaneously. This tends to improve overall cache performance and reduce total call overhead. More recently, Burns et al. [25] demonstrated the Flux language for generating high performance servers. Like TupleFlow, Flux encourages developers to write simple, small blocks of code that can be glued together using a domain-specific language. The more difficult tricks of server progarmming, including asynchronous I/O and error handling, are handled by the library. The result is that

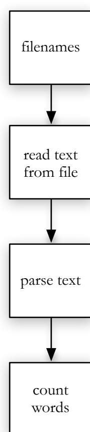


Figure 3.11. A simple TupleFlow execution graph.

inexperienced developers can quickly create very high performance servers. We hope to achieve a similar result, but with scalable distributed data processing.

3.4 Model of Computation

A TupleFlow computation is based on a set of tasks, called *steps*, connected together by typed streams of tuples. We suppose that all of the steps in the computation are running simultaneously, each on its own processor. Typically a step is event-driven, in that it waits for input to appear, processes that input and perhaps produces a result, and waits again for the next event. The steps can form an assembly line structure, or any directed acyclic graph.

This graph structure already gives us a certain level of parallelism, since we could have one computational thread for each step. However, suppose we have a computational cluster that includes thousands of processors. In order to use all those processors, we need thousands of steps. Since most computational tasks are not easily decomposable into thousands of unique steps, we replicate steps instead. Steps are organized into groups called *stages*, which are the atomic unit of replication.

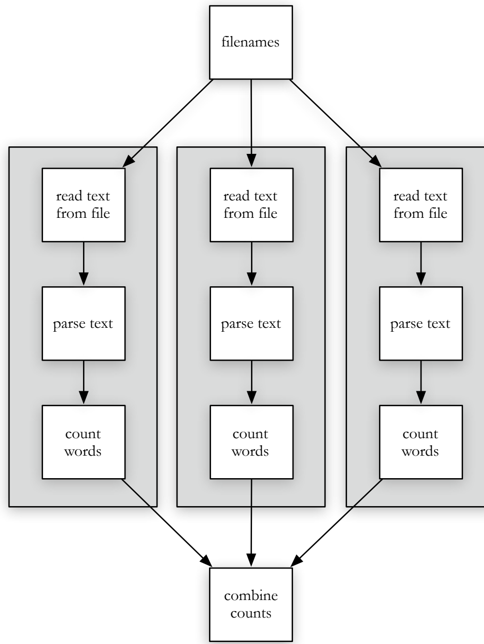


Figure 3.12. A TupleFlow execution graph with a replicated stage.

An example of how this works is shown in Figures 3.11 and 3.12. In Figure 3.11, we have a sequential pipeline of four steps. The first step generates a stream of filenames. The second step reads each filename, opens that file, extracts the text, then passes that text to the third step. The third step parses the text into word tokens, then generates a stream of word tokens for the final step, which generates a list of unique word counts. This graph might keep two processors busy parsing and counting, but the filename generation step and text extraction step are not particularly intensive.

Contrast this with Figure 3.12. The gray block represents a stage, which contains the extract, parse, and count steps. This stage has been replicated so there are three copies of it instead of just one. The filenames generated by the first step are distributed evenly between the three replicas. Each replica generates a list of word counts for its set of documents. Those word counts are then combined using a final word count combination step. The parallelism of the replicated graph is only limited by the number of input filenames, not by the number of steps in the computation.

The streams of data that connect steps are typed and ordered, meaning that a TupleFlow graph can be typechecked for not just the type of tuples used, but the order that they flow through the streams. Our implementation does both kinds of typechecks.

In order to handle stage replication, TupleFlow requires both data distribution and combination. TupleFlow uses hashing to distribute tuples across replicas. A hash function computes a hash value from each data tuple. The stage is chosen by the hash value modulo n. Since distribution is done by hash function, any tuples that are equal under the hash function will be sent to the same replica. For example, suppose we have a stream of word count tuples, and a hash function that is computed only over the text of the word. All word count tuples corresponding to a particular word will be sent to the same replica. This property is frequently exploited in practice, and TupleFlow allows for many different hash functions to be specified for the same tuple type in order to change the distribution patterns to fit the particular computation.

Streams of tuples in TupleFlow are usually sorted in some order, and that order is preserved during stream combination.

3.5 Step Implementation

TupleFlow includes only a few general purpose step classes for things like sorting and building lists of filenames. The remaining computational steps must be supplied by the developer.

A typical step class is a subclass of `StandardStep`. A `StandardStep` is a simple pipeline step which takes one stream of tuples as input, and produces another stream of tuples as output. Subclasses need to implement just one method, `process`. The process method is called by TupleFlow once for each tuple in the input stream. The developer can choose to emit any number of output tuples for each input tuple. This is done by calling the `process` method of the `processor` member variable.

The `CountsMaker` example from the example section shows how this is done (Figure 3.3). The process method is called once for each document object. The process method then generates one `WordCount` tuple for each term in the document, and sends those tuples on to the next processor object. The processor variable is set at runtime by TupleFlow based on the computation graph it is given.

As extensions, a step can implement a `close` method to handle the event when no more input remains. It can also supply a constructor that takes a `TupleFlowParameters` object to allow users to configure its execution.

In particular, some steps will pull data from more than one data stream. Often this is done to perform a data join between two types of data. For instance, suppose we have two streams of data. The first is a query log, where each tuple contains a query, a timestamp, and an IP address. The second stream contains location data, where each tuple contains an IP address, latitude, and longitude. Suppose both of these streams are sorted by IP address. A step can perform a merge by matching tuples from both streams by IP address, and emitting query tuples with location information.

To read from multiple lists simultaneously, we need to leave the event driven model behind. Instead, the step class uses a `TupleFlowParameters` object to retrieve `TypeReader`s, which are like iterators over streams of tuples. The step class implements a `run` method instead of `process`. The `run` method takes care of reading tuples from the streams and producing output. This kind of step (known as an `ExNihiloSource`) must be the first referenced in any stage.

3.6 Execution

TupleFlow executes execution graphs specified in a parameter file, using syntax we have already seen in the word count example. The first job of TupleFlow is to typecheck this graph specification to be sure that it can be executed. Since classes

are loaded dynamically, TupleFlow must check to be sure that all of the classes (both steps and tuple types) exist and that all data stream connections are type safe.

Stage replication is so effective at extracting parallelism that our implementation uses a single thread per stage instead of a single thread per step. The steps pass data between each other via method call, so there is no queueing or other communication overhead between steps. However, queues can be explicitly added if doing so might improve performance.

Stages communicate exclusively through files. A stage reads some set of files and produces another set of files as a result. Stages are scheduled so that no two processes are reading or writing to the same file at once. Although the computational model presumes that data is flowing immediately between stages, in fact it is buffered completely, so that one stage will not see any input until all of its input stages have finished running. This buffering is not strictly necessary for the system to function, but it made implementation simpler.

After the graph is deemed safe, TupleFlow detects which stages can be replicated and which stages cannot. It also generates a dependence graph which is sent to a stage scheduler. The scheduler sends out batches of stage replicas for execution, and as they complete, it consults the dependence graph to see what other stages can be executed next. Many different stages can be running simultaneously if there are no dependencies between them.

A variety of executors can be plugged into the stage scheduler. The most basic is a single-threaded executor used for debugging. There are also multi-threaded executors for parallel execution on a local machine, as well as a distributed executor that uses job execution engines like Condor [96] to schedule computation on other machines.

3.7 Code Generation

TupleFlow requires that computational steps exchange data, not just within a process, but across process and machine boundaries. To do that, we need to be able to serialize tuple data structures into binary streams. We also want to be able to sort and hash these tuples on arbitrary column sets.

The Java virtual machine, like many modern virtual machine architectures, allows for runtime object introspection. From an object pointer, it is possible to determine the methods it supports and the member variables it contains. Technically this is all we need to support our goals of serialization, sorting and hashing. In reality, introspection tends to be slow. Given the amount of data that TupleFlow is designed to use, serialization speed is a critical performance point of the code.

Instead, we generate Java code for every type we want to use in the TupleFlow system. As shown in the example, TupleFlow contains a program called TemplateTypeBuilder which converts a tuple specification into a TupleFlow Type class, complete with serialization, sorting functions and hash functions.

3.7.1 Hash functions

A hash function is a function $h : T \to \mathbb{Z}$, where T is the type of some object and $\mathbb{Z}$ is the set of integers. The specific integer value produced is not important, so long as for some relation $R_=$:

1. $R_=(a, b) \to h(a) = h(b)$
2. $\neg R_=(a, b) \to h(a) \neq h(b)$ with high probability
3. $\neg R_=(a, b) \to h(a) \neq h(b) (\text{mod} n)$ with probability approaching $\frac{1}{n}$

We often think of $R_=$ as the equality relation, so $R_=(a, b)$ can usually be replaced by $a = b$, while $\neg R_=(a, b)$ can be replaced by $a \neq b$.

The equality property (1) must always hold. A hash *collision* occurs when either (2) or (3) occurs. In practice, (2) is a special case of (3) for some fixed large value of n, since h typically returns a bounded integer datatype. It is impossible to prove the probability of a hash collision without a description of the distribution of object values, so we generally need to rely on heuristics to make reasonable hash functions.

Notice that all of the properties of the hash function depend on the definition of the relation $R_=$. It is easy to think of equality as an obvious property of an object, but in practice it can be a subtle and changing thing. For instance, when using floating point numbers, we often want to say that two numbers a and b are equal when they are less than some ϵ distance away from each other. Notice that this relation is not transitive, which is a standard property of an equality relation. In this case equality is then actually parameterized based on some value of ϵ, which creates an infinite space of possible equality relations, each with its own set of properties that a hash function must satisfy.

What does ϵ really mean, anyway? The factor ϵ creates an equality relation that, instead of "exactly the same thing," means "essentially the same thing." In TupleFlow we extend this idea to hash functions that mean "the same kind of thing," or "belongs to the same group," where the definition of "group" depends on the application.

For example, consider a tuple `WebLink` that represents a hypertext link between two web pages. The link has two parts, `source` and `destination`, where `source` is the URL of the web page that contains the link, and `destination` is the URL pointed to by the link. What makes two `WebLink`s the same?

Consider three possible equality relations:

1. $R(a, b) \leftrightarrow a.\texttt{source} = b.\texttt{source} \land a.\texttt{destination} = b.\texttt{destination}$

2. $R(a, b) \leftrightarrow a.\texttt{source} = b.\texttt{source}$

3. $R(a, b) \leftrightarrow a.\texttt{destination} = b.\texttt{destination}$

Relation (1) might be called the strict equality relation. We say that a and b are equal only if their sources and destinations match. Relation (2) is not so strict; it says that two tuples are equal if their sources match. Relation (3) says that two tuples are equal if their destinations match.

Why would we ever use relations (2) or (3)? Suppose we group all the links in a corpus based on equality relation (2); each group corresponds to the outgoing links for a particular web page. If we use relation (3) instead, each group corresponds to the incoming links for a particular web page. We typically extract links from web pages in an order that corresponds to relation (2). If the data is transformed into groups based on relation (3), we can easily count the number of links coming into a web page. This kind of operation is the basis of many static ranking functions, like PageRank or HostRank [78, 44].

Because of this, TupleFlow lets users specify equality relations based field names. In TupleFlow, relation (1) is called `+source +destination`, relation (2) is called `+source`, and relation (3) is called `+destination`. The hash functions are not interpreted. Instead, the hash function specification is sent to a factory which instantiates the appropriate hash function, which was already generated by `TemplateTypeBuilder`.

3.7.2 Comparators

As in hashing, sorting is based on a binary relation. In particular, we say that a list of objects L is sorted under relation R if, for all objects a and b in L:

$$R(a, b) \leftarrow a \text{ comes before } b \text{ in } L$$

Under this definition, the relation R does not need to specify a total order on objects. In the degenerate case R is an empty relation, so we say that every list L is

sorted. In TupleFlow, we say that if $\neg R(a, b)$ and $\neg R(b, a)$, then $a = b$. The equals sign has appeared again, so the previous discussion of equality is relevant here too.

Just like with hash functions, TupleFlow specifies comparison relations with order specifications, like `+source` or `+destination`. However, `-source` or `+source -destination` are possible. This last order specification means that `WebLink` tuples should be sorted in ascending order by source URL, with ties broken in descending order by destination URL. Each of these specifications corresponds to a comparison function generated by `TemplateTypeBuilder`. At runtime, a `Sorter` object will instantiate the appropriate comparison function to use for sorting.

Sorting is used for three primary reasons in TupleFlow. The first is to group like objects together. While a hash function does a good job of grouping like objects, its grouping is not guaranteed to be perfect because of hash collisions. The definition of an ordered list contains no talk of probabilities that could lead to imperfect results. The cost is that hashing tuples into bins is an $O(n)$ operation while sorting is $O(nlogn)$. The second reason to sort is for compression, which we will discuss later. The final reason is to enable efficient sorted merges and joins between different streams.

3.7.3 Order Compatibility

Notice that certain orders are compatible with others. For example, objects ordered by `+source +destination` are also ordered by `+source`, although the reverse is not true, as shown in the following theorem. This theorem is used to typecheck streams in TupleFlow.

Theorem 1. *Given a list of objects L_A sorted by order specification A, L_A is guaranteed to be sorted by order specification B iff B is a prefix of A (that is, $|A| \geq |B|$ and the first $|B|$ elements of A are B).*

Proof. We first prove sufficiency, then prove necessity.

Sufficiency. Suppose there exists some pair of elements x and y such that x comes before y in L_A, but B implies that x should come after y. However, since B is a prefix of A, any pair of elements that B orders are ordered identically under A (although A may order more elements than B). This is a contradiction, so x and y cannot exist.

Necessity. Suppose B is not a prefix of A. Let C be the common prefix of A and B, let A' be the suffix of A (not including C), and let B' be the suffix of B (not including C).

In the case where $A = C$, $|A'| = 0$ and $|B| > 0$, meaning that B is orders strictly more objects than A. Then L_A may contain such a pair of objects x and y such that x and y are ordered by B' but declared equal by C.

In the case where $|A| > |C|$, L_A may contain a pair of objects x and y which are considered equal under C, but where $x < y$ under A' and $x > y$ under B'.

In either case, a pair of objects x and y may exist in L_A that violate the order specification of B. □

3.7.4 Compression

The data streams used by a TupleFlow computation can cause enormous strain on a cluster of machines. If disks are shared, as they are in a NFS environment, many processors are competing for a small amount of disk bandwidth. In a distributed filesystem, the disks themselves are may not be the bottleneck, but the network switch could be, especially when data must pass through multiple switches to reach its destination. Even if neither the switch nor the disk are an impediment, the kinds of work that TupleFlow does tend to make poor use of L2 cache, causing the CPU to constantly wait on memory. Compressing the data in the data streams helps alleviate all of these problems.

Because the order of objects in each stream is known to TupleFlow, it can use that order information to aggressively compress data. Three compression schemes are

used in combination: run length encoding, delta encoding, and variable byte (vbyte) encoding [105].

Suppose we have ordered a list of `WebLink` tuples by `destination` URL. Intuitively we know that there are some web pages with huge numbers of incoming links. If our list contains 2832 links to the UMass web page, there will be 2832 consecutive tuples with `www.umass.edu` in the destination field. We can save on storage space by writing the number 2832 to the data stream, then writing `www.umass.edu`, the 2832 unique `source` URLs. For the cost of an integer, we are able to save the storage space of `www.umass.edu` repeated 2831 times. This is called run length encoding, because we encode a run (a set of tuples with a similar destination) with its length. TupleFlow automatically employs run length encoding for any tuple element which is a String, and in some cases of integer values.

For integer tuple elements we have more compression choices. Suppose we have a list of `WordCount` tuples ordered in ascending order by `count`. Since we know that each count field is at least as large as the last, we can store just the difference between each count value. A list with counts 1, 5, 7, 8, 10, 15, 21 can then be encoded as 4, 3, 1, 2, 5, 6. There are still six numbers in this list, but the maximum number in the second list is 6, versus 21 in the first list. This process is called delta encoding, and by itself it produces no compression gains.

However, if we use a numeric compression scheme that favors small numbers, delta encoding does pay off. TupleFlow uses variable byte (vbyte) compression for all numeric values. Using vbyte, all numbers smaller than 128 use just one byte. Numbers smaller than 16384 use two bytes. In the general case, for a number x such that $2^{7n} \leq x < 2^{7n+1}$, x requires n bytes to encode using vbyte. Vbyte does this by using just the low 7 bits of each byte to encode numeric information, while the high bit, when set, signals the end of a numeric code.

By itself, variable byte encoding does a good job of compressing numbers since most numbers are small. Think of the word count case; most unique strings appear just once in a corpus, while a few might occur millions of times. When combined with delta encoding, the space savings is even larger.

We will review some space saving experiments using TupleFlow at the end of the chapter.

3.8 Built-in Steps

The two most important steps in TupleFlow are sorting and hashing. The `Sorter` is explicitly instantiated by users, but hashing is done implicitly using the `hash` and `assignment` attributes in the computation parameter file.

The `Sorter` is a typical TupleFlow step, in that it has a `process` method which responds to incoming tuples, and it sends output to the `process` method of another step. Unlike most steps, however, `Sorter` emits no output tuples until all of the input has been seen. For small streams of tuples, the data is completely buffered in memory, sorted using a library sort function, then output to the next stream. Typically the input stream is larger than memory. In this case, the `Sorter` fills its internal buffer with tuples until the buffer is full. The buffer is then sorted and written to a temporary file using the compressed writing routines in the tuple class. This process continues until all of the tuples have been read from the input. At this time, the `Sorter` performs a merge operation on the temporary files in order to create a sorted output stream.

In the case where there are a large number of temporary files, `Sorter` merges batches of 20 temporary files at a time into larger temporary files, until less than 20 temporary files remain. While this may improve performance, the primary purpose of this is to reduce the number of files open at once, since some operating systems are very restrictive in the number of file handles allowed per process.

```
stream/
    0/
        0
        1
        2
        3
    1/
        0
        1
        2
        3
```

Figure 3.13. The on-disk representation of a stream with 4 inputs and 2 outputs.

```
stream/
    0/
        0
    1/
        1
    2/
        2
```

Figure 3.14. On-disk representation of a pass-through stream with 3 inputs and 3 outputs.

The `Splitter` handles hashing tuples, which is the only means of distributing computation in TupleFlow. The `Splitter` has one input stream and n output streams. Typically each of these n output streams is connected to a `Sorter`, which is then connected to an output file. The `Sorter` makes sure that the tuples are written to disk in sorted order, allowing the use of order-sensitive compression.

3.9 Storing Streams

Conceptually a TupleFlow stream is a tuple pipe that connects m inputs to n outputs. In practice this means that a TupleFlow stream can be composed of as many as mn files.

The files for a stream are stored in a directory hierarchy. The top level directory named after the stream. Under that directory is a set of numbered directories from

0 to $m-1$. Within each directory, there can be as many as n files, numbered from 0 to $n-1$. Figure 3.13 shows the directory structure for a stream with 4 inputs and two outputs.

For the case where $m > 1$ and $n > 1$, each of the m inputs hashes its output into n files. Input k stores its output in files $0/k$, $1/k$, $2/k$, ..., $n-1/k$. To read from output j, TupleFlow opens all files in directory j and performs an ordered merge operation.

In one common case, $m = n$, and the data is not hashed. This is known as a *pass-through* connection, since it directly connects one stage replica to another. The directory structure for this is shown in 3.14. As an optimization, there is only one file in each output directory instead of n, since the other $n-1$ files would be empty.

3.10 Checkpointing

TupleFlow requires that users define data dependencies explicitly between tasks, and also requires that users split their tasks into small pieces for distribution. While these requirements are useful for distribution, they also make it possible to add automatic failure recovery. The latest version of TupleFlow supports a semi-automatic failure recovery mode.

A try/catch block surrounds all TupleFlow stage executions. If the stage instance completes without throwing an exception, a checkpoint file for that stage is written to disk. If an exception is caught, an error file is written to disk instead. No attempt is currently made to stop other stages after an exception is caught, but an exception thrown in one stage typically causes exceptions to be thrown in all subsequent stages, since their input data is likely to be incomplete or corrupted.

Many exceptions are caused by simple code bugs or improper parameter settings. After fixing one of these kinds of errors, the TupleFlow job can be manually restarted. The TupleFlow execution code makes note of all stages that have already completed and does not try to run those stages again; instead, it only runs stages that have

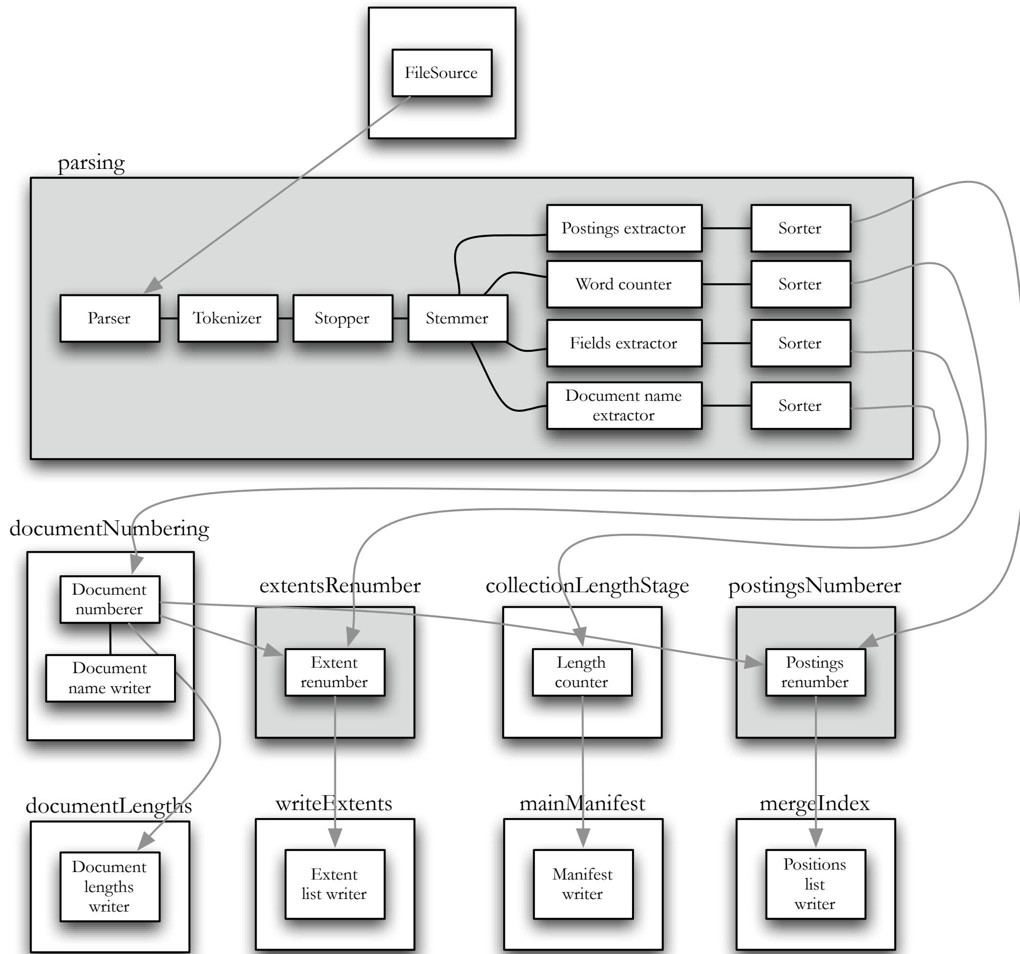


Figure 3.15. A TupleFlow computation graph for building a traditional, positions-based text index. Small boxes are steps, large boxes are stages, and gray boxes indicate stages that can be replicated.

either an error checkpoint file or no checkpoint at all. This can result in a substantial time savings, especially when developing a new kind of stage. When using this feature, users do need to be careful that code changes do not alter the results of the computation that has already been completed.

```
a = LOAD '/tmp/queries' AS (time, query, resultCount);
b = GROUP a by query;
c = FOREACH b GENERATE group as query, COUNT(a) as count;
d = FILTER c by count > '10';
```

Figure 3.16. A short Pig script to find all unique queries in a query log that appear more than ten times.

3.11 Sample Tasks

3.11.1 Building an Index

Figure 3.15 shows a TupleFlow computation graph for building a traditional document-sorted index.

3.12 Rapid Experimentation

TupleFlow is meant to be as general-purpose as possible, and like any flexible system it is more complicated to use than a simpler or less efficient one could be. However, TupleFlow can serve as the basis for a simpler system.

Pike et al. [80] built the Sawzall language for simpler MapReduce computation. The Dryad team [51] mention a language called Nebula which simplifies computation in their system. Following in their footsteps, we ported the Pig language[3] to TupleFlow. Pig is a language developed at Yahoo! Research for rapid experimentation on top of the Hadoop system [39].

Figure 3.16 shows an example Pig script. This script reads tab-delimited query log data from a text file called `/tmp/queries`. Each line in the log file contains a timestamp, query text, a session identifier and a count of results. The variable `b` contains groups of log lines, with one group for each unique query. The variable `c` measures the size of each group to form a count for each unique query. Finally, variable `d` contains all queries that occur at least 10 times in the log.

[3] http://research.yahoo.com/projects/pig

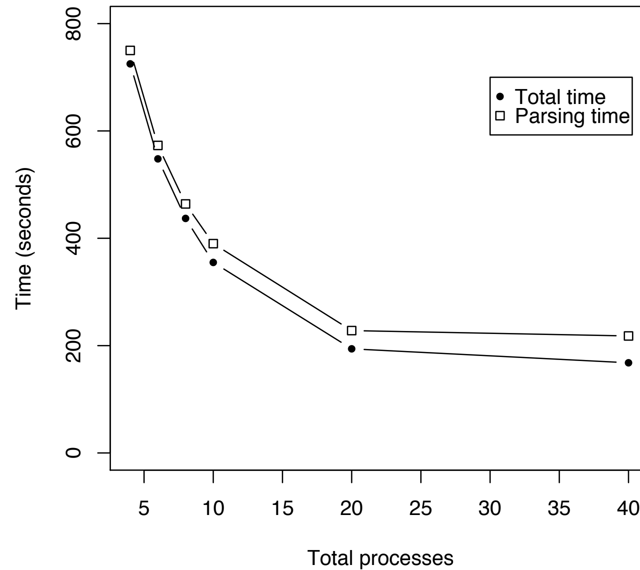


Figure 3.17. Total time and parsing time to count the words in the GOV collection for many levels of parallelism.

By implementing this language on top of a TupleFlow, this script is actually parallel. Each processor reads a small portion of the query log, groups by query, and counts the number of entries for each query. All of this data is merged together into one master list, which is then filtered so that only queries with at least ten results appear.

Using a query log with 15 million queries in it and 40 processors, this script takes less than 2 minutes to run. A hand-tuned TupleFlow job would probably be faster, but this script takes very little time to write. Using a scripting language like this allows users to rapidly experiment with large datasets using large clusters of processors without thinking about data processing details.

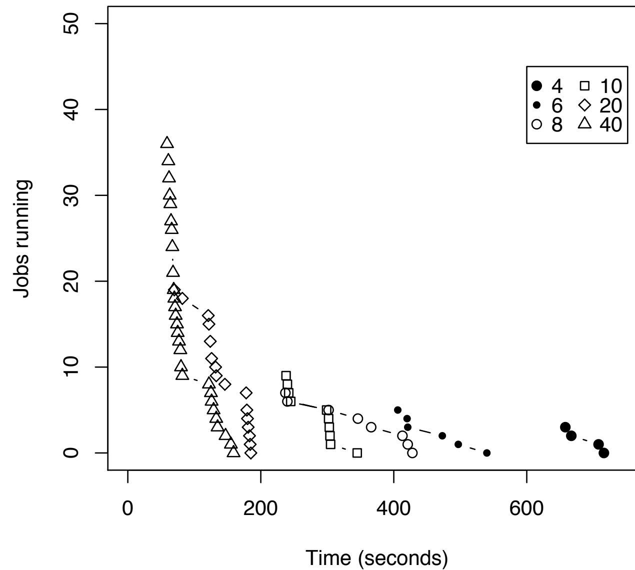


Figure 3.18. Processes running over time for many levels of parallelism.

3.13 Experiments

3.13.1 Word Count

To test the usefulness of TupleFlow, we ran our example word count program on the TREC GOV collection. This is a partial crawl of the .gov domain, containing 20GB of text in 1.2 million web documents. The input text was compressed with gzip so that it occupied 4.3GB on disk. The data was split into 46 files, most of which were about 100MB each (the final one was 13MB).

The execution graph followed the model shown in Figure 3.12. The filenames stage created a TupleFlow stream of 46 filename tuples. These were hashed to split them into many different streams. Each stream was then routed to a parsing process, which parsed the files sent to it and emitted word count tuples. Each word count tuple was a triple: word, document count and term count, where document count is the number of unique documents this word occurs in, and term count is the number of word occurrences regardless of document boundaries. A word count reducer object

was used within each parsing stage to reduce the number of objects written to disk. After all parsing stages completed, a single reducer stage merged all counts streams into one final word count list.

All jobs were run on a cluster of 32 machines, containing 70 cores in all, with core clock speeds between 3.0 and 3.2GHz. TupleFlow used a dedicated Lustre filesystem for all experiments.

Timing results are shown in Figure 3.17. The x-axis represents the number of parsing processes used. Adding processes gives a near-linear speedup until about 10 processes are used. At this point, workload imbalance starts to affect runtimes, as discussed further in the next paragraph. Notice that as the number of processes increases, the difference between total time and parsing time increases, reflecting the additional effort necessary to merge the results when many parsing processes are used.

3.13.1.1 Balance

TupleFlow uses hashing to distribute tuples evenly among stages. This strategy works well when there are many more stage instances than tuples, or many more tuples than stage instances. When there are more instances than tuples, the probability that any instance must process more than one tuple is

$$n\frac{k}{n}\frac{k-1}{n} \simeq \frac{k^2}{n}$$

where k is the number of tuples and n is the number of processes. So, when $k^2 << n$, most instances will receive no tuple, and the rest will receive no more than one tuple with high probability. Assuming each tuple takes approximately equal effort to process, we expect all of the processes that were assigned tuple to finish at roughly the same time.

Alternately, suppose we have more tuples k than instance n. Note that Chebychev's equality is:

$$P[|X - E[X]| > t] = \frac{Var(X)}{t^2}$$

We can apply this inequality to our problem. Let X be the number of tuples assigned to a stage. We expect each stage instance to receive k/n tuples, so $E[X] = k/n$. The variance of X is $np(1-p)$, where $p = 1/n$.

$$\begin{aligned}
P[|X - E[X]| > t] &= \frac{np(1-p)}{t^2} \\
&= \frac{1 - 1/n}{t^2} \\
&\simeq \frac{1}{t^2}
\end{aligned}$$

Note that the result here does not depend on k. In particular, suppose we want to know the probability that some instance gets a factor of c more tuples than the expected k/n:

$$\begin{aligned}
P[|X - E[X]| > (c-1)k/n] &= \frac{np(1-p)}{((c-1)k/n)^2} \\
&= \frac{1 - 1/n}{((c-1)k/n)^2} \\
&\simeq \frac{n^2}{ck - c}
\end{aligned}$$

This probability is nearly inversely proportional to k. This result shows that as k grows relative to n, we expect that the tuples will be distributed more evenly among the processes.

The problem area comes when k is close to n, as we saw in our previous experiment. When the number of instances (40) approached the number of files (46), some instances received two files to parse while others received just one, causing the reduce

```
congresslink,3,1
congresslink,3,1
congresslink,6,2
congressm,1,1
congressm,1,1
congressman,28,1
congressman,3,3
congressman,3,3
congressman,2,1
```

Figure 3.19. An example of the tuples emitted by the WordCount parsing stage.

step to have to wait for those slower processes to finish and making inefficient use of the processors. As the analysis above shows, either increasing the number of instances or increasing the number of files would improve this situation.

3.13.1.2 Compression

As explained before, all data sent between stages in TupleFlow is stored in a compressed binary format that depends on the order of the tuples. We illustrate this with a compression size example.

Figure 3.19 shows an example of some of the tuples emitted by a word count parsing stage. These are slightly different than the WordCount tuple defined in earlier examples, since these count both the total number of term occurrences, and the number of documents the term occurs in. Notice that the tuples are partially reduced already, because some of the tuples show multiple document counts (like `congressman,3,3`, which indicates the word "congressman" appearing 3 times in 3 documents). However, the tuples are not fully reduced, because the same term can appear many times in a row.

TupleFlow compresses this data in two ways. First, since this data is sorted in word order, it stores each word in the list just once by using run length encoding.

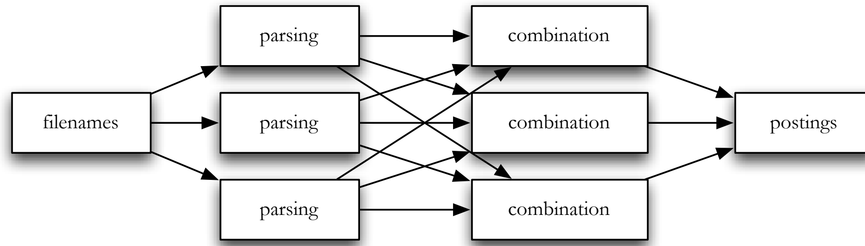


Figure 3.20. A stage diagram of the TupleFlow anchor text combination process.

Second, the integers in the last two columns tend to be small, making them excellent candidates for compression.

The result is that TupleFlow wrote 305,116K (298MB) of compressed word counts to disk during the 10-instance word count job, although that same data requires 1,123,188 (1.1GB) to store in the comma-separated form shown in Figure 3.19. This is a savings of 72%.

3.13.2 Anchor Text Combination

The previous experiments show that TupleFlow can count words efficiently, but that is also a canonical MapReduce example. This section covers anchor text combination, which is a task that is not well suited for a single MapReduce job.

Anchor text is the text found between link tags (`<a href...> </a>`) in web pages. Anchor text is an excellent source of descriptive text for the linked page. To use this text in query processing, the anchor text needs to be somehow copied to the linked page.

Figure 3.20 shows a diagram of the stages used in a TupleFlow anchor text combination job. Filenames flow from the filenames stage to replicated parsing stages. Each parsing stage parses its set of documents, and outputs two streams of data: a list of links found in the documents, and a list of document URLs. Some examples of document URL tuples from the GOV collection are shown in Figure 3.21. The first column is the TREC document identifier, followed by the document's URL. Some

```
G09-02-4188263,http://firstgov.gov/us_gov/doc.html
G01-71-2025464,http://fishclub.gsfc.nasa.gov
G01-81-1240304,http://fishclub.gsfc.nasa.gov/2000review.html
G01-12-2496787,http://fishclub.gsfc.nasa.gov/workshop.html
G01-83-2038485,http://fitness.gov/aboutpcpfs/aboutpcpfs.html
G09-85-2321207,http://fitness.gov/news/news.html
G12-18-2747694,http://fitness.gov/sports/sportsaward.pdf
G12-57-1635810,http://fitness.gov/video/getoffit/getoffit.html
```

Figure 3.21. DocumentURL tuples parsed from the GOV collection.

```
http://www.savingsbonds.gov/mar/mar01win.htm,
    ftp://208.131.225.4:21/mar01ny.jpg,,false
http://www.savingsbonds.gov/mar/mar01win.htm,
    ftp://208.131.225.4:21/mar01ny.jpg,jpeg format,false
http://www.treasurydirect.gov/mar/mar98wn1.htm,
    ftp://208.131.225.4:21/mar98ak.gif,see merellas poster,false
http://www.treasurydirect.gov/mar/mar98wn1.htm,
    ftp://208.131.225.4:21/mar98ca.gif,see laurences poster,false
http://www.treasurydirect.gov/mar/mar98wn1.htm,
    ftp://208.131.225.4:21/mar98fl.gif,see jordans poster,false
http://www.treasurydirect.gov/mar/mar98wn1.htm,
    ftp://208.131.225.4:21/mar98ga.gif,see miriams poster,false
```

Figure 3.22. ExtractedLink tuples parsed from the GOV collection.

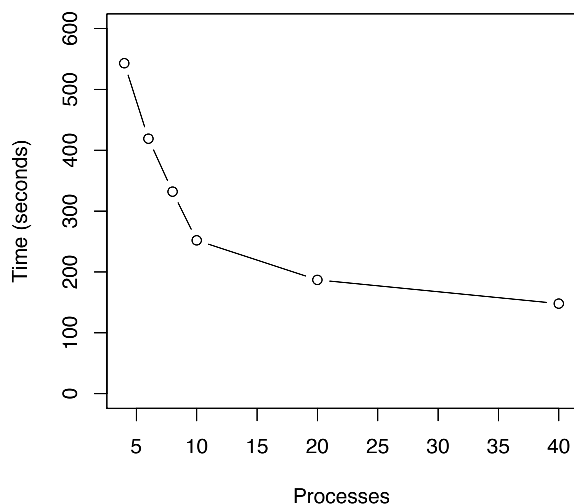


Figure 3.23. Speed of anchor text combination on the GOV collection for many levels of parallelism.

examples of extracted links from the GOV collection are shown in Figure 3.22. The source of the link is in the first column, followed by the destination of the link, followed by the anchor text, followed by a boolean value that is true if the link has a `nofollow` attribute set.

Notice that the data in Figure 3.21 is sorted by URL, and the data in Figure 3.22 is sorted by destination URL. In fact, the parsing stage hashes both of these data streams based on URL and destination URL respectively. This means that the DocumentURL and ExtractedLink tuples for the same pages arrive on the same machines during the combination stage. Since these lists are in the appropriate sorted order, they can be joined efficiently, which then can associate a TREC document identifier with anchor text that points to it. The key to making this work is the two streams of data that are simultaneously hashed to the combination stage instances. MapReduce does not have this capability.

Figure 3.23 shows a picture that is very similar to the word count picture. Between 4 and 10 processors, the speed increases roughly linearly. After that, the speed of the system levels off, primarily because of imbalance issues.

3.13.3 Indexing

TupleFlow is used to build every index used in this dissertation. The large variety of indexes built in this dissertation supports our assertion that TupleFlow is a flexible platform for distributed text processing.

Our focus for the rest of this thesis is on query effectiveness and efficiency, so our index timings are not precise. However, TupleFlow manages to index the 426GB GOV2 collection in an acceptable amount of time. In Section 5.2.1, we report that indexing score-sorted indexes requires 60 hours of CPU time, but 4 hours of wall-clock time, when using 20 processors. Anh and Moffat [11] report that their highly optimized C implementation takes 6.5 hours to build a very similar index on a single machine. Clearly our parser, which takes 42 hours of the indexing time, is not efficient. However, TupleFlow is able to use this slow parser in parallel, and therefore builds the index faster than the optimized single-machine implementation.

3.14 Weaknesses

The current implementation of TupleFlow has some weaknesses. One basic weakness is its reliance on a common filesystem available to all of the nodes, which means that the performance of the system depends critically on the performance of the shared filesystem. For early experiments, our computational cluster used NFS to access a particularly slow disk subsystem, and this was frequently a bottleneck for TupleFlow jobs. Recently we have been using the Lustre distributed filesystem instead, which has completely eliminated the storage bottleneck for our tasks.

Another problem with our current system is the number of files generated. It is not uncommon in our computations to have 100 tasks generate 100 files each, for a total of 10,000 files. Those 10,000 files are then accessed simultaneously by the next group of 100 stage instances. While Lustre has handled this admirably, the problem of file handle exhaustion remains. In order to scale to larger clusters (in the thousands of processors), we will need to better at merging small files together in intermediate stages. Currently the system knows how to insert some anonymous intermediate stages for file merging, but performance would improve with more sophisticated merge logic.

Most TupleFlow jobs spend the majority of runtime just sorting the tuples. This has caused us to consider a number of different ways to sort tuples more efficiently, often with frustrating results. One of the more promising developments is a multi-pivot quicksort implementation which appears to improve sort times by about 30% by exploiting the processor cache better than traditional sort methods.

Although it seems that this model of computation could be used for interactive tasks, the TupleFlow implementation is designed for batch-mode computation, which means there is some overhead in launching a job or a stage. This implementation is probably not appropriate for computations that require less than 15 seconds of processing time.

3.15 Summary

We have presented a distributed execution framework capable of processing using a graph of dataflows between execution stages. We call our system TupleFlow because of its explicit focus on the tuples that pass between stages. Our system has been used to support all of the remaining work in this dissertation, and allows us to quickly assemble distributed tasks from computational building blocks. TupleFlow is used

to build every index used in this dissertation, from the small collections in the next chapter to the large indexes in Chapters 5, 6, and 7.

TupleFlow is the first of three pieces necessary to build the navigational search indexes in 7. TupleFlow provides the flexibility and scalability to process the data. The next chapter introduces mathematical tools that can store language modeling probabilities in integers, so that we can translate structured query formulations into term weights. After that, Chapters 5 and 6 introduce the index structures and query processing techniques necessary for efficient search.

CHAPTER 4

BINNED PROBABILITIES

4.1 Introduction

In Section 2.2.1 we introduced the idea of storing integer values in inverted lists. In particular, we cited the work of Anh and Moffat [10], who developed document-centric impacts explicitly to be stored in indexes that require integer values. However, most query evaluation strategies rely on floating point term weights. In particular, language modeling probabilities are floating point, and are not necessarily 0 when the term does not appear in the document because of smoothing. Later, in Chapter 7, we will show how to use binned indexes to evaluate named page queries using detailed feature representations, and to do that, we need a way to store those feature representations in binned indexes.

In this chapter, we assume a generic query processing system which contains binned integer values $B_{w,D}$ in its inverted lists, where w is a word, and D is a document. The encoding and order of these values is unimportant. We assume that queries are evaluated by calculating the sum of these binned integer values for each candidate document:

$$B_D = \sum_{w \in Q} B_{w,D}$$

Previous work has shown how to produce integer values from pivoted TF-IDF values or from a document-centric weighting process [7, 10]. In this work, we consider computing these weights based on language model probabilities.

4.2 Method

Typical query likelihood ranking is based on a set of term probabilities combined by multiplication:

$$P(Q|D) = \prod_{w \in Q} P(w|D)$$

This can be extended to more complicated formulations, but typically the document is ranked based on a probability which is a product of other probabilities, like $P(w|D)$.

There are some important problems when adapting this equation to an impact-scored model. First, typically impact-sorted retrieval systems add values, while language modeling is based on multiplication. Second, impact-sorted evaluation rests on the idea that if a word doesn't appear in the document, the document-word score is 0. In language modeling, that score is a smoothed estimate for $P(w|D)$ which is small but non-zero. Finally, these sorted indexes rely on small integer values for good compression, while language models are defined as real values between 0 and 1.

We can trade multiplication for addition by taking the log of both sides:

$$\log P(Q|D) = \sum_{w \in Q} \log P(w|D)$$

Now we are computing a quantity that is rank-equivalent to $P(Q|D)$, but instead of a product we use a sum, as required. However, the $\log P(w|D)$ values are negative, instead of the positive integers we are looking for.

For each word w, we can compute the minimum value of $P(w|D)$ over the document collection:

$$C_w = \min_D \log P(w|D)$$

If we subtract this constant from the query for each query word, we maintain rank equivalence:

$$\log P(Q|D) - \sum_{w \in Q} C_w = \sum_{w \in Q} (\log P(w|D) - C_w)$$

Moreover, $\log P(w|D) - C_w$ is guaranteed to be positive, since C_w is defined as the smallest value of $\log P(w|D)$ over the collection.

When Jelinek-Mercer smoothing is used, $C_w = \log \lambda P(w|C)$. When Dirichlet prior smoothing is used instead, $P(w|D)$ depends on the length of D, even when $w \notin D$. In this case, $C_w = \log P(w|D')$, where D' is the longest document in the collection that does not contain w. Even in this case, C_w provides a decent approximation of $\log P(w|D)$ when $w \notin D$. Note that $\log P(w|D) - C_w$ will equal 0 when $\log P(w|D) = C_w$. This is also a desirable property, since $B_{w,D}$ will be implicitly zero when $w \notin D$.

Now, we define M, the negative minimum of $\log P(w|D)$:

$$M = -\min_{w,D} \log P(w|D)$$

Note that $P(w|D) \leq 1$, meaning that $\log P(w|D) \leq 0$. By the definition of M, we have:

$$\forall_w \qquad \log P(w|D) \leq 0$$
$$\forall_w \qquad \log P(w|D) - C_w \leq M$$

Now, we can compute term weights by binning the result:

$$B_{(w,D)} = \lfloor \frac{n}{M} (\log P(w|D) - C_w) \rfloor$$

where n is the number of integers we can use in the binning process. Note that while this formula could technically evaluate to 0 when $w \in D$, in practice we make sure that $B_{(w,D)}$ is at least 1 whenever $w \in D$. Because of the definition of M, $B_{(w,D)}$ will never have a value greater than n.

However, most probabilities from language models do not even approach 1. This means that many of the bin values $[0, n]$ will never be used by the previous formula, and therefore some of the resolution provided by n bins will be wasted.

Instead, we provide a saturation parameter, s. The saturation s is a probability value, less than one. We assert that all probabilities $P(w|D) > s$ should be treated as if they are equal. We saturate those probabilities to make more room in the integer space to resolve differences between important terms.

The final formula is:

$$B_{(w,D)} = \lfloor \frac{n}{M + \log s} (\log P(w|D) - C_w) \rfloor$$

4.3 Exploration

In this section, we will look at some examples of the data this procedure produces in order to explain how it works.

Looking at the Dirichlet-smoothed probability estimate for $P(w|D)$, we have:

$$P(w|D) = \frac{c(w;D) + \mu c(w;C)/|C|}{|D| + \mu}$$

For documents that do not contain the term w, this simplifies to:

$$P(w|D) = \frac{\mu c(w;C)/|C|}{|D| + \mu}$$

The second term in the numerator, $\mu c(w;C)/|C|$, is typically much smaller than 1. For instance, the common word "congress" occurs 43,778 times in the TREC45 collection out of a total of 252,109,637 total term occurrences. For a typical μ setting of 1500, this second term equals 0.26. For less frequent terms, this value is even smaller. Therefore, when $c(w;D) \geq 1$, $P(w|D)$ can be loosely approximated by:

$$P(w|D) = \frac{c(w;D)}{|D| + \mu}$$

Notice that this value does not depend on $c(w;D)$.

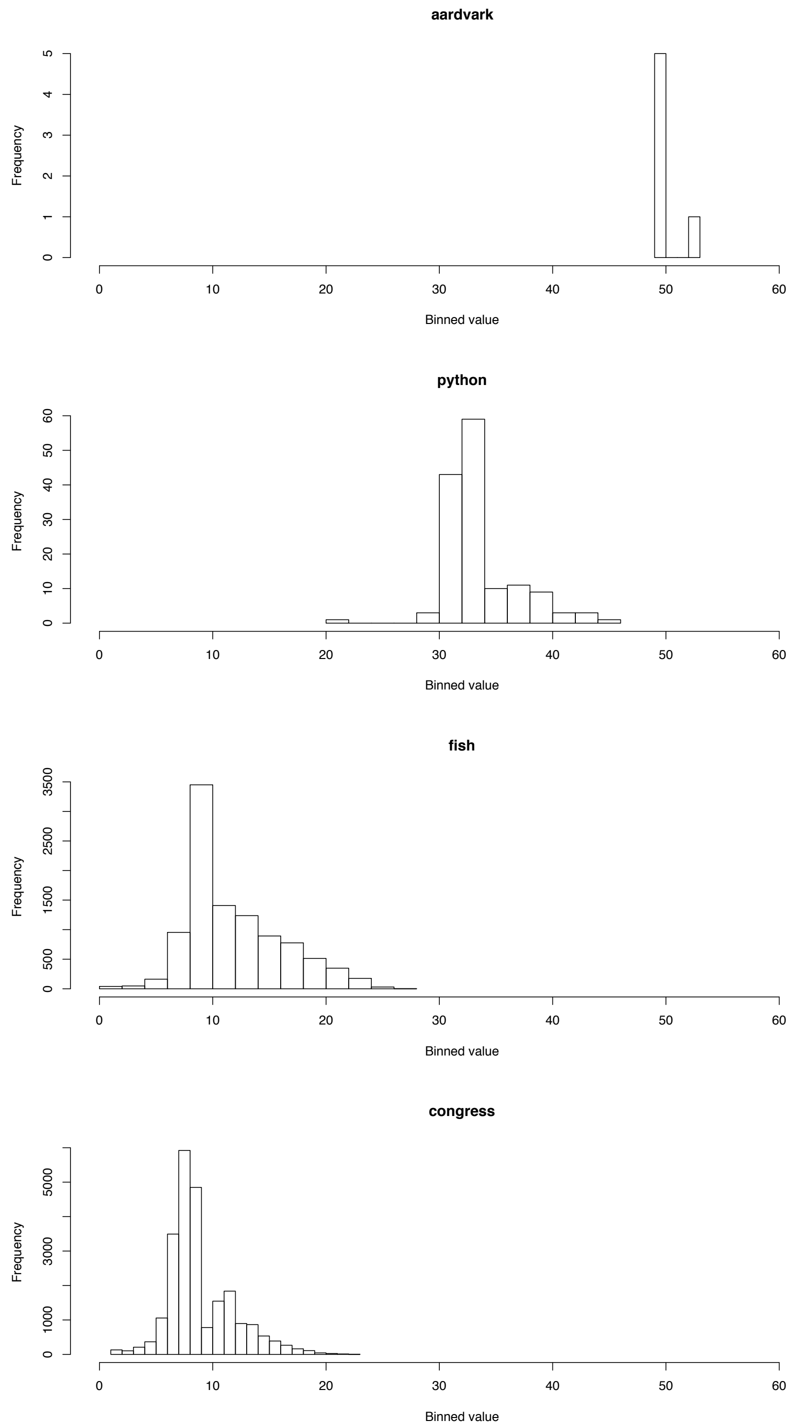


Figure 4.1. Distribution of binned values for four different terms. This data comes from the TREC45 collection, using 64 bins and saturation = 0.001.

For our purposes, this means that we expect the minimum value C_w to depend on the collection frequency $c(w;C)/|C|$, but the distribution of $P(w|D)$ values when $w \in D$ should follow $c(w;D)/|D|$ closely. Therefore, we expect the resulting $B_{(w,D)}$ distribution to have a similar shape across different terms, but we expect that C_w will cause this distribution to shift based on word popularity.

Figure 4.1 shows the kind of data generated from this procedure. The term "aardvark" occurs just 5 times in the TREC45 collection, and therefore instances of that term have very high bin values. In contrast, the word "congress" is very frequent, occurring over 23,590 times. Its occurrences have very low bin values. Most of the values lie between 7 and 9, with 26% (6,084) of the documents at bin value 8. As expected, the words share a common Gaussian-like look to the binned values, although the less common terms are shifted strongly to the right.

4.4 Evaluation

We evaluate this process using three TREC document collections. TREC12 is TREC disks 1 and 2, evaluated on TREC topics 51-150. TREC45 is TREC disks 4 and 5 minus the Congressional Record, evaluated on TREC topics 301-450 and 601-700. 10G is the TREC WT10G collection and topics 451-550. In all collections, only the title portion of each topic was used.

In all experiments, we use Dirichlet prior smoothing with $\mu = 1500$ and the Porter2 stemmer. All indexes were built with the Galago retrieval system. All saturation experiments use 32 integer bins, while all binning experiments use a saturation parameter $s = 0.001$. For all collections, 0.001 was greater than the smoothed probability estimate for any word in any document (Table 4.3). The baseline experiments are results from using traditional query likelihood ranking with no integer binning. For simplicity, we approximate C_w as $P(w|D)$ for a document of length 1500 that does not contain w.

Bins	10G	TREC45	TREC12
2	0.0492	0.1127	0.0808
4	0.1261	0.1432	0.0937
8	0.1833	0.2094	0.1778
16	0.1964	0.2393	0.2119
32	0.2027	0.2446	0.2206
64	0.2062	0.2475	0.2226
Base	0.2047	0.2469	0.2239

Table 4.1. Mean Average Precision over varying number of integer bins

Saturation	10G	TREC45	TREC12
1	0.1987	0.2434	0.2159
0.1	0.2006	0.2449	**0.2177**
0.01	0.1996	**0.2444**	**0.2176**
0.001	0.2027	**0.2446**	**0.2208**
0.0001	0.1982	**0.2470**	**0.2211**
0.00001	0.1970	**0.2467**	**0.2225**

Table 4.2. Mean Average Precision over varying saturation parameter values. Bold values are significant improvements over $s = 1.0$ (t-test, $p < 0.05$).

Saturation	TREC12	TREC45	10G
0.001	0.000	0.000	0.000
0.0005	0.001	0.001	0.002
0.0001	0.005	0.006	0.010
0.00005	0.007	0.009	0.015
0.00001	0.016	0.020	0.030

Table 4.3. Fraction of postings that are over the saturation probability for many saturation levels.

Size	Bins	Saturation	Increase
475M	1	0.001	0
475M	2	0.001	0
480M	4	0.001	5M
494M	8	0.001	14M
522M	16	0.001	28M
556M	32	0.001	34M
597M	64	0.001	41M
643M	128	0.001	46M
707M	256	0.001	64M
767M	512	0.001	60M

Table 4.4. WT10G index sizes for various numbers of bins

4.5 Results

Results for binning are shown in Table 4.1. Across the three collections, using 32 integer bins results in a drop of about 1% in search effectiveness, while using 16 bins results in a 5% drop. Note that in previous work, others have found peak effectiveness using just 8 bins. Typically in these systems, the weights stored in the inverted list is multiplied by an IDF-like constant during scoring. In this model, the weights from the inverted lists are added together without any additional weighting. This allows for arbitrary probabilities to be stored, including those that do not have an IDF component that is easy to factor out.

Results for saturation are shown in Table 4.2. In two of the three collections, changing the saturation parameter results in small but significant effectiveness gains. Table 4.3 shows that in all three collections, there exists no w and D such that $P(w|D) > 0.001$. The saturation parameter allows us to allocate the small number of bins we have more effectively by ignoring the probability space greater than s.

4.6 Summary

We have shown an algorithm for converting language modeling probabilities into small integer term weights with little loss in retrieval effectiveness.

Storing *ad hoc* weights in integers is not an important contribution, as there are already reasonable ways of doing that. This work is more important because it shows a generic way of translating language model probabilities into integers, including dealing with the problem of smoothing and background probabilities. This makes it possible to consider using more complicated query formulations than the *ad hoc* models we have seen so far.

We have now discussed two of the three pieces necessary to create the navigational indexes coming in Chapter 7. TupleFlow manages the data processing, and this work in binning probabilities shows how to translate query formulations into indexes. The final step, covered in the next two chapters, is to create the index structures and query processing algorithms to process queries efficiently.

CHAPTER 5

SCORE-SORTED INDEX OPTIMIZATION

This chapter is the first of two on efficient query processing strategies. This chapter describes a query processing strategy based on the same kind of score-sorted indexes previously described by Anh and Moffat. In order to compare directly with their work, the evaluation in this chapter stores their document-centric impacts in the inverted lists, although a novel query processing strategy is used to improve query throughput.

We provide the following research contributions in this chapter:

- We present a new method for continuous accumulator pruning in score-sorted indexes. Our method increases query throughput by 15% over the method proposed by Anh and Moffat [11] while still remaining *rank safe* results (i.e. documents are returned in the same order that they would be in an unoptimized evaluation)

- We show how our accumulator pruning technique can be combined with inverted list skipping to achieve a 69% total increase in throughput while maintaining the *rank safe* property.

- We provide a technique for optimizing the appropriate skipping distance to use during index based on simulation. Unlike all previous work we are aware of, we consider different skipping distances for different list lengths. We show that using list-length-dependent skip lengths can improve query throughput slightly.

- We add to results from Büttcher and Clarke [27] that indicate that storing inverted lists in memory can significantly improve performance. We show that the algorithm presented by Anh and Moffat [11] can evaluate queries 7 times faster on our system than the speed quoted in their paper.

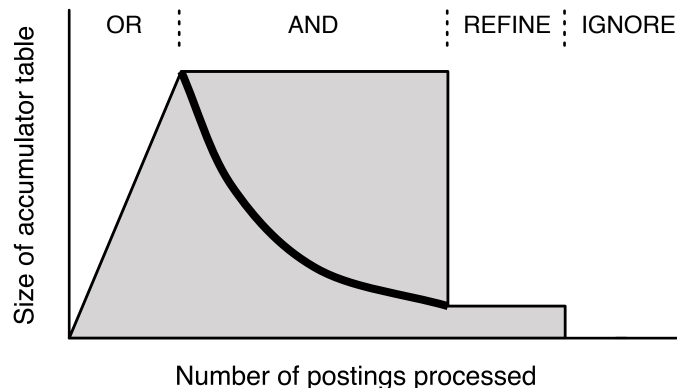


Figure 5.1. Relative number of accumulators used during the query evaluation process. The gray filled area represents the usage pattern in Anh and Moffat. The thick solid line represents the decreased accumulator usage of our approach.

5.1 Algorithm

Over the past six years, impact-sorted indexes have been shown to be an effective and efficient data structure for processing text queries [7, 10]. These indexes store term weights directly in the index, like the SMART system [21]. However, impact-sorted indexes use a very small number of distinct term weights; in this chapter we use just 8 different values. The small number of values used allows these indexes to store documents in impact order while still allowing for very high level of compression [7].

To generate effective retrieval results, care must be taken in selecting the impact values assigned to each term. Many different approaches are possible for this task. Anh and Moffat suggest a method for truncating BM25 values into integers, and also more recently have introduced a document-centric model. We presented another option in Chapter 4.

We use the Anh and Moffat document-centric impact model of for query evaluation [10]. The details of this model can be found in the references. For the purposes of this chapter, the important aspects of this model are that each term in a document receives a integer weight, based primarily on document statistics. This is roughly analogous to TF in a TF-IDF formulation. At query time, a query weight is com-

puted which is analogous to IDF. We call the document weight $w_{t,d}$ and the query weight $w_{t,q}$.

The retrieval score S_d for a document d, evaluated with a query Q is defined as:

$$S_d = \sum_{q \in Q} w_{t,q} w_{t,d} \tag{5.1}$$

The index contains an inverted list for each term t. In impact-sorted indexes, the list is separated into segments, one for each distinct value of $w_{t,d}$. Each segment contains a list of documents that share the same $w_{t,d}$ value.

The accumulator tuple we consider for this work is the same as used by Anh and Moffat. Each accumulator is a triple $\langle T, s, d \rangle$, composed of a term set T, a score s, and a document number d. The document number d and score s are traditional: the score s records the partial score for the document d based on query processing so far. The term set T records all terms that have been scored for this document so far. For our experiments in this chapter, we packed this triple into a 64-bit integer, using 8 bits for the term set, 24 bits for score, and 32 bits for document identifier.

In the code shown in this section, an accumulator for a particular document d is denoted A_d. In the system, each accumulator stores both a partial score and the set of terms that contributed to that partial score. Therefore, we overload the A_d symbol in this discussion so that it can be used as a score or as a set of terms. The particular meaning should be clear based on context.

Figure 5.1 describes the different phases of the Anh and Moffat algorithm we refer to in this section, as well as our new algorithm. The reduced accumulator usage of our algorithm is shown by the dark black line.

All of the query evaluation methods we consider in this chapter follow the pseudocode shown below, although with different implementations of ProcessSegment, TrimAccumulatorList, and CanQuit.

procedure PROCESSQUERY(Q)

$A \leftarrow \{\}$ ▷ The accumulator table

$S \leftarrow \{\}$ ▷ List of accumulator segments

for all query terms $t \in Q$ **do**

$I_t \leftarrow$ inverted list for term t

$w_{t,q} \leftarrow$ weight for term t

for all segments $I_{t,s} \in I_t$ **do**

add $\langle t, w_{t,q} \cdot s, I_{t,s} \rangle$ to S

end for

end for

sort S in descending order by segment score

for all segments $\langle t, w, I_{t,s} \rangle$ in S **do**

ProcessSegment(A, t, w, $I_{t,s}$)

TrimAccumulatorList(A, S)

if CanQuit(A, S) **then**

break

end if

end for

sort A by score, return top k results

end procedure

The simplest form of evaluation evaluates every posting in every segment. We define ProcessSegment below. TrimAccumulatorList is set to an empty function, and CanQuit is set to a function that returns false on all inputs. In unoptimized evaluation, the system is always in OR mode.

procedure ProcessSegment(A, q, w, $I_{q,s}$)

for all documents d in segment $I_{q,s}$ **do**

if $A_d \notin A$ **and** OrMode = **true then**

$A_d \leftarrow 0$

end if

if $A_d \in A$ **then**

$A_d \leftarrow A_d + w$

end if

end for

end procedure

5.1.1 AND Processing

If we know that all documents that could possibly be in the top k are already in A, we can update only accumulators that already exist. This is called AND mode, as opposed to the unoptimized OR mode. When in AND mode, ProcessSegment never adds new accumulators to the accumulator table. This is beneficial both because we do not incur the costs of building new accumulators, and because A stays small, leaving fewer accumulators to update and sort later.

The Anh and Moffat algorithm automatically detects when AND processing can be used safely by monitoring two quantities: a threshold value τ and a remainder function ρ. We describe these next.

The threshold value τ is a lower bound on the score of the last document that will be displayed to the user. Here, we assume that k represents the number of documents requested. We can compute a reasonable τ value by using the kth largest score in the accumulator table A. Since the score in any given accumulator is monotonically increasing, τ represents a lower bound on the score of the kth retrieved document at the end of an unoptimized retrieval. If k accumulators do not exist yet, $\tau = 0$.

For example, suppose at some point during the retrieval process, the kth largest score found in any accumulator was 50. Therefore, $\tau = 50$. Since none of the scores in the accumulator table can go down, we know that every document in the k returned to the user will have a score of at least 50.

The second monitored quantity is the score remainder function, ρ. This function computes an upper bound on the total additional amount of score that an accumulator could possibly gain through further processing of the inverted lists.

As an example, suppose we know the accumulator for some document d currently contains a score of 15, and that we are processing a four term query: t_1 t_2 t_3 t_4. The accumulator also records that we have already seen postings for document 15 in the inverted lists for t_1 and t_3. Any remaining score for document d must come from the inverted lists for t_2 and t_4. If we know that all remaining postings for t_2 have scores less than 5, and all remaining postings for t_4 have scores less than 6, we know that this accumulator can only increase by 11. Therefore, $\rho(\{t_2, t_4\}) = 11$.

We define the function ρ as follows:

- $\rho(\{t_i\})$ = the largest score remaining in the inverted list for term t_i
- $\rho(T) = \sum_{t \in T} \rho(\{t\})$

Remember that Q is the set all terms in the query. When $\tau > \rho(Q)$, we know that no more accumulators need to be created. This is true because no new accumulator will ever achieve a score greater than $\rho(Q)$, but a score of τ is necessary to enter the ranked list shown to the user.

With τ and ρ defined, we can now define a pruned processing algorithm:

procedure ProcessSegmentPruned(A, q, w, $I_{q,s}$)

  OrMode $\leftarrow \tau < \rho(Q)$

  ProcessSegment(A, q, w, $I_{q,s}$)

end procedure

5.1.2 Trimming Accumulators

Computing τ and ρ allows us to stop adding accumulators to A, but they can also help us identify accumulators that can be safely removed from the table.

As query evaluation continues, τ grows and ρ falls. If there comes a time when an accumulator A_d contains a score low enough that no additional postings could cause its value to rise above τ, we know it can never enter the final ranked list. Therefore, we can prune it from consideration.

```
procedure TrimAccumulatorList(A, S)
    for all accumulators A_d in A do
        if A_d + ρ(A_d) < τ then
            remove A_d from A
        end if
    end for
end procedure
```

The Anh and Moffat algorithm trims the accumulator table just once; only when the top k results have been determined. Instead, our algorithm trims accumulators constantly, after each inverted list segment is processed. This difference is highlighted in Figure 5.1 by the dark black line.

Having a smaller accumulator list improves speed because there are fewer accumulators to update. However, when the inverted list segments become much longer than the table of accumulators, inverted list skipping becomes possible. Since the inverted list is stored in document order, if we also store the accumulator table in document order we can identify when large sections of the inverted list are not worth decoding. By using skipping information, discussed in more detail later, we can skip over those regions without decompressing them.

5.1.3 Ignoring Postings

At some later point during query processing, it may be possible to determine the final order of the top k results without continuing to process postings. For this to happen, two conditions must be satisfied:

- The top k documents must be fixed (that is, the identity of the top k documents is not in question)
- No additional postings may be able to change the ordering of the top k documents.

We can check the first condition by checking all existing accumulators. All accumulators A_d containing scores less than τ must satisfy the condition $A_d + \rho(A_d) < \tau$. Essentially this means that if the accumulator is not in the top k now, it never will be, no matter how many additional postings we process.

We check the second condition by reviewing all accumulators with values of at least τ. If the accumulator of the document currently in rank i could surpass the document in rank $i - 1$ after processing additional postings, we cannot stop processing.

These checks lead to the following algorithm, which is used in both our algorithm and Anh and Moffat:

procedure CanQuit(A, S)
  for all accumulators A_d in A **do**
    if $A_d < \tau$ **and** $A_d + \rho(A_d) \geq \tau$ **then return** False
    end if
  end for
  $A' \leftarrow$ all accumulators A_i in A such that $A_i + \rho(A_i) > \tau$
  sort A' in ascending order by score
  for all accumulators A_i in A **do**
    if $A_i = A_{i+1}$ **and** $\rho(A'_i) > 0$ **then return** False
    else if $\rho(A_i) > A_{i+1} - A_i$ **then return** False
    end if
  end for
end procedure

5.2 Implementation

Our system follows the impact-ordered index design proposed by Anh and Moffat [7]. Inverted lists are stored in order of impact value. The bin value is encoded first, followed by a length value. After that, the document numbers that share the same impact value are delta encoded. This method produces indexes that are size-competitive with other space-efficient methods, typically around 7GB for the GOV2 collection.

The inverted file is compressed using standard variable byte encoding [105]. The vocabulary is compressed using 15-of-16 encoding, plus additional prefix encoding. The inverted file is segmented into blocks of approximately 32K in size, although no term spans multiple blocks. Any term with more than 32K of inverted list data gets its own block. An abbreviated vocabulary table is used to look up the appropriate block for a given term. A block contains a sub-vocabulary that can be efficiently searched to find the appropriate inverted list. This blocking technique efficiently packs infrequent terms together, so that the abbreviated vocabulary table can be very small. In our experiments, the vocabulary table required just 2MB of space. A similar technique has been used previously by Büettcher and Clarke [27].

We take advantage of the 64-bit address space of new commodity machines to memory map the inverted list file into the virtual address space of the process. This allows multiple retrieval processes to run simultaneously while sharing the same memory pages.

5.2.1 Indexing

All indexes in this chapter were created with TupleFlow. The first stage of the TupleFlow job processes text documents, converting them into compressed lists of (document, word, count) tuples. The next phase combines these tuples in order to determine the inverse document frequency for each term. In parallel, another process

combines the list of document names into a single table. The binning stage follows, where the IDF table is combined with the parsed word count tuples to generate binned term weights (in this case, document centric impacts). The final stage merges the binned lists together.

Our indexing system was written in Java, and is not optimized for speed. Indexing the compressed GOV2 collection requires 60 hours of CPU time, with 42 hours devoted to parsing, 16 for binning (impact generation), and the remaining 2 hours for merging. However, the parsing and binning stages are massively parallel. In practice, we can build a a GOV2 index in 4 hours using roughly 20 processors (this quantity varies, as the grid of processors is shared for other research tasks). The skipping information in the inverted lists is written in the final merge stage, so the skipping parameter can be changed with only two hours of additional work once an index has been built.

The relatively long indexing time required by our system should not be a reflection of the optimized indexing time for this task. Previous work in this area indicate that impact-sorted GOV2 indexes can be built in under 7 hours on a typical desktop computer using optimized implementations [11]. While our indexing process would take some additional time to build inverted list skipping information, we do not expect this would affect indexing time significantly.

5.3 Choosing Skip Lengths

Section 5.1.2 introduced the accumulator trimming process. Just trimming accumulators is enough to improve retrieval speed, but performance improves much more when trimming is used in conjunction with list skipping. When adding skip information to an index, it makes sense to ask how long the skip distance should be. Moffat and Zobel [74] suggest a method for determining this parameter. We use a slightly different formulation in this chapter which remains in the same spirit. We suppose that an inverted list segment is b bytes long and k entries exist in the accumulator

table. We also suppose that there are b_1 skip pointers, each of which can be encoded in 4 bytes, and that each skip pointer skips d_b bytes. Therefore, the expected total number of bytes processed is:

$$4b_1 + \frac{kd_b}{2}$$

The second term estimates the number of bytes decompressed in the inverted list. In reality, we will never decompress a byte of the inverted list more than once, so this has a natural upper bound of b:

$$4b_1 + \min(\frac{kd_b}{2}, b)$$

If the number of accumulators k is longer than the length of the inverted list segment, the system can quickly acknowledge this situation and ignore the skip information. This gives us the final revised time estimate:

$$T(b, k, d_b) = \begin{cases} b & \text{if } k > b \\ 4b_1 + b & \text{if } k \leq b \text{ and} \frac{kd_b}{2} > b \\ 4b_1 + \frac{kd_b}{2} & \text{otherwise} \end{cases} \tag{5.2}$$

We can estimate a reasonable value of d_b by using data collected from the system. We ran the 50,000 TREC 2005 Efficiency Track queries. For each inverted list segment the system processed while in AND mode, the system recorded the length of the segment b and number of accumulators k that the system had stored in the accumulator table at that time. This provided us with over 500,000 data points to use in simulation.

Note that b and k have an important relationship; if b is large, that indicates that the inverted list is perhaps also large. If so, it is likely to be processed at the very end of the query, when the accumulator table is almost empty. If b is smaller, we expect

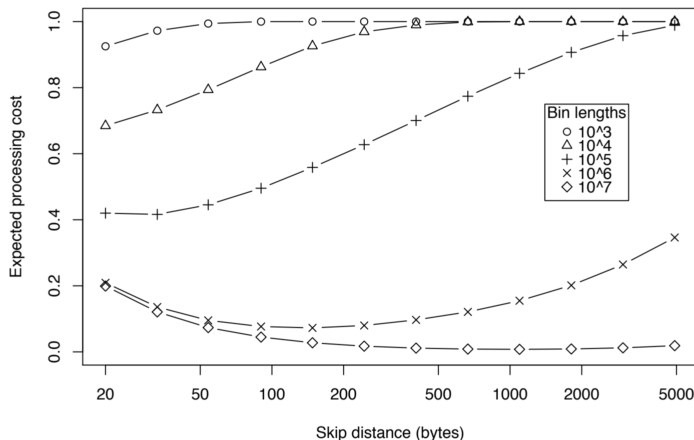


Figure 5.2. The expected cost of processing bins of varying lengths with varying skip sizes. The expected cost is expressed as a fraction of the cost of processing the data without skipping. Expectations are based on the 50,000 TREC 2005 Efficiency Track query set.

larger values of k. This suggests that perhaps different values of d_b should be used for different inverted list segment lengths.

Figure 5.2 shows this effect. For small bin lengths (about 1000 bytes long), skipping data helps very little, even at very small skip lengths. For moderate lengths (10^4 to 10^5 bytes long), very short skip distances give the largest expected performance increase. For long lengths (over 10^6 bytes long), short skip distances help, but longer skip distances are even better.

The relative frequency of encountering list segments of these different lengths is not shown in the graph. Analysis of our logs indicate that 95% of all inverted list bytes processed in AND mode are in segments larger than 100K. This leads us to consider skip lengths between 50 and 200 bytes.

The analysis in this section rests on the approximation that all byte accesses cost a similar amount. Of course, this is not the case. Reading a single byte from an address that is not currently in cache is expensive, but the process of fetching that

byte will cause L1 and L2 cache lines to fill. After a single miss, nearby byte accesses are inexpensive.

Previous work [105] indicates that skip information should be interleaved with inverted list data, but note that this leads to inefficient cache usage; reading a single skip entry of 3 bytes results in reading an additional 29 bytes of unwanted inverted list data. By storing skip information densely and separately from inverted list data, we avoid this problem.

5.4 Evaluation

We used the TREC GOV2 collection along with the TREC Terabyte Ad Hoc and Efficiency topics for evaluation. We processed the GOV2 collection using the Porter2 English stemmer,[1] and used a common list of 600 stopwords.[2]

In order to ensure that our system was producing reasonable results, we used the TREC Terabyte Track ad hoc queries to evaluate its effectiveness. The system returned 1000 results for each query. We computed mean average precision and precision at 20 figures for each set of ad hoc queries. The results here are close to, although slightly below, efficient systems participating in TREC. We suspect that these results will improve as we improve our document parser. Since all optimizations considered in this chapter are rank safe, these numbers represent the effectiveness of the system in all optimization modes, and any improvement in baseline effectiveness would be reflected in the optimized modes.

We used the same computer for all retrieval experiments. Our test machine is worth approximately US$3000 in 2006. It contains two dual-core Intel Xeon 5050 “Dempsey” processors, for a total of four compute cores. Each core runs at 3GHz,

---

[1]http://snowball.tartarus.org/algorithms/english/stemmer.html

[2]http://goanna.cs.rmit.edu.au/~jz/resources/stopping.zip

Query set	MAP	P@20
Topics 701-750 (2004)	0.2460	0.4949
Topics 751-800 (2005)	0.3004	0.5290
Topics 801-850 (2006)	0.2592	0.4590

Table 5.1. Effectiveness on TREC Ad Hoc Queries

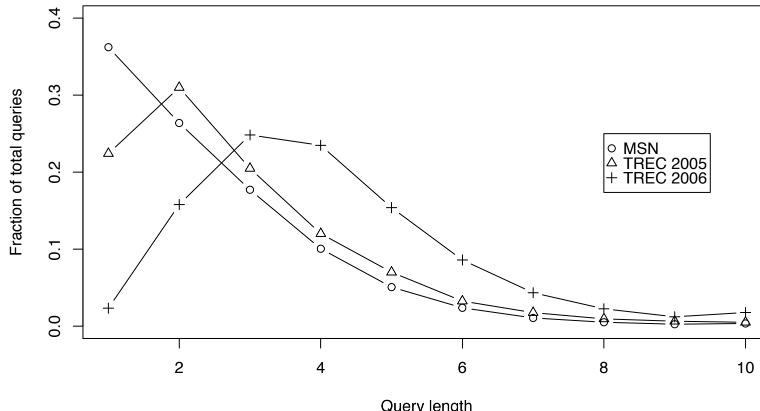


Figure 5.3. Distribution of query lengths (before removing stopwords) across collections.

and has a dedicated 2MB L2 cache. The cores share 8GB of RAM over a 667MHz bus. All experiments were done in 64-bit mode, in order to permit the inverted file to fit in the virtual address space.

Before timed runs, we ran a simple tool that memory mapped the inverted file and read it from start to finish, in order to ensure the inverted file was completely loaded into memory.

The results of our optimization experiments are shown in Tables 5.2 and 5.3. We compiled a separate version of the retrieval system for each optimization method in order to measure the highest possible speed. The mean query time for each method (except the unoptimized method) was measured by running each query set three times in immediate succession, then computing the mean query time for the batch. In all cases, total variation between runs of the same method was less than 2%. Because

Method	All queries		Query length (terms)									
	Mean	Throughput	1	2	3	4	5	6	7	8	9	10
Query count	47,543	47,543	10,899	17,347	10,888	5,489	1,965	683	233	32	6	1
Unoptimized	317.2	3.2	22	144	410	724	1149	1535	1972	3499	3197	1181
Anh/Moffat	19.0	57.8	0.2	5.5	21	44	79	119	178	366	376	89
Trimming	15.0	66.7	0.1	5.6	18	37	67	105	166	397	362	44
Trimming+Skips	10.3	97.5	0.2	2.9	10	24	49	84	145	360	342	35

Table 5.2. TREC 2005 Efficiency Queries, average query execution times, in milliseconds. Throughput is measured in queries per second.

Method	All queries		Query length (terms)									
	Mean	Throughput	1	2	3	4	5	6	7	8	9	10+
Query count	99,650	99,650	3,926	23,430	33,122	23,488	10,283	3,515	1,151	407	167	161
Unoptimized	1006	1.0	50.4	212.8	679	1364	2128	2855	3788	4751	5000	9644
Anh/Moffat	60.0	16.7	0.4	5.7	29	73	137	218	319	440	558	3014
Trimming	49.1	20.4	0.4	6.2	24	58	110	180	268	395	586	3180
Trimming+Skips	39.0	25.7	0.4	3.5	14	40	86	150	232	355	544	3122

Table 5.3. TREC 2006 Efficiency Queries, average query execution times, in milliseconds. Throughput is measured in queries per second.

Method	1 core	2 cores	4 cores
Anh/Moffat	57.8	108.3 (1.87x)	181.6 (3.14x)
Trimming	66.7	121.1 (1.81x)	178.2 (2.67x)
Trimming+Skips	97.5	161.2 (1.65x)	228.8 (2.35x)

Table 5.4. Speedup when using multiple cores. Throughput is measured in queries per second, while speedup is measured relative to each algorithm's performance on a single processor.

of the wide gulf in processing time between unoptimized and optimized methods, the unoptimized version was run just once, with logging turned on.

In the TREC 2005 log, we found 2457 queries that did not match any documents in the collection. In the TREC 2006 log, we found 350 queries like this. Although the full query log was processed in each test run, we averaged query performance only over the count of queries with at least some results returned.

To measure the average query speed for each query length, we compiled an executable for each method with logging turned on. The logging messages add approximately 10% to total runtime, and measure many aspects of query processing. Each log message contains a timestamp, which we used to measure the speed of each individual query; the mean times shown here are a result of aggregating that data. Because of the overhead of logging, the timings for various query lengths should be considered somewhat less reliable than the overall average speed figures.

In all of the tables except those about skipping lengths, the index was built with a fixed skip length of 128 bytes.

To measure performance on multiple processor cores, we ran multiple processes of the retrieval system simultaneously on the TREC 2005 query set. To avoid possible interaction between processes, we shuffled the query order so that each process executed the 50,000 queries in a different order. We measured the elapsed time between starting the first query process until the slowest query process completed. The number of queries executed was divided by total elapsed time to produce a throughput

	Throughput		
Skip Length	Mean	Low	High
64	98.94	98.80	99.20
96	94.73	93.99	95.45
128	97.95	97.60	98.18
256	97.05	96.68	97.36
Model	99.21	98.44	99.83
Model (Large)	99.14	99.05	99.27
Model (Small)	98.69	98.40	98.96

Table 5.5. Efficiency at varying skip lengths, TREC 2005 Efficiency Queries

number. We did not compute a mean query time in this case, since such a number would be misleading: adding multiple processors does not decrease individual query latency, but it does improve overall throughput.

5.4.1 Analysis

Figure 5.3 compares the distribution of query lengths in three different query logs. Note that the lengths reported here are measured as the number of whitespace breaks in the query plus 1; it therefore counts stopwords and words that do not appear in the collection. The query lengths reported in other tables refer to the number of inverted lists successfully fetched from the index. The TREC 2005 and TREC 2006 Efficiency queries are used for performance analysis, while the MSN query log data is provided for comparison purposes. [3] The TREC 2005 queries are a reasonable match for the actual distribution of query lengths on the web.

The gulf between all of the optimizations tested here and a full evaluation is striking, especially in the case of particularly short queries. Notice how all optimizations here complete single term queries in under a millisecond. With no disk seek overhead, the system can jump immediately to the necessary inverted list. After twenty results

[3] http://research.microsoft.com/ur/us/fundingopps/RFPs/Search_2006_RFP_Awards.aspx

are read, query processing halts. The unoptimized system is forced to read the entire inverted list and create an accumulator for every document in it, which takes much more time. The percentage difference in query time falls as the queries grow longer, but the Anh and Moffat approach completes queries in half the time of the unoptimized case in all cases, while our method completes them in less than a quarter of the unoptimized time.

Our experiments with different skip lengths show surprisingly similar performance. We ran each query set three times in succession, with the mean throughput shown, as well as the lowest and highest throughput recorded. The differences between many of these settings are well within the range of variability of our tests. The small differences here are reasonable given the results in Figure 5.2; note that the performance predicted for the largest inverted list segments (the most costly ones) are remarkably flat across differing skip lengths. However, we note that using the results suggested by our model (varying skip lengths based on inverted list segment length) results in a slight increase in performance. The large model came from manually slightly increasing the skip lengths suggested by the model, while the small model came from manually slightly reducing the skip lengths suggested by the model. These results seem to indicate that the predicted best setting matches the real best setting, but again, variability in measurement does not allow us to say this with confidence.

5.4.1.1 Multiple Cores

Our multiple process experiments show the limitations of shared memory bandwidth. We used a machine that is known to be bandwidth starved, and this shows in this experiment. Using four processors simultaneously, the Anh and Moffat algorithm is able to process 3.14 times as many queries as when just one processor is used. This speedup is much higher than the 2.35 speedup of our best algorithm.

When four cores are used, the Anh and Moffat algorithm equals the performance of the Trimming version of our algorithm.

We leave a detailed explanation of these numbers for future work, but it is clear that linear scalability is not assured with multiple processors, even when no disks are involved. However, notice that the Anh and Moffat algorithm and the Trimming algorithm access approximately the same amount of memory during evaluation, while the Trimming+Skips algorithm accesses less memory. We believe that the reduced memory access allows Trimming+Skips to maintain its edge over the other algorithms, even when memory bandwidth is scarce.

5.5 Related Work

Impact-sorted indexes are described in a series of papers by Anh and Moffat [7, 10, 11]. These indexes are a natural next step following the frequency-sorted indexes of Persin et al. [79]. The frequency-sorted indexes suggested by Persin et al. stored term counts instead of discretized document scores, and so these indexes still required query-time length normalization. Additionally, this meant that inverted list entries were not necessarily in score order. However, sorting lists by frequency allowed for a compact and compressible index representation that was amenable to early termination, and formed a basis for the impact-sorted work.

Roughly at the same time as the first impact-sorted work, Fagin et al. proposed a class of algorithms known as threshold algorithms [45]. These algorithms, like the ones shown in this chapter, provide a method for efficiently computing aggregate functions over multiple sorted lists by maintaining statistics about the data that remains to be read. This work also considers the added possibility of random access to elements in a list, which is somewhat similar to the skipping process we propose here. From an information retrieval perspective, this work can be seen as a combination of the max score work of Turtle and Flood [100] combined with the frequency and impact

sorted work of Persin et al. and Anh and Moffat [7, 79]. Both Brown and Strohman et al. considered supplemental lists of top scoring documents during query evaluation which can also be considered part of this tradition. [19, 94].

It is possible to have some of the benefits of impact-sorted lists while still sorting by document. Lester et al. [58] show how the memory footprint of accumulators can be significantly reduced without loss of effectiveness. Their algorithm scans inverted lists in document order, but processes only postings with term counts larger than some threshold. As in this work, smaller accumulator sets lead to faster query processing.

In our work, we process less index data by organizing the index for easy skipping and query termination. Another way to process less data is to store less data in the index. Static pruning methods remove information from the index that is unlikely to affect query effectiveness. Carmel et al. considered this process [32]. More recently, Büttcher and Clarke considered index pruning specifically so that the resulting index would fit in memory, although supplemental disk indexes are sometimes used for additional information. Query performance improves in part because of memory speed and in part because of the smaller amount of data, although these different factors were not analyzed in detail. More recent results from the TREC 2006 Terabyte Track shows that other researchers have considered static pruning [29].

While the actual process of storing precomputed scores in lists is not the subject of this chapter, there are many examples in the literature of researchers doing this. Cleverdon remarks how, in early experiments, human indexers were asked to choose integer term weights for documents for later use in automatic retrieval [35]. The SMART retrieval system, at least in the 1980s and beyond, used floating point weights stored directly in inverted lists for fast and flexible document scoring possibilities at query time [21]. More recently, Anh and Moffat have proposed two separate schemes for generating term weights; one involving assigning ranges of BM25 scores to integer values, and another using a document-centric approach. [7, 10] In more recent work,

researchers have attempted to create a unified theoretical framework for generating these weights [71].

Bast et al. extend the threshold algorithm ideas of Fagin et al. with an enhanced disk IO cost model [16]. Their system contains a score-ordered index as well as a document-ordered index (or, more accurately, an index of values accessible by document identifier). The system processes information in score order until the cost model indicates that it is more efficient to refine remaining accumulators by random access to score information using the document-keyed index. By contrast, our algorithm always accesses data in score order, but additional skip information allows us to jump rapidly to required information when the number of active accumulators drops, without requiring a separate index copy. Like us, the authors use a machine with 8GB of RAM to test their system, but data duplication causes their indexes to be too large to fit in system RAM.

While memory-optimized processing is a relatively new field for information retrieval research, it is well studied in the database community. The MonetDB system is the traditional example of a main-memory database system [111]. Along with the more recent C-Store system [90], these database systems store relational data by column instead of by row, which significantly increases the speed of certain classes of data warehousing database transactions. If these database systems are any indication, information retrieval data systems will continue to change as they are optimized for main memory access.

5.6 Summary

We have presented a study of efficient query processing techniques using score-sorted indexes. Our best technique improves query throughput by 69% over a strong baseline. In the process of developing this method, we have shown how log data can

be used to determine optimal skip lengths in inverted lists based on an estimation of total bytes read.

The combination of fast random access in memory and skip information in inverted lists allows us to achieve performance similar to previous heavily statically pruned systems, but without the potential loss in effectiveness [27]. Essentially the skipping information allows us to prune the index dynamically at query time, resulting in similarly efficient retrieval performance.

Although this chapter evaluated trimming and skipping based on an *ad hoc* query formulation, the results from Chapter 4 allow us to consider evaluating more complicated models, which will come in Chapter 7. The next chapter, Chapter 6, introduces another kind of index with its own efficient query evaluation technique.

CHAPTER 6

DOCUMENT-SORTED INDEX OPTIMIZATION

6.1 Introduction

In Chapter 5, we presented an efficient method for evaluating queries on score-sorted indexes. These score-sorted indexes are compelling for short queries, but have some disadvantages for longer queries. In the worst case, suppose we process a query like "cheese trees," which contains two very common terms that are not likely to occur in the same documents. The top scoring documents for "cheese" and "trees" are likely to be very different, which could cause a score-sorted algorithm to use a very large accumulator table. In the worst case, a score-sorted algorithm could create an accumulator for every document in the collection, which is a substantial memory cost.

By contrast, document-at-a-time retrieval on document-sorted indexes requires very little memory, and has some benefits when two lists have little in common. As we will consider in more detail in a moment, document-at-a-time retrieval computes full document scores in one step, which eliminates the need for an accumulator table to store partial results. This allows a document-at-a-time system to contain only the top k results seen so far. In addition, since these are full document scores and not partial scores, there may be some queries where document-at-a-time retrieval is able to compute a threshold τ more quickly than retrieval on a score-sorted index.

In this chapter, we investigate retrieval with document-ordered indexes using the binned probabilities we developed in Chapter 4. In the process, we develop a new

variant of Turtle and Flood's max-score algorithm, called score skipping, which can increase the throughput of short queries by as much as a factor of 5.

6.2 Algorithm

6.2.1 Traditional

We will use a canonical decomposition for all document-at-a-time retrieval algorithms discussed here, which looks like this:

```
procedure RETRIEVAL(Q, k)
    R ← new PriorityQueue
    L ← InvertedLists(Q)
    D ← undefined
    loop
        D ← FindNextDocument(D, L)
        if D is undefined then
            break
        end if
        s ← ScoreDocument(D, L)
        Push(R, D, s)
        if |R| > k then Pop(R)
        end if
        MovePastDocument(D, L)
    end loop
    return R
end procedure
```

For a basic, non-optimized implementation, we use these versions of the functions. Here we assume that we can retrieve document-ordered inverted lists from the index

using the function InvertedLists, and that functions CurrentDocument, CurrentScore, and NextDocument exist, and operate on a single iterator.

The FindNextDocument function determines the next document to score. This implementation considers the value of CurrentDocument for each list, and returns the lowest value, or returns undefined if CurrentDocument is undefined for each list (this is the end of list condition).

procedure FINDNEXTDOCUMENT(L)
  $D \leftarrow$ undefined
  for all lists L_i in L **do**
    if CurrentDocument(L_i) is undefined **then**
      continue
    end if
    if D is undefined **or** CurrentDocument(L_i) $< D$ **then**
      $D \leftarrow$ CurrentDocument(L_i)
    end if
  end for
  return D
end procedure

The ScoreDocument function computes a score for a document D. For each list L_i, this implementation considers whether L_i is currently pointing to D, and if so, adds its score contribution to S.

procedure SCOREDOCUMENT(D, L)
  $S \leftarrow 0$
  for all lists L_i in L **do**
    if CurrentDocument(L_i) $= D$ **then**
      $S \leftarrow S+$ CurrentScore(L_i)
    end if

end for

return S

end procedure

Finally, the MovePastDocument procedure scans considers each inverted list, and moves any list currently pointing to D to the next document in the list.

procedure MovePastDocument(D, L)

for all lists L_i in L **do**

if CurrentDocument(L_i) $= D$ **then**

NextDocument(L_i)

end if

end for

end procedure

This algorithm as shown will score every document that occurs in any of the lists in the set L. The result queue R, at the end of the process, contains only the top k scoring documents. At no time during the process does R contain more than $k + 1$ documents.

6.2.2 Max-Score

As we saw in the score-sorted index discussion, scoring every document in the lists is a very slow way to determine the top k documents for a query. Most documents in our set of lists L will be poor matches. We would like to quickly focus on just the best possible matches for our query, and ignore the rest of the documents. The max-score algorithm, proposed by Turtle and Flood [100], is a very effective way of doing this.

In the max-score algorithm, we introduce a threshold score, τ, which is a lower bound on the score of a document that could appear in the final ranked list when evaluation is done. Remember that R contains a set of document scores that we have

seen so far during evaluation. We set $\tau = 0$ when $|R| < k$. Once $|R| \geq k$, we set τ to the lowest score in R.

For max-score to work, we require a bound β_i on each list L_i. The value β_i represents the largest possible score we will see in list L_i.

The key insight in max-score is that when $\beta_i < \tau$, no document that contains only term t_i will be in the final ranked list. Remember that the scores from list L_i (which represents term t_i), are bounded above by β_i. Therefore, any document that contains only term t_i can have a score no larger than β_i. Since τ is a lower bound on the top-k scores in the ranked list, we know that no document that contains only t_i will be in the top-k matching documents. We can extend this to sets of multiple terms, too. Suppose we have a set of terms $Q' \subset Q$ such that:

$$\tau > \sum_{t_i \in Q'} \beta_i$$

Then, any document in the top k results must contain one term in Q that is not in Q'.

To use this insight, we sort the set of lists L in decreasing order by β_i, so that the highest scoring terms are first. We define a bound Θ_i as follows:

$$\Theta_i = \sum_{j \geq i} \beta_j$$

Intuitively, Θ_i is the largest possible score that a document can achieve when it contains only terms from the set $Q_i = \{t_j | j \geq i\}$.

We then define two new procedures, FindNextDocument and ScoreDocument. We start with a new FindNextDocument procedure:

```
1: procedure FindNextDocument(L)
2:     D ← undefined
3:     for all i < |L| do
```

```
4:         if τ > Θ_i then break
5:         end if
6:         if CurrentDocument(L_i) is undefined then
7:             continue
8:         end if
9:         if D is undefined or CurrentDocument(L_i) < D then
10:            D ← CurrentDocument(L_i)
11:        end if
12:    end for
13:    return D
14: end procedure
```

This is the same as the previous implementation, except for the if block at line 3. By adding this block, we ignore all documents D that contain only terms in Q_i. As we have seen, the condition $\tau > \Theta_i$ allows us to safely ignore those documents.

We also define a new MovePastDocument procedure:

```
procedure MovePastDocument(D, L)
    for all i < |L| do
        if τ > Θ_i then break
        end if
        if CurrentDocument(L_i) = D then
            NextDocument(L_i)
        end if
    end for
end procedure
```

Just as before, we have added an if block at line 2. With this change, we only move forward in lists that might actually affect how FindNextDocument operates.

We also define a new ScoreDocument procedure:

```
1:  procedure ScoreDocument(D, L)
2:      S ← 0
3:      for all i < |L| do
4:          if τ > S + Θ_i then
5:              break
6:          end if
7:          MoveToDocument(D, L_i)
8:          if CurrentDocument(L_i) = D then
9:              S ← S+ CurrentScore(L_i)
10:         end if
11:     end for
12:     return S
13: end procedure
```

The major change to this procedure comes at line 3. The if condition is $\tau > S+\Theta_i$. Notice that at this point, S is the partial score for this document based only on lists $\{L_j | j < i\}$, and Θ_i is the upper bound of the score contribution from the remaining lists. Therefore, if $\tau > S+\Theta_i$, we can stop scoring this document and move to another one.

Notice that we have added a call to the MoveToDocument procedure at line 6. We have do to this because MovePastDocument no longer moves every list forward.

The key to the efficiency of max-score is in the MoveToDocument function. As with our skipping optimizations in score-sorted indexes, max-score allows us to skip over large segments of some inverted lists. By adding skip pointers into the inverted lists, we can efficiently skip forward in the list to a particular document without decompressing many intermediate postings. In our experiments, the max-score optimization improves query throughput by a factor of 10.

6.2.3 Score Skipping

The efficiency of max-score comes from the ability to skip forward in an inverted list to a particular document number. In our indexes, we have added the ability to skip to the next document with a particular score. The resulting performance optimization is called score skipping, and it produces throughput gains of 40 to 80% over max-score alone. For two-term queries, score skipping improves query throughput by a factor of 4.

With score skipping indexes, we get some additional functions that act on inverted lists:

- CurrentBound – Returns an upper bound on the score of the current document.
- MoveSkipsToDocument – Advances the skip pointer list (and therefore the current bound) to the given document.
- MoveToBound – Moves the list iterator to the next posting that has a score at least as large as the score bound parameter.
- MoveToDocumentBound – Moves the list iterator to the document only if it might have a score at least as large as the parameter.

We keep MovePastDocument unchanged, but we change ScoreDocument and FindNextDocument. We start with FindNextDocument:

1: **procedure** FindNextDocument(L)
2: $\quad D \leftarrow$ undefined
3: $\quad$ **if** $\tau > \Theta_1$ **then**
4: $\quad\quad D \leftarrow$ MoveToBound($\tau - \Theta_1$)
5: $\quad$ **else if**
6: $\quad\quad$ **for all** **dothen**$i < |L|$
7: $\quad\quad\quad$ **if** $\tau > \Theta_i$ **then break**

```
8:         end if
9:         if CurrentDocument(L_i) is undefined then
10:            continue
11:        end if
12:        if D is undefined or CurrentDocument(L_i) < D then
13:            D ← CurrentDocument(L_i)
14:        end if
15:     end for
16:   end if
17:   return D
18: end procedure
```

This function is the same as the max-score version, except for the addition of an if block at line 2. The if condition here is $\tau > \Theta_1$. Recall that Θ_1 is an upper bound on the maximum score contribution of all lists $\{L_i | i \geq 1\}$. This means that the first list, L_0, must contribute at least $\tau - \Theta_1$ to the final score for a document to enter the top k query results. Therefore, we can safely skip forward in the first list to a document that has a term score of at least $\tau - \Theta_1$.

We also change the ScoreDocument function slightly:

```
1: procedure ScoreDocument(D, L)
2:   S ← 0
3:   for all i < |L| do
4:     if τ > S + Θ_i then
5:       break
6:     end if
7:     MoveSkipsToDocument(D, L_i)
8:     if τ > S + Θ_{i+1} + CurrentBound(L_i) then
9:       break
```

```
10:         end if
11:         MoveToDocument(D, L_i)
12:         if CurrentDocument(L_i) = D then
13:             S ← S+ CurrentScore(L_i)
14:         end if
15:     end for
16:     return S
17: end procedure
```

At line 6, we added a all to MoveSkipsToDocument, which sets up the call to CurrentBound on the next line. The added if statement at 7 mirrors the one at 3, but we use the tighter score bound given by the CurrentBound function instead of the whole list bound β_i.

6.3 Index Construction

6.3.1 Inverted lists

The inverted lists used in our experiments begin with a short header which includes an upper bound b on all weight values in the list. Skip information is stored next. Short lists, defined as those shorter than 2,048 bytes, contain no skip information. Longer lists contain a skip pointer for approximately each 128 bytes in the inverted list. Each skip pointer (d, o, b) consists of a document number d, a list byte offset o, and a binned weight bound b. All values are v-byte compressed, and the document d and offset o are also delta-encoded. Typically a skip pointer is about 5 bytes in size, with 2 bytes for the compressed d, 2 byte for the compressed o, and 1 byte for the compressed b. The exact meaning of the skip pointer values will be discussed later in this section.

Inverted list postings are 2-tuples, (d, b), containing a document d and a binned weight b. Both values are compressed, but the d value is also delta-encoded. For

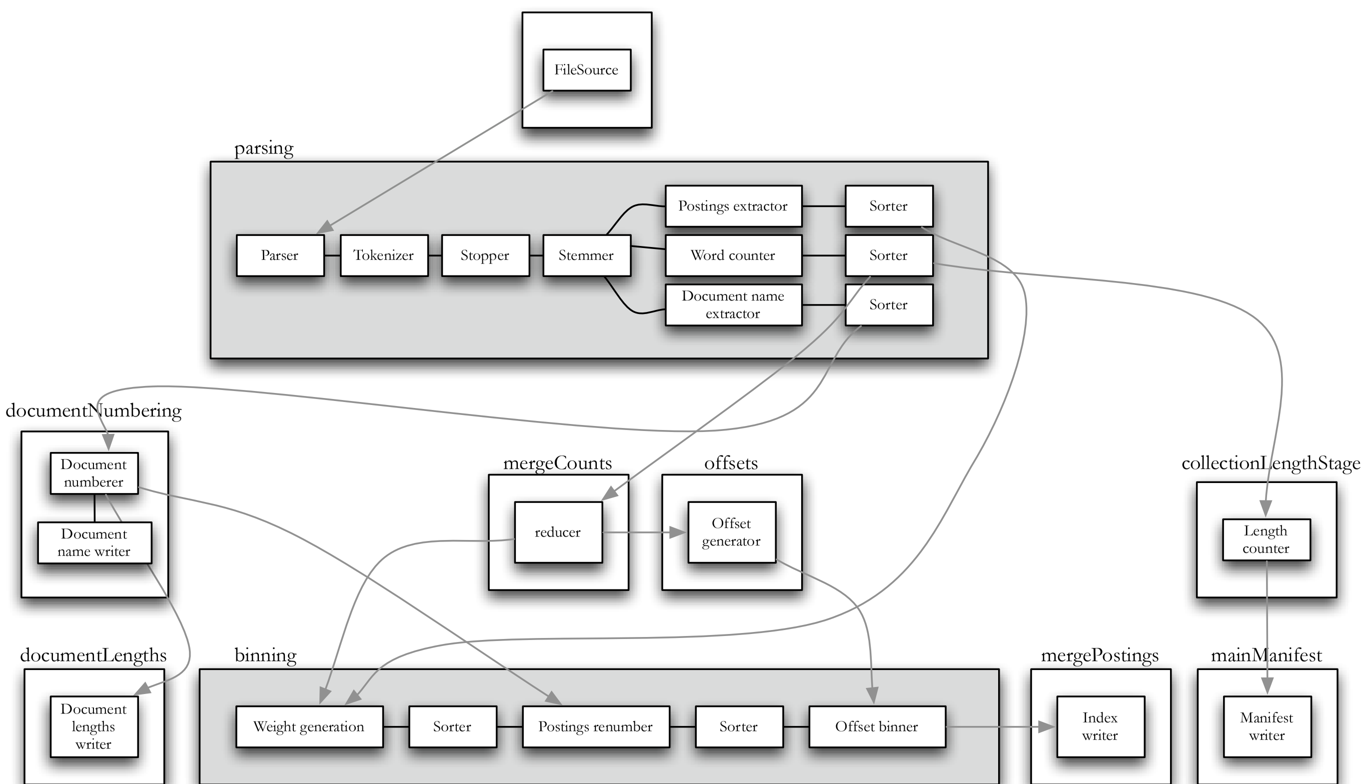


Figure 6.1. A picture of the TupleFlow execution graph that builds document-sorted indexes.

reasonably long lists, postings are approximately 3 bytes each, with 2 bytes used for d and 1 byte used for b.

For two consecutive skip pointers, (d_0, o_0, b_0) and (d_1, o_1, b_1), the second skip pointer asserts that:

1. The posting for document d_1 immediately precedes offset o_1 in the inverted list.

2. All postings between o_0 and o_1 have document numbers d_i such that $d_0 < d_i \leq d_1$.

3. All postings between o_0 and o_1 have binned weights b_i such that $b_i \leq b_1$.

The first property is necessary to allow skipping, because the postings are delta-encoded. The d value stored in the skip pointer is the key to delta-decoding starting at byte offset o. The second property allows efficient forward search for a posting with a particular document number, and the third property allows efficient forward search for a posting with a particular weight.

Note that the skip pointers are stored contiguously instead of scattered throughout the postings. As discussed in Chapter 5, this is necessary for good cache performance. Storing skip pointers contiguously allows the processor to efficiently prefetch skip pointers, reduces cache pollution, and reduces the number of cache misses necessary to skip through the list.

6.3.2 Other structures

The other structures in the index are identical to those used in the score-sorted indexes of Chapter 5. Inverted lists are stored in lexicographic order. Smaller lists are stored in blocks of approximately 32K each, while each big list (defined as longer than 32K) gets its own block. Each block contains a compressed vocabulary look up table, and the strings are delta-encoded in blocks of 16. There is also a vocabulary file stored separately. The vocabulary file contains one word per block, making it

about one-thousandth the size of the inverted file. Its size allows it to be loaded into memory quickly. This structure is meant to be efficient for disk operations: assuming the vocabulary file is loaded in RAM, the start of any inverted list can be found with one seek and less than 32K of sequential reading.

The document names file is also meant to be loaded into RAM, and is specifically optimized for looking up TREC document identifiers. Any kind of document name can be stored in the structure, but TREC document identifiers compress particularly well, allowing all 25 million of the GOV2 names to fit easily in RAM.

6.3.3 Construction with TupleFlow

Figure 6.1 shows the TupleFlow construction process for these indexes. Files enter the system from the filenames stage, and are passed to replicated parsing stage instances. Each parsing stage extracts word postings, word counts, and document names. Document names are passed to the numbering stage, which assigns an integer to each document name. Word counts are passed to the offset generator, which determines the minimum offset C_w for each term (as discussed in Chapter 4). Offset data, count data, and postings are passed to replicated binning stages, which combine all of this information into binned integer postings. These postings are then merged into a single inverted file.

6.4 Evaluation

To show the usefulness of this query evaluation strategy, we evaluated both the effectiveness and efficiency of our system. Since the contributions in this work are not about query effectiveness, the point of the effectiveness test is to prove that the results coming out of our system are reasonable. We spend additional time on the efficiency experiments to show the nature of the efficiency improvement.

Queries	Binned P@20	Binned BPREF	Binned MAP	MAP Real	MAP Indri
701-750	0.4969	0.3373	0.2660	0.2593	0.2870
751-800	0.5382	0.3649	0.3137	0.3029	0.3432
801-850	0.4610	0.3485	0.2899	0.2917	0.3071

Table 6.1. Effectiveness results from the GOV2 collection.

6.4.1 Effectiveness

Our effectiveness results are shown in Table 6.1. The three binned columns are results from the system presented in this chapter. Mean average precision values from two other systems, described below, are shown for comparison.

Some query optimization methods for text search are *unsafe*, in that they provide no guarantees of correctness. By contrast, our system is *score-safe*, which provides two guarantees: first, that the scores of each document retrieved are correct, and second, that no document with a score greater than τ is missed. This means that the effectiveness results shown here apply to all of the query evaluation methods mentioned in this chapter.

However, these guarantees only apply based on the weights stored in the index, which are integer approximations of real-valued weights. Therefore, we compare against a traditional query-likelihood evaluation using full real-valued computation, and those results are shown in the Real MAP column. To perform this evaluation, we used the same parsing and stemming pipeline used to create binned indexes, but instead stored the data into a traditional document-ordered index with word positions. We used an unoptimized document-at-a-time retrieval to evaluate the queries, using Dirichlet smoothing with $\mu = 1500$. The binned results are better for two of the three *ad hoc* query sets.

In addition, the effectiveness of a system can change based on how documents are parsed and which stopwords are removed. This helps account for the difference in the

		Query length (terms)									
Algorithm	Throughput	1	2	3	4	5	6	7	8	9	10+
Counts	47,625	10,682	17,264	11,004	5,606	2,059	721	241	39	7	1
Unoptitmized	5.2	16.2	111.4	249.0	405.8	594.7	783.5	931.3	1445.1	1388.7	727.1
Turtle/Flood max-score	46.7	3.1	22.4	23.4	31.1	45.4	61.4	88.0	146.8	173.0	11.9
Score skipping	86.6	1.3	4.4	14.5	27.1	39.6	54.9	82.9	140.2	167.5	10.7

Table 6.2. TREC 2005 Efficiency Queries, average query execution times, in milliseconds. Throughput is measured in queries per second.

		Query length (terms)									
Algorithm	Throughput	1	2	3	4	5	6	7	8	9	10+
Counts	99,946	3,291	22,536	33,170	24,146	10,883	3,788	1281	459	192	199
Unoptimized	2.0	33.2	145.1	367.0	649.7	937.9	1188.8	1471.1	1688.0	1892.1	2559.0
Turtle/Flood max-score	19.6	6.7	31.0	41.1	53.9	80.6	112.5	151.2	188.2	232.4	389.2
Score skipping	27.9	2.8	6.4	23.9	45.9	70.5	102.4	141.3	179.2	223.5	364.2

Table 6.3. TREC 2006 Efficiency Queries, average query execution times, in milliseconds. Throughput is measured in queries per second.

effectiveness of our system versus the Indri results shown here. The Indri results are supplied by Metzler et al. [70] and were official submissions to TREC.

6.4.2 Efficiency

We evaluated the efficiency of our technique using the 50,000 query TREC 2005 Efficiency query set, and the 100,000 query TREC 2006 Efficiency query set. Both of these query sets were used in Chapter 5 to evaluate our score-sorted index techniques.

We used a computer with two Intel Xeon 5355 microprocessors, each with 4 CPU cores, for a total of 8 CPU cores in the machine. Each core has a clock speed of 2.66GHz. Only one core was used for query processing experiments, and the rest were left idle. The cores share 16GB of memory over a 1333MHz bus.

Our results are shown in Tables 6.2 and 6.3. The table layouts are similar to those in Chapter 5.

Overall, we see a 85% overall improvement in throughput on the 2005 query set, and a 42% improvement on the 2006 query set. As in Chapter 5, the longer queries in the 2006 set not only run slower, but also are improved less than the 2005 query set. Notice, however, that max-score helps the longer 2006 queries somewhat more than the 2005 queries (an improvement of 9.8x versus 8.9x).

The benefit of our optimization is most pronounced on two-term queries, where it improves throughput by a factor of 5 over max-score. To understand this result, consider how two-term evaluation works in max-score. At the beginning of query evaluation, every document in both inverted lists is evaluated. After a short time, evaluation switches to where only documents that contain the less frequent term are considered. At this point, there is nothing more max-score can do except evaluate every document in the shorter inverted list. However, our method is able to skip through the shorter list, evaluating only documents that might enter the ranked list.

Score skipping is helpful for single term queries as well, but the cost of scoring a document in a single-term query is very low, so the benefit is not as great as with two-term queries. For three-term queries, we see a smaller benefit, since it is less likely that we will be able to ignore two of the three query terms. Interestingly, for longer queries, this method reduces overall query time by about 6 milliseconds (2005) or 10 milliseconds (2006).

Because the bulk of user queries are less than 5 words long, our efficiency gains on these shorter queries make a big impact on final throughput. Like our score-sorted optimizations, this technique is not as useful for longer queries.

6.5 Related Work

All modern retrieval systems use inverted lists to evaluate queries efficiently [105]. The differences between the optimizations discussed here lie in how these inverted lists are processed. In term-at-a-time systems, the inverted list for each term is considered separately. These systems use a table of score accumulators to keep track of partial scores for documents that have been seen. At the end of evaluation, the accumulators are sorted, and the top k documents (where k is a parameter given by the user) are returned to the user. In document-at-a-time systems, the inverted lists are considered simultaneously, much like the classical merge sort algorithm [55]. In this case, there are no partial scores; this allows the system to maintain a list of the top k scores it has seen so far.

One of the earliest papers on modern query optimization comes from Buckley [22]. In this approach, terms are evaluated one at a time, from the least frequent term to the most frequent term. At some point during query evaluation, it may be possible to show that it is not necessary to consider any more terms, as the document at rank $k+1$ cannot surpass the score of the document at rank k.

This approach, and approaches based on it, have the advantage that some inverted lists will not need to be read from disk. As the gap between disk access speed and processing speed continues to increase, this is an attractive feature. However, this approach has trouble with duplicate (or nearly duplicate) documents in the collection. If two documents at ranks k and $k+1$ will evaluate to the same score, the Buckley algorithm will read the inverted lists for all the terms in the query.

For large collections, the likelihood that two documents will evaluate to the same score is greatly increased, even among documents that are not exactly the same. For example, many legal documents follow specific templates, with only minor changes to the template text. It is likely that many documents generated with the same template will be the same length, and will therefore have the same score for many queries. In order to guard against this effect, the exactness conditions for ranking order must be relaxed. Buckley suggests a method for doing this.

Moffat and Zobel [74] evaluate two heuristics, *Quit* and *Continue*, which reduce the time necessary to evaluate term-at-a-time queries. The *Quit* heuristic dynamically adds accumulators while query processing continues, until the number of accumulators meets some fixed threshold. At this point, documents are ranked by the partial scores in the accumulators and returned to the user. The *Continue* strategy is similar, in that it uses only a fixed number of accumulators. However, when the accumulator threshold is reached, it continues query evaluation, but only considers those documents that already have accumulators allocated. The *Continue* method was found to be particularly effective; at times it was more effective than the baseline system.

Lester et al. [58] remark that the *continue* strategy is generally quite effective, but can have difficulties in web-scale collections. To see why this is so, suppose that the *continue* strategy is used with a fixed number of accumulators k. If a query appears where the term q_1 occurs $k/2$ times, but term q_2 appears $10k$ times, what should the system do? Based on the traditional implementation of the *continue* strategy,

the system adds accumulators for all $k/2$ documents that contain q_1, and all $10k$ documents that contain q_2, since after evaluation of q_1, the limit of k accumulators has not been reached. This overshoots the fixed accumulator budget and slows retrieval significantly. Instead, evaluation could switch from *full* to *continue* mode part of the way through evaluating q_2, thus efficiently processing the query and maintaining the proper accumulator budget, but this causes a strong bias for early documents in the collection, which may hurt retrieval effectiveness. Instead, the authors propose that only documents with many occurrences of q_2 should be allowed to have accumulators. As evaluation continues and the number of accumulators in use grows closer to k, higher frequencies of term q_2 are required to gain an accumulator. This biases retrieval effort toward the highest scoring documents, which is what we prefer. The authors find that this adaptive strategy results in efficient query evaluation with accumulator counts as low as 0.4% of the collection size.

Brown [19] presents a method for efficiently finding a small list of candidate documents for scoring. These candidate documents are considered to be the most likely set of documents to appear in the top k results. This short list of candidates can be scored quickly by skipping through inverted lists. To create the candidates list for a query, the search engine takes the union of term-specific candidates lists created at index time. For a given term t, its candidate list contains the top documents in the ranked list for the query t. Brown finds excellent speedups for this approach.

Broder, et al. [18], consider query evaluation in a two stage process. First, the query is run as a Boolean *and* query, where a document is only scored if all terms appear. If this process finds at least k documents, the process stops. However, if the number of documents found is less than k, a second query is issued that considers all documents that contain any query term.

As we discussed in some detail in the algorithms section, Turtle and Flood [100], consider a series of query optimization techniques, including some exact methods and

some approximate methods. In the document-at-a-time max-score method, the top candidate document for each term is stored in the index, like the approach taken by Brown, except only one document is stored. This is somewhat like the *Continue* approach of Moffat and Zobel, except the process is dynamic and guarantees correct results.

The term-at-a-time max-score method is similar to the method presented by Buckley. However, Turtle and Flood explore a rank-safe optimization, where terms are evaluated until the top k documents are guaranteed to be in the correct order.

We previously proposed another document-at-a-time evaluation strategy using top documents lists [94]. In this strategy, 1% of all postings are stored at the top of the inverted list. The documents in these postings are always scored. A bound on the other 99% of list postings is computed in the index. Essentially this strategy uses max-score on 99% of the postings, and does unoptimized retrieval on the top 1% of postings. The result is that the bounds on the lower 99% of postings can be much tighter, causing more efficient skipping, and better overall query throughput. To compare with our current work, this previous work splits each list into two pieces and computes bounds on each of them. Our current work instead breaks each list into many pieces. The result is a system that is faster, more flexible and less dependent on parameter settings.

6.6 Conclusion

We have presented a new efficient query processing strategy for document-sorted indexes, producing a up to an 85% throughput improvement for the kinds of short queries encountered on the web. Two-term queries receive the largest improvement of all, with overall throughput improving by a factor of 5.

While this chapter is primarily about fast document-sorted indexes, we have also shown that the binned language models from Chapter 4 can be used as a basis for an efficient retrieval system.

The speed boost we achieve with document-sorted indexes is encouraging because of the advantages of document-sorted indexes: there is no unbounded accumulator table to worry about, and most traditional indexes are document-sorted. One interesting avenue for research would be to see if our binned document-sorted indexes can be combined efficiently with a traditional document-sorted index that contains word position information. Such an index combination could combine the flexibility of a traditional index system with the efficiency of the technique presented here.

CHAPTER 7

NAVIGATIONAL SEARCH WITH COMPLEX FEATURES

7.1 Introduction

At this point in the dissertation we have presented two types of indexes, score-sorted and document-sorted, and have shown novel query optimization strategies for each. We showed how to perform binning on language model probabilities while maintaining retrieval accuracy. We also described TupleFlow, our framework for distributed computation.

In this chapter, we combine the work of all the previous chapters and apply them to the task of navigational web search. Our previous effectiveness evaluation has been on *ad hoc* query tasks, meaning the kind of recall-focused tasks that a paralegal or intelligence analyst might perform. By contrast, web users often use search engines as a tool for finding specific pages. This requires a different, more complex kind of query formulation than we have seen in previous chapters.

In Chapter 2, we explored some of the differences between navigational search and *ad hoc* search. In particular, the most successful navigational search systems use complex feature-based query formulations for the highest possible accuracy. Some of the more useful features for navigational search include title text, heading text, anchor text, document priors (like PageRank), and phrase text. Each one of these features provides useful evidence about the relevance of a web page to a navigational query. While combining all of this evidence is necessary for high effectiveness, traditional means for feature combination require large computational efforts at query time.

In this chapter, we move the work of feature combination and mixing into the indexer. TupleFlow makes it possible to adapt the indexing system to extract more features from each document, and to do so in a scalable way. The probability binning work gives us a method for combining that feature data and storing it in integers. The previous two chapters, on score-sorted and document-sorted indexes, give us an efficient platform for query evaluation on these combined binned feature lists.

Our goal in this chapter is not to create a new, highly effective ranking strategy. Instead, we want to show how a strategy that has previously been shown to be effective can be adapted to work on our system. To do this, we adapt the named page formulation used by UMass the TREC 2005 Efficiency Track for our system [72]. Our effectiveness results are usually comparable to the UMass Indri results. However, by using our efficient query processing strategies, we reduce query times from a minute each down to a small fraction of a second.

Adapting a new ranking strategy also gives us a chance to re-evaluate our query processing speeds. In the process, we found some very surprising results: our score-sorted indexes performed quite poorly under the new data, while our document-sorted indexes got faster. Later in this chapter we explore the reasons why this happened.

7.2 Model

7.2.1 Conversion

Table 7.1 shows the parameter settings developed by Metzler, and used by UMass at TREC 2005 for named page finding. For the remainder of this chapter, we will call this the UMass formulation. This table requires some extra explanation, because the meaning of the weight parameters is affected by Indri's weight normalization. If more explanation of the Indri system is needed, see the discussion in Chapter 2.

A sample query using the UMass formulation is shown in Figure 7.1. Notice how there are three sections to the query, starting with a priors section, then a section of

Feature	Mixture Weight	Dirichlet μ
Single Terms	0.8	-
Body	3	250
Title	1	10
Heading	1	40
Anchor	1	100
Phrases	0.1	-
Body Phrase	3	1000
Title Phrase	1	5
Heading Phrase	1	80
Priors	0.1	-
Inlinks	0.6	-
PageRank	0.4	-

Table 7.1. Parameter settings used in the UMass TREC 2005 experiments.

```
#combine(
0.1 #weight( 0.4 #prior(pagerank) 0.6 #prior(inlinks) )
1.0 #weight( 0.8 #combine( #wsum( 1.0 david.(anchor)
                                  1.0 david.(title)
                                  3.0 david.(mainbody)
                                  1.0 david.(heading) )
                           #wsum( 1.0 fisher.(anchor)
                                  1.0 fisher.(title)
                                  3.0 fisher.(mainbody)
                                  1.0 fisher.(heading) )
                           #wsum( 1.0 lemur.(anchor)
                                  1.0 lemur.(title)
                                  3.0 lemur.(mainbody)
                                  1.0 lemur.(heading) )
             0.1 #combine( #wsum( 1.0 #1( david fisher ).(anchor)
                                  1.0 #1( david fisher ).(title)
                                  3.0 #1( david fisher ).(mainbody)
                                  1.0 #1( david fisher ).(heading) )
                           #wsum( 1.0 #1( fisher lemur ).(anchor)
                                  1.0 #1( fisher lemur ).(title)
                                  3.0 #1( fisher lemur ).(mainbody)
                                  1.0 #1( fisher lemur ).(heading) )
) )
```

Figure 7.1. A sample query formed using the UMass TREC 2005 Named Page formulation.

single terms, then a section of phrases. The outer combination gives a weight of 0.1 to the priors and 1.0 to the text portion. Inside the text portion, 0.8 of the weight goes to body text, and 0.1 goes to two-term phrases. Within each `#wsum`, a weight of 3 is assigned to body text, and a weight of 1 is assigned to occurrences in particular fields.

Within each weighted operator, Indri normalizes weights so that they sum to 1. For example, the weights for body text, title, heading and inlink become 0.5, 0.16, 0.16, and 0.16 (instead of 3, 1, 1, and 1). If only one weight operator is used, this does not affect how the documents are ranked. However, in a nested query like the one we see here, the normalization affects the relative weight of the different operators. Also note that the `#combine` operator works the same way, so that each term receives a weight of $1/n$, where n is the number of terms in the operator.

Galago has no such normalization, so we approximate it in our converted weights, shown in Figure 7.2. Notice that Body, Title, Heading and Anchor are in the same 3:1:1:1 proportion as in the UMass model, but the weights are lower. The Body Phrase, Title Phrase and Heading Phrase components are also in the same 3:1:1 proportion as the UMass model, but the actual values are lower, to reflect the lower relative weight of the phrase component when mixed with the single terms. Finally, we give PageRank a weight of 0.1. In the original formulation, it gets a weight of 0.4 within an operator that has an unnormalized weight of 0.1. However, since our formulation does not use an inlink count prior, we boost PageRank somewhat to account for its loss.

The weighting of phrases versus single terms is especially tricky because of the natural normalization done by the #combine operator. The number of two-term phrases used in the query is $n - 1$, which means the proportion of phrase terms to text terms ($\frac{n-1}{n}$) changes as the query gets longer. The #combine operator accounts

Feature	Mixture Weight	Dirichlet μ	Length
Body	0.75	250	500
Title	0.25	10	20
Heading	0.25	40	80
Anchor	0.25	100	200
Body Phrase	0.15	1000	500
Title Phrase	0.05	5	20
Heading Phrase	0.05	80	80
PageRank	0.1	-	-

Table 7.2. Parameter settings for features in our experiments.

for this somewhat by normalizing the weights given to each phrase term. Our Galago system has no such query time normalization.

A final issue is how to set the length parameter that our binning process requires. Recall from Chapter 4 that we need to use a fixed document length to compute a background probability for documents that do not contain a particular term. For ad hoc search we typically use a length that is equal to μ. However, since the μ values in this section are so low, we found that many of our probabilities "bottomed out" in the binning process, resulting in many postings binning to a bin value of 1. By using a slightly larger length value, we were able to get acceptable fidelity for low probabilities.

Since Metzler et al. used PageRank values that were trained specifically for their query formulation, we used the same PageRank values used in their experiments instead of calculating our own at index time.

This conversion process is certainly not clean. In a production system we would have re-trained parameters for this kind of test instead of using this kind of error-prone conversion method. However, part of our goal for this system is to show how models built elsewhere can be adapted to work on this more efficient machinery, so we wanted to test using direct conversion. Because of the error in this process, however, this is certainly a lower bound on the effectiveness that could be achieved.

7.2.2 Feature Combinations

We used five different ranking functions for each query log:

- **Text** – Uses just single terms from the body text of each document.
- **Structure** – Uses title and heading text models mixed with the body text model.
- **+ Anchor Text** – Uses an additional anchor text model mixed with the title, heading and text models.
- **+ Phrases** – Mixes in body, title and heading phrase models with the previous models.
- **+ PageRank** – Mixes in a PageRank prior with the previous models.

Single term lists were constructed from, at most, the Body, Title, Heading, Anchor and PageRank features. Phrase lists were constructed from Body Phrase, Title Phrase, Heading Phrase and PageRank features. These features were mixed together using a weighted mixture (known as *#wsum* in Indri). Note that this is also a divergence from the original model, which uses PageRank as a separate feature. Here we mixed PageRank into each list in order to cause authoritative documents to drift to the top of the score-sorted index inverted lists. We hoped that this might improve query speeds.

All indexes were built with the recommended binning setting from Chapter 4: 64 bin values and saturation = 0.001.

7.2.3 Index Construction

Figure 7.2 shows a simplified diagram of the indexing process in TupleFlow. Filenames are generated from a filenames stage, then passed to many instances of a

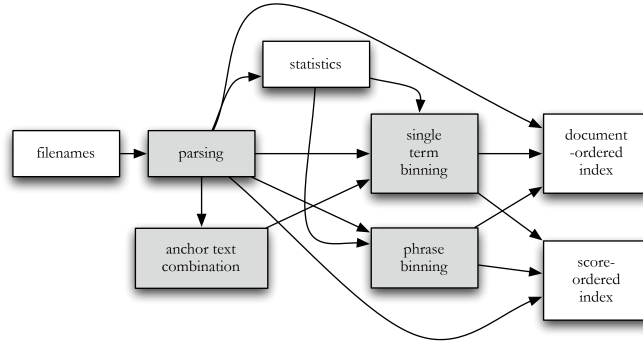


Figure 7.2. Simplified diagram of the TupleFlow job for constructing navigational indexes.

parsing stage. The parsing stage extracts all the necessary postings from each document, including single terms and phrases from a variety of fields. It also extracts links separately and sends them to an anchor text stage, which associates text from each link with its destination page. All of these postings are then sent to binning stages; one for phrases and one for single terms. All of this data is then transferred to index writing stages, which generates a document-ordered and a bin-ordered index from the data. We built both indexes simultaneously from the same input data to ensure that the contents of each index pair was identical.

All stages shown in gray were distributed to multiple processors. We built five separate feature combinations, for a total of ten indexes. The indexes were built simultaneously using a cluster of 70 processors, of which approximately 60 were free for computation. Four feature combinations completed indexing in 14 hours. The fifth combination halted because of a file system failure on the cluster. Because of TupleFlow's built-in checkpointing, this job was restarted at the failure point and completed normally.

In order to stay within the memory limits of our test machine when using phrases, we restricted our indexes to only the words and phrases used in our query logs. For the indexes that did not incorporate any phrases, this reduced index size by about 30% (11GB to 7.7GB for document-ordered indexes). For indexes with phrases, the

Run Type	NDCG@15	MRR	Success@10	Not Found
Text	0.5694	0.5203	0.7133	0.0267
Structure	0.7019	0.6661	0.8267	0.0133
+ Anchor Text	0.6999	0.6629	0.8400	0.0133
+ Phrases	0.6995	0.6626	0.8333	0.0133
+ PageRank	0.6935	0.6492	0.8133	0.0133

Table 7.3. GOV Collection, 2002 Navigational Queries

Run Type	NDCG@15	MRR	Success@10	Not Found
Text	0.4104	0.3579	0.5633	0.0867
Structure	0.5533	0.4989	0.7067	0.0367
+ Anchor Text	0.6776	0.6391	0.8300	0.0233
+ Phrases	0.6869	0.6481	0.8367	0.0233
+ PageRank	0.7144	0.6735	0.8633	0.0200

Table 7.4. GOV Collection, 2003 Navigational Queries

space savings was over 50%. The indexes with phrases were about 20% larger than without (5.9GB vs. 5GB for score-sorted, 9.2GB vs. 7.7GB for document-ordered). Note that there are many more two-term phrases than single terms, but that the average two-term phrase occurs much less often than the average term. This helps explain the small overhead of phrases in our vocabulary-pruned indexes.

7.3 Evaluation

To test the usefulness of our work, we need to test both effectiveness and efficiency. Our goal was to come close to the effectiveness of top TREC systems while showing large efficiency gains.

7.3.1 Effectiveness

Tables 7.3 and 7.4 show our results on the GOV dataset, a 20GB partial crawl of the .GOV domain. Our results for 2003 (Table 7.4), show exactly the kind of behavior we had hoped to see: as each new feature is added, effectiveness rises. Our results for 2002 do not show the same kind of behavior, although there is a strong boost in

Run Type	NDCG@15	MRR	Success@10	Not Found
Ad Hoc	0.3108	0.2826	0.4365	0.2262
Text	0.3486	0.3132	0.4683	0.2460
Structure	0.4638	0.4513	0.5952	0.1706
+ Anchor Text	0.4524	0.4414	0.5635	0.1825
+ Phrases	0.4528	0.4418	0.5556	0.1825
+ PageRank	0.4289	0.4200	0.5516	0.1706

Table 7.5. GOV2 Collection, 2005 Named Page Queries

Run Type	NDCG@15	MRR	Success@10	Not Found
Ad Hoc	0.2706	0.2362	0.3978	0.2210
Text	0.2961	0.2570	0.4420	0.2265
Structure	0.4314	0.4153	0.5912	0.1989
+ Anchor Text	0.4248	0.4234	0.5580	0.2099
+ Phrases	0.4223	0.4190	0.5580	0.2099
+ PageRank	0.4365	0.4227	0.5912	0.1823

Table 7.6. GOV2 Collection, 2006 Named Page Queries

effectiveness as we add in title and heading postings to the index. Based on the mean reciprocal rank (MRR) metric, our best 2002 run would have placed 11th among the 73 submitted runs [36]. Our 2003 result fares better, placing 6th of 70 submitted runs [37]. We also calculate other metrics here, like NDCG@15, Success@10, and Not Found, although they generally follow the trends in the MRR column. Since Indri did not exist until 2004, we cannot compare directly to Indri TREC runs.

Tables 7.5 and 7.6 follow the 2002 results, with strong improvements as title and heading text are added, but inconsistent results with extra features. In addition to all of our traditional models, we added in the *ad hoc* index used for our experiments in Chapter 6. This index uses the same postings as the Text index, but uses $\mu = 1500$ instead of $\mu = 250$. The extra smoothing results in a reasonable boost in effectiveness when used alone. However, the UMass parameter setting was trained in concert, so that $\mu = 250$ was found to be an optimal parameter when mixed with the other features.

Our best 2005 MRR result of 0.4513 actually slightly beats the Indri result of 0.441 at TREC 2005. Our TREC 2006 result of 0.4234 is, however, quite a bit worse the Indri result of 0.512. Even comparing against the Indri no-phrase result of 0.4980, our results are much worse. This result is somewhat perplexing, seeing as the Indri results are generated from the same query formulation, and it seemed to work well for us with the TREC 2005 queries.

In 2005 and 2006, the TREC Proceedings list just the best run from each institution instead of a list of all runs submitted. Our best 2005 result would have placed 2nd among the 8 participating institutions [34]. Our best 2006 run would have placed 5th among 11 participating institutions [29]. Again, we find our 2006 results to be less interesting than the 2005 results, but in both cases we are performing above the median.

Our overall results here are strong; we have successfully made use of additional feature information in the lists, and have produced results that are close to the best academic results.

7.3.2 Efficiency

We tested the efficiency of our system using the TREC 2005 and 2006 Efficiency query sets, which we used previously in both Chapters 5 and 6.

Our text computer contained two Intel Xeon 5355 microprocessors, each with 4 CPU cores, for a total of 8 CPU cores. Each core has a clock speed of 2.66GHz. The cores share 16GB of memory over a 1333MHz bus. This is the same machine used for our document-ordered index experiments in Chapter 6. For each test, we ensured that the entire inverted list was loaded memory before beginning the timing process. Although the machine has 8 CPU cores, only one was used for query processing while the other 7 were left idle.

Our query throughput figures are shown in Tables 7.7 and 7.8.

List Order	Run Type	Elapsed	Throughput
Score	Text	962	49.5
Score	Structure	3458	13.8
Score	+ Anchor Text	3357	14.2
Score	+ Phrases	3386	14.1
Score	+ PageRank	3153	15.1
Document	Text	804	59.3
Document	Structure	374	127.5
Document	+ Anchor Text	313	152.2
Document	+ Phrases	341	139.8
Document	+ PageRank	356	133.8

Table 7.7. Throughput results for 2005 TREC Efficiency queries.

List Order	Run Type	Elapsed	Throughput
Score	Text	5932	16.8
Score	Structure	21557	4.6
Score	+ Anchor Text	22924	4.4
Score	+ Phrases	22895	4.4
Score	+ PageRank	20981	4.8
Document	Text	5219	19.2
Document	Structure	2000	50.0
Document	+ Anchor Text	1623	61.6
Document	+ Phrases	1813	55.1
Document	+ PageRank	1974	50.6

Table 7.8. Throughput results for 2006 TREC Efficiency queries.

The score-sorted results cannot be compared directly with results from Chapter 5, since both the machine and the data stored in the index are different. However, we can make a comparison between these results and those in Chapter 6, where we evaluated our document-sorted indexes with binned language models on this same machine. Notice that query througput for the Text feature is about 30% less than we saw in Chapter 6. We can attribute this drop to the very different smoothing parameter used. Our Chapter 6 results used $\mu = 1500$, while these results use $\mu = 250$. When using a large parameter like $\mu = 1500$, weights for frequent terms like "cheese" are much higher than infrequenct terms like "abacus," because the collection frequency is emphasized. The smaller smoothing parameter $\mu = 250$ increases the effect that a word like "cheese" can have in a query, which makes it difficult for max-score to prune that inverted list.

Notice, however, that as we add features, the document-sorted results get much faster. Using the TREC 2005 query set, throughput rises from 59.3 queries per second to 127.5, which is 215% times faster. This is also 47% than our results with $\mu = 1500$. Adding in anchor text, throughput rises to 152.2 queries/second. Throughput drops somewhat as we add phrases, presumably because we now have to consider more inverted lists in the query evaluation process. We also see a large throughput increase in the 2006 queries, and again our indexes with feature mixtures are achieving large throughput gains over the Text feature alone.

As we said before, we cannot directly compare our score-sorted results here with those in Chapter 5, since they use a different binning strategy and were produced on a different machine. However, it does at least seem like we are seeing a similar smoothing effect with the Text feature results. The throughput figures we see here for just the Text feature are about 50% lower than those we saw in Chapter 5.

Interestingly, our efficiency results for the combined feature indexes are very low. While our document-sorted indexes improved with the additional feature data, notice

that the score-sorted index throughput drops by over 70%. The addition of PageRank improves speed by 10%, as the postings with high PageRank rise higher in the lists and make pruning easier. However, this small boost is not enough to counteract the earlier losses.

Comparing our results with those of the TREC 2006 named page task, our throughput results are dramatically faster than all other systems. The fastest system, from University of Melbourne, uses a score-sorted index and achieves a throughput of just under 4 queries per second, with MRR of 0.397. No other system processes queries faster than 1 per second.

In particular, the Indri system ran distributed across 6 CPUs and 12GB of RAM, but required an average of 60 seconds per query. This dissertation was inspired by Indri's low throughput on the navigational search task. We are encouraged that our best system runs over three orders of magnitude faster on a single CPU core (although of a newer vintage) and 4GB additional RAM.

7.4 Conclusion

In this chapter, we adapted our system to perform the navigational search task. We intentionally adapted our system without training to try to validate the claim that our methods are easily adaptable to new tasks. Our effectiveness results are close to some of the best known academic results, while setting new records in efficiency for this task.

The dramatic lesson from this chapter is the dependence of query optimization strategies on the data stored in the index. It is not surprising that there is some effect, since all of the optimization strategies we consider prune postings from consideration based on score bounds, so when document scores change we expect pruning to change. The extent to which the score affects speed is, however, surprising.

This chapter brought together all of our previous work, including TupleFlow, binned probabilities, score-sorted indexes and document-sorted indexes. All of these contributions were necessary to generate these efficient navigational results.

CHAPTER 8

EXTENSIONS

8.1 Introduction

This dissertation has explored how to efficiently process queries using feature-based query models. While building indexes and processing queries compose a major part of the document retrieval problem, more pieces are necessary to build a modern search engine. In this chapter we consider three key obstacles that would need to be overcome to deploy our research findings into a production system. For each challenge, we include a small survey of relevant state-of-the-art techniques, combined with specific guidance on how to integrate our work with these techniques.

8.2 Update

All experiments shown in this dissertation have relied on experimental text collections. For research purposes, it is desirable to use collections of documents that do not change so that experimental results can be accurately compared between researchers. However, few real text collections are unchanging. Typically users want to search web sites, their e-mail, internal corporate documents or a collection of news stories. All of these tasks require a search engine that can handle a collection that changes over time.

8.2.1 Update background

Whether documents are crawled or held locally in a single system, the indexing system needs some way to keep the index synchronized with the document collection.

This process is called *index maintenance* or *incremental indexing*. While there has always been practical interest in incremental indexing, academic work on this topic has been sparse. A small pool of researchers considered this topic between 1990 and 1995, then publications nearly stopped. In the past three years, new research has arrived to build upon and update the assumptions in earlier work.

Cutting and Pedersen [38] present the first incremental indexing work that we consider here. The authors use a variety of methods to store posting lists so that they may be dynamically updated. In the first method, they store postings (word and document location pairs) directly in the B-Tree, sorted first by word and second by document location. This straightforward approach is used more recently by the MySQL database engine for full-text search [2]. This method is simple and conceptually clean, but leaves room for improvement in both space and speed. The authors improve on space utilization by storing the word only once, instead of in each posting. Then, they improve on speed by using an external heap file to store list data instead of storing the data within the tree itself.

Tomasic, Garcia-Molina and Shoens [98] focus on the storage allocation policy in the inverted file. As in Cutting and Pedersen's approach, the inverted file is a heap that requires an allocation policy. Clearly the individual lists are expected to grow over time, but leaving large gaps in the file for extra postings wastes space and time (since the extra file space implies longer seek times between relevant data regions). The authors explore three allocation policies: constant, block, and proportional. In the constant case, each inverted list update operation reserves a constant amount of extra space for new postings. The block strategy is similar, except the extended list is forced to end on a block boundary. The proportional strategy increases leaves some percentage of the total length of the list as empty space; this means that longer lists have more room to grow. Additionally, the authors consider three update policies; new, whole and fill. The new strategy writes new postings to a new location, effec-

tively making each inverted list a linked list of segments. The fill strategy is similar, except each linked list segment is forced to be the same length. Finally, the whole strategy requires that each inverted list be copied at every update, so that lists remain contiguous. Not surprisingly, the authors find that the *new* strategy is quicker for updates, while the *whole* strategy is preferred for queries.

Brown, Callan and Croft [20] investigate similar approaches to those of Tomasic et al. [98]. The authors modify the INQUERY retrieval system to store its data in the Mneme object store [75]. This abstraction of the storage layer is similar to more recent work, such as de Vries et al. [41]. Brown et al. store inverted lists in this disk-based object store; small lists are segregated from large ones, and small lists are allocated in power-of-two sized blocks. Large lists are not necessarily stored sequentially, but may be stored in a linked list of blocks. The results are largely similar to Tomasic et al. [98].

Clarke et al. [33] present a system which, in contrast with the others shown so far, explicitly discusses query activity while new documents are added to the collection. Unlike the work shown in this paper, this method still adds documents to the index in batches, but these updates are committed quickly to the index, so that pauses in query operations are as short as possible.

The four papers mentioned here represented the state-of-the-art for index maintenance until recently. However, these papers were written in the mid-1990s; since that time, computer technology has changed drastically. While all aspects of disk performance have improved, disk capacity has outpaced transfer rates, and transfer rates have outpaced accesses per second (inverse of seek time). It appears that mean capacity, transfer rate, and accesses per second are following exponential curves, but each factor has a different exponent, meaning that capacity will continue to diverge from transfer rate, which will continue to diverge from accesses per second.

More details on this and the effect on retrieval system performance can be found in Zobel, Williams and Kimberly [110]. The important factor is that research that relies on disk performance must be revisited in order to maintain relevance.

Lester et al. [60] revived interest in the area of index maintenance with a 2004 study on the relative advantages of three strategies, in-place, re-build and re-merge. The in-place strategy is similar to Tomasic's *whole* strategy with a proportional allocation policy. The re-build strategy simply rebuilds the entire collection, while re-merge merges new postings into an existing index, forming a new index. The authors find that for updates of less than 10000 documents, both incremental strategies are better than rebuilding the index. Furthermore, for the smallest updates (under 100 documents), the in-place strategy is faster. However, for these small updates, the update time per document approaches 1 second.

Especially because of the issues in transfer time, a new class of update algorithms has been discussed in the literature recently. This strategy, called *geometric partitioning* by Lester et al. [59], does not force inverted lists to be merged together when postings are flushed to disk. Instead, new postings are flushed to disk in entirely new indexes, called *partitions*. Queries acting upon this data must check each partition for inverted list data, so there is a query speed penalty for maintaining too many partitions. Therefore, these partitions can be merged together to form larger partitions that are more efficient to query. Since merging is costly, an efficient system must balance the needs of query processing and efficient indexing in choosing its merging policy.

The name geometric partitioning refers specifically to a merging policy developed by Lester et al. [59]. For a system that can hold b document pointers in memory and some positive integer parameter r, the ith partition is limited in size to br^i pointers. For example, if $r = 3$, the first disk partition is limited to holding 3 times as many documents as the system memory can hold; if this partition grows beyond that size,

its data is merged into the second partition (which is limited to 9 times the system memory). By keeping this exponential distribution of partition sizes, the total number of indexes is kept small while still making the common case (merging in-memory data into the first partition) fast.

Many authors ([26], [28], [59], [91], [92]) have studied the partitioning strategy with good results. Büttcher and Lester focus primarily on balancing document insertion throughput with indexing throughput, while Strohman and Croft focus on reducing the latency of document insertions. However, previous work has focused on throughput instead of latency of operations.

8.2.2 Application Notes

The Strohman and Croft work [92] incorporates one in-memory index and perhaps multiple on-disk indexes. All new or updated documents are added to the in-memory index. When memory fills up, the memory index is written to disk. Queries are processed on each individual index, and results are merged to form a final result list. Deleted documents are marked in a bitmap so the query processor can ignore them. All document updates are performed as a pair of delete and insert operations.

We can use this strategy with either the score-sorted or document-sorted index types discussed in this thesis. Our document-sorted index is easy to integrate, since Strohman and Croft also use a document-sorted index. By assigning document numbers sequentially to new documents, all inverted list mutations are appends to the in-memory index, each of which happens in amortized constant time. Using score-sorted indexes for the in-memory index is trickier, since inverted list postings must be added in sorted order. Using a sorted structure like a B-Tree would give us $O(\log n)$ insertion times instead of the constant time updates with the document-sorted index. This B-Tree would also have to hold uncompressed data in order to make fast insertions possible, which would reduce query speeds and would reduce the number of

inverted list postings that could be held in memory before writing to disk. Because of these drawbacks, the best approach is to use a document-sorted index for the in-memory structure, although either score-sorted or document-sorted indexes could be used on disk.

As we discussed at the beginning of this dissertation, most retrieval approaches rely on some statistics about the collection for scoring, like the length of the collection and the frequency of particular vocabulary terms. This data changes in a dynamic collection. For traditional indexes that do not store score information, it is appropriate to update these statistics with each document insertion, although updating on document delete can be difficult to do efficiently. In our case, these statistics are used to compute binned term weights that are stored in the inverted list. If we use a changing language model, the term weights for early documents will be incompatible with those of later documents. Therefore, we need to keep a static statistical model of text that is used for all score computations.

Using an unchanging model could have unknown negative effects on retrieval effectiveness. For instance, the word *Flickr* was a misspelling ten years ago, but now it is a popular photo sharing website. A model of text produced ten years ago would necessarily assume that *Flickr* is a very rare word, but this assumption is now incorrect. Queries evaluated under that old text model will give far too much weight to the supposedly rare word *flickr*, and presumably these queries would return poor results.

The best approach is to periodically build a new, accurate text model and build all indexes again. Given that our approach also requires new indexes to be built whenever the basic feature set changes, indexes would probably need to be rebuilt periodically just to incorporate improved document representations. The ideal update frequency for the text model is unknown and is an appropriate topic for future research.

8.3 Distribution

The largest collections of documents are too large for a single machine to process efficiently. Our results suggest that servers using current technology can handle between 10 million and 100 million documents each. Collections larger than this size will require multiple machines in order to process queries in with acceptable interactive response times.

8.3.1 Background

There are two obvious ways to distribute a large collection over machines. One method is term distribution, where only a portion of the inverted lists in the index are stored on each machine. Another method is document distribution, where the collection is split into smaller sub-collections, with each sub-collection stored in a self-contained index on a single machine.

The term distribution method is attractive because it reduces data access costs versus document distribution. Suppose we wish to process a k word query on a cluster of n nodes. A document-distributed architecture will look up k inverted lists on n nodes, for a total of kn random accesses. The term-distributed architecture needs just k lookups, since the inverted lists are stored sequentially. Unfortunately, the probability that all k lists will be on the same machine is $1/n^k$; so even for a two word query using a two machine cluster, we only have a 25% chance that the data is properly located.

By contrast, a document-distributed retrieval system is very simple. Each server holds an index for a portion of the documents. One server, sometimes called the *query broker* or *receptionist*, sends the query out to all of the servers for processing. Each server returns a small set of top documents for the sub-collection, including the score for each document. The query broker merges these result lists into a single ranked list using the document scores as a sort key. The top results are returned to the user.

Moffat, Webber and Zobel [73] have produced the most comprehensive work to date on this subject. They improve term distribution performance by combining a query processing approach called *pipelining* with selective data replication and a log-based term assignment scheme. These techniques help to balance the load across a query cluster, but the result still fails to achieve the performance of the simpler document-distributed scheme. These results are disk-based, meaning that the high cost of random disk accesses has been taken into account. Presumably the results would skew even more in favor of document-distributed retrieval if all indexes were stored in memory.

8.3.2 Application Notes

Since a document-distributed retrieval system consists of many smaller information retrieval systems, almost any kind of retrieval algorithm can work in a document-distributed setting. There is only one requirement: the retrieval algorithm must return scores on the the sub-collections that are comparable, since we want to be able to merge the sub-results efficiently based only on score.

We have already discussed score compatibility in the previous section. For distribution, we need to be sure that the same statistical text model is used to build all of the sub-indexes. This is a relatively simple requirement and does not have the possible negative effect on retrieval effectiveness that could affect an updating system.

One other note of caution concerns the ignore phase of retrieval proposed both by Anh and Moffat [11] and Buckley and Lewit [22]. This optimization stops the retrieval evaluation process when the system determines that the order of the top ranked documents has been determined. Although the scores of each document may not be entirely determined at this point, the system can compute upper and lower bounds for each accumulator in the table. If these bounds do not overlap, retrieval can safely terminate.

Notice that this rests on the assumption that all of the retrieved documents are in the accumulator table. This is not a correct assumption in the distributed case, since the final ranked list of documents is a merged result from many different systems. If the ignore phase is used during retrieval, some documents may have scores that are incompletely specified. This may lead to the merge phase ordering documents inappropriately because of incomparable document scores.

8.3.3 Improvements

All of the query optimization methods shown in this thesis rely on a high quality threshold estimate in order to prune uninteresting documents. The threshold is a lower bound on the score of the kth document in the final ranked list. The tighter this lower bound can be, the more uninteresting documents can be skipped and the faster retrieval will go. Preliminary experiments showed that query throughput on score-sorted indexes improved over 50% when an oracle was used to produce tight threshold bounds instead of estimates.

Unfortunately, in distributed retrieval, the document collection is scattered across many machines, and this necessarily reduces the quality of threshold estimates. For large clusters, it may make sense to have the query nodes share threshold information. In the simplest case, all nodes could use the highest threshold computed on all of the query nodes. In a more complicated implementation, the query broker could maintain a running top k document list and periodically create a threshold estimate to share with the query nodes.

This section has used the term *distributed retrieval* to mean retrieval using a cluster of computers in parallel. This is different than the term *distributed IR*, which typically involves routing a query to the appropriate small collection that best matches the information need. There is a large amount of literature on this topic, including inference network [31] and cluster-based language model [106] approaches to the

problem. Routing queries to an appropriate collection may have important efficiency gains, since query evaluation can be restricted to just the collection of documents that contain a relevant match.

By combining threshold methods with a distributed IR technique, it may be possible to achieve a throughput increase while still returning exact answers. Running the query first on a small collection of good documents should generate a good set of candidate answers and a high threshold estimate. Retrieval can then proceed to the less relevant collections. The high threshold estimate should allow most of the documents in these less relevant collections to be skipped, but without compromising retrieval effectiveness.

Lu [63] suggests this kind of technique, where a query is routed first to a small collection of good documents. However, retrieval does not proceed to the rest of the collection, so exact results are not guaranteed. By contrast, Ntoulas [76] presents a system where exact results are guaranteed. However, there are only two collections: a small index of popular documents and a large index with less popular documents. Through clever mathematics, some queries can be provably satisfied on just the small index without looking at the large index. It would be interesting to investigate whether multiple small topical indexes would work even more efficiently.

8.4 Query caching

The fastest way to evaluate a query is to not evaluate it at all. This is the fundamental idea behind caching: if a query has already been evaluated, we can save the results and display them later if someone else asks for the same result.

We performed a quick analysis of query log data provided by Microsoft in order to estimate the potential for caching. In a query log of almost 15 million queries, we found 7.1 million unique queries. 5.5 million of those queries were issued only once, leaving about 1.6 million that were issues multiple times. This means that, within

this query log, 52.5% of queries could have been answered directly from a cache of unlimited size. Note that this log represents a sample of queries taken over a month, and we might find more duplication if we had all of the data for a particular day. We also would expect more duplication within larger query logs. Also note that we did no case normalization, stemming, or punctuation removal to these queries which might also improve caching results.

Recent work by Baeza-Yates et al. [14] shows the diminishing returns achieved by caching query results. As we might expect, a few queries are issued very often to search engines, and caching these results in a major reduction in load. As the cache gets larger, each additional result cached provides a smaller overall benefit. Baeza-Yates et al. show that if there is competition between result caches and inverted list data for memory space, that often storing inverted lists in memory is a better use of space than ever larger result caches. Long and Suel [62] propose a three-level caching scheme which caches results, inverted lists, and inverted list intersections. Fagni et al. [46], like others, recommend that part of the query cache be made dynamic and part static, so that the most popular queries are never evicted from the cache but that some space is left for temporal queries (like those for breaking news events) where popularity will change over time.

Stale data is a problem for any caching system. If we expect that the data in our collection is constantly changing, cached query results represent the results the system would have retrieved at some point in the past. However, our proposed index update strategy consists of old generations of data and new generations, with each generation stored in a different index. By caching accesses to older indexes separately from new indexes, the freshness problem can be tightly controlled.

CHAPTER 9
CONCLUSION

We took a complete view of the indexing and retrieval process informed by current information retrieval trends, like feature-based query models, massive distribution, and main memory processing. Our resulting work ties together a novel distributed computing architecture, a binning method for probabilities, and two optimization strategies for two types of indexes. We have shown that our system is suitable for producing high quality web search results efficiently.

9.1 The Broad View

Distributed computation is the future of text processing and indexing. We believe that TupleFlow (like the similar Dryad system) is a good choice as a basis for next generation text processing systems. We found that the tuple processing framework was a good fit overall for text processing tasks, and that it was easy to distribute load across a cluster of machines this way. The most difficult part of TupleFlow is its configuration, since a TupleFlow job can easy grow to dozens of interconnected stages. Making changes to a TupleFlow job can be tedious and requires a lot of concentration, since the connections between stages are complex and difficult to reason about when viewing the XML configuration files. Based on a suggestion, we plan to make a graphical tool for building jobs so that users can see connections visually. We also hope to allow for multi-stage sub-jobs that can be imported into a TupleFlow job. We notice that both Dryad and MapReduce have been extended with special-purpose

query languages that abstract away the mechanics of job scheduling, and perhaps this is the primary way these frameworks will be used in the future.

The optimizations presented for score-sorted and document-sorted indexes bring produce large throughput gains concentrated at the short query lengths common on the web. The score-sorted indexes appear to be the most efficient query processing route, but unfortunately they are not easily compatible with other kinds of data. With a document-sorted index, it is possible to imagine how a user might mix a binned index with a more traditional document-sorted index that contains word positions. This might be the appropriate approach for a system that is too disk-constrained to hold a lot of phrase information. Notice that score skipping can still be used even when only some of the lists have score skipping information.

In some ways, this work has been an experiment in how much work can be offloaded to the index process. The answer is, "almost all of it." However, building scoring logic into the index makes it difficult to quickly change query weighting. For instance, our named page indexes must be built again if the user wants to focus a little bit more on the titles of documents. For many search scenarios, especially ones where expert searchers expect a variety of search tools at their disposal, this kind of binned indexing is probably not appropriate. An interesting subject for future work is to determine how to make binned indexes coexist better with other types of indexes at query time.

One of the troubles with this kind of work is the massive investment in time necessary to build a reliable system that works on a large scale. To support this dissertation, we built Galago[1], which will soon be available as an open source toolkit. Surprisingly, much of the academic work into efficient systems results in code that is written but never released. Other systems, like our previous Indri system, are

---

[1]http://www.galagosearch.org

written to be useful as they are, but not extended. With Galago, we have attempted to build a system which is extensible and suitable for all kinds of information retrieval research. Galago represents over a year of development effort, and is based on lessons learned from the Indri project. We hope that Galago can save valuable research time, not only for end users and ranking researchers, but for systems researchers as well.

9.2 Contributions

We have established the following contributions to the literature:

- **Score-ordered query optimization.** We add skipping and trimming to the best known algorithm for score-ordered query evaluation, and show a query throughput increase of almost 70%. This algorithm is the fastest *rank-safe* query evaluation method known to us for *ad hoc* queries.

- **Document-ordered query optimization.** We show how score skipping can give some of the speed advantages of score-ordered indexes to document-ordered indexes. Score skipping improves throughput by over 80% on a set of web queries, with almost a 400% throughput increase for two word queries.

- **TupleFlow.** We present TupleFlow, a system for distributed computation on a grid of machines. TupleFlow extends MapReduce with comparators, compression, graph-based scheduling and arbitrary data flow, allowing for complex distributed task execution, but while still allowing for single-threaded tasks. TupleFlow is particularly suited for index building and text processing.

- **Binning Probabilities.** We show language model binning, which allows language model probabilities to be used in retrieval systems that require positive integers. We show how to solve the problem of smoothing, and how a saturation parameter can improve overall effectiveness.

- **Case study of named page retrieval.** We show how features used in a real web ranking function can be stored as integers to produce an effective and efficient search engine.

BIBLIOGRAPHY

[1] Open MPI: `http://www.open-mpi.org`.

[2] MySQL AB. MySQL.

[3] Eugene Agichtein, Eric Brill, and Susan Dumais. Improving web search ranking by incorporating user behavior information. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 19–26, New York, NY, USA, 2006. ACM Press.

[4] James Allan, James P. Callan, W. Bruce Croft, Lisa Ballesteros, John Broglio, Jinxi Xu, and Hongmin Shu. INQUERY at TREC-5. In *Text REtrieval Conference*, pages 119–132, 1996.

[5] James Allan and Giridhar Kumaran. Stemming in the language modeling framework. In *SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 455–456, New York, NY, USA, 2003. ACM.

[6] Thomas Anderson, David Culler, David Patterson, and NOW Team. A case for networks of workstations: NOW. In *IEEE Micro*, pages 54–64, February 1995.

[7] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 35–42, New York, NY, USA, 2001. ACM Press.

[8] Vo Ngoc Anh and Alistair Moffat. Impact transformation: effective and efficient web retrieval. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 3–10. ACM Press, 2002.

[9] Vo Ngoc Anh and Alistair Moffat. Improved retrieval effectiveness through impact transformation. In *ADC '02: Proceedings of the thirteenth Australasian database conference*, pages 41–47, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[10] Vo Ngoc Anh and Alistair Moffat. Simplified similarity scoring using term ranks. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 226–233, New York, NY, USA, 2005. ACM Press.

[11] Vo Ngoc Anh and Alistair Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 372–379, New York, NY, USA, 2006. ACM Press.

[12] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[13] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 243–254, New York, NY, USA, 1997. ACM Press.

[14] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 183–190, New York, NY, USA, 2007. ACM Press.

[15] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[16] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. IO-Top-k: index-access optimized top-k query processing. In *VLDB'2006: Proceedings of the 32nd international conference on Very large data bases*, pages 475–486. VLDB Endowment, 2006.

[17] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *WWW7: Proceedings of the seventh international conference on World Wide Web 7*, pages 107–117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[18] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 426–434, New York, NY, USA, 2003. ACM Press.

[19] Eric W. Brown. Fast evaluation of structured queries for information retrieval. In *SIGIR '95: Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 30–38, New York, NY, USA, 1995. ACM Press.

[20] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 192–202, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[21] Chris Buckley. Implementation of the SMART information retrieval system. Technical report, Cornell University, Ithaca, NY, USA, 1985.

[22] Chris Buckley and Alan F. Lewit. Optimization of inverted vector searches. In *Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 97–110. ACM Press, 1985.

[23] Chris Buckley, Mandar Mitra, Janet Walz, and Claire Cardie. Using clustering and SuperConcepts within SMART: TREC 6. *Information Processing and Management*, 36(1):109–131, 2000.

[24] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 89–96, New York, NY, USA, 2005. ACM Press.

[25] Brendan Burns, Kevin Grimaldi, Alex Kostadinov, and Mark Corner. Flux: A language for programming high-performance servers. In *USENIX*, 2006.

[26] Stefan Büttcher and Charles L. A. Clarke. Indexing time vs. query time tradeoffs in dynamic information retrieval systems. In *CIKM 2005: Proceedings of the 14th ACM Conference on Infromation and Knowledge Management*, Bremen, Germany, November 2005.

[27] Stefan Büttcher and Charles L. A. Clarke. A document-centric approach to static index pruning in text retrieval systems. In *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 182–189, New York, NY, USA, 2006. ACM Press.

[28] Stefan Büttcher and Charles L. A. Clarke. A hybrid approach to index maintenance in dynamic text retrieval systems. In *ECIR 2006: Proceedings of the 28th European Conference on Information Retrieval*, London, UK, April 2006.

[29] Stefan Büttcher, Charles L. A. Clarke, and Ian Soboroff. The TREC 2006 Terabyte track. In *TREC 2006*, Gaithersburg, Maryland USA, November 2007.

[30] James P. Callan, W. Bruce Croft, and Stephen M. Harding. The INQUERY retrieval system. In *Proceedings of DEXA-92, 3rd International Conference on Database and Expert Systems Applications*, pages 78–83, 1992.

[31] James P. Callan, Zhihong Lu, and W. Bruce Croft. Searching distributed collections with inference networks. In *SIGIR '95: Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 21–28, New York, NY, USA, 1995. ACM Press.

[32] David Carmel, Doron Cohen, Ronald Fagin, Eitan Farchi, Michael Herscovici, Yoelle S. Maarek, and Aya Soffer. Static index pruning for information retrieval systems. In *SIGIR '01: Proceedings of the 24th annual international ACM*

SIGIR conference on Research and development in information retrieval, pages 43–50, New York, NY, USA, 2001. ACM Press.

[33] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. Fast inverted indexes with on-line update. Technical Report CS-94-40, University of Waterloo, Waterloo, Canada, 1994.

[34] Charles L. A. Clarke, Falk Scholer, and Ian Soboroff. The TREC 2005 Terabyte track. In *TREC 2005*, 2006.

[35] Cyril W. Cleverdon. The significance of the Cranfield tests on index languages. In *SIGIR '91: Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 3–12, New York, NY, USA, 1991. ACM Press.

[36] Nick Craswell and David Hawking. Overview of the TREC-2002 web track. In *TREC 2002*, 2003.

[37] Nick Craswell, David Hawking, Ross Wilkinson, and Mingfang Wu. Overview of the TREC 2003 web track. In *TREC 2003*, 2004.

[38] D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. In *SIGIR '90: Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 405–411, New York, NY, USA, 1990. ACM Press.

[39] Doug Cutting. Hadoop: `http://lucene.apache.org/hadoop`.

[40] James R. Dabrowski and Ethan V. Munson. Is 100 milliseconds too fast? In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, pages 317–318, New York, NY, USA, 2001. ACM Press.

[41] Arjen P. de Vries, Johan A. List, and Henk Ernst Blok. The multi-model DBMS architecture and XML information retrieval. In *Intelligent Search on XML Data*, pages 179–191, 2003.

[42] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[43] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.

[44] Nadav Eiron, Kevin S. McCurley, and John A. Tomlin. Ranking the web frontier. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 309–318, New York, NY, USA, 2004. ACM Press.

[45] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 102–113, New York, NY, USA, 2001. ACM Press.

[46] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, 2006.

[47] Hui Fang, Tao Tao, and ChengXiang Zhai. A formal study of information retrieval heuristics. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 49–56, New York, NY, USA, 2004. ACM Press.

[48] Norbert Fuhr. Two models of retrieval with probabilistic indexing. In *SIGIR '86: Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 249–257, New York, NY, USA, 1986. ACM.

[49] Sanjay Ghemawat, Howard Goboff, and Shun-Tak Leung. The Google file system. In *ACM Symposium on Operating System Principles (SOSP)*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[50] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 902–903, New York, NY, USA, 2005. ACM.

[51] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, Lisbon, Portugal, March 2007.

[52] Kalervo Järvelin and Jaana Kekäläinen. IR evaluation methods for retrieving highly relevant documents. In *SIGIR '00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 41–48, New York, NY, USA, 2000. ACM.

[53] Thorsten Joachims, Laura Granka, Bing Pan, Helene Hembrooke, and Geri Gay. Accurately interpreting clickthrough data as implicit feedback. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 154–161, New York, NY, USA, 2005. ACM Press.

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

[55] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[56] Robert Krovetz. Viewing morphology as an inference process. In *SIGIR '93: Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 191–202, New York, NY, USA, 1993. ACM Press.

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

[58] Nicholas Lester, Alistair Moffat, William Webber, and Justin Zobel. Space-limited ranked query evaluation using adaptive pruning. In *WISE*, pages 470–477, 2005.

[59] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast on-line index construction by geometric partitioning. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 776–783, New York, NY, USA, 2005. ACM Press.

[60] Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *CRPIT '04: Proceedings of the 27th conference on Australasian computer science*, pages 15–23, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[61] Xiaohui Long and Torsten Suel. Optimized query execution in large search engines with global page ordering. In *VLDB*, pages 129–140, 2003.

[62] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 257–266, New York, NY, USA, 2005. ACM Press.

[63] Zhihong Lu and Kathryn S. McKinley. Partial collection replication versus caching for information retrieval systems. In *SIGIR '00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 248–255, New York, NY, USA, 2000. ACM Press.

[64] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[65] Marissa Mayer. Scaling Google for every user. Seattle Conference on Scalability, June 2007.

[66] Donald Metzler. *Beyond Bags of Words: Effectively Modeling Dependence and Features in Information Retrieval*. PhD thesis, University of Massachusetts, University of Massachusetts, September 2007.

[67] Donald Metzler and W. Bruce Croft. Combining the language model and inference network approaches to retrieval. *Inf. Process. Manage.*, 40(5):735–750, 2004.

[68] Donald Metzler and W. Bruce Croft. A Markov random field model for term dependencies. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 472–479, New York, NY, USA, 2005. ACM Press.

[69] Donald Metzler, Victor Lavrenko, and W. Bruce Croft. Formal multiple-bernoulli models for language modeling. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 540–541, New York, NY, USA, 2004. ACM.

[70] Donald Metzler, Trevor Strohman, and W. Bruce Croft. Indri at TREC 2006: Lessons learned from three Terabyte tracks. In *TREC 2006*, Gaithersburg, Maryland USA, 2007. electronic proceedings only.

[71] Donald Metzler, Trevor Strohman, and W. Bruce Croft. Revisiting document-centric impact-based retrieval models from a probabilistic perspective. In *In submission*, 2007.

[72] Donald Metzler, Trevor Strohman, Yun Zhou, and W. Bruce Croft. Indri at TREC 2005: Terabyte track (notebook version). In *TREC 2005 Notebook*, pages 175–180, Gaithersburg, Maryland USA, November 2006.

[73] Alistair Moffat, William Webber, and Justin Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 348–355, New York, NY, USA, 2006. ACM Press.

[74] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[75] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, 1990.

[76] Alexandros Ntoulas and Junghoo Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 191–198, New York, NY, USA, 2007. ACM Press.

[77] Paul Ogilvie and Jamie Callan. Combining document representations for known-item search. In *SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 143–150, New York, NY, USA, 2003. ACM Press.

[78] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[79] Michael Persin, Justin Zobel, and Ron Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society of Information Science*, 47(10):749–764, 1996.

[80] Rob Pike, Sean Dorward, Robert Grisemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):244–298, 2005.

[81] Jay M. Ponte and W. Bruce Croft. A language modeling approach to information retrieval. In *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 275–281, New York, NY, USA, 1998. ACM.

[82] M. F. Porter. An algorithm for suffix stripping. pages 313–316, 1997.

[83] Francois Raab. TPC-C - the standard benchmark for online transaction processing (OLTP). In Jim Gray, editor, *The Benchmark Handbook*. Morgan Kaufmann, 1993.

[84] Stephen Robertson, Hugo Zaragoza, and Michael Taylor. Simple BM25 extension to multiple weighted fields. In *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 42–49, New York, NY, USA, 2004. ACM Press.

[85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *USENIX*, pages 119–130, Portland OR (USA), 1985.

[86] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *SIGARCH Comput. Archit. News*, 31(2):422–433, 2003.

[87] Philip Schwan. Lustre: Building a file system for 1000-node clusters. In *Linux Symposium*, 2003.

[88] Margo Seltzer, David Krinsky, Keith Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, page 102, Washington, DC, USA, 1999. IEEE Computer Society.

[89] Tom Spring. Three minutes with Google's Eric Schmidt, 2002.

[90] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented DBMS. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[91] Trevor Strohman. Dynamic collections in Indri. Technical Report IR-426, University of Massachusetts Amherst, 2005.

[92] Trevor Strohman and W. Bruce Croft. Low latency index maintenance in Indri. In *SIGIR Workshop on Open Source Information Retrieval*, August 2006.

[93] Trevor Strohman, Donald Metzler, Howard Turtle, and W. Bruce Croft. Indri: A language model-based search engine for complex queries. In *Proceedings of the International Conference on Intelligence Analysis*, 2005.

[94] Trevor Strohman, Howard Turtle, and W. Bruce Croft. Optimization strategies for complex queries. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 219–225, New York, NY, USA, 2005. ACM Press.

[95] Jaime Teevan, Eytan Adar, Rosie Jones, and Michael A. S. Potts. Information re-retrieval: repeat queries in Yahoo's logs. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 151–158, New York, NY, USA, 2007. ACM.

[96] Douglas Thain, Todd Tannenbaus, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):232–256, 2005.

[97] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 242–249, New York, NY, USA, 2005. ACM Press.

[98] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 289–300, New York, NY, USA, 1994. ACM Press.

[99] H. Turtle and W. B. Croft. Inference networks for document retrieval. In *SIGIR '90: Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 1–24, New York, NY, USA, 1990. ACM Press.

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

[101] Sergei Vassilvitskii and Eric Brill. Using web-graph distance for relevance feedback in web search. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 147–153, New York, NY, USA, 2006. ACM.

[102] William Webber and Alistair Moffat. In search of reliable retrieval experiments. In Andrew Turpin and Ross Wilkinson, editors, *Proc. 10th Australasian Document Computing Symposium*, pages 26–33, Sydney, Australia, December 2005.

[103] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM Press.

[104] Hugh E. Williams, Justin Zobel, and Dirk Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.

[105] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[106] Jinxi Xu and W. Bruce Croft. Cluster-based language models for distributed retrieval. In *SIGIR '99: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 254–261, New York, NY, USA, 1999. ACM Press.

[107] Yun Zhou and W. Bruce Croft. Document quality models for web ad hoc retrieval. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 331–332, New York, NY, USA, 2005. ACM Press.

[108] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

[109] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, 1998.

[110] Justin Zobel, Hugh E. Williams, and Sam Kimberley. Trends in retrieval system performance. In *ACSC*, pages 241–248, 2000.

[111] M. Zukowski, P. A. Boncz, N. Nes, and S. Heman. MonetDB/X100 - a DBMS in the CPU cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, June 2005.