Search Engine Optimisation Using Past Queries

A thesis submitted for the degree of
Doctor of Philosophy

Steven Garcia  B.App.Sci. (Hons.),
School of Computer Science and Information Technology,
Science, Engineering, and Technology Portfolio,
RMIT University,
Melbourne, Victoria, Australia.

30th March, 2007
Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Steven Garcia
School of Computer Science and Information Technology
RMIT University
30th March, 2007
Acknowledgments

My venture into the realm of research began under the supervision of Hugh Williams and Adam Cannane. As a fledgling in the research environment, I found their support and enthusiasm highly motivating and inspiring. Half way through my candidature Adam and Hugh chose to leave RMIT. I was fortunate enough to be taken on as a student by Andrew Turpin and Justin Zobel. I owe a great deal to Andrew and Justin in shaping my views on what research means and how it should be conducted. They set a high standard for what a researcher should strive to be.

RMIT University provided a fantastic environment for research with a large group of students and staff dedicated to information retrieval. For informative and thought provoking discussions I would like to thank all the members of the RMIT Search Engine Group (SEG). I would especially like to thank Falk Scholer, Abhijit Chattaraj, Halil Ali, Bodo Billerbeck, Nick Lester, Jelita Asian, Ying Zhao, Pauline Chou, Timo Volkmer, Jovan Pehcevski, Yaniv Bernstein, Michael Cameron, Jonathan Yu, Iman Suyoto, Ranjan Sinha, Milad Shokouhi, Sarvnaz Karimi and Yohannes Tsegay, for their friendship over the past years, including the occasionally productive, but more often entertaining conversations over lunch or coffee-breaks.

Finally, none of this would have been possible without the encouragement and support of my father, Adrian Garcia, and fiancee, Monique Widjaja. To them I owe more than I can express in words.
Credits

This thesis contains material published in preliminary form elsewhere. Chapters 3, and 5 are based on material published in Garcia et al. [2004]. Chapter 4 is based on material published in Garcia and Turpin [2006].

This work was supported by the Australian Research Council and the Search Engine Group at RMIT University.

Note

Unless otherwise stated, all fractional results have been rounded to the displayed number of decimal figures.
Contents

Abstract

1 Introduction
  1.1 Information Retrieval .............................................. 3
  1.2 Exploiting Query Logs ............................................ 5
  1.3 Thesis Overview .................................................. 6

2 Background
  2.1 Queries and Results ............................................... 9
     2.1.1 Studies of User Query Logs ............................... 10
     2.1.2 Repetition and Reformulation ............................. 12
     2.1.3 Advanced Operators and Pages Examined ................ 12
     2.1.4 Changes in Query Patterns Over Time .................... 13
     2.1.5 Implications of Query Log Analysis ...................... 14
  2.2 Measuring Effectiveness .......................................... 16
     2.2.1 Precision and Recall ....................................... 16
     2.2.2 Reciprocal Rank ........................................... 17
     2.2.3 Single Value Measures .................................... 17
     2.2.4 TREC Project ............................................... 18
       TREC Collections ............................................... 19
  2.3 Example Collection: The Tempest ................................. 20
  2.4 The Inverted Index .............................................. 21
     2.4.1 Inverted List Compression ................................. 22
  2.5 Search System Architecture ..................................... 24
  2.6 Query Evaluation ................................................ 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.1</td>
<td>Vector-Space Model</td>
<td>29</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Probabilistic Model</td>
<td>30</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Language Models</td>
<td>31</td>
</tr>
<tr>
<td>2.7</td>
<td>Stopping, Stemming and Case-folding</td>
<td>33</td>
</tr>
<tr>
<td>2.8</td>
<td>Accumulator Limiting</td>
<td>34</td>
</tr>
<tr>
<td>2.9</td>
<td>Inverted List Reorganisation</td>
<td>35</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Frequency-Ordering</td>
<td>36</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Impact-Ordering</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>Collection-Oriented Impacts</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>Document-Centric Impacts</td>
<td>40</td>
</tr>
<tr>
<td>2.10</td>
<td>Collection Reordering</td>
<td>42</td>
</tr>
<tr>
<td>2.11</td>
<td>Search Engine Caches</td>
<td>43</td>
</tr>
<tr>
<td>2.11.1</td>
<td>Single Structure Caching</td>
<td>43</td>
</tr>
<tr>
<td>2.11.2</td>
<td>Multi Structure Caching</td>
<td>44</td>
</tr>
<tr>
<td>2.11.3</td>
<td>Advanced Cache Management Policies</td>
<td>45</td>
</tr>
<tr>
<td>2.11.4</td>
<td>Limitations of Caching</td>
<td>46</td>
</tr>
<tr>
<td>2.12</td>
<td>Query Difficulty Prediction</td>
<td>46</td>
</tr>
<tr>
<td>2.12.1</td>
<td>Measuring Prediction Effectiveness</td>
<td>47</td>
</tr>
<tr>
<td>2.12.2</td>
<td>Inverse Document Frequency Predictors</td>
<td>48</td>
</tr>
<tr>
<td>2.12.3</td>
<td>Machine Learning Based Predictors</td>
<td>49</td>
</tr>
<tr>
<td>2.12.4</td>
<td>Language Model Based Predictors</td>
<td>50</td>
</tr>
<tr>
<td>2.12.5</td>
<td>Other Forms of Difficulty Prediction Evidence</td>
<td>51</td>
</tr>
<tr>
<td>2.12.6</td>
<td>Modelling Query Difficulty</td>
<td>52</td>
</tr>
<tr>
<td>2.13</td>
<td>Query Independent Evidence</td>
<td>53</td>
</tr>
<tr>
<td>2.14</td>
<td>Summary</td>
<td>55</td>
</tr>
<tr>
<td>3</td>
<td>Access-Ordering</td>
<td>57</td>
</tr>
<tr>
<td>3.1</td>
<td>Document Access Frequencies</td>
<td>57</td>
</tr>
<tr>
<td>3.2</td>
<td>Index Design</td>
<td>60</td>
</tr>
<tr>
<td>3.3</td>
<td>Access Counting</td>
<td>62</td>
</tr>
<tr>
<td>3.4</td>
<td>Early Termination Heuristics</td>
<td>64</td>
</tr>
<tr>
<td>3.5</td>
<td>Index Compression</td>
<td>68</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Access-Block Compaction</td>
<td>69</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Other approaches</td>
<td>71</td>
</tr>
</tbody>
</table>
CONTENTS

3.6 Results
   3.6.1 Collection and Queries
   3.6.2 Overall Speed and Accuracy
   3.6.3 Access Counts
   3.6.4 Early Termination
       Maxpost Pruning
       Two-Phase Pruning
       Other Pruning Schemes
   3.6.5 Pruning with Term Offsets
   3.6.6 Index Compaction Schemes
3.7 Summary

4 Extensions to Access-Ordering
   4.1 Motivation
   4.2 Access-Reordering
   4.3 Measuring Change in Index Performance
       4.3.1 Change in User Queries
   4.4 Experiments in Access-Reordering
       4.4.1 Experimental Environment
       4.4.2 Effects of Pruning
       4.4.3 Stability of Access-Reordering
       4.4.4 Efficiency
   4.5 Summary

5 Static Index Pruning with Access Counts
   5.1 Access-Pruning
   5.2 Query Selection
   5.3 Related Work
   5.4 Results
       5.4.1 Experimental Environment
       5.4.2 Effects of Pruning
       5.4.3 Query Selection
   5.5 Summary
6 Query Caching ................................................. 125
   6.1 Motivation .............................................. 125
   6.2 Related Work ........................................... 127
   6.3 Caching Strategies .................................... 130
      6.3.1 General Cost Model ............................... 131
      6.3.2 Cache Management ................................. 132
   6.4 Experiments ............................................. 134
   6.5 Results .................................................. 136
      6.5.1 Individual Component Caching ...................... 136
      6.5.2 Combined Caches .................................. 139
   6.6 Summary ................................................. 146

7 Access Counts and Query Accuracy ............................. 149
   7.1 Access Counts as Query-Independent Evidence .......... 149
      7.1.1 Rationale .......................................... 150
      7.1.2 Methods of Incorporating Access Counts into the Ranking Function . 151
      7.1.3 Experimental Environment .......................... 154
      7.1.4 Results ............................................ 154
      7.1.5 Analysis .......................................... 158
   7.2 Access Counting for Difficulty Prediction .............. 160
      7.2.1 Determining Query Difficulty Using Access-Order .......... 160
      7.2.2 Evaluation of Query Prediction Performance .......... 163
      7.2.3 Analysis of Access-Order Based Prediction .......... 165
   7.3 Summary ................................................. 166

8 Conclusions .................................................. 169
   8.1 Directions for Future Research ......................... 173
   8.2 Final Remarks .......................................... 173

Bibliography .................................................... 175
# List of Figures

2.1 Sample documents from Tempest collection .......................... 19  
2.2 Example document-level inverted lists ................................ 20  
2.3 Example word-level inverted lists .................................... 20  
2.4 Example document-level inverted lists with difference compaction ................................................. 22  
2.5 Interaction of components within a search engine ..................... 24  
2.6 Sample impact-ordered inverted lists ................................. 39  
2.7 Document-oriented impact-ordering ................................. 41  
3.1 Document access frequency of all documents in WT10g ............ 58  
3.2 Document access frequency of known relevant documents in WT10g .................................................. 59  
3.3 Query result example .................................................. 61  
3.4 Access-ordered lists for Tempest collection ......................... 63  
3.5 TWO-PHASE prune scheme .............................................. 66  
3.6 Effect of FIXED BLOCK compaction ............................... 86  
4.1 Sample Access-reordered lists for terms in Tempest collection .......... 95  
4.2 Oracle result set comparison technique ............................... 96  
4.3 Query repetition frequency in MSN and Lycos query logs ........ 99  
4.4 Query evaluation result comparison using access-reordered indexes ................................................... 100  
4.5 Effect of pruning threshold and query training set size on accuracy .................................................. 102  
4.6 Recall/Precision curves for varying training set sizes .............. 103  
5.1 Sample access-pruned lists for terms in Tempest collection ........ 109  
5.2 Effect of pruning threshold on access-pruned index size, accuracy, and query evaluation time for WT10g collection .................................................. 114  
5.3 WT10g collection list length distribution ........................... 115
5.4 Effect of access-pruning on TREC-9 topics .................................................. 117
5.5 Effect of pruning threshold on access-pruned index size, accuracy, and query
evaluation time for Gov collection ................................................................. 119
5.6 Effect of access-pruning on named-page topics NP1–50 ............................. 120
5.7 Accuracy of access-pruning when combined with difficulty prediction ........ 122
6.1 Distribution of sequential pages requested in Excite and Anon query logs .... 136
6.2 Savings for individual search engine component caches ............................ 137
6.3 Savings for combined search engine component cache .............................. 139
6.4 Comparison of CPB and LRU cache policies ............................................. 142
6.5 Effect of modifying the CPB cost estimations ............................................ 143
6.6 Effect of varying maximum results pages cached per query ....................... 144
7.1 Count of known relevant documents per 17,000 document block in an access-
ordering of WT10g ....................................................................................... 151
7.2 Use of access-order to predict query difficulty .......................................... 161
7.3 Comparison of access-order based predictor to average precision ............. 163
7.4 Comparison of access-order based predictor to P@10 ............................... 164
7.5 Comparison of access-order based predictor to reciprocal rank .................. 165
# List of Tables

2.1 Data used in existing query log studies ........................................... 11  
2.2 Query logs used in this thesis ......................................................... 15  
2.3 TREC collections used in this thesis ................................................ 19  
2.4 Example vocabulary data structure entries ....................................... 25  
2.5 Example document mapping table structure entries ............................. 25  
3.1 Access counts for 25 most accessed documents in Tempest collection ........ 62  
3.2 Summary of pruning schemes ............................................................ 72  
3.3 Comparison of counting techniques ................................................... 75  
3.4 Pruning with the MAXPOST scheme .................................................. 77  
3.5 Pruning with the TWO-PHASE scheme ............................................... 78  
3.6 Exploration of $c_i$ and $c_a$ parameters in TWO-PHASE pruning ............. 80  
3.7 Pruning with the MINACCESS scheme ............................................... 81  
3.8 Pruning with the AVGACCUM scheme ............................................... 82  
3.9 Pruning with the AVGROLLACCUM scheme ....................................... 83  
3.10 Effect of including postings in inverted lists .................................... 84  
3.11 Comparison of compaction schemes ............................................... 85  
4.1 Access counts and ranks for 30 documents in Tempest collection ............. 94  
4.2 Access-reordered efficiency results ................................................. 104  
5.1 Effect of access-pruning $P$ threshold on accuracy for TREC-9 and 10 topics . 116  
5.2 Correlation between difficulty predictors and MAP .............................. 118  
5.3 Query evaluation time for 10,000 Excite queries evaluated on full and access- pruned indexes ............................................................ 123
6.1 Data structure access during query evaluation . . . . . . . . . . . . . . . . . . . 129
6.2 Inverted list sizes in WT10g and Anon collection . . . . . . . . . . . . . . . . . 134
6.3 Cache hit summary for WT10g collection and Excite queries . . . . . . . . . . . 145
7.1 Effectiveness of NORM-AC document prior . . . . . . . . . . . . . . . . . . . . 155
7.2 Effectiveness of LOG-AC document prior . . . . . . . . . . . . . . . . . . . . 156
7.3 Effectiveness of RANK-AC document prior . . . . . . . . . . . . . . . . . . . . 157
7.4 Effectiveness of language model and rank based document priors . . . . . . . . 158
# Table of Notations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V$</td>
<td>The set of terms in the collection.</td>
</tr>
<tr>
<td>$C$</td>
<td>The set of documents in the collection.</td>
</tr>
<tr>
<td>$q$</td>
<td>Query, a list of terms ${q_i,q_j,\ldots,q_n}$.</td>
</tr>
<tr>
<td>$d$</td>
<td>A document from the collection.</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of documents in the collection.</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of indexed terms in the collection.</td>
</tr>
<tr>
<td>$W_d$</td>
<td>Weight of document $d$.</td>
</tr>
<tr>
<td>$W_q$</td>
<td>Weight of query $q$.</td>
</tr>
<tr>
<td>$W_A$</td>
<td>Average document length.</td>
</tr>
<tr>
<td>$f_t$</td>
<td>The number of documents in which the term $t$ appears.</td>
</tr>
<tr>
<td>$F_t$</td>
<td>The number of occurrences of term $t$ in the collection.</td>
</tr>
<tr>
<td>$f_{d,t}$</td>
<td>Within-document frequency of term $t$ in document $d$.</td>
</tr>
<tr>
<td>$w_{d,t}$</td>
<td>Weight of term $t$ in document $d$.</td>
</tr>
<tr>
<td>$f_{q,t}$</td>
<td>Within-query frequency of term $t$ in query $q$.</td>
</tr>
<tr>
<td>$w_{q,t}$</td>
<td>Weight of term $t$ in query $q$.</td>
</tr>
<tr>
<td>$s$</td>
<td>Slope factor for document length pivoting.</td>
</tr>
<tr>
<td>$p$</td>
<td>Pivot point for document length pivoting.</td>
</tr>
<tr>
<td>$S_{q,d}$</td>
<td>Estimated similarity between document $d$ and query $q$.</td>
</tr>
<tr>
<td>$R_l$</td>
<td>The set of relevant documents for a query.</td>
</tr>
<tr>
<td>$R_t$</td>
<td>The set of retrieved documents for a query.</td>
</tr>
</tbody>
</table>
Abstract

World Wide Web search engines process millions of queries per day from users all over the world. Efficient query evaluation is achieved through the use of an inverted index, where, for each word in the collection the index maintains a list of the documents in which the word occurs. Query processing may also require access to document specific statistics, such as document length; access to word statistics, such as the number of unique documents in which a word occurs; and collection specific statistics, such as the number of documents in the collection. The index maintains individual data structures for each these sources of information, and repeatedly accesses each to process a query.

A by-product of a web search engine is a list of all queries entered into the engine: a query log. Analyses of query logs have shown repetition of query terms in the requests made to the search system. In this work we explore techniques that take advantage of the repetition of user queries to improve the accuracy or efficiency of text search. We introduce an index organisation scheme that favours those documents that are most frequently requested by users and show that, in combination with early termination heuristics, query processing time can be dramatically reduced without reducing the accuracy of the search results.

We examine the stability of such an ordering and show that an index based on as little as 100,000 training queries can support at least 20 million requests. We show the correlation between frequently accessed documents and relevance, and attempt to exploit the demonstrated relationship to improve search effectiveness. Finally, we deconstruct the search process to show that query time redundancy can be exploited at various levels of the search process. We develop a model that illustrates the improvements that can be achieved in query processing time by caching different components of a search system. This model is then validated by simulation using a document collection and query log. Results on our test data show that a well-designed cache can reduce disk activity by more than 30%, with a cache that is one tenth the size of the collection.
Chapter 1

Introduction

Search engines provide a gateway through which people can find relevant information in large collections of heterogeneous data. For both the web, and the collections maintained by enterprises, search engines efficiently service the information needs of people that require access to the data therein. Web search engines service millions of queries per day, and search collections that contain billions of documents. As the growth in the number of documents that are available in such collections continues, the task of finding documents that are relevant to user queries becomes increasingly costly. In this work, we propose several search engine optimisations based on the redundancy in past user queries. We propose techniques that work at several levels of the query evaluation process, ranging from inverted list organisation, to cache structures. While the primary focus of this work is search engine efficiency, we also examine techniques to improve search result accuracy using the information obtained from past queries.

1.1 Information Retrieval

Baeza-Yates and Ribeiro-Neto [1999] define Information Retrieval (IR) as the “part of computer science which studies the retrieval of information from a collection of written documents. The retrieved documents aim at satisfying a user information need usually expressed in natural language.” Salton and McGill [1983] note that “information retrieval is concerned with the representation, storage, organisation, and accessing of information items.” The tasks of an IR system are numerous. With growing amounts of electronically stored data to manage and maintain, the costs associated with accurate retrieval become increasingly high. Not only do growing collection sizes and types (such as image and sound) result in a wider
search space for the IR system to maintain and explore, the growth in the number of search system users requires adaptation to the differing expectations and needs of new users.

Over the past decade or so, the field of IR has undergone several changes. The focus of IR research has shifted from development of systems that are intended for expert searchers, such as librarians and intelligence officers, to systems that cater to the needs of the layman. Where systems in the past required expertise in query formulation, current IR systems aim to satisfy the needs of users with no expert knowledge in querying, by allowing simple query formulation. This shift has resulted in the development of query answer evaluation techniques that require minimal user information to generate query results, while retaining a high level of accuracy in those results.

Further, the scale of the collections indexed by search systems has grown dramatically. Within the research community, the focus on managing large volumes of data is highlighted by initiatives such as the Text Retrieval Conference (TREC), where collection sizes have increased since 1992 from hundreds of megabytes to hundreds of gigabytes [Voorhees and Harman, 2005]. Perhaps the single largest driving force behind such changes is the emergence and adoption of the internet, and more specifically the web. Since its inception in the early 90s, the web has grown into a staggeringly vast collection of documents on a multitude of topics. However, as documents on the web are distributed, there is no central point from which to begin accessing the collection. Indeed, without prior knowledge of the collection, it is not a trivial task to reach a desired resource.

Early attempts to resolve the information access problems focused on the generation of directory services, where users could traverse a manually maintained hierarchy of concepts with links to related pages at each node. Well known examples of such services are still in use today [YahooDir; DMOZ]. However, the manual effort required in the maintenance of such directories increases with the amount of data on the web.

A more scalable solution is the use of IR systems that crawl each page on the web and create an index of the collection which can be queried by users with key terms. Such systems, often referred to as Web Search Engines, are now an integral part of the web. Popular search engines such as MSN Live Search [MSN], Yahoo [Yahoo], and Google [Google] service many millions of queries per day, and index billions of documents. In this thesis we exploit logs of these past queries to improve the efficiency and accuracy of search engines.
1.2 Exploiting Query Logs

With millions of users, web search engines have access to large logs of user queries, as well as records of the documents that users choose to view from the search results. Over recent years, several studies of user query logs have analysed search behaviour and found that users tend to express their information need in less than three terms, and are unlikely to examine more than one page of search results [Jansen et al., 2005; Jansen and Spink, 2006]. Such studies allow web search engine vendors to refine their system interfaces to cater for the ways in which users utilise the search engine. However, to date few published works have examined how such information can be utilised to improve the efficiency and effectiveness of underlying search algorithms and data structures for web search engines.

Traditional search engine technology is based on estimating the similarity between a user query and the documents maintained in a collection through statistical ranking functions [Zobel and Moffat, 2006]. Fast access to statistics required at query time are supported through data structures such as inverted indexes. A typical inverted index stores, for each word or term in a document collection, the set of documents in which the term appears. A standard approach to inverted index organisation is to use document-ordering. A document-ordered index is arbitrarily ordered, based on the sequence in which documents are examined at index construction time. Some work has been done in optimising index ordering to improve query evaluation by taking advantage of document properties [Persin et al., 1996; Anh and Moffat, 2002b]. However, despite the maturity of search engine design, the relationship between past user queries and the history of documents retrieved from the collection has not been exploited in the structure of the inverted lists.

Spink et al. [2001] analysed over one million queries extracted from the Excite search engine query logs. They found that of the queries analysed, the top 75 terms in frequency account for 9% of all query terms. Furthermore, after common terms without context ("and", "of", "the", and so on) were removed, 67 terms accounted for 11.5% of all terms used in all queries. In Chapter 3, we show that a similar trend exists in the set of documents returned by a search engine over all submitted queries, such that a subset of documents account for a significant fraction of the search results.

The query evaluation process requires access to various data sources. Repetition in query logs suggests that search engines could benefit from effective caching policies. However, to date the focus of search engine caching research has been on result set and inverted list caching [Markatos, 2001; Lempel and Moran, 2003; Baeza-Yates and Saint-Jean, 2003], that
is, no work has examined caches for the query evaluation process as a whole.

This work addresses the following research questions:

- Can the repetition in query logs be exploited to improve the speed of query evaluation in search engines?

- Does access to the various data sources utilised during query evaluation skew with query repetition? If so, can such skew be used to improve the effectiveness of search engine caching techniques?

- Can the search results of past queries provide additional information that can be used to improve the quality of future search results?

We address the questions above by proposing techniques that incorporate past user queries into search engine data structures and algorithm design. Through knowledge of past user queries, we reduce search engine query evaluation time, without reducing the quality of the search results. Our techniques are applied at various levels of search engine architecture. We begin with a proposal for a new inverted list design. We also propose a caching technique that considers various query time data structures. Finally, we propose techniques to improve search accuracy based on the properties of past user queries.

1.3 Thesis Overview

Index design can greatly affect the performance of a search engine [Scholer et al., 2002]. Through index compression, space and time efficiencies can be gained that reduce query time. In Chapter 3 we investigate a novel technique for index organisation that reflects past user queries in the index structure. Our design is based on the observation that, over a large number of queries, only a fraction of the documents in a collection are ranked highly (by the search engine) in response to those queries. Moreover, as for other trends such as the frequency of occurrence of document and query words, the access frequencies follow an inverse power-law distribution. Our access-ordered approach takes advantage of this phenomena: postings for documents that are frequently returned in response to queries are stored at the beginning of the lists, while less-frequently retrieved documents are stored towards the end. Query evaluation stops processing an inverted list when a heuristic threshold is reached, with the aim of returning accurate results in less time than a conventional approach.

Access-ordering cannot take advantage of some well-known index compression techniques [Zobel and Moffat, 2006], because reorganisation of inverted lists reduces the ability to com-
1.3. THESIS OVERVIEW

Pact the index. We examine the impact of access-ordering on index size and propose various techniques to compress the modified index. We show that a compromise between document- and access-ordering works best, with only a minor increase in index size. A further benefit of efficient index design is the capacity for early termination of query processing. We experiment with several dynamic query time pruning techniques and show that access-ordering leads to query evaluation that is 39% faster than a standard document-ordered approach, while accuracy remains almost unchanged.

Our proposed index organisation is dependent on user queries. As such, it is possible that the performance of indexes based on access-ordering may deteriorate over time as patterns of user queries change. In Chapter 4 we extend our exploration of access-ordering and examine the durability of the index. That is, if the index ordering is based on past user queries, how long will the reorganisation be able to process queries efficiently if user requests change over time? We explore the effect of training set size and show that an access-ordered index based on as few as 100,000 queries can be used to produce results that are almost equivalent to that of an index ordered by 20 million queries. Given the durability of the ordering, we re-explore the issue of index compression with an emphasis on collection reorganisation. We propose a collection reorganisation based on access-ordering that provides the compression benefits of document-ordering while allowing dynamic query time pruning. Our results show that our access-reordered indexes are up to 73% faster than the baseline approach with less than 2% increase in index size, while returning highly similar query results.

In Chapter 5 we show that access-ordering can be a valuable component of a two-tiered search structure, where a much smaller access-pruned index is used to process queries that our system predicts to be easy to evaluate. By statically pruning lists at index construction time, a small cache friendly index can be created that supports rapid query evaluation for a sub-class of queries, while all other queries are evaluated by a full index. We employ two query difficulty predictors to select queries for evaluation by our access-pruned index. Using this approach, we demonstrate that for selected “non-difficult” queries, an access-pruned index can be used to evaluate queries up to 79% faster than a conventional index.

In Chapters 3, 4, and 5 we use past queries to reorganise search engine indexes to support faster query evaluation through reduced access to the index. The success of this approach is based on the repeated access of documents by the search engine over time. An alternative technique that capitalises on the frequent repetition of data access is caching. Previous work has proposed caching the data structures that are used by a search engine at query evaluation time. However, to date such work has focused on caching individual data structures such
as the inverted lists or the result pages. No work has considered access to the various other structures used by the search engine that exhibit similar repetition in access over time. Further, while such work empirically demonstrates the effectiveness of their approaches, they are dependent on the differing architectures, and on the collection sizes on which the experiments are run.

In Chapter 6 we propose a cost model for search engine caching that estimates the benefits of caching for differing hardware configurations. We use our model to explore the benefits of caching differing search engine data structures. We then propose a generic cache management policy that supports multiple query time data structures. Our policy is validated through simulation using a test collection and query log. Our results show that a cache that is approximately 10% the size of the index, and aware of search engine data structures, is capable of reducing disk access time by more than 30% over a conventional index without caching.

The primary body of work in this thesis focuses on search engine efficiency. However the skew in document access (over past user queries) that is used to reorder the index in earlier chapters may also be used to improve query accuracy. In Chapter 7 we consider document access frequencies as a form of query independent evidence. We apply this evidence to the estimation of query to document similarity. Although our analysis of the evidence suggests that access frequencies correlate with relevance, our application of this information to reordering of search results is shown to be ineffective. We also propose a new query difficulty prediction methodology, based on the discriminatory power of queries to produce results that differ significantly from the documents commonly returned in response to the “average query”, however our results show limited success.

In Chapter 8 we discuss the implications of our results and suggest directions for further work. We have shown that the information provided by users in the form of query logs provide a valuable source of information that should be used by search engine designers. In this work we present a selection of novel techniques that incorporate past user queries into the query evaluation process, thus reducing query processing times.
Chapter 2

Background

The diverse range of user needs in accessing digital information has resulted in a substantial amount of research in the fields of information systems, and more specifically, information retrieval. In this chapter we examine current and past research in the field of text-based information retrieval, focusing on web search.

2.1 Queries and Results

Before we examine the techniques employed by modern search systems to efficiently evaluate user queries, we need to understand the way in which users interact with these systems.

Users pose queries to describe an information need. That is, a user composes a query based on an understanding of the information for which they are searching. In the context of text search, queries are typically expressed in natural language, that is, as a sequence of words. The search engine takes the user query and extracts a set of query terms. The definition of a term can vary between systems, but is generally defined as a sequence of alphanumeric characters [Spink et al., 2001; Williams and Zobel, 2005].

Once formed, users submit their queries to a search system, and receive results. Results are typically presented as lists of documents that best match the user’s query by criteria that are defined by a similarity metric. For web search, systems return a ranked list of matching document links, with typically ten links listed per page of results. Each listed item is presented with a summary of the document, and where possible, the summary is biased to the user query [Tombros and Sanderson, 1998].

While most information retrieval tasks focus on the retrieval of whole documents, some researchers have investigated passage-based information retrieval [Salton et al., 1993; Liu and
Croft, 2002], and XML retrieval research has explored techniques that allow the retrieval of information segments from structured data [Fuhr et al., 2005; 2006]. Further, information retrieval is not limited to text, recent years have seen growth in the volume of work focusing on image, video, and sound retrieval [Smeaton, 2005]. In this thesis we concentrate on text document retrieval.

The information needs of users vary. Broder [2002] proposes a taxonomy that classifies web queries into three categories: navigational, informational, and transactional. Navigational queries are those for which a user is interested in reaching a particular webpage or document. Informational queries provide a user with information regarding a given topic; in this case the user may be interested in more than a single document. Finally, transactional queries are aimed at locating services with which the user will interact.

An information need is often not satisfied with a single query. A query session can involve multiple requests by the user, where for each request the user may choose to reformulate their query. Query reformulation can occur in several ways, with users adding, removing, or substituting query terms [Jansen et al., 2000]. In Section 2.1.1 we survey studies that analyse user queries.

The expressive power of queries can vary with each search system. In general queries are presented as a bag-of-words, however many systems offer further functionality such as Boolean and phrase querying, as well as other advanced query operators. Boolean query evaluation returns the set of documents that match a set of user defined predicates. Examples of such predicates include AND, OR and NOT. Phrase querying allows a user to specify a sequence of terms that must be in sequence in the matching documents. Other advanced query operators may include the functionality to force a term to be present or absent in a document; proximity restrictions, such as having term A within ten words of term B; and the ability to restrict the search to a specified range of documents, for example to those drawn from a particular domain. Most modern search systems default to ranked bag of words queries, where the user lists words that are related to their information need, and are presented with a ranked set of result documents.

2.1.1 Studies of User Query Logs

The need to find information on the web has resulted in the availability of several publicly accessible search systems. With billions of users, popular search engines process several millions of queries per day [Jansen and Spink, 2005; 2006]. A byproduct of the large volume
of user queries are *query logs*. A query log is a capture of the stream of queries submitted to a search engine by users of the system. Query logs at a minimum record the user query, but may also record:

- a timestamp for when the query was issued;
- an IP address for the computer from where the query was issued, or alternatively a unique identifier that anonymizes the IP address;
- the sequence of ranked results requested (this is useful if the user is interested in results beyond the first page); and
- the URLs of the documents that the user found worth examining (this is also referred to as *click-through data*).

At times, commercial search systems have made query logs available for research purposes. Table 2.1 lists some of the query logs that in the past have been made available for analysis.

**Table 2.1:** *Query logs that have been analysed by research groups in recent years. NR indicates that the value is not reported in the related study.*

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Queries</th>
<th>Terms</th>
<th>Sessions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excite’97-51K</td>
<td>Mar. 1997</td>
<td>51,474</td>
<td>113,793</td>
<td>40,068</td>
</tr>
<tr>
<td>Excite’97-1M</td>
<td>Sep. 1997</td>
<td>1,025,910</td>
<td>2,216,986</td>
<td>211,063</td>
</tr>
<tr>
<td>Fireball</td>
<td>Jul. 1998</td>
<td>16,252,902</td>
<td>NR</td>
<td>NR</td>
</tr>
<tr>
<td>AltaVista’98</td>
<td>Aug.–Sep. 1998</td>
<td>993,208,159</td>
<td>NR</td>
<td>285,474,117</td>
</tr>
<tr>
<td>Excite’99</td>
<td>Dec. 1999</td>
<td>1,025,910</td>
<td>1,500,500</td>
<td>325,711</td>
</tr>
<tr>
<td>BIWE</td>
<td>May 2000</td>
<td>105,786</td>
<td>116,953</td>
<td>71,810</td>
</tr>
<tr>
<td>AlltheWeb’01</td>
<td>Feb. 2001</td>
<td>451,551</td>
<td>1,350,619</td>
<td>153,297</td>
</tr>
<tr>
<td>Excite’01</td>
<td>Apr. 2001</td>
<td>1,025,910</td>
<td>1,538,120</td>
<td>262,025</td>
</tr>
<tr>
<td>AlltheWeb’02</td>
<td>May 2002</td>
<td>957,303</td>
<td>2,225,141</td>
<td>345,093</td>
</tr>
<tr>
<td>AltaVista’02</td>
<td>Sep. 2002</td>
<td>1,073,388</td>
<td>1,073,338</td>
<td>369,350</td>
</tr>
</tbody>
</table>
2.1.2 Repetition and Reformulation

Studies of query logs have explored many aspects of user search. Silverstein et al. [1999] examine the AltaVista’98 log and report that users enter queries with an average query length of 2.35 terms. They also report that the 25 most common queries — that is 0.00000016% of all queries — account for 1.5% of all unique queries. They show that 63.7% of the queries are unique, and that when pairs of terms appear together in queries frequently, they do so as part of a phrase.

Jansen et al. [2000] perform a similar analysis of the Excite’97-51K query log and show that 35% of queries are unique queries, while 22% are modified, and the remaining 43% are repeat queries. Modified queries are those that have been adjusted by the user after receiving a result set, while repeat queries request a sequential page of results. They explore how users refine their queries and show that 34.7% of modifications are word substitutions, while 19% of modifications are single word additions and 9.5% a double word additions. Word deletions represent 24.6% of the refinements, with 16.3% as single word deletions, and 8.3% as double word deletions. The results suggest that users prefer to refine queries by adding terms. They analyse individual term use and show that the 74 most common query terms account for 18.2% of all search terms.

Ross and Wolfram [2000] further examine the co-occurrence of query terms using clustering techniques, finding that the largest groupings of queries focus on the categories of sex, groups, places and pictures.

We assert that the repetition of both entire queries and individual query terms within query logs can be exploited to improve search engine efficiency. In Chapter 3 we show that the redundancy in user queries leads a skew in document access. We then use the access pattern as the basis of an index organisation that allows reduced query evaluation time. Index design is discussed in Section 2.4.

2.1.3 Advanced Operators and Pages Examined

Spink et al. [2001] extend their previous study with the larger Excite97-1M query log and verify many of their previous findings. They report that few users look beyond the first two pages of results, with 42% and 19% of users considering only the first and first two pages respectively. They observe that advanced query operators are rarely used with only 5% of queries using query modifiers.

Hölscher and Strube [2000] conduct a user study to explore the differences in query
formulation between novice and expert users. They report that advanced users express queries with more terms, and are more likely to use advanced query operators.

Spink et al. [2002] contrast the AllTheWeb’01 query log with the Excite’97-1M to highlight the differences between US and European users. They report that European users tended to be more patient in their searching patterns, leaning towards longer query sessions and viewing more result pages per query. European users averaged 2.9 queries per session, compared to US users with 2.3 queries per session, while the average number of pages viewed per query by Europeans was 2.2 compared to 1.7 by US users. They also noted that the European users query with a broader range of terms, showing that the 100 most frequent query terms account for 14% of all query terms in the AllTheWeb’01 log compared to 22% in the Excite’97-1M log. Cacheda and Vina [2001] examine the BIWE query log (from a Spanish search site), and show even less patience by searchers, with 68% only examining the first page of results, and 13% examining the second.

The reduction in pages viewed can perhaps be attributed to improved search technology. This view is supported by a later study of Jansen and Spink [2005] that suggests a trend towards the use of less complicated queries. In the study, European query logs from AllTheWeb are compared, where an increase in one term queries from 25% to 33%, and a reduction in the number of queries modified from 47% to 41% is shown.

With an increasing trend towards short queries, and users examining on average less than three pages of search results, the demand for search systems that produce accurate results with minimal information is increasing. In Chapter 7 we examine techniques that attempt to predict the effectiveness of queries, and then improve the quality of search results based on query independent evidence. Difficulty prediction is discussed further in Section 2.12, and query independent evidence is discussed in Section 2.13.

2.1.4 Changes in Query Patterns Over Time

Jansen and Spink [2006] compare the results of nine query log studies, and note trends in user search over time. They find that between 1997 and 2002 the percentage of users that consider only the first result page has increased from 29% to 73%. They also show a change in the subject matter sought, with a significant increase in searches for people, places and things, and a significant drop in searches for entertainment or recreation and sex and pornography. It should be noted that these results are reported across differing search systems. As such, it is not clear what proportion of the changes can be attributed
to different user behaviour, and what can be attributed to the difference between the search systems.

A more consistent analysis is that of Jansen et al. [2005], who compared AltaVista query logs from 1998 and 2002. Contrary to the nine query log study, they found that users have begun to interact more with search engines. They show an increase in query length from 2.35 terms per query, to 2.92 terms per query, and a related increase in session length with a decrease in single query sessions from 78% to 48%. A shift in user views about the web is suggested, with a move towards using the web as a source of general information rather than for entertainment purposes. This is shown with the marked increase in users requesting information on topics such as commerce, travel, employment, and economy. The increases in session and query lengths suggest that users find it more difficult to reach the data that they are searching for on the web.

The comparison of different query logs spanning multiple years has produced mixed results. However, despite the changes, the average length of a query remains less than three terms. Session length is skewed towards single query sessions, and a small proportion of users seem to examine more than one page of search results.

Beitzel et al. [2004] also examine temporal properties of queries. They examined the changes in query traffic during various hours of the day and found that during peak (afternoon and evening) hours, the volume of queries submitted increase. They also categorised queries into various topics and measured the changes in query volume for each category over the day. Their results show that topics such as music and entertainment exhibit a high volume of throughput in the early hours of the morning, then rapidly decline during the day, slowly rising again in the evening, while topics such as personal finance exhibit low volumes overnight, with a peak during the morning hours. However, not all topics exhibit such behaviour. Topics such as health remain relatively constant throughout the day.

The temporal analysis of query logs suggests that query patterns vary. Some queries occur regularly at a low frequency, while others can occur at high frequency irregularly. Such properties can have a significant impact on search engine caching techniques. Search engine caching techniques are discussed further in Section 2.11.

2.1.5 Implications of Query Log Analysis

The study of query logs has resulted in many discoveries about the ways in which users interact with search systems. Such information is valuable as it highlights the expectations
2.1. QUERIES AND RESULTS

Table 2.2: Attributes of query logs used throughout this thesis.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Queries</th>
<th>Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anon</td>
<td>2003 – 2005</td>
<td>576,649</td>
<td>1,060,666</td>
</tr>
<tr>
<td>Lycos.de</td>
<td>2003</td>
<td>186,589,623</td>
<td>375,999,907</td>
</tr>
<tr>
<td>MSN-Search</td>
<td>2003</td>
<td>569,239,727</td>
<td>1,501,704,646</td>
</tr>
<tr>
<td>MS-Gov</td>
<td>2005</td>
<td>2,097,447</td>
<td>6,520,345</td>
</tr>
</tbody>
</table>

of users, and allows researchers and developers to focus on producing systems that best match user needs. While it is at times unclear what proportion of the observed change is due to different user behaviour, and what proportion is due to differences in the search systems, several trends remain consistent among the various studies. For the purposes of web search, it is clear that the average query length is short, and that users tend to only examine one or two pages of results. Further, there is a large degree of repetition between user queries, both for entire queries and for the terms within each query. In this work we take advantage of the redundancy in query logs to improve efficiency and effectiveness.

Our experiments make use of the Excite query logs used by Jansen and Spink as well as a set of additional query logs listed in Table 2.2. Session information was not available for any of the additional query logs. The Anon query log contains all queries submitted to the on-site search feature of anon.edu.au between 2003 and 2005, where Anon is a large Australian academic institution, whose name has been anonymized for reasons of confidentiality. The Lycos.de log contains almost 200 million time-ordered queries from the German portal of the Lycos web search engine [Lycos]. Due to its origin, a high proportion of these queries contain German terms. Finally, two query logs are provided by Microsoft [MSN]. The first log, labelled MSN-Search, contains over 500 million queries ordered from most to least frequently occurring. Queries that occurred less than three times during the period in which the queries were gathered were not included in the log. Due to the format, this log does not provide a time reference for each query and does not show trends in query use.

The second Microsoft query log, labelled MS-Gov, is a time-ordered query log, composed of queries where the user selected a result document that was ranked within the top three results, and was located within the .gov domain. The log records a time-stamp for each query and click-through data. However, as only queries that resulted in a user click are recorded, the data is sparse.
2.2 Measuring Effectiveness

Before turning to finer details of search engine design, we first examine how retrieval systems can be evaluated. Several metrics to measure the effectiveness of search systems have been proposed, with each metric varying in the emphasis of the qualities measured.

2.2.1 Precision and Recall

Two intuitive measures for retrieval systems are the notions of recall and precision. Recall measures how many of the relevant documents available are returned by a search system, while precision measures how many of the retrieved documents are relevant. A relevant document is one that satisfies the user’s information need. Explicitly, recall is measured as:

\[
\text{Recall} = \frac{|R_l \cap R_t|}{|R_l|},
\]

while precision is:

\[
\text{Precision} = \frac{|R_l \cap R_t|}{|R_t|},
\]

where \(R_l\) is the set of relevant documents, and \(R_t\) is the set of retrieved documents.

Recall is an important measure for systems where the completeness of results is important. This may be the case for legal systems where lawyers seek to find all information about a specific topic. Conversely, precision is important for systems where users are not necessarily concerned with completeness, but do want access to a high proportion of relevant documents in the result set. Precision is often used when dealing with large collections, as recall is both costly and difficult to compute, requiring the evaluation of large numbers of documents.

One issue with the precision measure is that it can be taken over any number of returned results. For comparison purposes, it is ideal to measure precision over the same number of retrieved results per system. In TREC (discussed below), it is common to measure precision over 100 or 1,000 results per query. While this approach works well in a research environment, it is possible that it does not reflect all real world search engine users. Jansen and Spink [2006] report that 73% of users examine only one page of web search engine answers. As such, for web search it is useful to measure precision at 10 and 20 results reflecting the average number of answers examined by a user.
Further, in the case of web search, precision is likely to be preferred over recall as users are interested in receiving a high proportion of relevant documents in the few pages of results examined, but are less interested in completeness.

2.2.2 Reciprocal Rank

Some retrieval tasks are focused on finding a single relevant document. For example, such navigational type tasks include named- and home-page finding, where the user desires a single page in the collection. For such tasks it is ideal for the relevant document to be ranked as high as possible in the results set. One measure for such tasks is reciprocal rank, which measures the reciprocal of the rank at which the first relevant document was found [Shah and Croft, 2004]. This is defined as:

\[ \text{Reciprocal Rank} = \frac{1}{r}, \]

where \( r \) is the rank of the first relevant document in the result set. Over a set of queries the mean of reciprocal ranks for each query is referred to as the mean reciprocal rank (MRR).

2.2.3 Single Value Measures

Several metrics have been proposed that attempt to combine recall and precision. R-precision measures the number of relevant documents observed after retrieving \( |R_l| \) results, where \( |R_l| \) is the number of relevant documents for that query. A high value of R-precision indicates both high recall and high precision.

An alternate way of combining recall and precision is by using average precision. Average precision is calculated by averaging a set of precision values over the retrieved result set. For each relevant document, precision is calculated based on where that document is observed, and the mean of all precision values is taken:

\[ \text{Average Precision} = \frac{\sum_{r \in R_l} \text{Precision}(r)}{|R_l|}, \]

where \( \text{Precision}(r) \) is the precision at \( r \) retrieved results. Average precision reports a single value for an individual query. Mean average precision (MAP) combines the average precision scores of multiple queries to produce a single value over a query set.
2.2.4 TREC Project

A rich set of metrics to evaluate effectiveness is of little use without a common testbed in which to evaluate each system, thus allowing comparisons in a uniform environment. The Text Retrieval Conference (TREC) aims to encourage research in the area of information retrieval [Voorhees and Harman, 2005]. A large part of TREC’s contribution to information retrieval research is the provision of standard collections, queries, and relevance judgements for use by its participants.

Each year the TREC conference organises a set of tasks that focus on varying aspects of information retrieval. Tasks vary from genomic and legal search tasks, to spam detection and question-answering. For each track, a standard collection is provided along with a set of queries for the participants to resolve. Participants generate result sets and return them to the TREC organisers who then pool the results and produce relevance judgements [Voorhees, 2001]. The pooled judgements are then used to evaluate the performance of each system. The judgements are also made available so that participants can refine their systems for participation at the following TREC conference.

Query topics consist of a unique numeric identifier, a query title, a description, and a narrative. The titles are often actual queries extracted from web query logs, while descriptions and narratives contain information added to expand on the information needs of the user. In Chapters 3, 4 and 5 we make use of 100 query topics (numbered 451–500 and 501–550 from TREC-9 and TREC-10 respectively) and use only the query title field in the search process.

Along with query topics, the TREC consortium provides relevance judgements that list those documents that are deemed to be relevant for each query topic. The relevant documents are chosen from a pool of results that are returned by TREC participants. For topics 451–500 the mean pool size per topic was 1,401, with a mean of 52.34 documents judged to be relevant per topic. For topics 501–550 the mean pool size per topic was 1,408, with a mean of 67.26 judged relevant documents per topic. Although not all documents in the collection are tested for relevance, Zobel [1998] found that the pooling technique used by TREC is adequate for the domain in which it is used. After the runs are assessed, the relevance judgements are publicly available for research purposes. Relevance judgements are lists of documents that specify whether the information need described by each topic is, or is not satisfied by each assessed document.
Table 2.3: TREC collections used in this thesis.

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
<th>Size</th>
<th>Documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>WT10G</td>
<td>Web data (1997)</td>
<td>10 GB</td>
<td>1,692,096</td>
</tr>
<tr>
<td>GOV</td>
<td>Crawl of .gov domain (2002)</td>
<td>18 GB</td>
<td>1,247,753</td>
</tr>
<tr>
<td>GOV2</td>
<td>Crawl of .gov domain (2004)</td>
<td>426 GB</td>
<td>25,205,179</td>
</tr>
</tbody>
</table>

DOC #612
FERDINAND.
This is a most majestic vision, and
Harmonious charmingly; may I be bold
o think these spirits?

DOC #613
PROSPERO.
Spirits, which by mine art
I have from their confines call’d to enact
My present fancies.

DOC #614
FERDINAND.
Let me live here ever:
So rare a wonder’d father and a wise,
Makes this place Paradise.

DOC #615
[JUNO and CERES whisper,
and send IRIS on employment.]

Figure 2.1: Sample documents from Tempest collection.

TREC Collections

To date, TREC has contributed several collections for use in information retrieval research. Some notable collections are listed in Table 2.3. The TREC collections vary in type and volume. Earlier collections were composed of mostly news articles, while later collections are drawn from the web.

In this work we primarily use the WT10g and GOV2 collections. The WT10g collection contains 1.6 million documents in 10GB of uncompressed data. The average document size is 6.2KB. The collection represents a subset of the documents taken from an Internet Archive 1997 crawl. The GOV2 collection contains 25 million documents in over 400GB of data with an average document size of 17.7KB. The collection represents a complete crawl of the .gov domain.
CHAPTER 2. BACKGROUND

| fairy  | (642, 2) | (649, 1) | (649, 1) | (649, 1) |
| farewell | (47, 2)     | (434, 3) | (464, 1) | (464, 1) |
| fish | (283, 1)     | (386, 6) | (430, 1) | (436, 1) | (480, 2) | (750, 1) |
| state | (77, 2)    | (79, 1)  | (471, 1) | (606, 1) | (606, 1) |
| storm | (22, 1)     | (386, 3) | (408, 2) | (408, 2) | (408, 2) |
| swords | (565, 1)   | (566, 2) | (566, 2) | (566, 2) | (566, 2) |
| sycorax | (128, 1)  | (132, 1) | (136, 1) | (159, 2) | (511, 2) |
| vision | (612, 1)  | (624, 1) | (718, 1) | (718, 1) | (718, 1) |

Figure 2.2: Selected inverted lists in a document-level index from Tempest collection.

| fairy | (642, 2, [4, 11]) | (649, 1, [14]) |
| farewell | (47, 2, [11, 16]) | (434, 3, [2, 4, 5]) | (464, 1, [10]) |
| fish | (283, 1, [64]) | (386, 6, [77, 84, 89, 107, 122, 203]) | (430, 1, [13]) | ... |
| state | (77, 2, [39, 85]) | (79, 1, [52]) | (471, 1, [32]) | ... |
| storm | (22, 1, [19]) | (386, 3, [16, 217, 254]) | (408, 2, [29, 44]) |
| swords | (565, 1, [6]) | (566, 2, [29, 66]) |
| sycorax | (128, 1, [13]) | (132, 1, [26]) | (136, 1, [50]) | ... |

Figure 2.3: Selected inverted lists in a word-level index from Tempest collection.

2.3 Example Collection: The Tempest

Throughout this work, we present several techniques and propose optimisations for text retrieval. At times, techniques are illustrated with a sample collection. The criteria for our sample collection is that it be detailed enough to illustrate the benefits of the various techniques discussed, yet that it also be simple enough to allow the reader to conceptualise the impact of each approach with regard to the collection as a whole. With these goals in mind, we make use of The Tempest by Shakespeare [1564–1616]. For simplicity, we treat each paragraph in the text as a separate document. Figure 2.1 illustrates some sample documents from the collection. Based on our partitioning of the text, the collection has 775 documents, including 17,505 terms of which 3,231 are unique. The average document length is 23 terms.
2.4 The Inverted Index

For search systems to service requests efficiently, they must have rapid access to candidate results. An inverted index is a data structure that provides direct access to the locations at which terms appear within a document collection. Akin to the index found in a text book, an inverted index records, for each term in the collection, the documents in which the term occurs. Inverted indexes are an essential component of all web search engines and text retrieval systems [Zobel and Moffat, 2006].

An inverted index has two components: first, a vocabulary of the terms that occur in the searchable collection, and second, a set of postings that list the locations of occurrence of the search terms. Typically, searchable terms are words extracted from the text [Williams and Zobel, 2005].

For each term, there is one posting for each document that contains the term. Zobel and Moffat [1998] proposed a notation to formally describe such postings: each term $t$ has postings $\langle d, f_{d,t} \rangle$, where $f_{d,t}$ is the frequency of $t$ in document $d$. An inverted list is the set of postings for a single term of the collection. Inverted lists take the form:

$$\langle d_1, f_{d_1,t} \rangle, \langle d_2, f_{d_2,t} \rangle, \ldots, \langle d_{n-1}, f_{d_{n-1},t} \rangle, \langle d_n, f_{d_n,t} \rangle,$$

where $d_n$ is the largest document identifier in the set of postings.

Although other data structures, such as signature files [Faloutsos, 1985], have been proposed for search systems in the past, inverted indexes have been shown to be more functional and efficient. Zobel et al. [1998] compare signature files with inverted indexes, and present results showing the benefits of the later for most search related tasks.

Figure 2.2 shows a fraction of the inverted index for the Tempest collection from Section 2.3. In the inverted list for the term “storm”, we can see that the term appears once in document 22, thrice in document 386, and twice in document 408.

The internal document identifiers assigned by the search system can be resolved into disk locations using a document mapping table. The document mapping table is constructed as the collection is parsed at indexing time.

This basic document-level inverted index structure is sufficient to support the popular ranked and less-popular Boolean query evaluation models. However, a document-level index cannot support query types such as phrase queries, where the ordering and adjacency of words determines which documents are matches. Nor can it support ranking methods based on term proximity. To support such fine-grained measures, word position information is...
also stored in the postings lists. Extending the notation of Zobel and Moffat, each posting becomes a triple of the form:

\[(d, t, f(d, t), [o_1, \ldots, o_{f(d, t)}])\]

where \(d\), \(t\), and \(f\) are as previously described, and the \(o\) values are the positions in \(d\) at which \(t\) occurs. We refer to an inverted index containing term offset information as a word-level index.

Figure 2.3 shows a fraction of the word-level inverted index for the Tempest collection. We can now see that the term “storm” appears once in document 22, at the 19th position, thrice in document 386, at the 16th, 217th and 254th positions, and finally twice in document 408, at the 29th and 44th positions.

For the majority of terms, the number of postings are small. It is not uncommon for a large collection to have a large proportion of terms that occur only once. In the Tempest collection, 1,935 terms, that is 59.9\% of the unique terms in the collection, appear only once.

Conversely, for common terms the number of postings in each inverted list can be significantly large. Returning to the Tempest collection, the five most common terms: “and”, “i”, “the”, “to” and “a”, appear in 26.1\% to 35.1\% of the documents in the collection, and individually account for 1.8\% to 3.0\% of all term occurrences.

### 2.4.1 Inverted List Compression

The efficiency of ranked querying is highly dependent on the effective organisation and storage of postings lists. Conventionally, postings lists are organised so that the postings are sorted by increasing document order. As postings are processed sequentially, this permits

<table>
<thead>
<tr>
<th>Term</th>
<th>((d, t, f(d, t), [o_1, \ldots, o_{f(d, t)}]))</th>
</tr>
</thead>
<tbody>
<tr>
<td>fairy</td>
<td>((642, 2, 7, 1))</td>
</tr>
<tr>
<td>farewell</td>
<td>((47, 2, 387, 3, 30, 1))</td>
</tr>
<tr>
<td>fish</td>
<td>((283, 1, 103, 6, 44, 1, 6, 1, 44, 2, 270, 1))</td>
</tr>
<tr>
<td>state</td>
<td>((77, 2, 2, 1, 392, 1, 135, 1))</td>
</tr>
<tr>
<td>storm</td>
<td>((22, 1, 364, 3, 22, 2))</td>
</tr>
<tr>
<td>swords</td>
<td>((565, 1, 1, 2))</td>
</tr>
<tr>
<td>sycorax</td>
<td>((128, 1, 4, 1, 4, 1, 23, 2, 352, 2))</td>
</tr>
</tbody>
</table>

*Figure 2.4: Selected inverted lists in a document-level index from Tempest collection with difference compaction.*
differences to be taken between adjacent document identifiers prior to storage on disk. Stor-
ing differences between document numbers improves the ability to compress lists by reducing
both the magnitude and range of values stored [Zobel and Moffat, 2006]. An inverted list
with difference compaction takes the form:
\[
(d_1, f_{d_1}, t), (d_2 - d_1, f_{d_2}, t), \ldots, (d_{n-1} - d_{n-2}, f_{d_{n-1}}, t), (d_n - d_{n-1}, f_{d_n}, t)
\]
An original value is restored by summing all previous values. Figure 2.4 shows document-
level lists for the Tempest collection, with difference compaction. For example, for the term
“storm” the second posting document is obtained by taking the sum of 364 and 22, which
gives the original value of 386.

A similar approach can be followed for the offsets used in a word-level index:
\[
(d, f_{d,t}, [o_1, o_2 - o_1, o_3 - o_2 \ldots, o_{f_{d,t}} - o_{f_{d,t-2}}, o_{f_{d,t}} - o_{f_{d,t-1}}])
\]
For the word-level list of the term “storm”, the list becomes:
\[
(22, 1, [19]), (364, 3, [16, 201, 37]), (22, 2, [29, 15])
\]
This indicates that, for example, the term occurs in document 386 at word positions 16, 217
and 254.

With differences taken, various coding techniques can be employed to compress the in-
tegers that compose the index. Traditional bitwise coding techniques [Golomb, 1966; Elias,
1975] have been shown to effective, but less efficient [Trotman, 2003]. We use a purpose-built
variable-byte integer compression scheme that is designed for very fast decoding [Williams
and Zobel, 1999; Scholer et al., 2002]. More recently, Anh and Moffat [2005a] have proposed
word aligned codes that combine the benefits of bitwise compression with fixed-width align-
ment, allowing for rapid access through the encoded data. While we do not use this more
recent coding technique, we note that it is compatible with the work reported here.

Improving compression improves query evaluation times for two reasons: first, since disk
bandwidth is a bottleneck in modern retrieval engines, compression allows more data to be
transferred per disk read than when data is uncompressed; and, second, compression permits
more data to be stored in main-memory, improving caching effects. Indeed, Scholer et al.
[2002] show that compressing postings lists reduces average query evaluation times to around
one-third of that of an uncompressed representation. However, a disadvantage of document-
ordered lists is that each must be decoded in its entirety in response to a query. As document
identifiers are assigned incrementally as documents are observed, the position of a document
in the postings list has little correlation with its similarity to most queries.
A search engine is composed of the data structures that form the index, and the processes that interact with the index components. Several works have explored the architecture of efficient search engines [Arasu et al., 2001; Brin and Page, 1998; Zobel and Moffat, 2006]. Figure 2.5 models the interaction of the components within a search system.

The index is composed of four key structures. First, the vocabulary structure provides access to the set of terms that appear in the collection of documents that are indexed. At a minimum, the vocabulary maintains a pointer to the inverted list of each term in the collection. The vocabulary also maintains statistics about each term such as the number of documents in which the term appears, or the total number of occurrences of the term in the collection. Table 2.4 provides an example of the data that would be found in a vocabulary. For example, we can see that the term “swords” occurs a total of three times within two distinct documents.
Table 2.4: Vocabulary entries for selected terms in Tempest collection. For each term the vocabulary records the document- and collection-frequency of the term, the location of the term’s inverted list on disk, and the size of the list.

<table>
<thead>
<tr>
<th>Term</th>
<th>$f_t$</th>
<th>$F_t$</th>
<th>Offset</th>
<th>List Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>fairy</td>
<td>2</td>
<td>3</td>
<td>12718</td>
<td>8</td>
</tr>
<tr>
<td>farewell</td>
<td>3</td>
<td>6</td>
<td>12828</td>
<td>13</td>
</tr>
<tr>
<td>fish</td>
<td>6</td>
<td>12</td>
<td>13537</td>
<td>26</td>
</tr>
<tr>
<td>state</td>
<td>4</td>
<td>5</td>
<td>37739</td>
<td>15</td>
</tr>
<tr>
<td>storm</td>
<td>3</td>
<td>6</td>
<td>38128</td>
<td>14</td>
</tr>
<tr>
<td>swords</td>
<td>2</td>
<td>3</td>
<td>38858</td>
<td>8</td>
</tr>
<tr>
<td>sycorax</td>
<td>5</td>
<td>7</td>
<td>38878</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 2.5: Document mapping table entries for selected documents in the Tempest collection. For each document, the table records the length of the document in bytes and terms; the count of unique terms in the document; the location of the document on disk; and the weight assigned to the document by the document ranking function.

<table>
<thead>
<tr>
<th>Doc. ID</th>
<th>Length</th>
<th>Words</th>
<th>Unique Words</th>
<th>Offset</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>611</td>
<td>294</td>
<td>45</td>
<td>39</td>
<td>84050</td>
<td>7.055118</td>
</tr>
<tr>
<td>612</td>
<td>125</td>
<td>18</td>
<td>18</td>
<td>84344</td>
<td>4.220472</td>
</tr>
<tr>
<td>613</td>
<td>117</td>
<td>17</td>
<td>17</td>
<td>84469</td>
<td>4.094488</td>
</tr>
<tr>
<td>614</td>
<td>116</td>
<td>18</td>
<td>17</td>
<td>84586</td>
<td>4.283464</td>
</tr>
<tr>
<td>615</td>
<td>69</td>
<td>9</td>
<td>8</td>
<td>84702</td>
<td>3.118110</td>
</tr>
</tbody>
</table>

Second, for each term in the vocabulary, an inverted list of postings is stored that records the locations of where that term appears in the collection. Inverted lists were discussed in Section 2.4.1.

Third, a document mapping table records statistics for each document such as document length, the number of distinct terms in the document, and the location of the document on disk. Table 2.5 shows sample entries in a document mapping table. For example, in the table we see that document number 612 contains 18 words, and has a length of 125 bytes.

Finally, some form of the collection is optionally maintained to generate the document snippets that are often presented with the query results.
Several processes interact with the index with diverse purpose. At index construction time a document parser tokenizes documents into terms. As each new document is encountered, it is assigned a document identifier, and an entry is added to the document mapping table. If required, a copy of each document is stored in the collection archive. Tokens and document identifiers are passed to the indexer. For each token, if the term has not been previously encountered, an entry is added to the vocabulary structure. If the term is not new, the vocabulary structure is updated to record another occurrence of the term. Then a posting is appended to the corresponding inverted list.

At search time, a query parser tokenizes the user query into tokens that match the format of those in the index. The query processor makes use of a document ranking metric to produce a result set. Regardless of the ranking metric employed, the following process is observed. First, for each query term the associated entry is retrieved from the vocabulary for access to term specific statistics, then its inverted list is loaded from disk. For each posting in the inverted list a partial similarity score is calculated between the document to which the posting refers and the query. If the document that the posting refers to has not been previously encountered, an accumulator is initialised with the partial score for the document. If the document has been previously encountered, its accumulator is updated with the partial score. After all query terms have been processed the set of accumulators is partially sorted to obtain the top ranked results, and document summaries are optionally produced by fetching the best matching documents from the collection archive. Result pages are then presented to the user after a query had been processed. The result pages are typically ranked by predicted similarity and often a summary or snippet of the document is presented with the results [Tombros and Sanderson, 1998].

It is clear that the evaluation of a query requires access to a wide selection of structures. Often these structures are located on disk, but optionally can also be stored in memory, or alternatively a combination of both. The size of the collection has an effect on such design decisions. For example, for small collections, the vocabulary structure can reside in main memory. However, as it has been shown that the rate at which new terms are encountered remains almost constant as collection size increases [Williams and Zobel, 2005], for larger collections it is likely that the vocabulary, or at least part of it, will reside on disk. Inverted lists form the largest part of an index, and the lists for individual terms vary in size with the lists for common language terms such as “the”, “and”, and “as” becoming extremely long. As this component forms the largest part of the index, the inverted lists are typically stored on disk. The document mapping table is directly proportional to the number of documents.
2.6. QUER Y EVALUAT I ON

in the collection. The document mapping table can reside on either disk or in main memory. Finally, result summaries of matching documents require access to the document collection itself. The document collection is significantly larger than the index and is most often stored on disk [Zobel and Moffat, 2006]. Fast access to frequently accessed on-disk data can be achieved through the use of a cache. We discuss caching further in Chapter 6.

As new pages are regularly added to the web, and existing pages change frequently [Fet terly et al., 2004], the ability to update the search engine index is crucial. Index update strategies fall into one of two broad categories: index rebuilds and dynamic updates. An index rebuild builds a new index for the entire collection, then replaces the existing index, while dynamic updates attempt to modify the existing index [Lester et al., 2005b; Büttcher and Clarke, 2005a]. To limit the complexity of the experimental environment in this work, we only consider static document collections requiring a single index build, and note that index updates for the techniques presented in this work are possible via index rebuild approaches.

2.6 Query Evaluation

Throughout the course of information retrieval research, a large body of work has focused on improving the ability of search systems to rank documents with respect to user queries. Although early publications focus on Boolean based retrieval, the inherent limitations of Boolean keyword matching prevent a search system from providing a meaningful ordering to the retrieved results [Cooper, 1988]. As such, without an exhaustive examination of the search results, users have no way of knowing where they may find the information they seek. In contrast, ranked retrieval, where documents are presented in predicted order of utility, provide users with a set of documents from which they can intuitively derive an understanding of the quality of the search results. However, to achieve this end, a mechanism by which documents can be evaluated with respect to the query is necessary. Despite the diversity of models of document ranking proposed, most ranking methods are based on similar kinds of features of documents and queries.

Primarily, three features form the building blocks of most ranking formulas. First, \textit{term-frequency}, \( f_{d,t} \), is used to indicate the importance of a term in a document. Term-frequency is a count of the number of occurrences of term \( t \) in document \( d \). The more often a term is encountered in a document, the more likely it becomes that the topic of the document is related to the term.
Second, document-frequency, $f_t$, indicates the discriminatory power of a term within a collection by measuring the number of documents in which term $t$ occurs. Most query evaluation models make a unigram term independence assumption where term co-occurrence is not considered in the ranking. The reason that such an assumption is made is that the computational complexity of bi-gram models is typically high, and to-date, the effectiveness of such models has been limited. Although some work has attempted to factor bi-grams into ranking models [Song and Croft, 1999; Srikanth and Srihari, 2002], most attempts have resulted in computationally expensive systems with at best minor improvements to system accuracy. Therefore, under the independence assumption, it is desirable to value terms that reduce the query search space. Consider the query “the king”. If we consider the terms of the query independently, then the term “the” is of little value as it is likely to appear in most English documents in the collection. However, the term “king” is likely to only appear in a smaller subset of documents. Returning to the Tempest example, the term “the” occurs in 265 documents, that is 34% of the collection, while the term “king” occurs in only 32 documents, or 4% of the collection. It is therefore likely that the term “king” is a more effective discriminator.

Finally, values such as term-frequency are inherently biased towards long documents as they are likely to contain more terms. The statistic of document length, $|d|$, measures the length of document $d$, and often used to normalise document based statistics with operations such as division. For example, it is common to see term-frequency divided by document length in several document ranking metric deviations [Zobel and Moffat, 1998].

Apart from term-frequency, document-frequency, and document length, other features (and variations of those features) are used by the many models of ranking. We will use the notation of Zobel and Moffat [2006], presented on page xii.

We now discuss three well-known ranking models, as well as common implementations of each. We begin with the vector-space model in which each document and query is represented by a multi-dimensional vector, and where similar vectors represent similar information. Then we discuss the probabilistic model where documents are ranked by the inferred likelihood that they are relevant to a given query. Finally we present the language model approach where it is assumed that each document is generated by a specific language model, and documents are ranked by the probability that the user query was generated by the same language model as that of the document model.

The vector-space and probabilistic models are used in Chapters 3, 4 and 5 to measure the effectiveness of our proposed inverted index structures, and in Chapter 7 we extend
language modelling to consider a new form of statistical information based on the likelihood of document access by the search system.

### 2.6.1 Vector-Space Model

The vector-space model is one of the earliest rank based models proposed for information retrieval. Explained by Salton et al. [1975], variations of the model are noted as being in use as early as 1962 [Zobel and Moffat, 2006]. Under the vector-space model, documents and queries are represented by \( n \)-dimensional vectors, where \( n \) is the number of distinct terms in the collection. The vector for each document \((w_{d,1}, w_{d,2}, \ldots, w_{d,n})\), and query \((w_{q,1}, w_{q,2}, \ldots, w_{q,n})\), is composed of term weight components. The vector-space model determines ranking by measuring the similarity of two vectors. A benefit of this model is that it is neatly represented with vector algebra, where the similarity coefficient between a query and document can be measured as the dot product between the two vectors:

\[
S_{q,d} = q \cdot d = \sum_{t=1}^{n} w_{q,t} \times w_{d,t},
\]

where \( w_{q,t} \) and \( w_{d,t} \) are the weights of query term \( t \) and document term \( t \) respectively.

One problem with the simple dot product measure is that it is biased towards longer documents as they have a greater chance of containing more query terms. To normalise for document length, we take the angle \( \theta \) between the two vectors which is given by the dot product when normalised by the Euclidean length of each vector. This variant is known as the cosine similarity measure:

\[
S_{q,d} = \frac{q \cdot d}{||q|| \cdot ||d||} = \frac{1}{W_q W_d} \sum_{t=1}^{n} w_{q,t} \times w_{d,t},
\]

where \( W_d \) is the Euclidean length of document \( d \) given by:

\[
W_d = \sqrt{\sum_{t=1}^{n} w_{d,t}^2},
\]

and \( W_q \) is the Euclidean length of the query \( q \):

\[
W_q = \sqrt{\sum_{t=1}^{n} w_{q,t}^2}.
\]

However, as \( W_q \) is constant for a single query, it has no effect on the ranking of the documents.
Singhal et al. found that the cosine measure is less likely to retrieve longer documents, than is indicated by their likelihood of relevance. They propose a normalisation technique for the cosine measure that favours long documents by a tunable factor $s$ relative to a pivot point $p$ [Singhal et al., 1996a;b]. Documents on one side of the pivot are down-weighted, while documents on the other are up-weighted. The value of $p$ is generally given by the average document normalisation factor, which in this case is the mean document weight $W_d$ over the collection. The pivoted cosine ranking measure is therefore:

$$S_{q,d} \simeq \frac{1}{(1-s)p + (s)W_d} \sum_{t=1}^{n} w_{q,t} \times w_{d,t}.$$ 

Several variants have been explored for determining term $w_{d,t}$ and query $w_{q,t}$ weights [Salton and Buckley, 1988; Zobel and Moffat, 1998], but no single combination has been shown to dominate in all scenarios. In this work we use the weights recommended by Zobel and Moffat [1998], where:

$$w_{d,t} = 1 + \log_e f_{d,t},$$

and

$$w_{q,t} = \log_e \left( 1 + \frac{N}{f_t} \right),$$

where $N$ represents the number of documents in the collection.

### 2.6.2 Probabilistic Model

Instead of determining the similarity between documents and a query, the probabilistic model attempts to estimate, for each document, the likelihood that it is relevant to the information need [Robertson and Sparck Jones, 1976; Fuhr, 1992]. Results are then ranked by their probability of relevance.

The underlying assumption of the probabilistic model is that terms are distributed differently between relevant and non-relevant documents. Given a set of relevant documents, weights can be assigned to terms based on their likelihood of being present or absent within relevant and irrelevant documents. This leads to the term weighting function of Robertson and Sparck Jones [1976]:

$$w_t = \frac{P(t|R)P(\bar{t}|\bar{R})}{P(t|R)P(\bar{t}|\bar{R})},$$
where $P(t|R)$ is the probability of term $t$ being present in a relevant document; $P(\overline{t}|R)$ is the probability of term $t$ being absent in a relevant document; and where $P(t|\overline{R})$ and $P(\overline{t}|\overline{R})$ are the probability of term $t$ being present or absent in a non-relevant document respectively.

Given the term independence assumption, the probability of a document being relevant is given by the product of the weight of each term:

$$P(R|q,d) = \prod_{t \in d} w_t .$$

However, as we only require an ordering of documents by probability of relevance, it is often practical to compute the sum of logarithms instead:

$$\log P(R|q,d) = \sum_{t \in d} \log w_t .$$

A key difficulty with the probabilistic model is the dependence on relevance judgements. Sparck Jones et al. [2000] propose a metric, Okapi BM25, based on probabilistic principles, but without the need for pre-judged documents. This measure computes the similarity of a document $d$ to a query $q$ that contains the terms $t$ as follows:

$$S_{q,d} = \sum_{t \in q} \log \left( \frac{N - f_t + 0.5}{f_t + 0.5} \right) \times \frac{(k_1 + 1) f_{d,t}}{K + f_{d,t}} .$$

$K$ is $k_1((1 - b) + b \times |d|/W_A)$, where $|d|$ is the document length, $W_A$ is the average document length in the collection, and $k_1$ and $b$ are constants that are set to 1.2 and 0.75 respectively. While the derivation of this formula is beyond the scope of this work, a detailed explanation of the formulation is presented elsewhere [Robertson and Sparck Jones, 1996; Sparck Jones et al., 2000], and has consistently been shown to be very effective in practice [Voorhees and Harman, 1998; Craswell and Hawking, 2004; Zaragoza et al., 2004; Craswell et al., 2005a;c].

### 2.6.3 Language Models

Language models have been used extensively in the field of data compression, optical character recognition and voice recognition [Cole, 1997]. Conceptually, a language model describes the likelihood of a token occurring in a stream of tokens. For our purposes a token is a term, but it could optionally be an individual character or group of characters, also referred to as a gram.

In information retrieval, under the language modelling approach each document is assumed to be generated by an underlying language model. Queries are also assumed to be
generated by an underlying language model [Ponte and Croft, 1998]. Documents are ranked with respect to the query by the likelihood that the document language model is the one that generated the query. This is referred to as a *query likelihood* approach.

Conversely, one can use a *document likelihood* approach, where we must estimate the probability of the query language model generating the document. However, as queries are often only a few words, they do not provide substantial information about their underlying language model, this approach is less common in practice.

The ranking of documents is determined by the probability of the document model generating the query. Given a query $q$ and a document $d$, we are interested in calculating $P(d|q)$. After applying Bayes rule we can rearrange the equation into:

$$P(d|q) = \frac{P(q|d)P(d)}{P(q)} .$$

As we are only interested in the ranked order of the documents, we can ignore the probability of observing the query $P(q)$. Further, we assume a uniform probability of observing a document $P(d)$. Therefore, the ranking of documents is based on:

$$P(d|q) \simeq P(q|d) .$$

Given the term independence assumption, the probability of a query given a document is based on the product of query term probabilities:

$$P(q|d) = \prod_{t \in q} P(t|d) .$$

To reduce the loss of precision associated with the multiplication of small probabilities, we take the logarithm:

$$\log P(q|d) = \sum_{t \in q} \log P(t|d) ,$$

which is rank equivalent. As the language model of each document is unknown, the probability of each term in the model must be estimated. Using the maximum likelihood estimator, we assume that each document is the best possible representation of its model. This leads to a term probability $P(t|d)$ based on the number of occurrences of the term in the document:

$$P(t|d) = \frac{f_{t,d}}{|d|} ,$$
where \(|d|\) is the length of the document in terms. One problem with the maximum likelihood estimate is that terms that do not occur in the document are assigned a zero probability. Smoothing can be used to assign some of the probability mass in the document to terms that do not occur within it. Zhai and Lafferty [2004] investigate various smoothing techniques for use with language models and find that for short web-style queries, Dirichlet and absolute discounting smoothing techniques work best. In this work we use Dirichlet smoothing, where:

\[
P(t|d) = \frac{f_{d,t} + \mu F_t}{|d| + \mu},
\]

where \(|C|\) is the size of the collection, measured in terms, and \(\mu\) is a tuning parameter used to control the amount of smoothing. Using these smoothed term probabilities, the order preserving ranking function is given by:

\[
P(q|d) \approx |q| \cdot \log \left( \frac{\mu}{\mu + |d|} \right) + \sum_{t \in q \cap d} \log \left( \frac{f_{d,t}}{\mu} \cdot \frac{|C|}{F_t} + 1 \right),
\]

where \(\log \left( \frac{\mu}{\mu + |d|} \right)\) is the document length normalisation applied for each query term, and \(\log \left( \frac{f_{d,t}}{\mu} \cdot \frac{|C|}{F_t} + 1 \right)\) is the contribution of each term that appears in both the document and the query. Finally, the use of \(\log\) allows an additive computation while retaining rank order.

### 2.7 Stopping, Stemming and Case-folding

The formulas in Section 2.6 assume all terms in the query are processed, however in practice many optimisations to index processing have been proposed. One method is to reduce the average number of postings lists processed per query. A well-known, simple technique to do this is stopping [Salton and McGill, 1983], that is, ignoring the long lists of function terms such as “the”, “of”, and “therefore”. This saves time in query evaluation for queries containing function terms but may lower average accuracy. Moreover, some queries that consist of only function terms — such as “the who” — cannot be evaluated at all. It is therefore unclear when stopping should be used.

Stopping can be applied at either index construction time, or at query evaluation time. When applied at index construction time, the stopped terms are not stored in the index, significantly reducing the index size. When applied at query evaluation time, stopped terms are not retrieved from the index to be processed during the similarity calculation, and query processing time is significantly reduced.
Another optimisation approach to reduce the average number of lists processed per query is stemming [Frakes, 1992]. Typically, English stemming iteratively removes suffixes from words to find the morphological root, allowing plurals to match singulars and so on. For example, stemming reduces “reducing”, “reduced”, and “reduction” to the common root “reduc-”. Using this approach, all words that stem to “reduc-” are indexed as postings in one list, with the overall effect that the list is typically much longer than the list of any of the unstemmed variants. This improves compression — since the differences between adjacent values are smaller — and fewer disk seeks (but longer reads) are required on average to answer each query. From a search perspective, the process permits a query such as “reducing words” to match a document containing the terms “word” and “reduction”. As with stopping, effectiveness varies between tasks. Stemming is generally applied at index construction time, and reduces the size of the on-disk index as less terms are stored in the vocabulary.

Finally, character case can result in variations of a term. That is, “THIS”, “This” and “this” are by default considered different terms with regard to general string comparison operations. A simple technique to reduce the number of distinct terms in an index, is to case-fold all terms into either upper- or lower-case.

In this work we apply both case-folding and stemming to all experiments. Stopping is used in some experiments and is noted where it is present.

2.8 Accumulator Limiting

Yet another approach to improving efficiency is to limit the number of accumulators that are created during query evaluation. As discussed previously, an accumulator stores the sum of similarity computations for each query-document pair, where similarity is computed using one of the approaches listed in Section 2.6. Reducing the number of accumulators has the dual benefits of limiting both main-memory use and the computational cost of maintaining values for documents with very low similarity to the query. In previous work [Moffat and Zobel, 1994; 1996] it has been shown that limiting the number of accumulators to around 1% of the documents of the collection works well in practice. However, to do this, it is necessary to decide which accumulators to create and what to do when the threshold limit is reached.

A simple, effective approach for creating accumulators for only documents that are highly similar to the query is to process postings lists by increasing length. In this way, rare terms are processed first, that is, those postings with the highest inverse document frequency are
used to create accumulators. Once a threshold value of postings is reached, no further accumulators are created.

When the accumulator creation threshold is reached, two possible approaches can be taken. First, to QUIT processing the current and remaining postings lists, and rank by the partial similarity values stored in the accumulators. Second, to CONTINUE processing lists, but to only update existing accumulators so that they reflect the total similarity between a query and the documents represented in the accumulator set. In experiments, Moffat and Zobel showed that the CONTINUE strategy was as effective as using unlimited accumulators, and that the QUIT strategy was less effective. However, unsurprisingly, QUIT is the fastest approach.

Recently Lester et al. [2005a] proposed an adaptive accumulator limiting scheme based on the setting of dynamic thresholds for the minimum allowable contribution towards accumulators. Their approach reduces bias towards those postings that are located at the beginning of each inverted list, and they show a significant reduction in the number of accumulators required at query time. However, a limitation of the approach is that it requires processing of the entire inverted list for each term in the query.

In this work we propose a reorganisation of inverted lists to reduce the amount of disk access required at query time. We make use of the QUIT and CONTINUE schemes, as well as variations that are discussed in Chapter 3.

2.9 Inverted List Reorganisation

The techniques listed above avoid processing all the terms of the query, or all of the postings lists in response to a query. A different approach is to reorganise the lists themselves so that an individual list may not need to be completely decoded. There are two general approaches in this area: first, inserting additional pointers into the lists so that values can be skipped [Moffat and Zobel, 1996]; and, second, storing postings in a different order so that those that are most likely to be similar to a query are at the beginning of the lists. However, in modern systems skipping is unlikely to be effective as the relative performance of CPUs, CPU cache, main-memory, and disk is now very different to when skipping was proposed [Zobel and Moffat, 2006]. Therefore, we do not experiment with skipping here.

Various works have explored inverted list reorganisation. Some work has focused on partial reorderings where a subset of postings that are considered to be valuable are placed at
the head of each inverted list [Brown, 1995]; while others propose inverted lists of normalised \( f_{d,t}/W_d \) values, with the goal of reducing the computations required when processing the postings [Buckley and Lewit, 1985; Lucarella, 1988]. A problem with storing normalised values in the lists is that they cannot be represented by integers, and therefore they effect compression of the index; while partial reorderings offer limited possibilities to prune inverted lists during query processing.

Two techniques that allow effective compression, while maintaining list-wide ordering are frequency- and impact-ordered indexes. Frequency-ordering organises postings by the within-document frequency of a term, while impact-ordering arranges lists by the impact that a posting is expected to have on the similarity metric. Each approach is discussed below.

### 2.9.1 Frequency-Ordering

Persin et al. [1996] propose organising postings lists by decreasing frequency \( f_{d,t} \) to give a frequency-ordered index. The motivation for storing postings by frequency is to correlate list ordering to impact on the ranking function. Using this approach, our example postings list for the term “storm” is ordered as follows:

\[ \langle 386, 3 \rangle \langle 408, 2 \rangle \langle 22, 1 \rangle. \]

A disadvantage of this approach is that differences can no longer be taken between ordinal document numbers in the inverted list. Instead, differences can be taken between frequency values to improve compressibility. Interestingly, Persin et al. showed that by taking the difference between ordinal document numbers within each block of postings that share a common within document frequency, a slight overall reduction in index size can be achieved. This gain arises because document term frequencies need no longer be stored per individual posting, but instead once per group of postings that share the same value.

The advantage of frequency-ordering is that at query time, heuristics can be applied to abandon processing after partial list decoding. To do this, Persin et al. propose computing metrics that use two thresholds based on, the weight of the current term; the maximum accumulator value seen so far; and two constant parameters. This works as follows: first, before processing each list, the values of two parameters \( f_i \) and \( f_a \) are calculated:

\[ f_i = \frac{c_i S_M}{f_{q,t} W_t^2}, \quad f_a = \frac{c_a S_M}{f_{q,t} W_t^2}. \]
where $S_M$ is the maximum accumulator value seen so far, $w_t$ is the weight of the current term, and $c_i$ and $c_a$ are constants. Second, each posting in the current list is considered sequentially:

1. If the similarity computation value is greater than or equal to $f_i$, then a new accumulator is created if the document does not have one, or the accumulator value updated otherwise.

2. If the similarity computation value is greater than or equal to $f_a$ but less than $f_i$, then an existing accumulator value is updated. If an accumulator does not exist, the computation is ignored.

3. If the similarity computation is less than $f_a$, then the processing of the current list is abandoned.

As for the continue and quit strategies, lists are processed from shortest to longest.

The constants $c_i$ and $c_a$ require careful consideration. In practice, the constant $c_i$ is set to a value greater than $c_a$ so that lists are processed in three phases: accumulators are added and updated, then accumulators are only updated, and then processing is abandoned. However, since $S_M$ is zero before the first list is processed, the first list creates accumulators for all postings. Persin et al. found that values of $c_i = 0.12$ and $c_a = 0.007$ worked well in practice for searching the Wall Street Journal (WSJ) newswire data from TREC disks 1 and 2. Indeed, they found that this pruning function produced the same retrieval effectiveness as processing all postings through a document-ordered approach, while processing of as little as 10% of the postings lists.

In their experiments, Persin et al. found frequency-ordered indexes were efficient and effective. On newswire data with long queries ranging from 66 to 313 terms, they found that querying was over 62% faster for stopped queries and more than 83% faster for unstopped queries than using a conventional document-ordered index while achieving comparable accuracy.

The idea of frequency-ordering has been used in other works. Brown [1995] proposed a hybrid of frequency-ordering where the postings with the highest frequency values are stored in a single disk page at the start of each inverted list. At query time, documents that occur in the first disk page are given priority for accumulator creation. Wong and Lee [1993] also make use of frequency-ordered inverted lists, but apply a different thresholding approach. When the value of $f_M \times \log_2(N/f_t)$ falls below a threshold, list processing terminates, where $f_M$ is the largest $f_{d,t}$ value in a page of postings.
2.9.2 Impact-Ordering

A recent scheme for list reorganisation is *impact-ordering* [Anh et al., 2001; Anh and Moffat, 2002a;b; Anh, 2004; Anh and Moffat, 2005b]. In the impact-ordered indexing scheme, postings are ordered within a list by decreasing effect on the ranking function.

Two general approaches to impact-ordering are proposed. The first utilises a *collection-oriented* approach to determine the value of each impact, where impacts are based on the contribution of the posting to the similarity metric. The second view, takes a *document-centric* approach in determining impacts, where each impact is based on the significance of the term within the document in which it occurs. In both cases, impacts reflect the contribution of the term to the similarity metric, however in the case of collection-oriented impacts, the relationship is direct, whereas in the document-centric approach it is implicit.

In Chapter 3 we compare our novel index ordering approach to impact-ordering.

**Collection-Oriented Impacts**

In a collection oriented impact-ordered index, each posting is a tuple \( \langle d, m_{d,t} \rangle \) where \( m_{d,t} \) is the impact of term \( t \) in document \( d \), and is defined as:

\[
m_{d,t} = w_{d,t}.
\]

The value \( w_{d,t} \) is the weight of term \( t \) in document \( d \) as determined by the document similarity measure employed in the search engine. Anh and Moffat [2002b] found that the pivoted cosine metric as defined by Singhal et al. [1996b] worked best with impact-ordering. Using this metric the impact of a term in a document is given as:

\[
m_{d,t} = \frac{1 + \log_e f_{d,t}}{(1 - s) + s \cdot W_d/W_A}.
\]

The constant \( S \) is usually set to 0.7, \( W_d \) is the weight of document \( d \), and \( W_A \) is the average value of \( W_d \) across all documents in the collection. The value of \( m_{d,t} \) as defined above is a floating point value that can be used directly in the similarity measure as follows:

\[
s_{d,q} = \sum_{t \in d \cap q} m_{q,t} \cdot m_{d,t},
\]

where the similarity between a document \( d \) and query \( q \) is the sum of the products of the document-term impact \( m_{d,t} \) and the query-term impacts \( m_{q,t} \) for each term that appears in both the document and the query. For compactness, we do not define query term impacts in detail here, but the principles follow that of document impacts.
2.9. INVERTED LIST REORGANISATION

<table>
<thead>
<tr>
<th>Term</th>
<th>Postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>fairy</td>
<td>⟨642, 24⟩ ⟨649, 12⟩</td>
</tr>
<tr>
<td>farewell</td>
<td>⟨434, 31⟩ ⟨47, 23⟩ ⟨464, 1⟩</td>
</tr>
<tr>
<td>fish</td>
<td>⟨386, 25⟩ ⟨430, 21⟩ ⟨750, 13⟩ ⟨283, 1⟩ ⟨436, 1⟩ ⟨480, 1⟩</td>
</tr>
<tr>
<td>state</td>
<td>⟨77, 18⟩ ⟨606, 13⟩ ⟨79, 1⟩ ⟨471, 1⟩</td>
</tr>
<tr>
<td>storm</td>
<td>⟨408, 21⟩ ⟨386, 19⟩ ⟨22, 1⟩</td>
</tr>
<tr>
<td>swords</td>
<td>⟨566, 15⟩ ⟨565, 14⟩</td>
</tr>
<tr>
<td>sycorax</td>
<td>⟨159, 17⟩ ⟨511, 17⟩ ⟨128, 1⟩ ⟨132, 1⟩ ⟨136, 1⟩</td>
</tr>
<tr>
<td>vision</td>
<td>⟨612, 1⟩ ⟨624, 1⟩ ⟨718, 1⟩</td>
</tr>
</tbody>
</table>

Figure 2.6: Postings of selected terms in an impact-ordered document-level index for the Tempest collection.

The impact-ordered approach encompasses much more than a method of organising postings. As part of their body of work, the authors propose normalising impacts so that all terms in short web queries contribute significantly towards similarity computations. In addition, they propose mapping the normalised impact values into fixed precision integers using a uniform quantisation scheme. Together, these steps have three advantages: first, quantisation allows integers to be stored instead of floating point values; second, the uniformly quantised impacts can be used directly in the similarity measure as surrogates for the actual impacts; and last, normalisation was found to improve search accuracy.

As with frequency-ordering, as postings are ordered by decreasing impact value, the differences between adjacent values can be stored. In addition, when postings share the same impact, they are sorted by document identifier and differences between postings in each equi-impact block are stored instead of the full document identifiers. Finally, as the normalised quantised impacts are used directly in the similarity measure, there is no need to store the within document term frequency \( f_{dt} \) values per posting.

Figure 2.6 shows the quantised impact-ordered lists for selected terms in the Tempest collection. Note that for each list, the postings are ordered by decreasing impact, so that at query time, those postings that are likely to contribute the most to the similarity metric will be processed first. Further, although there is a correlation between the document frequency of a term and its impact, this is not always the case. For example, although document 386 contains term “storm” three times, its impact in that document is less than that of document 408 that only contains the term “storm” twice.
To reduce query processing time, impact-ordering utilises a heuristic that determines when to abandon list processing. Anh and Moffat proposed two new schemes: first, **TERM-FINE** where a penalty is applied to the impact of each query term, and this penalty is increased as each new term is processed; and, second, **BLOCK-FINE**, where a penalty is applied to posting contributions, and the penalty is increased as each new impact block is processed. For both schemes, the penalty is applied to the posting contribution before updating the relevant accumulator. Processing is abandoned when the penalised contribution of a posting falls below zero.

The benefits of these heuristics are two-fold: first, list processing can now be abandoned before processing an entire postings list; and second, for long queries with many terms, the penalties re-establish the differences between posting contributions that are diminished by the normalisation step. This is important in long queries as terms that do not discriminate well between documents — that is, traditionally those that have a low impact — may have high impacts after normalisation, and without penalties may dominate the accumulator contributions for a document.

Use of a combination of **TERM-FINE**, **BLOCK-FINE**, and the **CONTINUE** strategy discussed in the previous section, produced the best compromise between speed and accuracy. On the same web collection we use, Anh and Moffat [2002b] report relative accuracy improvements of 23%–43% over a pivoted cosine baseline.

There are three disadvantages to collection-oriented impact-ordering. First, Anh notes that impact-ordering is as yet ineffective for state-of-the-art Okapi BM25 measure due to the inability to separate query term and document term impacts [Anh, 2004]. Second, impact-ordering is dependent on the ranking function, making it difficult to change or tune the ranking function without rebuilding the index. Finally, there is an unclear relationship between normalisation and the **FINE** strategies: one increases the impact of terms, while the other lowers it.

**Document-Centric Impacts**

The document-centric approach to impacts simplifies the assignment of impact values to each posting. Each posting remains a tuple \(\langle d, m_{d,t} \rangle\) where \(m_{d,t}\) is the impact of term \(t\) in document \(d\), but now the impact \(m_{d,t}\) is assigned based on the importance of the term within the document [Anh and Moffat, 2005b]. Terms within a document are ranked by importance, and the number of terms that are assigned a given impact is determined by a
2.9. INVERTED LIST REORGANISATION

Figure 2.7: Process of impact value allocation to the terms in a document. First, the frequency of occurrence of each unique term in a document is established. Second, terms are ordered by decreasing frequency within the document. Third, impacts are assigned using a geometric distribution.

geometric distribution given by:

\[ x_i = B \cdot x_{i+1}, \]

where:

\[ B = \left( |d| + 1 \right)^\frac{1}{k}. \]

The equation makes use of the number of terms in the document \(|d|\), and the number of impact buckets that are allowed \(k\). Anh and Moffat suggest eight impact buckets, that is, \(k = 8\).

Anh and Moffat explored various ranking techniques to assign impact values to each term, ranging from ranking by the value of the product of document term frequency, \(f_{d,t}\), and inverse document frequency, \(\frac{N}{f_t}\); to ranking by term frequency alone; and variants combining both, with one as the primary sort key, and the other as a secondary key. They suggest an ordering based on a primary key of document term frequency, with a secondary key of inverse document frequency performs best, but note that a primary sort key of document term frequency alone is comparable. A problem with the dual key approach is that it requires some collection oriented information to produce impacts, while the single key term frequency ranking does not.

Figure 2.7 illustrates the process of impact assignment with a document-centric approach. Here we see that impacts are assigned based on the importance of the term to the single document. Unlike collection-oriented impacts, collection wide statistics are not necessary to assign the impacts. With all the terms in a document assigned an impact, the postings can
be added to the inverted lists. Once impact values have been assigned to each term in each document, the inverted lists are reordered by impact and query processing continues in the same manner as the collection-oriented impact-ordered index.

The benefits of document-centric impact-ordering are that collection-wide statistics such as inverse document frequency are not necessarily required to assign impacts. Further, the addition of new documents to the collection does not require the recalculation of all the existing impacts.

### 2.10 Collection Reordering

An alternative to reorganising the inverted lists of individual terms is to reorganise the entire collection. Despite the formidable label, *collection reordering* does not require that the collection itself be changed. Instead, a collection reordering can be trivially accomplished by assigning each document in the document map a new identifier based on a specified ordering. In a document-ordered index, the reorganisation process has the effect of determining which documents are processed first in each inverted list.

Blandford and Blelloch [2002] propose a collection ordering based on the clustering of similar documents. In their approach they generate a document-to-document graph over the collection, with each edge weighted by the similarity of the documents using the cosine measure. The collection is then repeatedly partitioned into clusters to establish a new document ordering.

Under this approach, documents of similar content, and therefore similar terms, are likely to be located within a close proximity of each other. Benefits in compression arise as the average distance between documents in the inverted lists of the terms is reduced. Blandford and Blelloch achieve index compression improvements of up to 21.8% over a random ordering of documents, and show that this saving is likely to increase as the collection size grows.

Like Blandford and Blelloch, Shieh et al. [2003] also build a document-to-document graph over the collection, where the weight of an edge between two documents is simplified to a count of the terms that co-occur between each document node. Shieh et al. experiment with various algorithms to traverse the graph and find that a greedy nearest-neighbour approach produces the best ordering for their collections. In their work, they show collection compression improvements of up to 15%.

Over several works Silvestri et al. propose a variant collection reordering technique that improves the efficiency of the clustering step [Silvestri et al., 2004b,a]. In their approach, they
2.11. SEARCH ENGINE CACHES

segment the collection into \( k \) clusters as follows. First, they select the largest document from the unallocated set to be the center of the next cluster. Second, they compute the Jaccard similarity between each unallocated document and the current center. Third they allocate the \( \frac{N}{k} \) most similar documents to the new cluster, where \( N \) is the size of the collection. After \( k - 1 \) iterations, the collection is partitioned. Silvestri et al. achieved compression improvements of up to 23% with their test collections, and showed that their approach scales linearly with the size of the collection.

To date, collection reordering approaches have relied on the within collection information to establish an ordering with the goal of reduced index size. However, as we show in Chapter 4, external information can also be used to establish a collection ordering that supports rapid query evaluation through list pruning at query time.

2.11 Search Engine Caches

A complement to altering the composition of inverted lists to achieve efficiency is to use caching. For many applications, retaining frequently accessed data from disk in memory reduces the cost of disk access. Search engine query optimisation using an in-memory cache has been explored by several authors. Xie and O’Hallaron [2002] examine several search engine query logs and highlight the repetition and locality of the queries within their data. While they do not propose an approach to caching, they suggest that any caching approach for search engines should account for the popularity of queries and the locality of repetition.

Work on search engine caching falls into three broad categories. Some work focuses on the benefits of caching individual data structures used during the search process. These structures can include items such as inverted lists, or result pages. A second branch of research builds on the above by combining the data structures that are considered by the cache. Finally, a third branch of research has investigated caching policies that best manage the different components that can be cached by a search engine.

2.11.1 Single Structure Caching

An early contribution to search engine caching research is that of Brown et al. [1994], who attempt to cache inverted lists with a least recently used (LRU) replacement policy. In their work they maintained three manually chosen fixed-size caches for small, medium, and large lists respectively. They found that by tuning the separate cache sizes correctly they could
achieve speed improvements of up to 25% on their test collections, which they attributed to reduced disk usage costs. The approach of Brown et al. relies on manually tuning lists sizes for inclusion into each of the small, medium, and large caches, and is unlikely to work for dynamic collections, where the distribution of list sizes change over time.

Jónsson et al. [1998] take a lower-level approach. They propose a buffer management technique that caches the pages of inverted lists, and favours pages from the inverted lists that contain high weight postings. At query time, inverted lists are processed in an order that minimises the number of disk reads by fetching those lists that contain the most pages in the buffer. Using their approach, they show a 70% reduction in disk reads over non-caching. However, as their approach approximates results, a variance of ±5% in effectiveness is observed.

Inspired by the caching of result pages by web proxy servers [Markatos, 1996], Markatos [2001] explored the caching of search engine result pages. Markatos compared the benefits of caches composed of fixed pre-calculated result pages, to dynamic caches that adapt based on the stream of queries posed to the search system. The findings suggest that, for small cache sizes, a fixed cache of the most frequently requested result pages outperformed a dynamic cache. However, as the cache size was increased, the performance of dynamic caching improved. This outcome was attributed to the fact that, for small caches, dynamic policies were not able to maintain popular pages within the cache long enough for them to be reused but, as the cache size grew, the probability of retaining a popular result page in the dynamic cache increased.

In similar work, Luo et al. [2000] cache result pages, but assume a Boolean environment where the results of individual queries can be combined to form the results for larger queries. They label this active caching and report a two-fold improvement in cache hit ratios over the results of Markatos. However, this approach does not account for the complexity of most modern similarity ranking functions, where the weight of query terms has a significant influence on their contribution to the final results.

2.11.2 Multi Structure Caching

Saraiva et al. [2001] combined the above approaches, proposing a two-level cache for search engines that at one level caches result sets, and at a second level caches inverted lists. The caching of result sets allows query processing for cached queries without accessing disk, while caching inverted lists reduces the I/O costs associated with query processing for queries that
contain popular terms, but do not have a result set in the cache. In their work they found an increase in query throughput of up to a factor of three.

Baeza-Yates and Saint-Jean [2003] propose a similar two-level cache with both inverted lists and pre-computed answer sets in memory. In their approach, lists are selected such that the largest set of the most commonly queried terms are placed in the cache. Result sets are also retained in the cache, and the proportion of space allocated to each data type is trained. Their results show a 7% reduction in query time evaluation.

Long and Suel [2005] extended the work of Saraiva et al. [2001] and proposed a three-level cache, in which result sets and inverted lists are cached in memory, and a representation of merged inverted lists that are computed during query evaluation are stored on disk. For combinations of query terms where a pre-computed merged list is in the on-disk cache, a single disk access fetches the required information, rather than a fetch to the inverted list of each term in the query. They report a 75% reduction in disk blocks read at query time, with a cache that is 2.5% the size of the index.

2.11.3 Advanced Cache Management Policies

Focusing on result caching, Lempel and Moran [2003] propose a more specialised cache replacement policy targeted at result pages. In their work, they present a cache replacement policy based on the probability of a result page being requested. The probability of a page being requested is a function of the history of the result page’s prior access, and the number of users that are currently viewing result pages for the same topic. Further, Lempel and Moran introduce the pre-fetching of subsequent result pages. Working with a query log from AltaVista, the authors measured the number of queries that were satisfied by the results stored in the cache. They found that their approach up to doubled the cache hit ratio over a cache using a least recently used queue.

Fagni et al. also propose a result page oriented cache that is divided into static and dynamic components [Fagni et al., 2004; 2006]. The static part of the cache is populated with the results for the most common queries observed, while the dynamic component caches results for queries that do not appear in the static component. Further, the dynamic cache may be managed by any cache management policy. In their work, Fagni et al. experiment with least recently used, segmented least recently used, frequency based replacement, two-queue, and probabilistic driven cache policies [Karedla et al., 1994; Robinson and Devarakonda, 1990; Johnson and Shasha, 1994; Lempel and Moran, 2003]. While the optimal proportion of
cache allocated to the static and dynamic caches varied with the data-set, the authors found that their combined approach outperformed an individual static or individual dynamic cache. Further improvements were obtained by applying pre-fetching of query results, however the authors suggest that only a few pages of results — three pages are shown to work well — per query is optimal.

Other work has examined distributed caching. In some cases the caching of results is suggested on the client side [Alonso et al., 1990], in others within web proxy servers [Meira Jr. et al., 1999]. However, such work is beyond the scope of this thesis.

2.11.4 Limitations of Caching

To date, search engine caching research has focused on the caching of result sets and inverted lists. The caches for inverted lists and results have been maintained independently, and the ratio between the two has been manually tuned with regard to the query log and collection in use. To our knowledge, there has been no examination of the costs of caching each data type with respect to underlying system architecture, or of how to tune cache sizes in practice. Further, there has been no exploration of the use of a single heterogeneous cache for all the various kinds of data used during query processing. We address these issues in Chapter 6.

2.12 Query Difficulty Prediction

The inverted list reorganisation, collection reordering, and caching techniques presented in the above sections focus on improving query evaluation time. A second branch of information retrieval research focuses on techniques that improve the accuracy of search results. One such technique is query difficulty prediction. Query difficulty prediction is the task of determining the effectiveness of search without any further information about the query from the user. By detecting those queries that are likely to perform poorly, a search system can:

- inform the user that the query may fail, thus prompting for query refinement;
- automatically refine the query in a effort to improve the search results;
- choose to use different metrics to resolve the query; or
- in the case of a distributed system, choose to use a different collection to resolve the query.
2.12. QUERY DIFFICULTY PREDICTION

The task of difficulty prediction is indeed difficult. As with any other task that involves natural language, it is not always clear from the query what users want. A query can be difficult because a user does not provide enough information, or because the query conveys a complex meaning that a token-based search system fails to understand. Research of query difficulty in recent years has broadened. Query difficulty prediction was included in the TREC Question Answering track in 2002 [Voorhees, 2002], and subsequently added to the TREC Robust track in 2004 [Voorhees, 2004]. In 2004, a workshop was also organised at SIGIR to investigate the cause of failure in IR systems [Harman and Buckley, 2004] and in 2005 a SIGIR workshop focused on query difficulty prediction [Carmel et al., 2005]. In this section we present a selection of the key works in the area.

2.12.1 Measuring Prediction Effectiveness

Several techniques have been proposed to measure the performance of prediction systems. At TREC 2002, Kendall’s \( \tau \) [Long, 2006] was used to measure the correlation between the actual and predicted accuracy of search systems [Voorhees, 2002]. Kendall’s \( \tau \) is one of several correlation metrics. In other works Pearson’s coefficient [ESS, 2006] and Spearman’s rank correlation [Pirie, 2006] have also been used [Carmel et al., 2006; Cronen-Townsend and Croft, 2002; Cronen-Townsend et al., 2002; Scholer et al., 2004].

At TREC 2004, Voorhees [2004] argued that order based correlation metrics such as Kendall’s \( \tau \) are too sensitive to small permutations in the ranking. Voorhees proposed an inverse area measure using two curves. The first curve plots mean average precision (MAP), over \( N \) topics, where the next best performing topic is successively removed from the pool of topics prior to the calculation of MAP. A similar curve is plotted for the predictions, where the next best predicted topic is sequentially removed from the pool. The area difference between the two curves shows the effectiveness of the prediction system. An accurate prediction produces a curve that closely follows the actual performance curve, resulting in a small area between the two curves. A poor prediction would result in dissimilar curves, with a large area between the two curves. However, at TREC 2005, it was shown that the area-under-the-curve metric can favour systems that produce poor effectiveness results [Voorhees, 2005].

An alternative measure of performance is to count the number of correctly predicted best and worst performing queries [Kwok et al., 2005; Kwok, 2005]. The benefit of this approach is that the metric is less sensitive to small changes in the ordering of actual and predicted performance. Further, Kwok argues that the best-worst metric can be used to
identify systems that can predict bad queries separately to those systems that can predict good queries.

2.12.2 Inverse Document Frequency Predictors

One issue that leads to difficult queries is query ambiguity, that is, without further information, several distinct interpretations of the query are possible. Ambiguity generally arises when a query term has several definitions and the query does not provide enough context to distinguish which definition is appropriate for the query information need. It could be argued that an ambiguous term is one that appears in a wide range of documents [Pirkola and Järvelin, 2001]. For example in the query “Tasmanian tiger”, the term “Tasmanian” is likely to be found in documents for a wide range of topics, such as those about the Tasmanian forest, Tasmanian culture, and Tasmanian government; while the range of documents containing the term “tiger” is likely to focus on topics about the animal. The inverse document frequency (IDF) value used in the term weighting function of several similarity metrics is based on the principle that terms which appear least frequently should carry a higher weight. Therefore, a simple approach to query difficulty prediction is to consider the document frequency of the query terms.

Several works examine variant IDF based techniques including the minimum, mean, maximum and standard deviation of query term IDF values and report mixed results. Scholer et al. [2004] report that the minimum IDF is a poor predictor, while maximum IDF works well. Plachouras et al. [2004] use the mean and the standard deviation of IDF values and report a slightly better correlation between performance and prediction with the mean based approach. He and Ounis [2004] use the ratio between the maximum and minimum query term IDF values to measure the “distribution of informative amount in the query terms”, however this seems less intuitive as a query that contains two common (low IDF) terms could be considered as difficult as one that contains two rare (high IDF) terms.

One benefit of the IDF approach is that difficulty prediction can be made without a need to evaluate the query. Other simple pre-retrieval metrics such as the length of the query, or a count of the number of documents containing at least one of the query terms have been proposed. However, reports show that such metrics do not work well [He and Ounis, 2004]. Several systems even use the results from the search system similarity metric directly [Piatko et al., 2004; Tomlinson, 2004], however such approaches have not been shown to be effective.
While the IDF of a term indicates its rarity in the collection, it does not indicate its relationship with the other query terms. One way to measure this relationship is to compare the set of documents returned when each term is evaluated independently, to the set of documents returned when the query is evaluated as a whole. A large overlap in results indicates that the query terms are closely related, and that the query is not difficult to resolve, while a disjoint set of results suggests that the terms are unrelated and the query is difficult. Yom-Tov et al. combine this approach with the IDF of the query terms as a second feature of query difficulty. They then use machine learning to train query difficulty estimators [Yom-Tov et al., 2004; 2005b]. They report correlations of up to 43% with Kendall’s $\tau$ on TREC newswire data, and significantly lower correlations of up to 20% on web data.

### 2.12.3 Machine Learning Based Predictors

Machine learning is used in various works to combine features for query prediction [Kwok et al., 2004; 2005; Jensen et al., 2005; Kwok, 2005; Grivolla et al., 2005], features used include:

- inverse document frequency (IDF);
- query term frequency within query;
- average query term frequency in the collection;
- average percentage of character n-grams appearing in the query and document title;
- counts of the number of documents retrieved that contain all, all but one, and all but two of the unique query terms;
- the entropy of the top retrieved documents; and
- the distribution of collection frequencies of the query terms.

Sehgal and Srinivasan [2005] make use of machine learning with a specialised feature set to predict performance, but found that a combination of features was no more effective than a single reliable feature.

Taking a different approach, Kowalczyk et al. [2004] apply machine learning to classify queries into general query classes, such as: location, name, number, person, time, and other. The queries of each class are then treated differently at query evaluation time to improve accuracy. In their work, Kowalczyk et al., manually derived the various classes applied to their system. It is unclear how a large scale search system with a diverse range of user
information needs would adapt to the limited set of classes defined, or how new classes could be automatically identified without user intervention for the generation of training queries and results.

2.12.4 Language Model Based Predictors

One might assume that documents that discuss a particular topic share a similar vocabulary. Further, the language used in the documents that belong to a particular topic may vary to some degree to the language model used throughout the rest of the document collection. As such, if it were possible to measure the distance between the language model of the topic’s documents, and the language model of the collection, it would be possible to find poorly performing topics. The rationale is that the documents for topics that are easy to resolve stand out from the collection. Conversely, topics that perform poorly are expected to have a similar term distribution to the rest of the collection, therefore making those documents difficult to extract from the collection at query time. The above principles form the basis of the query clarity approach to difficulty prediction [Cronen-Townsend and Croft, 2002; Cronen-Townsend et al., 2002].

In the query clarity approach, a language model is generated for each query. The query language model is then compared to the collection language model using Kullback-Leibler divergence [Kullback, 2006]:

\[ QC = \sum_{t \in V} P(t|q) \log_2 \frac{P(t|q)}{P(t|C)}. \]

One problem with query clarity is that the query does not contain enough information to create a model that sufficiently distinguishes it from the collection. As such, a model for the query is generated by sampling the documents returned in response to the query. The authors suggest a sample size of 500 documents. Based on the sample, the likelihood of observing a term is defined as:

\[ P(t|q) = \sum_{d \in R_t} P'(t|d)P(d|q), \]

where \( R_t \) is the set of sampled documents from the result set. The value of \( P(d|q) \) is obtained from \( P(q|d) \) with Bayesian inversion assuming a uniform prior for the documents in \( R_t \), and the value of \( P'(t|d) \) is obtained with linear smoothing against the collection model:

\[ P'(t|d) = \lambda P(t|d) + (1 - \lambda) P(t|C). \]
Cronen-Townsend and Croft have produced successful results with query clarity, reporting significant correlations of up to 57% between predicted difficulty and actual difficulty on TREC newswire collections. More recently, Cronen-Townsend et al. [2003] have applied query clarity to the task of question answering by indexing documents at a passage level.

However, it has been found that query clarity does not perform well on web-oriented tasks [Scholer et al., 2004; He and Ounis, 2004]. Further, in user based studies, Turpin and Hersh [2004] failed to find a correlation between query clarity and the user’s ability to find documents with a search system.

He and Ounis [2004] propose a variant query clarity measure that does not generate the query language model with the query results, but instead estimates the probability of observing a term in the query model as:

$$P(t|q) = \frac{f_{q,t}}{|q|}.$$  

While this simplified measure is shown to not perform as well as the original query clarity, the reduced computational complexity is significant. Further, when compared to a range of other pre-retrieval predictors, their simplified clarity score is shown to be an effective predictor, producing more accurate predictions than their IDF based techniques.

In an approach that is very similar to the implementation of query clarity, Amati et al. [2004] measure the divergence in term distributions between the top-ranked documents retrieved for a query and the collection itself. Their difficulty ranking metric is defined as:

$$D = \sum_{t \in q} \log_2 \frac{f_{R,t}}{\sum_{t' \in R} f_{R,t'}} \cdot \log_2 \frac{f_{R,t} \cdot \sum_{t' \in C} F_{t'}}{\left(\sum_{t' \in R} f_{R,t'}\right) \cdot F_t},$$

where $R$ is the set of top retrieved documents, and $f_{R,t}$ is the frequency of term $t$ within the retrieved set, $C$ is the set of all documents in the collection, and $F_t$ is the frequency of term $t$ in the collection. Using their approach, Amati et al. selectively apply query expansion and show minor improvements in accuracy to a baseline where all queries have expansion applied.

### 2.12.5 Other Forms of Difficulty Prediction Evidence

External evidence can potentially help predict performance. Swen et al. [2004] make use of external information from WordNet [Miller, 1995] to measure the ambiguity of query terms with limited success. Other groups have attempted to extract information directly from the web [Yom-Tov et al., 2005a], however the dependency on web data limits the applicability this approach to tasks utilising data that is similar to the content of the web.
Linguistics provide a different means by which we can assess the difficulty of resolving a query term. Mothe and Tanguy [2005] examine various linguistic features as a means of query difficulty prediction. They find that the average syntactic link span between query terms and the average polysemy value of the query terms show a significant negative correlation with query difficulty over a number of test collections. Syntactic link span is a measure of the distance between a term pair tagged by a syntactic analyser. For example, where two terms appear adjacent to each other the distance is one. It should be noted that syntactic link span is unlikely to be effective for web queries where the average query length is low, as the span will exhibit minimal variance. Polysemy approximately measures the number of different meanings a word has. This value is available from tools such as WordNet [Miller, 1995].

Diaz and Jones [2004] introduce a temporal aspect to difficulty prediction. They explore variant temporal features that measure properties such as the burst rate of queries, and the temporal divergence of a query to the temporal distribution of all queries. Individually their predictors are not conclusively shown to be effective, but, when combined with query clarity, the results presented show an improvement to the accuracy of predictions. However, the approach of Diaz and Jones is dependent on the availability of a creation time for each document in the collection. In the case of web documents, such data is both difficult to obtain, and likely to be unreliable.

2.12.6 Modelling Query Difficulty

Carmel et al. [2006] propose a model to describe the forms of evidence that can be obtained from topics for difficulty prediction. In their model a topic is composed of a set of queries, \( Q \), that describe the topic; documents, \( R_l \), that are relevant to the topic; and a collection, \( C \), from which the relevant documents are selected. Factors that can be used to estimate topic difficulty include:

- the divergence between the queries and collection: \( d(Q, C) \);
- the divergence between the relevant documents and the collection: \( d(R_l, C) \);
- the divergence between the queries and the relevant documents: \( d(Q, R_l) \); and
- further, the divergence between the queries: \( d(Q, Q) \), and between the relevant documents: \( d(R_l, R_l) \), gives an indication of the scope of the topic.

Divergence values can be estimated using various techniques. Indeed, to some extent, all prediction approaches presented in this section can be categorised into one of the distances
presented in the model. For example, the IDF based predictors measure the divergence between queries and the collection, \(d(Q, C)\), while query clarity attempts to measure the divergence between the relevant set and the query, \(d(Q, R_t)\). Finally, counting the number of distinct terms in a query measures the divergence within the topics, \(d(Q, Q)\).

In their work, Carmel et al. use Jensen-Shannon divergence (JSD) [Lin, 1991] to estimate the divergence between distinct objects, whereas to measure the divergence between the relevant documents, or between the queries, they count the number of aspects covered by each set. In these cases, an aspect is a concept category and is determined by clustering.

Experimental results show a strong positive correlation between the query and collection distance \(d(Q, C)\) to average precision, suggesting that queries containing terms that stand out from the collection are likely to perform well. On a similar note, a strong positive correlation between the distance of relevant documents and the collection \(d(R_t, C)\) to average precision is also shown, which indicates topics with documents that stand out from the collection are less difficult. A weaker, but positive, correlation is presented for the distance between the documents themselves \(d(R_t, R_t)\). This suggests that topics with a diverse range of results have the potential to be difficult to resolve. The results of Carmel et al. reflect those of other difficulty prediction research, where to date, the most successful approaches have revolved around differentiating either queries or estimated relevant documents from the collection.

In Chapter 7 we propose a new approach to query difficulty prediction that considers the document access from past queries to determine the difficulty of the query.

### 2.13 Query Independent Evidence

A second method proposed to improve query evaluation accuracy is the use of query independent evidence. The focus of most query evaluation research is centered on the relationship of query terms to the documents in the collection. However, it has been argued that documents alone have qualitative properties that can help determine their likelihood of being relevant [Singhal et al., 1996b; Lafferty and Zhai, 2001]. By associating each document with a prior probability of relevance, these qualitative properties can be incorporated into similarity metrics, and therefore improve the ranked results presented to the user.

The use of query-independent evidence in evaluation has been explored in several works. Singhal et al. [1996b] show a positive correlation between the length of a document and relevance. Lafferty and Zhai [2001] suggest potential gains with the use of document priors based on document length or link-analysis.
CHAPTER 2. BACKGROUND

To date, the use of document priors in information retrieval has primarily been explored within the context of entry page and named page search. Entry page search is the task of finding the home-page of the topic associated with the user query. For example, for the query “Coke”, the user would be after the Coca-Cola home-page http://www.coca-cola.com. In named page search, the user is after a specific page, however that page may or may not be a home-page. Unlike an ad-hoc query, there is typically one specific page that will satisfy the user need. Such queries can therefore be categorised as navigational under the classification of Broder [2002].

Upstill et al. [2003] examine priors for the entry-page-finding task. They examine the applicability of in-link count, URL length, and PageRank [Brin and Page, 1998] based priors to improve the effectiveness of search. In their approach, they obtain a ranked result set for each query, and then filter the results based on the value of the document prior. They find that URL priors work best on both full document text and anchor only search. Where an anchor is the text description linking one web document to another web document. They also report moderate, yet mostly insignificant improvements with link based and PageRank priors. They suggest that a possible reason for the lack of success of link based evidence in their work is the scale of their test collections and the lack of interlinked documents therein.

Kraaij et al. [2002] also make use of document priors for entry page search. They examine priors based on document length, in-link count, and URL type. For each feature they present correlations that demonstrate the relationship between the feature and relevance. They apply the priors both individually and in combination to a language model based similarity metric and report improved accuracy for in-link count priors, URL type priors, and a combination of the two. However, they report that document length based priors do not work well for the entry page search task. This is to be expected as the length prior of Singhal et al. [1996b] is based on principle that longer documents discuss a wider range of topics, and is therefore oriented towards informational user needs.

Craswell et al. [2005b] combine document priors with the Okapi BM25 similarity metric. In their work they examine priors based on PageRank, in-link count, URL length, and click distance (the minimum number of clicks required to reach a particular page from a specified root document). Craswell et al. propose various techniques to linearly combine the document priors with the Okapi BM25 document score. However, they find that this fails to improve accuracy as simple linear combination of values assumes that the result set obtained from the Okapi BM25 ranking is independent of the document priors. The authors then proceed to propose an adjustment that estimates the proportion of the Okapi BM25 document weighting
that includes the prior to be offset before ranking. The results show that, after adjustment, all features can improve accuracy, with PageRank producing the best results.

Within TREC, other groups have also made use of URL length and structure based priors to improve the performance of their systems for both named page and entry page search [Ra et al., 2001; Tomlinson, 2003; Collins-Thompson et al., 2002; Xi et al., 2002; Ogilvie and Callan, 2003]. While such evidence is intuitive for the task of named- and home-page finding, its place in an ad-hoc (informational) search task is questionable as the structure or length of a URL are less likely to be related to the content of the document. For example, the following URLs are taken from articles on three well-known news web-sites:

- ABC News: http://www.abc.net.au/news/newsitems/200703/s1871033.htm,
- BBC News: http://news.bbc.co.uk/2/hi/business/6446193.stm, and

While the URLs provide information regarding the date of the articles, no other features indicate the content of the documents. Further, the application of such techniques in enterprise search — where documents are not necessarily maintained on a web server — is likely to require specific knowledge of the file system structure.

In Chapter 7 we apply a prior based on the likelihood of a document appearing in the result set to the similarity ranking function. Unlike other work in the area, our proposed predictor is not dependent on URL features or document link structures. As such, the prior is applicable to the task of ad-hoc search.

2.14 Summary

The field of information retrieval is both vast and varied with a great number of significant contributions in recent years. Such advances can be found in fundamental principles, such as the similarity metrics used; and interesting search system augmentations, such as query difficulty prediction algorithms and prior-based result enhancement. Improvements in disk capacities, processor performance, and the vast quantities of data that are becoming available have contributed to these advances. However, these factors also continue to drive the demand for faster and more accurate search.

The studies of query logs discussed in Section 2.1.1 have shown that users have high expectations of search systems. Providing on average less than three query terms, and rarely
examining more than two page of search results, search engines must maximise the limited information provided by users to produce high quality results. These studies also suggest a high volume of redundancy that can be taken advantage of at various phases of the query evaluation process.

As the scale of collections grow, demands for access to disk based inverted lists, vocabulary and document mapping table entries, and collection files also increases. Previous caching techniques have focused on the caching of inverted lists and result sets, but have not considered other data structures. Index organisations that allow dynamic query time pruning have the potential to further improve inverted list caching, yet to date have not considered the history of queries that users pose to the search system. Similarly, while the most successful techniques to predict query difficulty have relied on the relationship between the query and the collection, they have yet to consider past queries.

In subsequent chapters we build upon the background work presented in this chapter. We examine the repetition in query logs and propose techniques to reorganise the index for more efficient query evaluation. In this regard we explore query evaluation optimisation techniques such as static and dynamic list pruning, index compression and collection reordering. Continuing our exploration of search engine efficiency techniques we explore the effects of search engine caching and propose a model to measure the impact of differing caching techniques at search time using real query logs and collections. Finally, we derive information from query logs for use as external evidence and consider this first, as a form of document prior, and second, for the task of difficulty prediction.
Chapter 3

Access-Ordering

In this chapter, we investigate whether indexes can be reorganised based on past query usage patterns so that they better support future queries. Our aim in developing this approach is to explore whether past queries can be used to identify documents that are likely to be relevant responses to future queries, and whether this can be used to organise an index for faster or more accurate query evaluation.

We begin by presenting observations on the access frequencies of documents in Section 3.1. In Section 3.2, we propose an index design that takes advantage of document access frequency. In Section 3.3, we examine alternate approaches that can be used to derive the index structure. List pruning techniques that allow efficient evaluation of user queries are discussed in Section 3.4. Compression techniques are discussed in Section 3.5. We present experimental evaluations of our methods in Section 3.6.

3.1 Document Access Frequencies

The frequency of distributions of many items in web retrieval are often very skew. For example, the frequency of word occurrences in large newswire collections is such that the commonest word, “the”, occurs twice as often as the second most common word, “of”, which in turn occurs 1.1 times as often as “to”, and so on; the ratios appear to vary slightly between collections, but the trends are the same [Williams and Zobel, 2005]. Similar behaviours are seen in the frequency of new words occurring in collection document sizes, where the number of words grows sublinearly with the collection size [Williams and Zobel, 2005]; the use of query terms, where 9% of all query terms are drawn from a set of only 0.05% of unique terms [Spink et al., 2001]; and in the caching behaviour of web documents [Breslau et al., 1999].
To investigate whether this skew distribution extends to documents that are returned in response to queries, we examined more than 1.9 million ranked queries on a collection of around 1.6 million documents. The queries were taken from the Excite 1997 and 1999 query logs after removing any queries that contained terms that were deemed to be offensive, while the collection used was the TREC WT10g collection (described in Table 2.3, page 19).

For each query, we evaluated the similarity of the query to the documents in the collection using the Okapi BM25 formulation described in Section 2.6.2 (page 30), and counted the number of times each document appeared in the top $n$ documents of any query (we used $n = 10$, $n = 100$, and $n = 1,000$). We refer to this count as the access count of each document.

A plot of the document access counts for the query set is shown in Figure 3.1. On the y-axis, the graph shows the access count (the frequency with which the document appeared in the top $n$ results). On the x-axis, the documents are organised by decreasing rank of frequency so that the most-frequently retrieved document (with the highest access count) is shown at 0 and the least-frequently retrieved (with the lowest access count) at 1.6 million.

**Figure 3.1:** A plot showing the number of times each document in the WT10g collection is ranked in the top 10, 100 and 1,000 responses for around 1.9 million Excite queries. Documents are sorted and labeled in access count order.
3.1. DOCUMENT ACCESS FREQUENCIES

Figure 3.2: A plot showing the number of times each document in the WT10g collection is ranked in the top 1,000 responses for around 1.9 million Excite queries. Points on the line are documents that have been judged as relevant for any of the TREC-9 queries used in our experiments.

Not surprisingly, Figure 3.1 shows that a small fraction of the documents in the collection are frequently ranked highly in response to the queries. At \( n = 1,000 \), the most frequently accessed document (TREC document WTX030-B33-366) has an access count of 76,740, that is, it appears in the top 1,000 results for around 4% of the queries. Further, the 10% most frequently accessed documents account for almost 40% of all accesses.

Many documents in the collection are accessed infrequently or not at all. For example, even for \( n = 1,000 \), 13% of the documents in the collection were accessed by 100 queries or less, that is, by less than 0.005% of queries, while 1.8% of the collection is not accessed at all. With smaller values of \( n \), these effects are more pronounced: for \( n = 10 \), almost 40% of the collection is not accessed, and for \( n = 100 \) around 8% of documents are never retrieved.

The line plotted in Figure 3.2 does not show all documents from the collection. Instead, we have only shown the 2,605 documents that have been judged as relevant to any of the fifty TREC-9 queries that are used to compare the behaviour of web retrieval techniques; the TREC framework and queries are discussed in Section 2.2.4 (page 18). This plot illustrates
that documents that have been judged as relevant to user information needs are also those with high access counts: around 39% of the relevant documents are in the 10% of those with highest access counts. Only 7% of the relevant documents are in the half of the collection with the lowest access counts. Note that due to the pooling technique used to judge relevant documents in the TREC environment, a total of 59,720 unique documents are judged over the 50 topics, with the rest of the collection unjudged.

We conclude from this initial investigation that document accesses are highly skew when averaged over a large number of queries. In addition, we empirically observed that there is a correlation between the access count of a document and its likelihood of being relevant to the user. With this motivation, we propose access-ordered indexes in the next section.

### 3.2 Index Design

Based on our observations in the previous section, we propose that postings be organised within each postings list by decreasing access count. Our motivation is to store those postings that are most likely to appear in the top answers to queries at the beginning of each list and, as in the frequency- and impact-ordered approaches described in Section 2.9 (page 35), to partially process the postings lists at query time. However, in contrast to previous approaches, our scheme does not rely only on the document collection and ranking function. Instead, it includes the properties of the queries — the temporal information that users provide — to further refine index design. A successful access-ordered index should permit accurate query evaluation compared to evaluation based on processing all postings, while being faster because less data is used.

Consider an example of an access-ordered index using the Tempest collection introduced in Section 2.3 (page 20). Suppose that we have a query log of 100 queries, and that we choose to increment access counts when a document appears in the top ten answers in response to a query.

Figure 3.3 shows the ranked results when evaluating two sample queries on the Tempest collection. To establish an access-ordering, we would increment a counter each time a document appeared in the top ten results for a query. Therefore, after evaluating the first query, documents 33, 69, 81, 100, 352, 385, 624, 637, 700, and 744 are each assigned an access count of one. After the second query, the access counts for documents 5, 108, 175, 178, 344, 385, 396, 511, 682, and 744 are all incremented by one. At this point, documents 385 and 744 each have appeared in the top ten results for both queries and have an access count of two.
3.2. INDEX DESIGN

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>Doc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.218642</td>
<td>81</td>
</tr>
<tr>
<td>2</td>
<td>0.757494</td>
<td>385</td>
</tr>
<tr>
<td>3</td>
<td>0.748940</td>
<td>744</td>
</tr>
<tr>
<td>4</td>
<td>0.737558</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>0.728168</td>
<td>69</td>
</tr>
<tr>
<td>6</td>
<td>0.725724</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>0.686610</td>
<td>637</td>
</tr>
<tr>
<td>8</td>
<td>0.685745</td>
<td>700</td>
</tr>
<tr>
<td>9</td>
<td>0.621150</td>
<td>352</td>
</tr>
<tr>
<td>10</td>
<td>0.545150</td>
<td>624</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>Doc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.774413</td>
<td>511</td>
</tr>
<tr>
<td>2</td>
<td>0.718244</td>
<td>175</td>
</tr>
<tr>
<td>3</td>
<td>0.716931</td>
<td>178</td>
</tr>
<tr>
<td>4</td>
<td>0.523467</td>
<td>108</td>
</tr>
<tr>
<td>5</td>
<td>0.511644</td>
<td>385</td>
</tr>
<tr>
<td>6</td>
<td>0.479165</td>
<td>344</td>
</tr>
<tr>
<td>7</td>
<td>0.436609</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>0.435824</td>
<td>744</td>
</tr>
<tr>
<td>9</td>
<td>0.433951</td>
<td>682</td>
</tr>
<tr>
<td>10</td>
<td>0.433951</td>
<td>396</td>
</tr>
</tbody>
</table>

Figure 3.3: Top ten ranked results for two sample queries evaluated on the Tempest collection. For each of the top ten documents returned by the system, we show the score assigned to the document by the ranking function, and the rank of the document based on the assigned score.

Preliminarily, we might expect documents 385 and 744 to be highly ranked in response to future queries. After processing further queries the access-ordering becomes more refined. Table 3.1 lists the most frequently accessed documents from the Tempest collection after processing a further 98 queries from a sample query log.

Consider now the term “state” which occurs in documents 7, 79, 471 and 606. The conventional document-ordered postings (with frequencies, and taking document differences) are:

\[
\langle 77, 2 \rangle, \langle 2, 1 \rangle, \langle 392, 1 \rangle, \langle 135, 1 \rangle.
\]

To reorganise this postings list by access frequency, we inspect the frequencies gathered in the training phase, and sort the list as follows:

\[
\langle 471, 1 \rangle, \langle 7, 2 \rangle, \langle 79, 1 \rangle, \langle 606, 1 \rangle.
\]

Note that the document identifiers are no longer stored as differences, since an absolute ordering of postings by document is no longer a guarantee; as we show later, this has a significant negative impact on the compression of the lists. Note that the access frequency,
Table 3.1: Access count values for 25 most accessed documents in Tempest collection after processing 100 queries, and considering the top 10 results per query as accessed documents.

<table>
<thead>
<tr>
<th>Document</th>
<th>Access count</th>
<th>Document</th>
<th>Access count</th>
</tr>
</thead>
<tbody>
<tr>
<td>386</td>
<td>14</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>87</td>
<td>11</td>
<td>73</td>
<td>5</td>
</tr>
<tr>
<td>385</td>
<td>10</td>
<td>83</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>93</td>
<td>5</td>
</tr>
<tr>
<td>56</td>
<td>7</td>
<td>359</td>
<td>5</td>
</tr>
<tr>
<td>548</td>
<td>7</td>
<td>408</td>
<td>5</td>
</tr>
<tr>
<td>576</td>
<td>7</td>
<td>443</td>
<td>5</td>
</tr>
<tr>
<td>731</td>
<td>7</td>
<td>567</td>
<td>5</td>
</tr>
<tr>
<td>77</td>
<td>6</td>
<td>624</td>
<td>5</td>
</tr>
<tr>
<td>115</td>
<td>6</td>
<td>641</td>
<td>5</td>
</tr>
<tr>
<td>159</td>
<td>6</td>
<td>695</td>
<td>5</td>
</tr>
<tr>
<td>383</td>
<td>6</td>
<td>744</td>
<td>5</td>
</tr>
<tr>
<td>690</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$a_d$, is not stored in the postings since it need only be stored once per document and, therefore, can be compactly represented as an additional component of the document mapping table. Figure 3.4 shows the access-ordered inverted lists for the terms depicted in Figure 2.2 (page 20).

### 3.3 Access Counting

The goal of the access count generation process is to produce the set of access counts that reflect the frequency with which each document is presented in the search results. Access counts establish an ordering of document accesses beginning with the most frequently accessed document in the collection and continuing to the least frequently accessed document.

Several considerations arise when assigning counts to a document. First, the number of queries used to generate the access counts can have a significant effect on the final ordering. It is desirable that the ordering reach a level of stability where running further queries does not change the access-order of the documents in the collection. A related consideration is the number of results considered per query. That is, if the top $n$ documents per query are
3.3. ACCESS COUNTING

<table>
<thead>
<tr>
<th>Term</th>
<th>Access Count 1</th>
<th>Access Count 2</th>
<th>Access Count 3</th>
<th>Access Count 4</th>
<th>Access Count 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fairy</td>
<td>(642, 2)</td>
<td>(649, 1)</td>
<td>(642, 2)</td>
<td>(434, 3)</td>
<td>(430, 1)</td>
</tr>
<tr>
<td>farewell</td>
<td>(47, 2)</td>
<td>(464, 1)</td>
<td>(47, 2)</td>
<td>(436, 1)</td>
<td>(480, 2)</td>
</tr>
<tr>
<td>fish</td>
<td>(386, 6)</td>
<td>(283, 1)</td>
<td>(386, 3)</td>
<td>(430, 1)</td>
<td>(750, 1)</td>
</tr>
<tr>
<td>state</td>
<td>(471, 1)</td>
<td>(7, 2)</td>
<td>(471, 1)</td>
<td>(606, 1)</td>
<td>(22, 1)</td>
</tr>
<tr>
<td>storm</td>
<td>(386, 3)</td>
<td>(408, 2)</td>
<td>(386, 3)</td>
<td>(79, 1)</td>
<td>(22, 1)</td>
</tr>
<tr>
<td>swords</td>
<td>(565, 1)</td>
<td>(566, 2)</td>
<td>(565, 1)</td>
<td>(606, 1)</td>
<td>(22, 1)</td>
</tr>
<tr>
<td>sycorax</td>
<td>(159, 2)</td>
<td>(511, 2)</td>
<td>(159, 2)</td>
<td>(136, 1)</td>
<td>(132, 1)</td>
</tr>
<tr>
<td>vision</td>
<td>(624, 1)</td>
<td>(612, 1)</td>
<td>(624, 1)</td>
<td>(136, 1)</td>
<td>(132, 1)</td>
</tr>
</tbody>
</table>

Figure 3.4: Access-ordered lists for selected terms from Tempest collection.

awarded an access count, what value should \( n \) take? A final consideration in generating access counts involves utilizing the ranked order of the search results. That is, should the top \( n \) documents be awarded equal weight for appearing in the search results?

As shown in Section 3.2, our initial technique to generate access counts is to increment a counter for each document that appears in the top \( n \) search results of the training query set. Such a method awards all documents equal weight regardless of their rank. We label this the **Static** access counting scheme.

**Static** access counting considers all top ranked documents equally. It may be argued that those documents that are highly ranked in response to a query are more valuable than those that are ranked further down in the result set. An alternate technique that can be used to incorporate ranked results into the access counts is to award a larger increment to documents that rank highly.

We propose a second scheme, which we label **Linear** counting, where increments to document access counts are based on the rank of the document in the results set. For example if \( n \) results are to be awarded an access count, the first ranked document has its access count incremented by \( n \), the second ranked document is awarded \( n - 1 \), and so on down to the \( n \)th document which has its score incremented by 1. Such a scheme reduces the possibility of over-valuing those documents that appear frequently in the result sets at low ranks.

The **Linear** counting scheme assumes a uniformly decreasing value for each document retrieved in response to a query. However, the magnitude of the increment is dependent on the value of \( n \). For example, in the case of \( n = 10 \), the document returned at rank 9 is allocated an increment that is equivalent to one-ninth the value of the document at rank 1.
In contrast, for the case of $n = 1,000$, the document at rank 9 is allocated an increment that is only fractionally smaller than that of the top ranked document, while a document that is equivalent to one-ninth the value of the top rank is not encountered until the 900th ranked result.

An alternative is to weight documents in a non-linear manner, such that the value of $n$ has a reduced impact on document scores as the rank of the document decreases. To this end we propose a third weighting scheme, Logarithmic, where the highest ranked documents are assumed to be twice as valuable as the following set of ranked documents, which in turn are assumed to be twice as valuable at the next set, and so on. In this approach, the ranked results are partitioned into five buckets of uniform size. The documents appearing in the highest ranked partition are awarded an access count increment of $2^4$, documents in the next highest partition are awarded an increment of $2^3$, and so on to the final partition where documents are awarded an increment of $2^0$.

The scope in variation of possible counting schemes is large. Here, we have proposed three schemes that cover distinct principles. First, the Static scheme treats all top ranked documents equally. Second, the Linear scheme values documents directly proportional to their rank in the results. Third, the Logarithmic scheme values documents on an exponential scale, thus valuing those documents in the top ranks significantly more than others. In Section 3.6.3 we report results across varying values of $n$, for each of the Static, Linear, and Logarithmic access count generation schemes. In Chapter 4, we further explore access counting by examining the effect of differing quantities of training queries.

3.4 Early Termination Heuristics

To take advantage of an access-ordered index, a strategy is required to heuristically abandon postings list processing before entire lists are processed. Perhaps the simplest approach is to only process a fixed number of postings $P$ from each list, with the motivation that short lists that may have a larger impact on the ranking function (because of smaller $f_t$ values) will be processed in their entirety, and longer lists will be partially processed. We refer to this scheme as MAXPOST.

For example, consider a threshold value of $P = 2$. For the single word query “storm” from Figure 3.4, only the postings for documents 386 and 408 would be processed. The computational saving is that the posting for document 22 is not decoded and processed, but
the disadvantage of a fixed threshold is that document 22 can never be returned in response to the query; we return to this discussion in Section 3.6.4.

There are several other possible approaches to partial list processing. These include:

- **MINACCESS**: Processing only those postings with an access count greater than a threshold value $M$. This scheme favours postings with a high access count. Like the MAXPOST scheme, this approach has the disadvantage that postings with an access count below the specified threshold cannot be returned in response to their respective terms.

- **AVGACCUM**: Calculating an average accumulator contribution for the postings that have been processed in a list, and stopping when this falls below a specified threshold $A$. Although the lists are not directly ordered by accumulator contribution, we assume that for documents to be highly ranked in an access-ordering, their postings on average contribute more to accumulators than do postings with low access counts. We report results with this AVGACCUM scheme in Section 3.6.4.

- **AVGROLLACCUM**: Extending AVGACCUM so that the average is computed over a moving window of $D$ documents, with the aim of detecting more local change in accumulator contributions. We call this AVGROLLACCUM.

- **TWO-PHASE**: Setting two threshold values based on the access count of the next posting processed. The first threshold determines when list processing will stop adding new accumulators. The second determines when to stop updating existing accumulators. The two-phase scheme is described in detail below.

Based on the work of Persin et al. [1996], described in Section 2.9.1 (page 36), we propose a modified version of the frequency-ordered termination heuristic that can be applied to access-ordered indexes. Using this approach, values of $c_i$ and $c_a$ respectively are chosen based on access counts. However, we propose two minor variations to the threshold calculation. First, the initial work on frequency-ordering was based on long-topic style queries drawn from early TREC experiments, where the within-query frequency of a term, $f_{q,t}$, can have a significant effect on the final similarity score. In our work, we focus on short web queries that typically contain one to three terms, and we therefore assume a value of $f_{q,t}$ equal to one. Second, the formulation for term weight $w_t$ varies with the selected similarity metric. In our work, we use the Okapi BM25 function described in Section 2.6.2 that can result in zero-valued term weights, and so, to prevent divide-by-zero errors we add a constant value of 1 to the threshold calculation denominator.
Figure 3.5: TWO-PHASE pruning scheme on a three term query: king and magic. Terms are processed in ascending IDF order. Prior to processing each inverted list, the thresholds values $a_n$ and $a_i$ are updated. As each term is processed the threshold values increase and further restrict the postings to be processed. The plot above each inverted list shows the access count distribution of the postings in that list, and the effect of the increasing thresholds on the processing of that list.

The results of adapting the termination heuristics are the following functions:

$$a_i = \frac{c_i S_m}{w_i^2 + 1},$$

$$a_a = \frac{c_a S_m}{w_i^2 + 1},$$

where $c_i$ and $c_a$ are the tunable constants that determine when to cease accumulator creation and when to cease accumulator updates, $S_m$ is the maximum accumulator value seen so far, and $w_t$ is the weight of the current term.
3.4. EARLY TERMINATION HEURISTICS

We refer to this early termination heuristic as two-phase pruning. Query processing proceeds as follows: first, query terms are ordered by inverse document frequency, the value $S_m$ is initialised to 0, and the first query term is selected to be processed. Second, for the selected query term, the two threshold values $a_i$ and $a_a$ are calculated. Third, the postings list for the term is processed with accumulators being created or updated while the access counts for each posting remains greater than or equal to $a_i$ and $a_a$ respectively; processing of the current list ceases when an access count is less than $a_a$. Last, after processing a list, the thresholds are recalculated for the next term and the process repeated from the second step for that term. If during the processing of the postings the accumulator contribution of a posting is greater than $S_m$, the value of $S_m$ is updated to reflect the new highest accumulator contribution. This process is illustrated in Figure 3.5. Although the threshold values $a_a$ and $a_i$ increase as more terms are processed, the proportion of postings processed per individual list is variable and dependent on the access count distribution of the postings therein.

The continue scheme described in Section 2.8 (page 34) is an effective technique for saving memory and reducing computational cost. Anh and Moffat [2002b] experimented with the combination of various pruning techniques for impact-ordered indexes. They reported that a combination of their impact-order prune schemes with continue produced the best overall compromise in results between accuracy and efficiency.

To restrict the memory consumption of query evaluation, we limit the number of accumulators initialised at query time using the continue scheme in combination with our pruning approaches. When the threshold amount of accumulators initialised has been reached by any of the proposed processing schemes, postings are processed in update-only mode, allowing existing accumulators to have their scores incremented, but not allowing the addition of new accumulators.

In their impact-ordered work, Anh and Moffat [2002b] proposed several index pruning schemes that build on the quit and continue strategies of Moffat and Zobel [1996]. They defined a block-fine approach where the product of transformed document and query impacts are sequentially penalised until the accumulator contribution of the processed postings block is zero, at which point processing of the list is stopped. As the ordering of our access-ordered indexes is not directly related to the effect of a posting on the similarity function this approach cannot be applied to our index. However, the avgaccum and avgrollaccum schemes allow for a similar granularity of pruning by terminating the processing of a list when the average contribution of preceding postings falls below a given threshold. Anh and Moffat also defined a variation of the block-fine scheme, term-fine, where a penalty
is applied to each sequential term—as opposed to each block in a single list—therefore increasing the probability of pruning postings lists for terms that are processed later during the query evaluation process. We have not applied a penalty at a term level as in the **TERM-FINE** approach, and leave this open as an area of further work.

Recently, Lester et al. [2005a] showed that the **CONTINUE** scheme is biased towards documents that appear early in the collection. They proposed an accumulator management strategy that reduces this bias and limits the number of accumulators initialised at query time. In their approach, an accumulator threshold is determined by sampling the contribution of postings in each inverted list, and is updated as postings are processed. When accumulators are encountered that do not exceed the current threshold, they are removed from the accumulator set. For an access-ordered index, such an approach is unlikely to work. As postings are ordered by their likelihood of occurrence in the result set, a correlation between accumulator contribution and position in the list is present. Therefore, any estimates of the expected contribution of postings based on a sequential segment of the inverted list will be skew and not representative of the entire list. As such, the application of the scheme proposed by Lester et al. would likely result in a regular reduction in the adaptive threshold (as lower contributing postings are processed), and a bias against the frequently accessed documents (which are likely to be skipped by an artificially high threshold early in the list processing).

### 3.5 Index Compression

We now consider the techniques used to compress postings lists and propose techniques for the effective storage of access-ordered indexes. Section 2.4.1 (page 22) describes standard approaches to inverted list compression. We use a variable-byte oriented technique based on the work of Scholer et al. [2002] that has been shown to provide effective compression with the benefit of rapid decompression.

Integer compression schemes are applied to postings lists, after the postings are organised using document-, frequency-, or impact-ordering, and compaction techniques applied. For example, Persin et al. [1996] proposed grouping postings in each list with the same \( f_{d,t} \). This has two advantages for index compaction: first, grouping postings into blocks of postings with the same \( f_{d,t} \) permits storing each \( f_{d,t} \) value once; and, second, postings within a block can be sorted by document identifier to permit document differences to be stored within each block. However, overall, the largest contribution to index compaction in frequency-ordered
3.5. INDEX COMPRESSION

indexes is the elimination of the redundant storage of $f_{d,t}$ values. With the organisation in place and compaction applied, variable-byte integers can be used to represent the list.

Access frequencies are unrelated to both the position of a document in a collection and the frequency of the term in each document. Therefore, since the postings are not ordered by a property stored in the lists, taking differences between adjacent values does not yield a list of small integers. Accordingly, as we show in Section 3.6.6, without additional compaction, access-ordered indexes are around 160% of the size of a conventional document-ordered index. This large size affects index space requirements on disk, query evaluation speed, and memory caching. We therefore need to consider approaches to reducing index size, based on storing postings in blocks where document order can be maintained.

3.5.1 Access-Block Compaction

We now consider techniques for compaction of access-ordered indexes. Each scheme we describe aims to form blocks of postings, and then to organise the postings within the block by document order to permit differences to be stored. We report results with the schemes in Section 3.6.6.

Basic access-block compaction. Our first approach to compacting access-ordered indexes is motivated by the work of Persin et al. [1996]. It is likely that many documents within a collection share the same access count — particularly for low access counts — and so postings can be blocked together by that access count. Then, within each block, postings can be sorted by document identifier and differences taken. However, unlike blocking by $f_{d,t}$ values, this does not reduce the information that is required to be stored per block. Indeed, an additional integer must be stored per block that indicates the number of postings in the block; we refer to this value as $f_b$.

Consider an example. The term “fish” that has the following document-ordered postings list:

$\langle 283, 1 \rangle, \langle 386, 6 \rangle, \langle 430, 1 \rangle, \langle 436, 1 \rangle, \langle 480, 2 \rangle, \langle 750, 1 \rangle$.

Assuming that the access counts in the collection are 2, 14, 1, 2, 1 and 0 for documents 283, 386, 430, 436, 480 and 750 respectively, a basic access-block compacted inverted list for the term “fish” would appear as:

$[1]\langle 386, 6 \rangle, [2]\langle 283, 1 \rangle, \langle 153, 1 \rangle, [2]\langle 430, 1 \rangle, \langle 50, 2 \rangle, [1]\langle 750, 1 \rangle$.

Each block begins with a block-size value $f_b$ shown in square brackets, and where the block-size is greater than one, the differences between adjacent document identifiers are stored.
For short lists, the net benefit to index compaction through blocking is outweighed by the requirement to store the extra $f_b$ value per block.

**Fixed access-block compaction.** A simple extension of the basic access-blocked technique is to use a constant block size, that is, to set $f_b = k$ for a block size $k$. This avoids the requirement to store $f_b$ for each block, since all blocks (except, in most cases, the last in each list) contain $f_b$ postings. However, it relies on careful choice of $k$: when $k = 1$, exact access-ordering is maintained and no compaction achieved, while when $k$ is set to the length of the largest list in the index, strict document-ordering is enforced.

Returning to our example list for the term “fish”, assuming $k = 3$, and access count values as above, the fixed access-block inverted list becomes:

\[
\begin{array}{l}
\langle 283, 1 \rangle, \langle 153, 6 \rangle, \langle 50, 1 \rangle, \\
\langle 430, 1 \rangle, \langle 50, 2 \rangle, \langle 270, 1 \rangle,
\end{array}
\]

Each block is shown within square brackets. Fixed access-block compaction does not guarantee that all postings with the same access count are stored in the same block. Similarly, postings in a block may have different access counts; this is particularly likely for the postings with high access counts in each list or for short lists. Because of this, early termination heuristics must be carefully considered.

The two-phase and minaccess pruning approaches rely on an absolute ordering of postings that have decreasing access count values. However, several of the compaction schemes proposed in this section rely on breaking this ordering. For these schemes, pruning requires that the thresholds are no longer compared to the access count of each posting, but instead are compared to the highest access count value in the block of postings.

In Section 3.6.6, we investigate choices of $k$, and show the effect of this compaction technique when combined with the early termination heuristics proposed above.

**Exponential access-block compaction.** Another approach to block-based compaction is to allocate block sizes based on a function with zero or more parameters. We have experimented with one approach in this class, though there are many options that can be considered. As we have shown, document access counts follow an inverse power law distribution: this has the effect that few documents have unique access counts, some share access counts with a few other documents, and the majority share low access counts with most others. Therefore, to permit blocks to contain most postings with identical access counts and for identical access counts not to span blocks, we propose a global scheme, where the first block in each list has $f_b = 1$ postings, the second $f_b = 2$, the third $f_b = 4$, and so on, with each block storing twice as many postings as its predecessor.
For the example list “fish”, the exponential access-block compacted list is as follows:

\[
\langle 386, 6 \rangle, \langle 283, 1 \rangle, \langle 153, 1 \rangle, \langle 430, 1 \rangle, \langle 50, 2 \rangle, \langle 270, 1 \rangle.
\]

This approach permits early termination between blocks, with reduced likelihood that postings with identical access counts span blocks. The drawback is that compaction will be most effective for long inverted lists, while, for short lists, compaction will be limited by small sized blocks. Compaction of short lists can be improved by setting an initial block size greater than 1. However, in experiments not reported here, we were unable to further reduce index size significantly by increasing the initial block size.

### 3.5.2 Other approaches

We have considered other approaches to achieving compaction. For example, we experimented with schemes where the postings at the beginning of each list — those with access counts greater than a threshold — are access-ordered and remaining postings are document-ordered. We did this based on the observation that absolute access-ordering permits careful application of termination heuristics for postings with high access counts, while compaction can still be achieved for the majority of postings with low access counts. Unfortunately, such approaches work well only for long lists, achieving low compaction overall for reasonable parameter choices. We therefore report experiments with only the schemes described previously in this section.

### 3.6 Results

We now present the results of our experiments with access-ordered indexes. We begin in Section 3.6.1 with an overview of our experimental environment including collection and query set details. Section 3.6.2 describes our overall results, and the following subsections detail the effect of parameter choices. Overall, our results show access-ordered indexes reduce the query evaluation costs of web search.

#### 3.6.1 Collection and Queries

The NIST Text Retrieval Conference (TREC) is an annual forum for experimentation, discussion, and advancement of information retrieval research [Voorhees and Harman, 2005]. It provides to participants collections, search topics, and relevance judgements for those topics and collections. In this chapter, we make use of the WT10g web track document collection.
Table 3.2: *Overall results showing the efficiency and effectiveness of document-, impact-, and access-ordered indexes with the best parameter settings described in subsequent sections. Effectiveness is measured using TREC topics 451–500, and efficiency measured using 10,000 queries drawn from the Excite logs.*

<table>
<thead>
<tr>
<th>Scheme</th>
<th>MAP</th>
<th>R-Prec.</th>
<th>P@10</th>
<th>Postings processed</th>
<th>Elapsed time per query (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doc. Ord. Cont.</td>
<td>0.1865</td>
<td>0.2170</td>
<td>0.2708</td>
<td>93.5</td>
<td>80.0</td>
</tr>
<tr>
<td>Doc. Ord. Cont. Stop</td>
<td>0.1779</td>
<td>0.2147</td>
<td>0.2562</td>
<td>10.7</td>
<td>37.9</td>
</tr>
<tr>
<td>Impact-Ordered</td>
<td>0.1916</td>
<td>0.2251</td>
<td>0.2511</td>
<td>11.8</td>
<td>28.8</td>
</tr>
<tr>
<td>Access Ord. Maxpost</td>
<td>0.1867</td>
<td>0.2172</td>
<td>0.2604</td>
<td>11.5</td>
<td>41.2</td>
</tr>
<tr>
<td>Access Ord. Two-phase</td>
<td>0.1856</td>
<td>0.2193</td>
<td>0.2667</td>
<td>10.0</td>
<td>48.5</td>
</tr>
</tbody>
</table>

of 1.69 million documents in 10 GB of data taken from a web snapshot of 1997. A discussion of TREC and the WT10g collection is presented in Section 2.2.4 (page 18).

To measure effectiveness we make use of 100 query topics (numbered 451–500 and 501-550 from TREC-9 and TREC-10 respectively) and use the query title field only in the search process. We use the TREC-9 topics as training data, and verify our results with the TREC-10 topics. The title fields of these topics are actual queries extracted from web query logs. For topics 451–500, the mean pool size per topic was 1,401, with a mean average of 52.3 relevant documents per topic. For topics 501–550, the mean pool size per topic was 1,408, with a mean of 67.3 documents per topic.

Access count data was gathered with queries taken from the Excite 97 and 99 query logs. The first log, collected in 1997, contains just over 1 million queries, while the second log, collected in 1999, contains a further 1.7 million queries [Spink et al., 2001]. The Excite query logs are discussed in Section 2.1.1 (page 10). Both have been filtered to remove queries on topics that may offend.

Finally, to determine whether our techniques have a significant impact on the performance of the system, we use Wilcoxon signed rank testing [Sheskin, 1997]. Our choice is based on the use of this measure by Zobel [1998] for similar purposes.
3.6. RESULTS

3.6.2 Overall Speed and Accuracy

Table 3.2 shows the results of our experiments with access-, impact-, and document-ordered indexes. The results are for the parametrised schemes that best trade efficiency and effectiveness; details of parameter choices for access-ordered indexes are discussed in subsequent sections. A continue strategy is used in all schemes with a limit of 20,000 accumulators. Such a limit accounts for over 1% of the documents in the collection which is recommended by Moffat and Zobel [1996] and Lester et al. [2005a]. The access-ordered schemes are trained using all 1.9 million queries from the Excite log using the Static access counting with \( n = 1,000 \).

We present three comparative baseline systems based on document- and impact-ordering. All schemes are implemented in the efficiency optimised Zettair search engine [Zettair] and are otherwise identical. Unless otherwise noted, term offsets are not stored in the index.

For document-ordered lists, we show results, both with and without stopping. The effect of stopping is clear. Although processing time is greatly reduced, this improvement is obtained at the cost of accuracy. The baseline document-ordered continue scheme has a mean average precision (MAP) of 18.5% with an average of 80.0 milliseconds per query. We also tested the quit strategy, but found a significant reduction in accuracy for MAP (at 5% significance level) and did not pursue this further as a baseline. Moffat and Zobel [1996] recommend not using the quit scheme in practice. Impact-ordering uses the collection-oriented impact-ordering approach described in Section 2.9.2 (page 38), using block-fine and term-fine pruning, with a continue strategy using 20,000 accumulators.

Overall, our results show that access-ordered indexes are effective and efficient: query evaluation is over 39% faster than with a document-ordered index (without stopping), and as accurate with no statistically significant difference (at 1% significance level) in the accuracy of MAP. When compared to document-ordering with stopping, the access-ordered index is as efficient without the reduction in accuracy.

However, when compared to the impact-ordered index, access-ordering is as accurate, with no significant difference in MAP at the 1% significance level, but slower. Although both schemes process approximately the same number of postings, the pruning techniques differ in the method of list processing termination. For an impact-ordered index, pruning is more likely to ignore terms during query evaluation. In our experiments, 26% of the query terms were bypassed by the impact pruning scheme, while for access-ordering all lists had at least some postings processed.
In the next chapter we explore further optimisations that improve the query evaluation costs of access-ordering in an effort to provide gains comparable to impact-ordering.

The column labelled “Postings processed” in Table 3.2 reports the fraction of all postings that were actually processed under the scheme. For the impact- and access-ordered schemes, the value of “Postings” is much lower than that of document-ordering with continue. In effect, this column shows where the query evaluation time savings are achieved. For continue, the values are not 100% because list processing can be abandoned when the accumulator limit is reached, and the postings list has been processed to the point that exceeds the document identifier value of the highest initialised accumulator.

Even without compaction, the maxpost approach is fast. The presented results use \( P = 55,000 \) with an uncompacted access-ordered index. We show that although this scheme allows efficient pruning, the robustness of the pruning parameter is questionable. Arguably, this approach may be unacceptable for all applications, as it may prevent a document being returned as an answer to a difficult information need; this can occur when a posting is the \( P \)th or later posting in the lists for all terms in the query. However, this problem can be addressed, by dynamically adjusting the threshold \( P \) depending on the query (perhaps by pre-computing a query quality measure [Cronen-Townsend et al., 2002]). We have not investigated this approach in detail.

The avgaccum and avgrollaccum schemes are either accurate and inefficient, or efficient and inaccurate compared to the maxpost scheme; however, in general avgrollaccum is the preferred accumulator-based scheme because the local window-based scheme is more sensitive to a drop in accumulator contributions, and therefore more effective in abandoning a list. Both schemes process more postings than maxpost for two reasons: first, but less importantly, an average has to be maintained; and, second, the schemes are less sensitive to list length as evidenced by the closer values in the “Postings” column of Table 3.2.

Finally, the two-phase pruning scheme is as accurate as the baseline, with no statistically significant difference in MAP at the 1% significance level, and as we show later stable, but remains slower than maxpost. The values shown use two-phase pruning with \( c_i = 250 \) and \( c_a = 250 \). Compaction using the fixed access-block scheme is also used, with a block-size \( k = 1,000 \). These parameters represent the best combination of values found during the training phase and are discussed below. This approach is significantly faster than document-ordered processing without stopping, and only slightly slower than document-ordered processing with stopping, while maintaining the accuracy of the unstopped approach.
3.6. RESULTS

Table 3.3: Effectiveness of variant counting schemes. Each row shows the fraction of the reordered collection that on average accounts for 10, 25, 50, 75, and 100 percent of the documents judged as relevant for TREC topics 451–500.

<table>
<thead>
<tr>
<th>Counting Scheme</th>
<th>10%</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static_10</td>
<td>5.9</td>
<td>10.1</td>
<td>22.9</td>
<td>36.3</td>
<td>68.0</td>
</tr>
<tr>
<td>Static_100</td>
<td>5.9</td>
<td>10.1</td>
<td>18.2</td>
<td>28.9</td>
<td>55.1</td>
</tr>
<tr>
<td>Static_1000</td>
<td>5.9</td>
<td>10.2</td>
<td>16.7</td>
<td>27.0</td>
<td>53.5</td>
</tr>
<tr>
<td>Linear_10</td>
<td>6.1</td>
<td>10.6</td>
<td>23.7</td>
<td>37.1</td>
<td>68.5</td>
</tr>
<tr>
<td>Linear_100</td>
<td>5.7</td>
<td>9.5</td>
<td>18.2</td>
<td>29.3</td>
<td>54.5</td>
</tr>
<tr>
<td>Linear_1000</td>
<td>5.9</td>
<td>9.9</td>
<td>16.7</td>
<td>26.7</td>
<td>53.4</td>
</tr>
<tr>
<td>Logarithmic_10</td>
<td>6.0</td>
<td>11.0</td>
<td>24.8</td>
<td>38.1</td>
<td>69.3</td>
</tr>
<tr>
<td>Logarithmic_100</td>
<td>5.6</td>
<td>9.2</td>
<td>18.7</td>
<td>29.9</td>
<td>55.2</td>
</tr>
<tr>
<td>Logarithmic_1000</td>
<td>6.0</td>
<td>9.8</td>
<td>17.0</td>
<td>27.2</td>
<td>53.1</td>
</tr>
</tbody>
</table>

3.6.3 Access Counts

We explored three different award schemes to generate the access counts, and for each scheme examined the effect of considering the top 10, 100, and 1,000 results per query for the scoring. To compare the various techniques we examined the distribution of relevant documents in the final ordering produced by each scheme over a set of TREC topics. For each of the fifty TREC-9 topics, we calculated the proportion of the ordered collection that accounts for the top 10, 25, 50, 75, and 100 percent of the relevant documents. The average for each interval was then taken. A relevant document was considered to be any document that was marked as relevant for that topic in the collection. Using this technique, the better scheme is the one that orders the collection such that the relevant documents are earlier in the ordering. Table 3.3 shows the results for each counting scheme. For example, using the Static scheme with 10 results per query, 10% of the relevant documents can be found in the first 5.9% of the access-reordered documents.

No scheme is clearly dominant, however, all three schemes improve as the number of results allowed per query is increased to 1,000. This suggests that it is better to allow more results per query with the possibility of noisy data than to restrict the results to the best matches while losing information. Further, as the number of results per query
increases, the difference in performance between the schemes reduces. This indicates that the various counting schemes converge over a large number of queries to produce a similar ordering.

Based on these results, we proceed with Static counting with 1,000 documents per query. We acknowledge that there is scope for further exploration in counting techniques, but do not explore this due to the similarity in results observed among the three variants shown here.

3.6.4 Early Termination

The success of access-ordering depends on the early termination of list processing. We experimented with a wide range of parameter settings in our exploration of early termination of query processing. We present some of these parameters here, including different thresholds in MAXPOST, MINACCESS, AVGACCUM, AVGROLLACCUM, and TWO-PHASE.

Maxpost Pruning

Table 3.4 shows the effect of varying the threshold \( P \) for the MAXPOST pruning scheme. For threshold values of \( P \) below 10,000, the scheme is fast but significantly less accurate than the CONTINUE approach on a document-ordered system. For values of \( P \) larger than 10,000, MAP negatively deviates from the baseline by at most 1%, and at no time produces results that are significantly different. The trade-off is that query evaluation is at best 68% faster than processing a document-ordered list with the CONTINUE approach without stopping, and 32% faster than the CONTINUE with stopping.

Query processing time savings can be attributed to the significant reduction in postings processed per query. Ranging from 3.9% to 14.7% of total postings (for values of \( P = 15,000 \) to \( P = 75,000 \)), access-ordering with pruning has an effect similar to that of stopping. Indeed, 88.2% of the query postings used in our comparisons belong to terms that are stopped. While stopping prevents such postings from contributing to the final ranking, access-ordering with pruning allows those postings that belong to frequently accessed documents to contribute, without the overhead of entirely processing the lengthy inverted lists of these terms.

It is also interesting to note that for values of \( P \) between 15,000 and 20,000, the MAP of access-ordering with MAXPOST pruning is greater than the baseline CONTINUE scheme, although not significant. This outcome reflects the results of Persin et al. [1996] and Anh and
3.6. RESULTS

Table 3.4: Overall accuracy and timing results for the maxpost scheme. Marked items are significantly worse than the baseline document ordered scheme with continue († for 95% and ‡ for 99% confidence levels).

<table>
<thead>
<tr>
<th>Threshold $P$</th>
<th>MAP</th>
<th>R-Prec.</th>
<th>$P@10$</th>
<th>Postings processed</th>
<th>Elapsed time per query (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>0.1676†</td>
<td>0.2015</td>
<td>0.2187†</td>
<td>1.5</td>
<td>17.7</td>
</tr>
<tr>
<td>10,000</td>
<td>0.1855</td>
<td>0.2118</td>
<td>0.2542</td>
<td>2.8</td>
<td>22.2</td>
</tr>
<tr>
<td>15,000</td>
<td>0.1891</td>
<td>0.2173</td>
<td>0.2542</td>
<td>3.9</td>
<td>25.5</td>
</tr>
<tr>
<td>20,000</td>
<td>0.1926</td>
<td>0.2183</td>
<td>0.2583</td>
<td>5.0</td>
<td>28.0</td>
</tr>
<tr>
<td>25,000</td>
<td>0.1854</td>
<td>0.2152</td>
<td>0.2604</td>
<td>6.0</td>
<td>30.8</td>
</tr>
<tr>
<td>30,000</td>
<td>0.1868</td>
<td>0.2178</td>
<td>0.2583</td>
<td>7.0</td>
<td>32.3</td>
</tr>
<tr>
<td>35,000</td>
<td>0.1863</td>
<td>0.2164</td>
<td>0.2583</td>
<td>8.0</td>
<td>34.3</td>
</tr>
<tr>
<td>40,000</td>
<td>0.1880</td>
<td>0.2179</td>
<td>0.2583</td>
<td>8.9</td>
<td>36.1</td>
</tr>
<tr>
<td>45,000</td>
<td>0.1872</td>
<td>0.2177</td>
<td>0.2583</td>
<td>9.8</td>
<td>37.6</td>
</tr>
<tr>
<td>50,000</td>
<td>0.1865</td>
<td>0.2178</td>
<td>0.2562</td>
<td>10.7</td>
<td>39.4</td>
</tr>
<tr>
<td>55,000</td>
<td>0.1867</td>
<td>0.2172</td>
<td>0.2604</td>
<td>11.5</td>
<td>41.2</td>
</tr>
<tr>
<td>60,000</td>
<td>0.1847</td>
<td>0.2172</td>
<td>0.2542</td>
<td>12.4</td>
<td>42.3</td>
</tr>
<tr>
<td>65,000</td>
<td>0.1851</td>
<td>0.2190</td>
<td>0.2583</td>
<td>13.2</td>
<td>44.6</td>
</tr>
<tr>
<td>70,000</td>
<td>0.1847</td>
<td>0.2183</td>
<td>0.2604</td>
<td>13.9</td>
<td>45.4</td>
</tr>
<tr>
<td>75,000</td>
<td>0.1849</td>
<td>0.2187</td>
<td>0.2583</td>
<td>14.7</td>
<td>46.6</td>
</tr>
</tbody>
</table>

Moffat [2002b], who report similar increases in accuracy when limiting the number of postings processed per query. However, given the greatly reduced number of postings processed at these levels of pruning, there is the possibility that the parameters are over-fitted to this particular search task.

We suggest setting the maxpost $P$ threshold to a value that processes approximately 11% of the inverted list postings, which is similar to the amount processed by both the continue scheme with stopping, and the impact-ordered scheme (both shown in Table 3.2).

Based on this criterion, for pruning with the maxpost scheme, we recommend a $P$ threshold of 55,000. In our experiments, setting $P = 55,000$ on the training data resulted in accuracy that approximates the baseline, with a query processing time improvement of 48% over document-ordered processing. However, our approach was not as efficient as document-ordered processing with stopping, where our query evaluation time was 9% slower, or impact-ordering, which evaluated queries 30% faster than our approach.
Table 3.5: Overall accuracy and timing results for the two-phase scheme when \( c_i = c_a \).

<table>
<thead>
<tr>
<th>Threshold</th>
<th>MAP</th>
<th>R-Prec.</th>
<th>P@10</th>
<th>Postings processed</th>
<th>Elapsed time per query (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>0.1869</td>
<td>0.2192</td>
<td>0.2708</td>
<td>32.1</td>
<td>79.8</td>
</tr>
<tr>
<td>125</td>
<td>0.1860</td>
<td>0.2182</td>
<td>0.2729</td>
<td>19.8</td>
<td>67.2</td>
</tr>
<tr>
<td>250</td>
<td>0.1860</td>
<td>0.2197</td>
<td>0.2667</td>
<td>9.9</td>
<td>53.3</td>
</tr>
<tr>
<td>500</td>
<td>0.1874</td>
<td>0.2202</td>
<td>0.2604</td>
<td>5.3</td>
<td>41.5</td>
</tr>
<tr>
<td>750</td>
<td>0.1925</td>
<td>0.2219</td>
<td>0.2708</td>
<td>3.8</td>
<td>36.4</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1904</td>
<td>0.2176</td>
<td>0.2604</td>
<td>3.0</td>
<td>32.7</td>
</tr>
<tr>
<td>1,250</td>
<td>0.1905</td>
<td>0.2127</td>
<td>0.2583</td>
<td>2.5</td>
<td>30.6</td>
</tr>
<tr>
<td>1,500</td>
<td>0.1870</td>
<td>0.2196</td>
<td>0.2604</td>
<td>2.1</td>
<td>28.7</td>
</tr>
<tr>
<td>2,000</td>
<td>0.1800</td>
<td>0.2139</td>
<td>0.2583</td>
<td>1.7</td>
<td>26.0</td>
</tr>
</tbody>
</table>

To verify the selection of our parameters, we applied our selected threshold to the TREC-10 topics and observed reductions of 10%, 8%, and 13%, to MAP, R-precision, and precision at 10 respectively when compared to the baseline document-ordered approach. The results were not significantly different at a 5% significance level. It is possible that the reduction in accuracy is due to differences between the training and test topics. The TREC-10 topics contain both longer queries and a larger proportion of terms that occur at high frequency in the collection. For the TREC-9 training data, the \textit{maxpost} threshold resulted, on average, in the early termination of 1.5 lists per query, whereas, for the TREC-10 data, early termination occurred for 2.9 lists per query.

Although the \textit{maxpost} scheme reduces accuracy, large reductions in the amount of postings processed at query evaluation time were observed. In Chapter 5 we propose a two-tiered index design based on this approach that further reduces query processing time. Application of the technique is dependent on predicting when a query can be accurately evaluated with the pruned index.

**Two-Phase Pruning**

Our proposed two-phase pruning scheme makes use of two parameters. The first parameter \( c_a \) affects the proportion of the inverted list postings that are decoded at query time, while the second parameter \( c_i \) determines which postings are allowed to initialise new accumulators.
3.6. RESULTS

We experimented with a wide range of values for both parameters.

The $c_i$ has a similar role to the accumulator threshold in the CONTINUE scheme. In this work, we combine the TWO-PHASE scheme with the CONTINUE scheme so that postings initialise new accumulators only if the access count of the posting being examined is greater than the calculated $a_i$ threshold, and if the CONTINUE accumulator limit has not been reached. In preliminary experiments not presented here, we found that the effect of combining the two schemes has a negligible impact on accuracy with the greatest difference in MAP across our runs being no greater than 0.04%. The combination with the CONTINUE scheme however significantly improves timings as it is possible to allocate a fixed limited amount of memory for accumulators where the stand alone TWO-PHASE scheme can potentially require an accumulator for each posting examined. All results presented here utilise the modified TWO-PHASE scheme, which has a limit of 20,000 accumulators or approximately 1% of the collection size.

We explored the effects of the two pruning parameters independently. Initially, by setting the value of $c_i$ equal to that of $c_a$, we reduce the pruning scheme to a single phase where the same threshold is used to determine when to cease processing an inverted list, and when to stop adding accumulators. In this approach, the second phase of the pruning scheme, where only existing accumulators are updated is not utilised, and all examined postings are considered for the accumulator set.

Table 3.5 shows the effect of pruning when $c_i$ and $c_a$ are equal. We found that, for values of $c_i$ and $c_a$ up to 2,000, the accuracy of the TWO-PHASE scheme was comparable to the baseline approaches with no statistically significant differences observed. Further, for values of $c_i$ and $c_a$ greater than 700, the query processing time was less than that of document-ordered processing with CONTINUE and stopping.

Applying the 11% processing criterion of Section 3.6.4 suggests a threshold of $c_i = c_a = 250$. At this threshold, effectiveness remains competitive, while the efficiency of query processing is 33% faster than document-ordered processing without stopping, but 41% slower than document-ordered processing with stopping enabled.

Varying the value of $c_i$ determines which postings can initialise new accumulators, while $c_a$ determines which postings can update existing accumulators. We experimented with varying the $c_i$ parameter for differing values of $c_a$. As we increased the value of $c_i$ for each $c_a$ setting, accuracy varied slightly. Table 3.6 shows the effect of varying the $c_i$ parameter for three different settings of $c_a$. The table shows, for each combination of $c_i$ and $c_a$, MAP, the average number of accumulators initialised per query, and average query processing time. It
Table 3.6: Effect of varying $c_i$ and $c_a$ parameters for two-phase pruning scheme. Table entries record triples of MAP, average accumulators initialised per query, and average query processing time in milliseconds.

<table>
<thead>
<tr>
<th>$c_i$</th>
<th>$c_a = 250$</th>
<th></th>
<th></th>
<th>$c_a = 500$</th>
<th></th>
<th></th>
<th>$c_a = 750$</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MAP</td>
<td>Acc.</td>
<td>Time</td>
<td>MAP</td>
<td>Acc.</td>
<td>Time</td>
<td>MAP</td>
<td>Acc.</td>
<td>Time</td>
</tr>
<tr>
<td>250</td>
<td>0.1860</td>
<td>14,058</td>
<td>53.3</td>
<td>0.1874</td>
<td>13,927</td>
<td>41.5</td>
<td>0.1925</td>
<td>13,376</td>
<td>36.4</td>
</tr>
<tr>
<td>500</td>
<td>0.1860</td>
<td>13,927</td>
<td>52.1</td>
<td>0.1875</td>
<td>13,376</td>
<td>46.9</td>
<td>0.1926</td>
<td>12,538</td>
<td>39.3</td>
</tr>
<tr>
<td>750</td>
<td>0.1861</td>
<td>13,349</td>
<td>57.9</td>
<td>0.1876</td>
<td>12,517</td>
<td>44.1</td>
<td>0.1927</td>
<td>11,543</td>
<td>40.6</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1863</td>
<td>12,504</td>
<td>56.3</td>
<td>0.1877</td>
<td>11,533</td>
<td>45.5</td>
<td>0.1930</td>
<td>10,842</td>
<td>40.3</td>
</tr>
<tr>
<td>1,250</td>
<td>0.1864</td>
<td>11,526</td>
<td>56.0</td>
<td>0.1877</td>
<td>11,533</td>
<td>45.5</td>
<td>0.1933</td>
<td>10,266</td>
<td>40.0</td>
</tr>
<tr>
<td>1,500</td>
<td>0.1867</td>
<td>10,827</td>
<td>56.1</td>
<td>0.1880</td>
<td>10,833</td>
<td>45.6</td>
<td>0.1936</td>
<td>9,731</td>
<td>40.0</td>
</tr>
<tr>
<td>1,750</td>
<td>0.1870</td>
<td>10,255</td>
<td>55.6</td>
<td>0.1883</td>
<td>10,260</td>
<td>45.5</td>
<td>0.1936</td>
<td>9,731</td>
<td>40.0</td>
</tr>
<tr>
<td>2,000</td>
<td>0.1873</td>
<td>9,721</td>
<td>55.6</td>
<td>0.1888</td>
<td>9,726</td>
<td>45.5</td>
<td>0.1936</td>
<td>9,731</td>
<td>40.0</td>
</tr>
</tbody>
</table>

is interesting to note although the number of accumulators initialised per query declines as the insertion threshold $c_i$ is incremented, the effect on accuracy and query processing time is negligible. This is due to the fact that list processing is not likely to reduce with the increment of the $c_i$ threshold. As less documents have accumulators initialised, postings processed during the the update phase have a higher probability of being skipped, and therefore the calculated $a_i$ and $a_a$ thresholds (page 66), that are partially based on the highest processed posting contribution to date ($S_m$) will be lower. In general we found that using the two phases of processing gave a minimal improvement over the single parameter approach where $c_i$ and $c_a$ are equal.

The additional cost of reading the uncompressed inverted lists from disk, and the extra processing costs associated with the two-phase pruning scheme, can be clearly seen. For example, at $c_i = c_a = 75$, in Table 3.5, the two-phase scheme is almost as slow as continue without stopping, even though only roughly 30% of the lists are being processed. The time difference can be attributed to the increased index size and the extra processing required by the termination scheme.

We recommend use of the two-phase scheme with a single parameter. By reducing the scheme to a single parameter, the computational overhead of the scheme is reduced without the need of calculating two thresholds for every query term processed. We conclude this section by recommending use of a single phase of processing that is equivalent to setting
Table 3.7: Overall accuracy and efficiency results for the minaccess scheme. Marked items are significantly worse than the baseline document ordered scheme with continue († for 95% and ‡ for 99% confidence levels).

<table>
<thead>
<tr>
<th>Threshold M</th>
<th>MAP</th>
<th>R-Prec.</th>
<th>P@10</th>
<th>Postings processed (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.1866</td>
<td>0.2177</td>
<td>0.2708</td>
<td>99.5</td>
</tr>
<tr>
<td>10</td>
<td>0.1866</td>
<td>0.2177</td>
<td>0.2708</td>
<td>99.3</td>
</tr>
<tr>
<td>50</td>
<td>0.1867</td>
<td>0.2177</td>
<td>0.2708</td>
<td>96.7</td>
</tr>
<tr>
<td>100</td>
<td>0.1868</td>
<td>0.2177</td>
<td>0.2708</td>
<td>93.4</td>
</tr>
<tr>
<td>500</td>
<td>0.1873</td>
<td>0.2215</td>
<td>0.2792</td>
<td>68.5</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1810</td>
<td>0.2237</td>
<td>0.2875</td>
<td>42.6</td>
</tr>
<tr>
<td>1,250</td>
<td>0.1657</td>
<td>0.2045</td>
<td>0.2812</td>
<td>33.1</td>
</tr>
<tr>
<td>1,500</td>
<td>0.1514†</td>
<td>0.1968</td>
<td>0.2604</td>
<td>25.8</td>
</tr>
<tr>
<td>2,000</td>
<td>0.1147‡</td>
<td>0.1521‡</td>
<td>0.2125‡</td>
<td>15.8</td>
</tr>
</tbody>
</table>

$c_i = c_a$. Further the parameter should be set to a value for the collection where approximately 11% of the inverted lists are processed.

Overall, for the best balance between various measures of accuracy and processing cost we suggest the parameters $c_i = c_a = 250$ for our test collection. To test the effectiveness of our selected parameters we applied the values to the TREC-10 data and found non-significant differences in accuracy between the baseline continue scheme and our two-phase pruning approach. Unlike maxpost, the two-phase scheme takes a more dynamic approach to list pruning, and therefore is able to adapt to the TREC-10 queries, which have a different style, without a loss in accuracy.

One disadvantage of the two-phase pruning approach is the increased processing time to evaluate a query. This is a result of the larger index size, and the extra computational overhead. In Section 3.6.6 we reduce query processing time by applying compaction techniques to the access-ordered index.

Other Pruning Schemes

We explored various values of $M$, $A$, and $W$ for the minaccess, avgaccum, and avgrollaccum schemes with limited success. Table 3.7 shows results for the minaccess approach with a selection of $M$ values. For values of $M$ between 1,000 and 1,300 a small, but insignifi-
Table 3.8: Overall accuracy and efficiency results for the \textit{avgaccum} scheme. Marked items are significantly worse than the baseline document ordered scheme with \textsc{continue} († for 95% and ‡ for 99% confidence levels).

<table>
<thead>
<tr>
<th>Threshold</th>
<th>MAP</th>
<th>R-Prec.</th>
<th>P@10</th>
<th>Postings processed (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.1866</td>
<td>0.2177†</td>
<td>0.2708</td>
<td>37.0</td>
</tr>
<tr>
<td>0.05</td>
<td>0.1672‡</td>
<td>0.1999†</td>
<td>0.2234‡</td>
<td>30.8</td>
</tr>
<tr>
<td>0.10</td>
<td>0.1678‡</td>
<td>0.1974†</td>
<td>0.2239‡</td>
<td>30.7</td>
</tr>
<tr>
<td>0.15</td>
<td>0.1647‡</td>
<td>0.1965†</td>
<td>0.2295‡</td>
<td>30.6</td>
</tr>
<tr>
<td>0.20</td>
<td>0.1630‡</td>
<td>0.1939‡</td>
<td>0.2273‡</td>
<td>30.3</td>
</tr>
<tr>
<td>0.25</td>
<td>0.1565‡</td>
<td>0.1870‡</td>
<td>0.2163‡</td>
<td>30.1</td>
</tr>
<tr>
<td>0.30</td>
<td>0.1568‡</td>
<td>0.1833‡</td>
<td>0.2095‡</td>
<td>27.3</td>
</tr>
</tbody>
</table>

cant improvement in precision at 10 is observed. In general the \textsc{minaccess} scheme proved to be ineffective at reducing the number of postings processed per query without a significant reduction in accuracy. Indeed, even at $M = 1,500$, a significant reduction in MAP of over three percentage points is observed, when compared to document-ordered processing. Further, this requires that processing of over 25% of the total postings, which is substantially greater than the amount processed by the \textsc{maxpost} and \textsc{two-phase} for higher levels of accuracy.

The result is interesting as it shows that documents with low access counts still need to be selectively processed to achieve high accuracy. In contrast to the \textsc{maxpost} and \textsc{two-phase} schemes, \textsc{minaccess} is unlikely to process a large fraction of the postings in short inverted lists, even though short lists — those for terms with high inverse document frequencies — are highly weighted by the search system.

Table 3.8 presents the results of pruning with the \textit{avgaccum} scheme. For all runs, accuracy was significantly worse even with a high fraction of the inverted lists processed per query. We found that this scheme was unstable as it would often process either an entire list, or cease processing within the first few postings. However, early termination occurred most frequently for the long lists as their accumulator contribution per posting was low.

By adding a sliding window, the \textit{avgrollaccum} scheme guarantees a minimum number of postings to be processed per list. Further, by considering only those postings that occur within the sliding window, changes in accumulator contribution are localised. Table 3.9
3.6. RESULTS

Table 3.9: Overall accuracy and efficiency results for the AVGROLLACCUM scheme. Marked items are significantly worse than the baseline document ordered scheme with CONTINUE († for 95% and ‡ for 99% confidence levels).

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Window size</th>
<th>MAP</th>
<th>R-Prec.</th>
<th>P@10</th>
<th>Postings processed (%)</th>
<th>Elapsed time per query (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.02</td>
<td>100</td>
<td>0.1926</td>
<td>0.2142</td>
<td>0.2625</td>
<td>2.7</td>
<td>29.7</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.1841</td>
<td>0.2169</td>
<td>0.2604</td>
<td>7.8</td>
<td>48.1</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.1854</td>
<td>0.2198</td>
<td>0.2562</td>
<td>11.7</td>
<td>58.0</td>
</tr>
<tr>
<td></td>
<td>5,000</td>
<td>0.1868</td>
<td>0.2173</td>
<td>0.2687</td>
<td>28.1</td>
<td>76.3</td>
</tr>
<tr>
<td>0.04</td>
<td>100</td>
<td>0.1832</td>
<td>0.2165</td>
<td>0.2396</td>
<td>1.6</td>
<td>24.7</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.1859</td>
<td>0.2175</td>
<td>0.2542</td>
<td>5.2</td>
<td>38.7</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.1840</td>
<td>0.2169</td>
<td>0.2604</td>
<td>7.9</td>
<td>47.1</td>
</tr>
<tr>
<td></td>
<td>5,000</td>
<td>0.1873</td>
<td>0.2197</td>
<td>0.2708</td>
<td>20.7</td>
<td>68.3</td>
</tr>
<tr>
<td>0.06</td>
<td>100</td>
<td>0.1689†</td>
<td>0.2021</td>
<td>0.2229†</td>
<td>1.1</td>
<td>22.0</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.1864</td>
<td>0.2209</td>
<td>0.2604</td>
<td>4.0</td>
<td>34.3</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.1856</td>
<td>0.2161</td>
<td>0.2625</td>
<td>6.3</td>
<td>42.0</td>
</tr>
<tr>
<td></td>
<td>5,000</td>
<td>0.1858</td>
<td>0.2195</td>
<td>0.2667</td>
<td>16.1</td>
<td>64.7</td>
</tr>
<tr>
<td>0.08</td>
<td>100</td>
<td>0.1666†</td>
<td>0.2010</td>
<td>0.2146†</td>
<td>0.9</td>
<td>20.4</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.1912</td>
<td>0.2209</td>
<td>0.2521</td>
<td>3.3</td>
<td>31.6</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.1856</td>
<td>0.2175</td>
<td>0.2542</td>
<td>5.3</td>
<td>38.7</td>
</tr>
<tr>
<td></td>
<td>5,000</td>
<td>0.1851</td>
<td>0.2204</td>
<td>0.2604</td>
<td>13.5</td>
<td>60.2</td>
</tr>
<tr>
<td>0.10</td>
<td>100</td>
<td>0.1524†</td>
<td>0.1835†</td>
<td>0.2083†</td>
<td>0.8</td>
<td>19.2</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.1925</td>
<td>0.2139</td>
<td>0.2583</td>
<td>2.8</td>
<td>29.5</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.1867</td>
<td>0.2172</td>
<td>0.2604</td>
<td>4.5</td>
<td>36.0</td>
</tr>
<tr>
<td></td>
<td>5,000</td>
<td>0.1852</td>
<td>0.2190</td>
<td>0.2562</td>
<td>11.9</td>
<td>56.5</td>
</tr>
<tr>
<td>0.15</td>
<td>500</td>
<td>0.1881</td>
<td>0.2195</td>
<td>0.2625</td>
<td>2.1</td>
<td>26.3</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.1861</td>
<td>0.2206</td>
<td>0.2542</td>
<td>3.5</td>
<td>31.8</td>
</tr>
<tr>
<td></td>
<td>5,000</td>
<td>0.1841</td>
<td>0.2196</td>
<td>0.2583</td>
<td>9.5</td>
<td>51.1</td>
</tr>
<tr>
<td>0.20</td>
<td>500</td>
<td>0.1848</td>
<td>0.2148</td>
<td>0.2458</td>
<td>1.7</td>
<td>24.0</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.1917</td>
<td>0.2137</td>
<td>0.2542</td>
<td>2.8</td>
<td>29.2</td>
</tr>
<tr>
<td></td>
<td>5,000</td>
<td>0.1841</td>
<td>0.2169</td>
<td>0.2604</td>
<td>8.1</td>
<td>46.6</td>
</tr>
</tbody>
</table>
Table 3.10: Query processing time for indexes with offsets stored in the inverted lists and relative difference with baseline scheme.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Threshold</th>
<th>Elapsed time (ms)</th>
<th>Relative Diff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTINUE</td>
<td>n/a</td>
<td>121.8</td>
<td>0%</td>
</tr>
<tr>
<td>CONTINUE stopped</td>
<td>n/a</td>
<td>50.2</td>
<td>-59%</td>
</tr>
<tr>
<td>MAXPOST</td>
<td>55,000</td>
<td>51.4</td>
<td>-58%</td>
</tr>
<tr>
<td>TWO-PHASE</td>
<td>$c_i = c_a = 250$</td>
<td>70.3</td>
<td>-42%</td>
</tr>
</tbody>
</table>

shows the results of pruning with the sliding window scheme for varying windows sizes $W$ and accumulator contributions $A$.

For this scheme various parameter sets produced a desirable combination of accuracy while processing a low number of postings. Larger window sizes of 500 to 1,000 postings are more robust and less sensitive to highly localised minima. However, due to the overhead of maintaining the windowed average, query evaluation time is slower than that of the TWO-PHASE and MAXPOST schemes when processing similar quantities of postings.

### 3.6.5 Pruning with Term Offsets

The above experiments make use of inverted lists without term offsets. To process phrase queries, or similarity metrics that utilise the proximity of query terms, term offsets must be stored in the inverted lists. Due to the global ordering of an access-ordered index, query modes that make use of term offsets can be applied without modification to the evaluation process.

Table 3.10 shows the average query processing time when postings are stored within the inverted lists for both the document- and access-ordered indexes using the best pruning parameters from above. We see that with offsets included in the inverted lists, the relative differences between the baseline scheme and access-ordering remain similar to the results presented above without term offsets (Table 3.2 page 72).

### 3.6.6 Index Compaction Schemes

The document-ordered index (without term offsets) for the WT10g collection is 1.05 GB or around 10% of the size of the collection. The access-ordered index is 58% larger, requiring
Table 3.11: Accuracy and efficiency results for the various compaction schemes using two-phase with $c_i = c_a = 250$. Document-ordered index results are also listed for comparison.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>MAP</th>
<th>Postings processed (%)</th>
<th>Elapsed time per query (ms)</th>
<th>Index size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access-Ordered Uncompacted</td>
<td>0.1860</td>
<td>9.9</td>
<td>53.3</td>
<td>1.63</td>
</tr>
<tr>
<td>Basic access-block</td>
<td>0.1860</td>
<td>9.9</td>
<td>52.7</td>
<td>1.64</td>
</tr>
<tr>
<td>Exponential-block</td>
<td>0.1836</td>
<td>13.0</td>
<td>52.4</td>
<td>1.10</td>
</tr>
<tr>
<td>Fixed access-block $k = 1,000$</td>
<td>0.1856</td>
<td>10.0</td>
<td>48.5</td>
<td>1.22</td>
</tr>
<tr>
<td>Document-Ordered</td>
<td>0.1865</td>
<td>93.5</td>
<td>80.0</td>
<td>1.05</td>
</tr>
</tbody>
</table>

1.63 GB or around 16% of the collection size. This is the result of the loss of redundancy due to removing the document-ordering, as discussed in Section 3.5. All indexes are compressed using variable-byte coding [Williams and Zobel, 1999], a scheme that has been shown to be less effective in storage than well-known bitwise schemes [Zobel and Moffat, 2006] but faster for query evaluation [Scholer et al., 2002].

The performance of the proposed compaction schemes varied greatly between both the parameters used and in the schemes themselves. For each scheme we constructed the relevant index and examined index size, accuracy, and query evaluation time.

We expected that the distribution of access counts in an inverted list would be similar to that of the collection. Given this hypothesis, the performance of the Basic access-block compaction approach would be effective. Surprisingly, after compacting the index using this scheme we found that the index size slightly increased to 1.64 GB. The increase in size was due to the distribution of postings throughout the inverted lists. Short inverted lists with few postings form a significant portion of the index. For these short lists, blocks of postings sharing the same access count are uncommon, and therefore each list posting with a unique access count results in an additional integer. The need to store extra integers for such postings more than offset the benefit of grouping the postings in the larger inverted lists.

Due to the lack of index compaction, query processing time did not improve. For the two-phase approach an insignificant difference in evaluation time was observed. Table 3.11 shows the evaluation times for each compaction scheme along with index size and effectiveness.
Figure 3.6 shows the performance of the Fixed access-block compaction technique. Unsurprisingly, at low values of $k$ the scheme produces indexes that are equivalent to an uncompacted access-ordered index, where at the extreme of one posting per block the scheme is equivalent to an uncompacted index. As the number of postings per block increases, compaction improves and by $k = 50,000$ the index is 36% smaller, and only 1% larger than the document-ordered index. While accuracy remains relatively uniform, the increasing block sizes initially have a beneficial impact on query processing time, but, as the block sizes increase, an adverse effect on evaluation time is observed. As the value of $k$ increases, the pruning scheme is forced to process entire blocks of postings, and is therefore unable to prune lists to the same degree. Between $k = 2,000$ and $k = 50,000$, the average fraction of inverted list postings processed per query increases from 10% to 14% due to the increased minimum number of postings processed per list. As a result, a slight increase in query processing time is observed.
3.7. **SUMMARY**

More notable is the improvement in query processing time between low values of $k = 1$ where the index remains completely access-ordered, and $k = 1,000$, where partial access-ordering is observed. The reduction in query evaluation time of 12.7%, from 54.4 milliseconds to 47.5 milliseconds, can be attributed to the reduced cost of reading lists from disk.

For the WT10g collection we suggest a Fixed-block compaction with $k = 1,000$, where postings processed per query is fractionally greater than that of the uncompacted scheme. At this setting query evaluation time is reduced by 9% when compared to query evaluation time using an uncompacted access-ordered index, and is 39% faster than processing a document-ordered index using the CONTINUE scheme without stopping. However, even with the compaction, access-ordered processing with **TWO-PHASE** pruning remains 28% slower than document-ordered processing using CONTINUE when stopping is applied.

Finally, our experiments with the **EXPONENTIAL**-block scheme show that index compaction is very effective, with the index size reducing by 32% when compared to an uncompacted access-ordered index. Although the **EXPONENTIAL**-block index is only 7% larger than the document-ordered index, query processing time does not reduce as, on average, a greater number of postings are processed per query. As with the Fixed-block scheme, entire blocks of postings must be processed atomically. Our experiments show that using the **TWO-PHASE** pruning scheme, 13% of the postings per list are processed, resulting in a negligible improvement in query evaluation time.

Table 3.11 summarises the results of the compaction techniques. Although intuitive, the **Basic** access-block compaction scheme is ineffective at reducing index size due to the large proportion of short lists in the index. The **EXPONENTIAL** scheme, while effective at compacting the index, results in a large number of postings processed per query. We recommend use of the Fixed-block scheme with a $k$ value of 1,000. This scheme offers a balance between index compaction and accuracy with the **TWO-PHASE** pruning approach.

**3.7 Summary**

We have proposed a fast and accurate query evaluation approach based on a new organisation of inverted indexes. Our access-ordered index is motivated by the observation that a small number of documents in a collection account for a large proportion of search results. As such, we propose organising the index so that those documents are stored at the beginning of inverted lists. The benefit of this approach is that they can be processed early in the
query evaluation process, and the process can be abandoned when less popular documents are encountered.

To establish an access-ordering we have examined three unique counting techniques. We found that despite the fundamental differences in the scoring methods proposed, the ordering obtained by the variant schemes resulted in a similar distribution of relevant documents.

Based on the principles of related index reorganisation techniques, we have proposed several dynamic query-time pruning heuristics. In training, the MAXPOST scheme is fast and accurate, but when applied to our test data, the results varied. We do not recommend use of this technique for the query-time pruning of access-ordered indexes. However, given a fixed threshold $P$, those postings that exceed the threshold will not be processed by the search engine. As such, the MAXPOST pruning scheme can be adapted for static index construction time pruning resulting in a smaller index. At query time, such an index can be used to evaluate queries with the benefits of reduced I/O and improved caching. In Chapter 5 we examine the application of the MAXPOST scheme for static pruning in a two-tiered index.

A second pruning scheme that we label TWO-PHASE pruning, resulted in a system that was both accurate and efficient. After training we recommend the pruning parameters set to $c_i = c_a = 250$. Using this technique, we reduced query evaluation time by 33% from an unstopped document-ordered index using the CONTINUE scheme.

We explored several other pruning techniques based on the properties of the list ordering, but found that they were either too aggressive at pruning time, resulting in fast but inaccurate query evaluation; or not aggressive enough, resulting in accurate but slow query evaluation.

By default, our indexes are document-level and do not store term offsets. For comparison we tested the TWO-PHASE pruning scheme on an access-ordered word-level index. Although evaluation is slower, our results show similar relative improvements over the baseline document-ordered evaluation.

A side effect of the access-ordering is that the index can no longer be compacted by taking differences between adjacent postings. We propose index compaction techniques that trade-off strict access-ordering with document-ordering, while still allowing pruning techniques designed for access-ordered indexes. Our results show that the FIXED-block scheme, where postings are grouped into fixed blocks of size $k$ provides the best compromise between index compaction and pruning. Using this approach we further reduced query evaluation time using TWO-PHASE pruning to a saving of 39% over the baseline unstopped document-ordered index using the CONTINUE scheme.
Access-ordering provides a means by which lists can be pruned at query evaluation time. The greatest benefit of such pruning comes from the avoidance of processing long lists. Stopping performs a similar role by removing the need to process the long lists of terms that tend to contribute less to the similarity ranking function. Our results show a significant improvement in query evaluation time when stopping is not applied. Indeed, even when stopping is applied to the baseline, our approach without stopping is only slightly slower, and more accurate.

In the next chapter, we continue our investigation of access-ordered indexes with further investigation of techniques to compress the index and reduce query evaluation time. We also explore the effect of query drift on an access-ordering and further consider the effect of query training set size when establishing an access-ordering.
Chapter 4

Extensions to Access-Ordering

Access-ordered indexes allow query-time inverted list pruning, without a reduction in average evaluation accuracy. One of the major drawbacks of the access-ordered index is that standard index compression techniques cannot be used to compress the index effectively. In the previous chapter we explored dynamic query-time inverted list pruning techniques for access-ordered indexes. In this chapter we extend the previous work on access-ordered indexes and present a variant that we label an access-reordered index, with the benefit that access-reordering allows the application of standard methods of index compression.

We also address index stability; that is, if the index ordering is based on past user queries, then will the reorganisation be able to process queries effectively as user requests change over time? To address this question, we explore the effect of training set size and show that an access-ordered index based on a small number of queries can be used to produce results that are almost equivalent to that of an index ordered by queries gathered over a much larger period of time.

In Section 4.1 we outline the motivation behind this adaption of access-ordering. In Section 4.2 we introduce our variant index organisation technique. In Section 4.3 we outline the methodology that we employ to compare the ranked results produced by two indexes based on different access-orderings. In Section 4.3.1 we explain our approach to observing the effect of change in user queries. Finally, in Section 4.4 we present experimental results that demonstrate the effectiveness of access-reordered indexes, as well as the stability of indexes based on access counts.
4.1 Motivation

In Chapter 3, we proposed a list reordering technique that we called access-ordering. Based on the frequency of document appearance in previous search results, an access-ordered index provides an effective means for terminating query evaluation without processing all the postings within a query term’s inverted lists.

Early termination is dependent on a query evaluation time heuristic to determine when the decoding of an inverted list can cease. In Section 3.4 (page 64), we explored several early termination heuristics ranging from simple approaches such as processing a fixed number of postings per list, and only processing postings with a minimum access count, to more elaborate techniques such as tracking the average accumulator contribution of a list as it is processed and terminating when a sentinel condition is met. However, the most effective pruning technique we found was an adaption of the frequency-ordered pruning technique of Persin et al. [1996], where processing terminates when the contribution of a posting falls below a threshold value that is calculated once per inverted list. In our adoption of the pruning scheme, a threshold value is calculated for each term prior to processing its inverted list as follows:

\[ a_a = \frac{(c_a S_m)}{(w_t^2 + 1)} , \]

where \( w_t \) is the weight of the query term \( t \), \( S_m \) is the largest accumulator seen to date while processing this query, and \( c_a \) is a global tuning parameter. The terms are processed in descending inverse document frequency order, with list processing terminating when the access count of a term falls below the \( a_a \) value. Accumulator limiting via the \texttt{continue} scheme is also used to bound main-memory usage as required.

In Section 3.6 (page 71), we reported processing time savings of up to 39% over the standard document-ordered approach. However, as access-ordered lists do not have ordered document identifiers or ordered document contributions, the advantages of d-gap compaction are minimal. Indeed, without compaction the index size grew by up to 60% when ordering lists by access count. The best compromise between compaction and performance resulted in an index that remained almost 20% larger than the baseline document-ordered index.

An alternate to index reorganisation is \textit{collection reordering}. In a reordered collection, instead of reorganising the inverted lists independently, the document identifiers of the entire collection are remapped to meet a specified criteria. Typically, similar documents are assigned identifiers in the same proximity. The key benefit of this approach is that if similar documents
are clustered together, then the inverted lists of the terms appearing in those documents will contain postings that have a reduced average d-gap. This in turn leads to compression gains.

Various works propose collection reordering techniques based on the clustering of similar documents [Blandford and Blelloch, 2002; Silvestri et al., 2004a;b]. They report reductions in index size of as much as 23% over a conventional index organisation. Like the approaches of Blandford and Blelloch, and Silvestri et al., access counts provide an absolute final ordering for the documents in a collection. As such, a collection reordering approach based on access counts may be an effective solution to the inefficient compression observed in the previous chapter.

4.2 Access-Reordering

Unlike other index reorganisation techniques, such as frequency-ordering [Persin et al., 1996] or impact-ordering [Anh and Moffat, 2002b; 2005b], access-ordered indexes are organised by a collection-wide ordering. That is, all lists are ordered by the same set of access counts. As such, we propose the following extension. After establishing access counts, documents are assigned an identifier based on their rank in the access-ordering. That is, the document with the highest access count is assigned the document identifier 1, the document with the second highest access count is assigned the document identifier 2, and so on.

Consider the following example using the Tempest collection of Section 2.3 (page 20). After processing a query log of 100 queries, each document in the collection has an associated access count and access rank. Table 4.1 lists the access counts and ranks for a portion of the collection. For example, document 136 has an access count of 3, and is ranked 57th by access-ordering.

Given the original, document-ordered list for the term “sycorax”:

\[
\langle 128, 1 \rangle, \langle 132, 1 \rangle, \langle 136, 1 \rangle, \langle 159, 2 \rangle, \langle 511, 2 \rangle,
\]

the access-reordered equivalent list becomes:

\[
\langle 10, 2 \rangle, \langle 40, 2 \rangle, \langle 57, 1 \rangle, \langle 354, 1 \rangle, \langle 357, 1 \rangle.
\]

Note that with ascending document identifiers, d-gaps are now possible, leading to a reduced index size. Figure 4.1 shows the access-reordered lists, with d-gap compaction, for selected terms in the Tempest collection.

In the basic access-ordered index, changes to the access counts can be reflected in the index by re-sorting the inverted list of each term. Lists can be updated independently as
Table 4.1: Access counts and ranks for 30 documents in Tempest collection after 100 training queries have been processed to establish an access-ordering. The column “Access count” shows the frequency with which a document appeared in the result set, and the column “Rank” shows the rank of that document after sorting the documents by access count.

<table>
<thead>
<tr>
<th>Document</th>
<th>Access count</th>
<th>Rank</th>
<th>Document</th>
<th>Access count</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
<td>3</td>
<td>56</td>
<td>145</td>
<td>0</td>
<td>366</td>
</tr>
<tr>
<td>131</td>
<td>0</td>
<td>356</td>
<td>146</td>
<td>0</td>
<td>367</td>
</tr>
<tr>
<td>132</td>
<td>0</td>
<td>357</td>
<td>147</td>
<td>0</td>
<td>368</td>
</tr>
<tr>
<td>133</td>
<td>0</td>
<td>358</td>
<td>148</td>
<td>0</td>
<td>369</td>
</tr>
<tr>
<td>134</td>
<td>2</td>
<td>111</td>
<td>149</td>
<td>0</td>
<td>370</td>
</tr>
<tr>
<td>135</td>
<td>0</td>
<td>359</td>
<td>150</td>
<td>0</td>
<td>371</td>
</tr>
<tr>
<td>136</td>
<td>3</td>
<td>57</td>
<td>151</td>
<td>0</td>
<td>372</td>
</tr>
<tr>
<td>137</td>
<td>0</td>
<td>360</td>
<td>152</td>
<td>0</td>
<td>373</td>
</tr>
<tr>
<td>138</td>
<td>0</td>
<td>361</td>
<td>153</td>
<td>0</td>
<td>374</td>
</tr>
<tr>
<td>139</td>
<td>0</td>
<td>362</td>
<td>154</td>
<td>0</td>
<td>375</td>
</tr>
<tr>
<td>140</td>
<td>0</td>
<td>363</td>
<td>155</td>
<td>0</td>
<td>376</td>
</tr>
<tr>
<td>141</td>
<td>0</td>
<td>364</td>
<td>156</td>
<td>2</td>
<td>113</td>
</tr>
<tr>
<td>142</td>
<td>1</td>
<td>190</td>
<td>157</td>
<td>2</td>
<td>114</td>
</tr>
<tr>
<td>143</td>
<td>0</td>
<td>365</td>
<td>158</td>
<td>2</td>
<td>115</td>
</tr>
<tr>
<td>144</td>
<td>2</td>
<td>112</td>
<td>159</td>
<td>6</td>
<td>10</td>
</tr>
</tbody>
</table>

needed without reference to the rest of the collection. Remapping the document identifiers makes updates difficult as it requires the updating of all postings in all lists. In order for the access-reordering approach to be practical we must show that the need to remap the index is not often required. One way to show this is to explore the stability of a generated ordering. This raises two questions, first, do changes in user queries over time cause the index to become unstable for list pruning? Second, how many training queries are required to generate an access-reordered index that can efficiently service user queries?
4.3 Measuring Change in Index Performance

As described in Section 2.2.4 (page 18), a standard means by which to measure effectiveness is to make use of the well known TREC collections, queries and relevance judgements. Using this approach, search systems can be compared and contrasted based on their ability to retrieve relevant documents.

While it is likely that variance in the number of training queries used to establish an access-ordering will have an effect on the performance of the query time pruning heuristics, it is unlikely that the effect of training set size can be adequately measured with the small number of static TREC queries that are not derived from changing user information needs over time.

To measure the effect of training set size and query drift, a large number of queries and relevance judgments are required. While the query logs that we work with meet the former requirement, relevance judgments are unavailable. However it is still possible to measure change in system performance by measuring the change in the results returned by two search systems when processing the same query log.

In our approach, we process a query log using system $A$ over our collection and record the top $R$ returned results in the set $R_A$. We label the set $R_A$ the oracle run. A second run with the same query log and collection is processed using search system $B$ and the results are stored in $R_B$. By treating $R_A$ as the set of relevant documents, and $R_B$ as the returned result set, we can compare the difference between the two runs using precision and recall. Figure 4.2 illustrates the process. Under such an approach, precision and recall values of

<table>
<thead>
<tr>
<th>fairy</th>
<th>(698,2)</th>
<th>(4,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>farewell</td>
<td>(52,2)</td>
<td>(92,1)</td>
</tr>
<tr>
<td>fish</td>
<td>(0,6)</td>
<td>(125,1)</td>
</tr>
<tr>
<td>state</td>
<td>(67,1)</td>
<td>(21,2)</td>
</tr>
<tr>
<td>storm</td>
<td>(0,3)</td>
<td>(18,2)</td>
</tr>
<tr>
<td>swords</td>
<td>(652,1)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>sycorax</td>
<td>(10,2)</td>
<td>(30,2)</td>
</tr>
<tr>
<td>vision</td>
<td>(21,1)</td>
<td>(24,1)</td>
</tr>
</tbody>
</table>

Figure 4.1: Access-reordered lists for selected terms from Tempest collection based on an ordering established with 100 training queries. Lists are ordered by ascending remapped document identifier, and compaction using d-gaps is employed.
Figure 4.2: Proposed oracle based approach to comparing two information retrieval systems when relevance judgements are unavailable. Queries are evaluated on two distinct systems, System A and System B. The results returned by the System A are recorded as oracle relevance judgements. The results returned by System B are evaluated against the oracle results of System A using standard IR effectiveness measures.
100% indicate that both systems produce identical results. Lower values of recall indicate dramatically different result sets, and low values of precision at $N$ indicate that system $B$ produces different results ranked within the top $N$ documents compared to the results returned by system $A$.

While our approach intuitively measures differences in the result documents returned by two search systems, it is less clear about the degree of change in the ordering of the results. By repeating the above process over varying values of $R$ per query, we can establish an understanding of the similarity, of not only the difference in the results returned by each system, but differences in the ordering of those results. In this work we sample with the following values of $R$: 10, 100, and 1,000.

In related work, Carmel et al. [2001b] compare the similarity of the results returned by their pruned index to full evaluation using Kendall’s $\tau$. They choose to compare the top 10 ranked documents per query. While such comparison contrasts the ordering of results presented by two search systems, small differences in low ranked documents are valued as equally to the correlation metric, as differences in highly ranked documents. We do not utilise this method of comparison, as we consider up to 1,000 ranked results per query.

### 4.3.1 Change in User Queries

Changes in the pattern of user queries will, over time, reduce the effectiveness of an access-reordered index. In order for our approach to be practical, it is necessary to measure the effect of query change on the performance of the system.

To measure this effect, we propose sampling the first $Q$ queries of a large query log, where $Q$ is incremented between samples. For each sample, we construct an access-reordered index based on the set of $Q$ sampled queries. We then compare the performance of the index trained on the sample queries, to that of baseline oracle system trained on a significantly larger set of queries. At each phase, comparisons are made using the approach outlined in Section 4.3.

Upon conclusion of processing the entire query log, the effects of training set size and query drift on the retrieved results are both observable. The effect of query training set size is observed as each query set sample subsumes the previous sample, and the effect of query drift is observed, as the oracle system has total knowledge of the query log, while each sample only contains partial knowledge. Details of the query log and collection are presented next.
CHAPTER 4. EXTENSIONS TO ACCESS-ORDERING

4.4 Experiments in Access-Reordering

We now present experimental results that demonstrate the effectiveness of access-reordering. We begin in Section 4.4.1 with a discussion of the experimental environment and test collections used in this chapter. In Section 4.4.2, we revisit the effects of query time pruning, and further compare the ranked results returned by a query-time pruned access-ordered index, to a standard system with no pruning. Then, in Section 4.4.3, we show that an index trained on as little as 100,000 queries can be used to process as many as 20,000,000 queries with effective query time pruning. In Section 4.4.4 we demonstrate the efficiency of access-reordering with comparisons of index size and query evaluation time.

4.4.1 Experimental Environment

Experiments in this chapter are on a dual Pentium Xeon 3.0 Ghz server. The server uses the Linux Fedora Core operating system with a 2.6.9 kernel. For all timed experiments the system cache was flushed between runs. Our implementation is based on the publicly available open-source Zettair search engine [Zettair].

The data used is a subset of the Gov2 collection from TREC 2004 [Clarke et al., 2004]. The Gov2 collection is composed of web documents from the .gov domain crawled during 2004. Our subset consists of the first 7.5 million documents in the collection as distributed by TREC, accounting for the first 100 GB of documents.

To explore the effect of training query set size, a large query log is required. We use two distinct query logs with different properties. The first query log from Lycos.de [Lycos] contains almost 200 million time-ordered queries from the Lycos web search engine. Unfortunately, a high proportion of these queries contain German terms that do not occur frequently in the Gov2 collection.

The second query log provided by Microsoft [MSN] in 2003 contains over 500 million queries ordered from most to least frequently occurring. Queries that occurred less than three times during the period in which the queries were gathered were not included in the log. While more compatible with the collection, this log does not provide a time reference for each query, therefore simulating a stream of queries over time requires randomisation of the query log and does not show trends in query use such as those explored by Ozmuthu et al. [2004] and Diaz and Jones [2004].

To make use of the sorted Microsoft queries we generate a query log by randomising the queries in the log based on their frequency of occurrence. For our experiments we generate
4.4. EXPERIMENTS IN ACCESS-REORDERING

![Comparison of query repetition rates in Lycos and MSN query logs. For each unique query, the average distance (number of queries) between repetitions of that query is calculated. The plots show the number of unique queries that share a repetition range frequency.](image)

(a) Lycos  
(b) MSN

Figure 4.3: Comparison of query repetition rates in Lycos and MSN query logs. For each unique query, the average distance (number of queries) between repetitions of that query is calculated. The plots show the number of unique queries that share a repetition range frequency.

Randomly sampling from the Microsoft log based on query frequency produces a log with a similar distribution of queries to the original data. However, such an approach is unable to capture the temporal properties of a time-ordered query log, where queries can repeat frequently in small temporal clusters. Figure 4.3 shows the difference between our artificial Microsoft (MSN) based query log, and the Lycos time-ordered query log. The figure shows, the number of distinct queries that share the same average interval between repetitions. We can see that the Lycos log has many distinct queries that repeat at short intervals, while the distribution for the MSN log is flatter. That is, the queries that repeat in the Lycos log are more likely to repeat frequently during a short interval, and then not repeat again; while in the MSN log, queries that repeat are more likely to be distributed evenly, due to the randomisation.

Both query logs are described in greater detail in Section 2.1.1 (page 10).
CHAPTER 4. EXTENSIONS TO ACCESS-ORDERING

Figure 4.4: Difference in results returned by an access-reordered index with query time pruning, to that of a standard index with no pruning. $N$ is the number of results returned per query. Oracle data is based on the top $N$ results returned when evaluating queries on a document-ordered index with continue. The oracle run is compared to a access-reordered index trained on 20 million queries. Access-reordered pruning is set to process an average of 10% of the inverted lists.

4.4.2 Effects of Pruning

In contrast to the work in the previous chapter, where pruning schemes were trained and evaluated on small TREC data sets, our exploration of change in user queries requires access to a large query log. As the TREC environment provides no topic sets with a sufficient volume of queries, we employ a different methodology to measure the performance of a system.

To further explore the effects of pruning at query time we compared the difference in the returned result sets of our access-reordered index with query time pruning, to the results returned by evaluation with a document-order index using the continue scheme. Using 1,000 queries from each log, we marked the top 10, 100, and 1,000 results returned by the baseline approach as oracle runs, and compared these to the results returned by our access-reordered scheme. The 1,000 queries selected for this experiment were not among of the queries used to train the access counts. For the access-reordered index, the pruning parameter $c_a$ was
4.4. EXPERIMENTS IN ACCESS-REORDERING

trained to process on average 10% of the inverted lists, with a single-phase of pruning as discussed in Chapter 3.

Figure 4.4 shows the difference in result sets between the two systems when we consider the top 10, top 100, and top 1,000 returned results per query. Figure 4.4(a) compares an index reordered by access counts based on the first 20 million Lycos queries, while Figure 4.4(b) compares an index reordered by access counts generated by 10 million sampled Microsoft queries. For the collection reordered by access counts from the Lycos queries, we can see that, when considering the top 10 results per query, for 100% recall, precision remains above 70%. That is, on average our pruned scheme produces the same top 10 documents in the result set as full evaluation, for over 70% of the queries. As we increase the number of results considered per query, the similarity between our approach and the standard differ more, but it should be noted that across all three levels of considered results, precision at 90% recall does not drop below 80%. That is, nine out of ten queries produce results that match in over 80% of the documents presented. Figure 4.4(b) shows that an access-reordered index based on the Microsoft log produces similar results.

While different collections and queries are used, the results of this experiment show that, with pruning, access-reordered indexes produce similar results to that of full list evaluation. As such, these results are consistent with those presented in the previous chapter.

4.4.3 Stability of Access-Reordering

For a search system, the cost of reordering a collection is significant. When the collection is reordered, the entire index must be rebuilt. In this section we show that with as little as 100,000 queries, we can generate a collection ordering that will service results that do not differ greatly from an index reordered on access counts based on 20 million queries.

To examine the effects of query drift over time we generated indexes based on query logs from 50,000 queries up to 20,000,000 queries from the Lycos log. For each reordered index we ran 1,000 test queries and collected the top 10, 100, and 1,000 results returned per query. We treated the results returned by the index trained on 20 million queries as the oracle run and compared the difference in result sets returned by indexes trained on the other amounts of queries. Assuming that the ordering produced is stable, then an index trained on a relatively small amount of queries should be able to return similar results to that of an index trained on a large number of queries.

Figure 4.5 shows the mean average precision of results returned by indexes trained on
Figure 4.5: Comparison of results returned by indexes trained on varying numbers of training queries when compared to an index trained on 20 million queries. $N$ is the number of results returned per query. Oracle data is based on the top $N$ results returned when evaluating queries on an access-reordered index trained on 20 million queries. The oracle run is compared to access-reordered indexes trained on 50,000 to 20 million queries. Figure(a) shows the results when query evaluation is set to process an average of 10% of the query term inverted list postings, while Figure(b) shows the results for 20% processing.

The figure shows that, when pruning so that 20% of the inverted lists are processed on average, with 50,000 training queries, mean average precision between the two runs is over 90%, suggesting a high correlation in the returned query results between the two indexes. Further, with 100,000 training queries the results are 96% similar.

The figure also shows the similarity between the indexes when pruning the inverted lists to process 10% of the postings on average. In this case, little performance is lost to that of processing 20% of the lists, although for less than 100,000 training queries the result sets begin to visibly differ.

As in Section 4.4.2, we see that the similarity between the query results returned by the indexes remains high when examining the top 10 results per query, up to the top 1,000 results per query. This suggests that there is not a great amount of movement in the ranking of the documents returned by the varying indexes.
Figure 4.6: Difference in results returned by access-reordered indexes, trained on varying number of queries, to that of an index trained on 20 million queries. Oracle data is based on the top $N$ results returned when evaluating queries on an access-reordered index trained on 20 million queries. The oracle run is compared to access-reordered indexes trained on 50,000 to 20 million queries. Access-reordered pruning is set to process an average of 10% of the inverted lists.

Figure 4.6 shows precision at varying levels of recall for the indexes trained on different amounts of queries when query processing is set to process on average 10% of the inverted lists. The figure further illustrates the effect of training set size, and the ability of a small number of queries to produce an ordering similar to that of processing a significantly larger query log. When considering the top 10 results per query, we can see that for 100,000 training queries and above, precision remains relatively high, above 85%, at all levels of recall.

For the top 1,000 results per query, it is difficult to maintain high precision at 100% recall, however, for all the access-reordered indexes shown, precision at 100% recall remains above 50%. That is over 50% of the test queries return the same documents in the results. It is also promising that at 90% recall, precision remains above 84% for all but the index trained on 50,000 queries.

These experiments did not use the generated Microsoft log, as that log does not exhibit the time-ordering and clustering of queries that is an essential query log property for this task.
Table 4.2: Comparison of average time taken to process a query with a document-ordered index and an access-reordered index when $c_a$ is tuned to process 10% of inverted lists. Index sizes are also reported. For the stopped runs, stopping is applied at query evaluation time.

<table>
<thead>
<tr>
<th>Index</th>
<th>Milliseconds per query</th>
<th>Index Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document-order</td>
<td>89</td>
<td>5.3</td>
</tr>
<tr>
<td>Access-reordered</td>
<td>24</td>
<td>5.4</td>
</tr>
<tr>
<td>Document-order (stopped)</td>
<td>39</td>
<td>5.3</td>
</tr>
<tr>
<td>Access-reordered (stopped)</td>
<td>18</td>
<td>5.4</td>
</tr>
</tbody>
</table>

4.4.4 Efficiency

We have shown that an access-reordered index with pruning produces results similar to that of a standard index. We now compare the efficiency of query evaluation with an access-reordered index to the standard approach.

To compare the efficiency of query evaluation, we timed both systems using 100,000 queries from the Lycos log. To avoid bias, the timing queries were separate to those used to train the access-reordered index. An index trained on 1 million queries was used, with the $c_a$ parameter set to process 10% of the inverted lists on average. Main memory was flushed between each run to avoid caching effects.

Table 4.2 shows the average time to process a single query using a standard document-ordered index and an access-reordered index. The access-reordered approach with pruning is 73% faster. In our experiments, the access-reordered index was 2% larger than the standard index, as a result of storing a single access count value for each document in the collection. This is a significant improvement over the original access-ordered approach in Chapter 3, where index size grew by as much as 60%.

When pruning inverted lists at query time, a large saving comes from not processing postings in terms that occur frequently in the collection. These terms, often referred to as stop-words are often disregarded by search systems. To remove the above effect we performed timings on stopped queries using the same $c_a$ parameter.\footnote{Stopping was performed using a list of 600 common terms available from: http://www.cs.rmit.edu.au/~jz/resources/stopping.zip} In the case of stopping the access-reordered approach remains 54% faster than the standard approach.
4.5 Summary

In this chapter, we have presented a new index representation that builds on access-ordered indexes. This technique overcomes the compression difficulties raised by the access-ordered approach by generating an index that is equivalent in design to a standard document-ordered index. We have shown that, even when pruning inverted lists at query time by up to 90%, the returned results are similar to the results returned by a standard index processing with no pruning.

The costs of rebuilding the index are high. By examining the effect of the index size of the query training set on access-reordered indexes, we have shown that an access-ordering is robust. An index based on as little as 100,000 queries can produce similar results to orderings based on larger query training sets. This suggests that an access-reordered index has a durable life span and that the index need not be rebuilt frequently to allow for query drift.

Finally, we have shown that access-reordered indexes with pruning are significantly faster than the standard approach, both with and without stopping. The improvements in query evaluation time over the previous chapter can be attributed both to the reduced index size, and the reduced complexity of decoding lists. Under an access-reordered index, list decoding is no different to a document-ordered index, with the added benefit that list pruning can be applied during query processing.
Chapter 5

Static Index Pruning with Access Counts

In Chapter 3 we observed the query evaluation time savings that can be achieved through access-ordering. Access-ordering allows fast query evaluation by processing a fraction of each query term’s inverted list. However, the required inverted list reorganisation has the effect of increasing index size. Although dynamic query-time pruning resulted in high levels of accuracy while processing around one tenth of the total query term postings, the overall query evaluation time did not reduce substantially.

In Chapter 4 we integrated access-ordering and document-ordering. The result was a compact index that supports dynamic query time pruning for fast query evaluation. However, dynamic pruning techniques require full access to inverted lists, and the point at which list processing terminates is often unknown until a terminating condition is met; where, such conditions are often dependent on the properties of individual postings in the inverted lists. This limitation results in the unnecessary disk reads of postings that will not be considered at query time.

In this chapter we utilise the information provided by access counts to generate a compact inverted index with the benefits of document-ordering. By statically pruning the index to maintain only those postings that are frequently accessed in the inverted lists, we reduce index size and query evaluation time. We show potential query processing time reductions of up to 75%.

As static index pruning has the unfortunate side effect of reducing accuracy for some queries, we propose a two-tiered system where prior to query evaluation, queries are selected
for evaluation on either, a full index, or on a pruned index. We conclude by exploring methods for the selection of queries that are best suited for evaluation with a pruned index.

5.1 Access-Pruning

As noted in Section 3.6.2 (page 73), the maxpost pruning scheme ignores postings that fall beyond the first $P$ postings in each inverted list. Given that only the first $P$ postings per list are to be processed, the following extension can be applied. Instead of storing postings in access-order and dynamically pruning at query time, the index can be statically pruned at construction time, and be organised in document-order, maintaining only those postings with the top $P$ access counts. This approach has the benefits of both access- and document-ordered indexes. First, the postings for documents that appear in highly ranked documents are stored in the index, and second, the compression effectiveness of a document-ordered index is maintained with no additional processing required at query time. Further, by adjusting the parameter $P$, we can reduce the index size substantially, and therefore improve query performance with both reduced disk accesses and a higher likelihood of caching benefits. We refer to this approach as an access-pruned index.

Successful static pruning techniques require that those postings that are most significant at query time be available in the pruned index. As observed in Section 3.6.3 (page 75), access counting is an effective technique for the production of document orderings that favour relevant documents.

To illustrate the process of access-pruning, we use the sample Tempest collection of Section 2.3 (page 20). Similar to the access-ordering work of the previous chapter, we begin by processing a set of training queries and reordering the lists by access count. After the access-ordering procedure, we have the same inverted lists as shown in Figure 3.4. Now, assume that we seek to produce a small index and set the pruning threshold to a maximum of three postings per list (that is, $P = 3$). For the access-ordered list of the term “fish”:

$$\langle 386, 6 \rangle, \langle 283, 1 \rangle, \langle 436, 1 \rangle, \langle 430, 1 \rangle, \langle 480, 2 \rangle, \langle 750, 1 \rangle,$$

we take the three most accessed postings, and produce the access-pruned list:

$$\langle 283, 1 \rangle, \langle 103, 6 \rangle, \langle 50, 1 \rangle,$$

where 103 is the gap between 283 and 386. Note that, after pruning, we return the list to document-order, and allow d-gaps. Even though this approach leads to a wider distribution
of \(d\)-gap values, and an increase in the average size of the \(d\)-gaps in the lists, an appropriate selection of a threshold \(P\) will result in an overall reduction in index size. Figure 5.1 presents access-pruned lists for selected terms in the Tempest collection. The benefits of access-pruning are clear. With a low \(P\) threshold, index size dramatically decreases, resulting in query evaluation time reductions through reduced disk access.

Pruning an access-ordered index with a fixed threshold is an extension of the \textsc{maxpost} pruning scheme. In Chapter 3, we observed that the \textsc{maxpost} prune scheme was less accurate when applied to the TREC-10 topics than it was for the training topics of TREC-9. One possibility is that the topics in the TREC-10 data set are more difficult to resolve. Indeed, as we show later in Table 5.1, when using a standard document-ordered index with full list evaluation, the mean average precision (MAP) for the TREC-10 topics is 14\% lower, at 16.0\%, than that of the TREC-9 topics at 18.7\%.

Removing postings from the index has the effect of preventing some documents from being retrieved in response to a query. For all queries, especially those that are difficult for the search engine to resolve, the reduced volume of information available at query evaluation time can reduce accuracy. We propose a two-tiered search system where those queries that are predicted to be “simple” to resolve are processed by the efficient pruned index, while remaining queries are relegated to evaluation on a full index. “Simple” queries are those that are expected to have highly accurate results.

A sufficiently small first-tier index could potentially be stored in memory, rapidly reducing disk access costs for those queries that are supported by the index. Alternatively, selected lists of the first-tier index may be cached in memory as they are repeatedly requested in a stream of queries. In Chapter 6 we explore search engine caching, and consider heterogeneous caches.

<table>
<thead>
<tr>
<th>Term</th>
<th>Access-Pruned List</th>
</tr>
</thead>
<tbody>
<tr>
<td>fairy</td>
<td>(642, 2) (7, 1)</td>
</tr>
<tr>
<td>farewell</td>
<td>(47, 2) (387, 3) (30, 1)</td>
</tr>
<tr>
<td>fish</td>
<td>(283, 1) (103, 6) (50, 1)</td>
</tr>
<tr>
<td>state</td>
<td>(7, 2) (72, 1) (392, 1)</td>
</tr>
<tr>
<td>storm</td>
<td>(22, 1) (364, 3) (22, 2)</td>
</tr>
<tr>
<td>swords</td>
<td>(565, 1) (1, 2)</td>
</tr>
<tr>
<td>sycorax</td>
<td>(136, 1) (23, 2) (352, 2)</td>
</tr>
<tr>
<td>vision</td>
<td>(612, 1) (12, 1) (94, 1)</td>
</tr>
</tbody>
</table>

Figure 5.1: Access-pruned lists for selected terms from Tempest collection with \(P = 3\).
that support the wide variety of data sources used by search engines in the query evaluation process.

In the next section, we discuss techniques that can be used to select queries for use with the pruned index, and in Section 5.4 we present results that show the effect of access-pruning on index size, query evaluation time, and accuracy.

5.2 Query Selection

Pruning an index reduces the amount of information that the search system has access to at query evaluation time. While some queries can be resolved accurately with such limited information, others may request “hard to find” documents. In the case of access-pruned indexes, such queries may be those that seek documents that do not frequently appear in the search results.

Differentiating between the queries that are likely to be successful when evaluated with a pruned index, and the queries that are likely to be unsuccessful, is a difficult task. Various studies have investigated the task of query difficulty prediction. Approaches have ranged from simple measures such as query length, to the use of combinations of features with machine learning, to techniques that compute the divergence between term distributions in the results and the collection [Cronen-Townsend et al., 2002; He and Ounis, 2004; Yom-Tov et al., 2005b]. Section 2.12 (page 46) outlines several approaches to the task of query difficulty prediction.

For our purposes, we desire a difficulty predictor that can categorize queries prior to evaluation. In Section 5.4.3 we examine the application of the following predictors to select queries for processing with access-pruned indexes:

- **Query Length**: The length of a query may be a good indicator of difficulty. Longer queries contain more information, and make use of evidence from multiple inverted lists. With access to a wider selection of information, it is likely that long queries are easier to resolve.

- **Maximum IDF**: Most ranking functions value those terms that appear infrequently within a collection. The rationale is that matching a rare collection term allows the a greater degree of discrimination between documents than do terms that appear frequently throughout the collection. As such, a natural measure for query difficulty is to measure the rarity of its terms. Maximum IDF rates the difficulty of a query by the
5.3 RELATED WORK

document frequency of the rarest query term. Scholer et al. [2004] found this to be a simple and effective predictor for web queries.

- Query Clarity: To date, one of the most successful approaches to difficulty prediction is that of Cronen-Townsend et al. [2002], who propose a technique that measures the divergence in frequency of terms in the query results, to that of the collection. The calculation of the query clarity measure requires evaluation of the query before determining difficulty. While this approach is in practice not viable for the task of selecting queries rapidly, we examine it as a state of the art predictor. We also note that other works [He and Ounis, 2004] have explored variations of query clarity that determine query difficulty prior to retrieval, but do not explore such approaches further here.

Once selected, queries deemed to be simpler to resolve are candidates for processing with an access-pruned index. Prediction metrics may require selection of a parameter or threshold by which to select “simple” queries. To avoid the need for such training we rank the set of test queries from least to most difficult based on the result of each predictor, then starting from $N = 1$, we incrementally select the $N$ queries that are predicted to be least difficult, and measure the aggregate performance of the access-pruned index over those $N$ queries.

In Chapter 7 we further explore predictors of query difficulty, and attempt to use collection access-ordering as an indicator of difficult queries.

5.3 Related Work

Unlike the dynamic query-time pruning that is applied to an access-ordered index, access-pruning makes use of static index pruning. That is, pruning is performed only once at indexing time. Other works have proposed methods for static inverted list pruning. Carmel et al. [2001b] propose pruning lists based on the contribution of each posting to the final results. Akin to the impact-ordering work of Anh and Moffat [2002b], for each posting in the lists a query time contribution is calculated. However, instead of reordering the lists by these precomputed values, list postings are removed from the index based on the magnitude of their contribution. Each posting in a list is compared to the highest contributing posting for that list, and those postings that contribute less than a user-specified fraction of the largest contribution are removed. Carmel et al. report query evaluation time savings of approximately 50% when pruning over 60% of the index while retaining around 60% correlation in the top 10 results per query. They later applied their approach at TREC
[Carmel et al., 2001a] and reported that, for pruning of up 40% of the index, precision at 10 remains stable, with a slight reduction in mean average precision.

The query evaluation time reductions observed by Carmel et al. are a direct result of reduced disk traffic. While the dynamic pruning schemes examined in previous chapters reduce list decoding time, they still require that the entire inverted list for a term be read from disk prior to decoding. A statically pruned index reduces the size of the lists on disk. This has dual benefits. First, less disk activity is required to fetch an inverted list at query evaluation time; and second, system caching effects are likely to be greater as less data is moved from disk to main memory.

In related work, de Moura et al. [2005] propose an extension of the work of Carmel et al. to allow phrase- and proximity-based querying. This is accomplished by retaining the postings of all terms that appear in the same sentence as those selected by the original approach. While this approach allows improved accuracy for conjunctive queries, it does so by sacrificing compression, requiring a larger index to retain the postings selected by the Carmel et al. algorithm along with the postings of all terms that appear in the same sentence. The presented results suggest that the approach of de Moura et al. [2005] outperforms the original algorithm at similar levels of pruning.

Finally, Büttcher and Clarke [2005b] propose a multi-tiered index, where the most frequently occurring collection terms are maintained in a small index for fast query evaluation, while the remaining collection terms are stored in a larger index that is only used on demand. Further, Büttcher and Clarke apply a derivative pruning approach, like that of Carmel et al., where for lists in the small index only the highest contributing postings are retained. They report a 67% improvement in query evaluation time, with reductions of 8.6% and 4.3% in mean average precision and precision at 10 respectively.

Previous approaches to static index pruning focus on removing those postings that have a minimal impact on the ranking function. In contrast, our approach bases the decision of postings to prune on past user queries. Although we do not explicitly seek out those postings that contribute the most to the similarity metric, through training the postings for those documents that frequent the results are maintained, and a similar goal is achieved. However, in contrast to other approaches, access-pruning also retains postings for those documents that are popular in user queries despite the individual posting’s relationship to the similarity ranking function.

Like Büttcher and Clarke [2005b], we propose a two-tier search system. However, while Büttcher and Clarke partition their index by terms and re-sort to their larger index when
the query demands it, our approach bases the decision to use the pruned index on the result of a pre-evaluation difficulty measure.

5.4 Results

We now compare the performance of access-pruned indexes to full list evaluation. Our results show that access-pruning can reduce query evaluation time by around 25% while maintaining the accuracy of full list evaluation. We also explore various predictors of query difficulty, and show that, in combination with access-pruned indexes, simple predictors based on IDF work well for web style queries. Finally, based on training thresholds established by TREC topics, we show that up to 37% percent of real user queries may be considered “simple” and evaluated using a pruned index.

5.4.1 Experimental Environment

Our experiments make use of the WT10g and Gov TREC collections. Collection selection was based on the availability of a collection for which a matching query log and topic set is available. For the WT10g collection, generation of access counts is based on processing of the Excite 1997 and 1999 query logs, using the Static counting scheme (of Section 3.3, page 62), and considering the top 1,000 results per query. For the Gov collection, we use the first 250,000 queries of the MS-Gov query log to establish an access-ordering. The MS-Gov query log is detailed in Section 2.1.5 (page 14).

Timings were conducted on a Linux Fedora system running a 2.6.9 kernel. The system has dual Pentium Xeon 3.0 Ghz processors and 4 GB of RAM. To reduce caching effects, system memory was flushed between runs. All timings were undertaken when other load was minimal.

We used the open source Zettair [Zettair] search engine using document-ordering and the continue strategy with a 20,000 accumulator limit. Predicted similarity was measured with the Okapi BM25 metric. Query clarity was computed using the Lemur toolkit [Lemur].

For the WT10g collection, parameter training was on the TREC-9 topics 451–500, and tested with the TREC-10 topics 501–550. For the Gov collection we use TREC named-page topics 1–150. WT10g experiments use the title field, while Gov experiments use the description field. To reduce index size and for consistency with the previous chapters, document offsets were not stored within the indexes.
Figure 5.2: Plots showing the effect of $P$ threshold on index size, accuracy, and query processing time for WT10g collection. The access-pruned index is based on Static access counting with the Excite query logs. Accuracy is measured with the TREC-9 topics. Query time is averaged over 10,000 queries. The dashed lines represent performance for the baseline unpruned index.

Finally, to measure the degree of correlation between predicted difficulty and our test data, we use Spearman’s ranked correlation co-efficient [Sheskin, 1997].

5.4.2 Effects of Pruning

We applied access-pruning to the WT10g index for values of $P$ ranging from 1,000 to 100,000. As with maxpost pruning, accuracy remained high using the TREC-9 topics for pruning thresholds as low as 10,000. However, as noted in Section 3.6.4 (page 76), performance between the training and test topics differ significantly. For the recommended threshold of 55,000 postings per list, index size dropped to 856 MB, resulting in an index size that is 18.6% smaller than an unpruned document-ordered index, at 1052 MB. Query evaluation time re-
5.4. RESULTS

Figure 5.3: Distribution of inverted list lengths, measured in number of document postings for the WT10g collection. The x-axis shows terms ordered by decreasing length, the y-axis shows the number of document postings in each list.

duced to 33 ms per query from 80 ms, for document-ordered processing on an unpruned index. Further, at $P = 55,000$, access-pruning is faster than the equivalent MAXPOST approach of Chapter 3, which required 41 ms to evaluate a query on an unpruned access-ordered index, and 38 ms per query with stopping applied. The savings observed are a direct result of reduced disk activity. Access-pruning reduces the on-disk size of inverted lists. When these lists are required at query evaluation time, the reduced amount of data transferred results in query time savings. Further, as the lists loaded from disk are smaller, they have a greater chance of being cached in memory.

Figure 5.2 shows the effect of pruning on index size, accuracy, and query evaluation time, for varying values of the threshold $P$ on the WT10g collection. Accuracy is measured with MAP using the TREC-9 topics, while query evaluation time is measured over 10,000 queries taken from the Excite log. The dashed lines represent the performance with the baseline unpruned index. Note that index size drops slowly with the diminishing threshold. However, this is not surprising when considering the distribution of list lengths, where the majority
Table 5.1: Accuracy for access-pruned indexes on TREC-9 and 10 topics for low $P$ threshold values. The row labelled “full index” shows the baseline values for evaluation using an unpruned index.

<table>
<thead>
<tr>
<th>Threshold $P$</th>
<th>TREC-9 MAP</th>
<th>TREC-9 P@10</th>
<th>TREC-10 MAP</th>
<th>TREC-10 P@10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.073</td>
<td>0.158</td>
<td>0.085</td>
<td>0.190</td>
</tr>
<tr>
<td>2,000</td>
<td>0.096</td>
<td>0.185</td>
<td>0.094</td>
<td>0.196</td>
</tr>
<tr>
<td>5,000</td>
<td>0.168</td>
<td>0.219</td>
<td>0.123</td>
<td>0.224</td>
</tr>
<tr>
<td>10,000</td>
<td>0.186</td>
<td>0.254</td>
<td>0.129</td>
<td>0.224</td>
</tr>
<tr>
<td>15,000</td>
<td>0.189</td>
<td>0.254</td>
<td>0.135</td>
<td>0.230</td>
</tr>
<tr>
<td>Full index</td>
<td>0.187</td>
<td>0.271</td>
<td>0.160</td>
<td>0.318</td>
</tr>
</tbody>
</table>

of terms appear in very few documents. Figure 5.3 illustrates the list length distribution in the WT10g collection. It is clear that even for large values of $P$ only a small fraction of the indexed terms are pruned. Even for the recommended $P = 55,000$ threshold, less than 1,200 term lists are affected by our pruning method. However, it is worth noting that those same 1,200 terms appear in over 21% of the Excite queries used to generate the access-ordering. As such, selecting an effective pruning threshold that produces accurate query results is a non-trivial task.

For values of $P$ below 15,000 postings, query evaluation time drops below 20 ms per query, a reduction of over 75% in query evaluation time over full list processing. However, when applied to the TREC-10 test data, a significant reduction in accuracy is notable for evaluation with the access-pruned index, compared to the full index. Table 5.1 shows the accuracy of access-pruned indexes for values of $P$ below 15,000 for both sets of data. While a loss in accuracy is observed as the threshold is reduced, the quality of results only begins to rapidly decline as $P$ is reduced to values below 5,000.

Figure 5.4 shows a per query breakdown of average precision for the TREC-9 topics when pruning with $P$ at 1,000, 5,000 and 15,000. While the low threshold of $P = 1,000$ results in decreased accuracy for 81% of topics, we rapidly see improvement as the threshold is increased. At $P = 5,000$, only 52% of the queries are negatively affected by the pruning, with 9 of 25 negatively affected topics reducing in absolute average precision by less than 1%. Finally at $P = 15,000$, only 29% of the topics are negatively affected by pruning. These results show that with a balanced amount of pruning, efficiency improvements are possible if
5.4. RESULTS

Figure 5.4: Comparison of accuracy for TREC-9 topics when evaluating with access-pruned indexes to full index evaluation for varying $P$ thresholds. Topics are ordered by the difference in average precision of evaluation on a $P = 1,000$ access-pruned index, and full list evaluation. As the pruning threshold $P$ increases, the difference in accuracy between topics diminishes.
Table 5.2: Spearman’s ranked correlation between predictor and MAP (over 1,000 documents) when evaluating the TREC-9 topics with an access-pruned index with $P = 5,000$ and the WT10g collection.

<table>
<thead>
<tr>
<th>Predictor</th>
<th>Correlation</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query length</td>
<td>0.003</td>
<td>0.984</td>
</tr>
<tr>
<td>Max. IDF</td>
<td>-0.322</td>
<td>0.026</td>
</tr>
<tr>
<td>Query Clarity</td>
<td>0.484</td>
<td>$5.7 \times 10^{-4}$</td>
</tr>
</tbody>
</table>

those queries that are expected to work well with a pruned index can be selected prior to evaluation. We examine techniques for query selection in the next section.

The effect of the $P$ threshold on the Gov collection is shown in Figure 5.5. As the TREC topics for this collection represent a named-page task, accuracy is measured with mean reciprocal rank. The results follow the trends of the WT10g collection. The reduction in query evaluation time is significantly greater for the Gov collection. This is due to the increased average query length of the timing queries. While the WT10g timing queries from the Excite log have a mean length of 2.2 terms, the MS-Gov queries used for the Gov collection average 3.4 terms per query.

Although the Gov collection contains significantly fewer documents than the WT10g collection, access-pruning has a more pronounced negative effect on accuracy for this data set. This effect can be attributed to the named-page task, where there is commonly only a single relevant document. With such limited relevance judgements, it is possible for the relevant document to be unavailable in the index after pruning. Despite this limitation, for $P > 45,000$, access-pruning produces accuracy that is within 11% of the accuracy of an unpruned index, while reducing query evaluation time by as much as 77%.

Figure 5.6 shows the effect access-pruning on the Gov collection for the first 50 named page topics. Like the WT10g collection, accuracy improves as the pruning threshold is relaxed. Even at $P = 15,000$, where processing is 87% faster than full list evaluation, the access-pruned approach achieves equivalent or higher accuracy than full evaluation for 50% of the topics.

5.4.3 Query Selection

We experimented with various predictors of query difficulty. Table 5.2 shows the correlation between the average precision per query for a WT10g access-pruned index, and the different
Figure 5.5: Plots showing the effect of $P$ threshold on index size, accuracy, and query processing time for Gov collection. The access-pruned index is based on Static access counting with the MS-Gov query logs. Accuracy is measured with the TREC NP1–150 topics. Query time is averaged over 10,000 queries. The dashed lines represent performance for the baseline unpruned index.
Figure 5.6: Comparison of accuracy for named-page topics when evaluating with access-pruned indexes to full index evaluation for varying $P$ thresholds. The first 50 topics of the data set are shown. Topics are ordered by decreasing difference in reciprocal rank, when using evaluation on a $P = 15,000$ access-pruned index, and full list evaluation.
predictors. The poor results of the query length predictor are consistent with the work of He and Ounis [2004] who report similar findings. Although query length proved to be a disappointing predictor, both query clarity and the IDF based predictor resulted in an acceptable level of correlation between prediction and accuracy. We found query clarity to be the best predictor at 48% correlation between actual and predicted accuracy, which while high for the task of query difficulty prediction, is not ideal.

We applied the IDF and query clarity based predictors to the TREC-10 topics. Commencing with the predicted least difficult query, we incrementally processed topics for each predictor using access-pruned indexes. At each step we calculated the accuracy of the run using mean average precision with 1,000 results per query, and compared the results to evaluation on a full index. Figure 5.7 shows the MAP for the $N$ least difficult predicted topics. At $N = 50$, MAP is the same as that of the complete run.

For the query clarity based predictor, we see that accuracy of evaluation using the access-pruned indexes is limited. The index pruned with $P = 15,000$ closely tracks the performance of full list evaluation for 70% of topics (up to $N = 35$). The high accuracy is not reflected in the $P = 5,000$ or $P = 1,000$ access-pruned indexes. For $P = 5,000$, MAP over the $N \leq 8$ least difficult predicted topics is noticeably lower than that of full list evaluation. This suggests that query clarity is not a good predictor for access-pruning when the index is heavily pruned.

Conversely, the maximum IDF predictor, effectively selects those topics for which accuracy between the access-pruned index and the full index are comparable. For all values of $P$, the IDF predictor selects topics that, when evaluated with an access-pruned index, closely track the accuracy results of evaluation using a full index. While this holds for the early queries, the effectiveness of the two types of indexes begin to differ after 15% ($N < 8$) of the queries have been processed.

The IDF predictor and query clarity predictor both show that an access-pruned index can be effective for queries that are predicted to be easy to resolve. Computation of the clarity measure requires sampling from the collection [Cronen-Townsend et al., 2002], and therefore counteracts the query time savings of processing with a pruned index. However, pre-retrieval predictors, such as maximum IDF, are ideal for the task of routing queries between different search systems. We continue our discussion of difficulty prediction in Chapter 7.

Finally, the distribution of predicted query difficulties in TREC topics may not be representative of web queries. To explore the savings that can be achieved using our approach, we classified 10,000 topics from the 1997 Excite query log into difficult and non-difficult queries
Figure 5.7: Accuracy of evaluation with access-pruned index when queries ordered by query clarity or maximum IDF predictor. Evaluation is on WT10g collection using the TREC-10 topics with varying $P$ index pruning thresholds. Accuracy is measured with mean average precision over the $N$ predicted least difficult topics.
Table 5.3: Query evaluation time for 10,000 Excite queries when partitioned between full evaluation, and evaluation on a pruned index. Query difficulty is determined using the IDF predictor. Difficulty thresholds are based on the 8 lowest IDF values of the TREC-10 topics. For each threshold, the table shows the fraction of queries that are deemed as “simple” and “difficult”. Pruned evaluation performed on an index with $P = 5,000$.

<table>
<thead>
<tr>
<th>Difficulty rank threshold</th>
<th>Queries</th>
<th>Evaluation time (ms)</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simple</td>
<td>Difficult</td>
<td>Simple</td>
<td>Difficult</td>
<td>Average</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>15.5%</td>
<td>84.5%</td>
<td>18.4</td>
<td>81.9</td>
<td>72.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>21.5%</td>
<td>78.5%</td>
<td>18.1</td>
<td>81.1</td>
<td>68.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>26.2%</td>
<td>73.8%</td>
<td>17.5</td>
<td>83.0</td>
<td>65.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>33.0%</td>
<td>67.0%</td>
<td>17.0</td>
<td>84.8</td>
<td>62.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>33.5%</td>
<td>66.5%</td>
<td>16.9</td>
<td>85.0</td>
<td>62.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>33.5%</td>
<td>66.5%</td>
<td>16.8</td>
<td>85.4</td>
<td>62.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>35.2%</td>
<td>64.8%</td>
<td>16.8</td>
<td>85.3</td>
<td>61.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>36.9%</td>
<td>63.1%</td>
<td>16.6</td>
<td>85.6</td>
<td>60.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full evaluation</td>
<td>0%</td>
<td>100.00%</td>
<td>n/a</td>
<td>80.1</td>
<td>80.1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

using the IDF based predictor. We then processed each batch of queries sequentially on either a full index or an access-pruned index.

Query classification requires the selection of a maximum IDF value with which to segment the queries. Selection of a maximum IDF value was based on the data in Figure 5.7, where the eight least difficult topics using access-pruning, closely matched the accuracy of full evaluation. For each of the eight least difficult topics, we noted the maximum IDF value of the topic, and used that value to segment the queries used for timings. For all 10,000 queries in the timing set, the maximum IDF of each query was calculated. Those topics for which maximum IDF was found to be lower than the threshold, were added to the non-difficult query set and processed on an access-pruned index with $P = 5,000$, while all other queries were added to the difficult query set, and processed on a full index.

Table 5.3 shows the fraction of queries categorised as “simple” and “difficult” for each of the IDF based threshold levels obtained from the eight predicted simplest queries. The table also reports the average query processing time for each subset of queries, and the overall average processing time per query.

Using the threshold obtained from the simplest training query alone, produces a substantial query evaluation time saving over full evaluation, with average query time reducing 10%
from 80 ms to 72 ms. This is an interesting result as, although the threshold is based on the simplest predicted query of the fifty TREC-10 topics, over 15% of the 10,000 timing queries were found to have a lower maximum IDF value, suggesting differences in the properties of TREC topics and web queries.

As the categorisation threshold is relaxed, and more queries are allowed into the “simple” set, the query time savings improve slowly but substantially. By the threshold based on the eighth least difficult query, average query evaluation time is reduced by 25% to 60 ms.

While in Chapter 3, we saw a reduction in query evaluation time of up to 39% using dynamic query time pruning, the techniques explored in this chapter provide an alternative yet compatible means of query optimisation. As “difficult” queries are processed on a full index, it remains possible to use an access-ordered index with query time pruning to process the more difficult queries, while using the access-pruned index to process the “simple” queries. We leave this extension as further work.

5.5 Summary

In this chapter we have shown that access counts can be utilised to generate small indexes for fast query evaluation. We demonstrated improvements of up to 75% in query evaluation time for queries, by pruning inverted lists of the WT10g index to maintain only the postings with the highest accesses. However, improved efficiency was obtained at the cost of accuracy. Indeed, a 75% reduction in query evaluation time resulted in a reduction of over 22% in MAP. We then explored query difficulty prediction measures and showed that for predicted “simple” queries, access-pruning can be as effective as full index evaluation.

The work in this chapter presents an initial exploration of static index pruning using access counts. We note that the fixed length pruning method presented here is simplistic, and that other approaches where lists are pruned dynamically based on the importance of their postings may result in further improvements. We also note that our approach is compatible with term based pruning, where entire lists are removed from the index [Shokouhi et al., 2007], and may result in further gains.

Similarly, our exploration makes use of a small subset of the query difficulty predictors available. There is scope for the exploration of more accurate pre-retrieval predictors. We explore query prediction further in Chapter 7.
Chapter 6

Query Caching

In previous chapters we have explored methods to both improve the efficiency and effectiveness of search by exploiting the frequency of document repetition during the query evaluation process. While our studies of access-ordering have resulted in both interesting and promising results, the speed of query resolution depends on more than inverted list processing. Hence, we step back and consider the overall search engine software architecture.

A search engine makes use of a wide variety of data structures to resolve user queries. In this chapter, we commence an investigation into the access frequencies for these data structures. We provide cost models to analyse the benefits of caching, and propose a heterogeneous caching technique to improve the management of data in search engine caches.

Our results contrast the effect of caching different query time data structures. Of these, result page caching offers the greatest reduction in disk access. However, for smaller sized caches, the caching of lower level data structures, such as vocabulary entries, can prove to be just as beneficial. Finally, a caching policy that is able to heterogeneously deal with the wide variety of the data required at query evaluation time, can substantially improve caching performance over traditional single-item caching approaches.

6.1 Motivation

A typical search engine must process a stream of queries, rapidly identifying likely answers to each and returning those answers to the user. As detailed in Section 2.5 (page 24), efficient search of a large text collection requires use of an inverted index, where each term in the collection has an inverted list containing a pointer to each document that contains that term. To resolve a query the individual terms must be validated by accessing a vocabulary, then
the inverted lists for the component terms are fetched and combined to give similarity scores for the documents containing the terms. The documents themselves are then accessed, to produce snippets or document summaries for presentation to the user.

Some of the cost of query processing can plausibly be avoided by caching information about past queries, in the hope that the queries or the query terms will recur. Studies of query logs suggest that many queries are frequently repeated [Jansen et al., 2000; 2005]. Some are posed to search systems at regular intervals, such as queries for travel information, while others are temporal and are posed frequently over a brief period, such as queries relating to current news events [Diaz and Jones, 2004]. There are other forms of recurrence. Some words appear frequently in distinct queries [Jansen and Spink, 2005], while some queries may be reformulated, so that part of a recent query may recur in a new form. Also, as we have noted in Chapters 3 to 5, some documents appear more frequently than others across different queries. All of these elements are candidates for caching.

To date, much of the research on caching has focused on the final result pages presented to the user [Lempel and Moran, 2003; 2004; Markatos, 2001]. These are expensive to compute and, compared to inverted lists, relatively small, providing good reasons to cache this material. If other kinds of data, such as a vocabulary entry, are cached, there are potential benefits, but in the context of a finite cache it is conceivable that more valuable material might be displaced. The separate kinds of data accessed in the process of answering a query could be allocated separate caches, leading to the problem of determining the size of each cache and, potentially, requiring that the number of caches multiply with the number of query types supported.

In this chapter we argue that a more attractive alternative is to use a single heterogeneous cache. A key problem of caching heterogeneous data is choosing a replacement strategy for the items in the cache; that is, of finding a mechanism for dynamically balancing the different kinds of data items. Consider the case of the enterprise server that runs several services: the organization web server, local databases, and in our case the search system. In such a system, memory is required by several applications, and the amount that may be available to the search system is likely to be limited. In such cases, the memory available for the cache needs to be populated with the best possible data to reduce disk access when servicing user queries. It follows that the decision about what is cached in this limited space should be based on two parameters: what is most likely to be required, and what is the most costly to retrieve.
For these reasons, a simple Least Recently Used (LRU) strategy is unlikely to be effective, as it takes no account of the cost required to recreate a piece of data. For example, it makes little sense to discard an expensive item used at time units ago when a cheap item used $t-1$ units ago is retained. Likewise, the argument that explicit caching is unnecessary, as the operating system already provides disk caching, does not stand scrutiny. We need not disk blocks or memory pages, but individual data entries, which may range in size from a few bytes to many megabytes. For example, consider the search engine’s vocabulary component where each entry is often less than 100 bytes, with multiple entries stored in a single page of disk. Operating systems usually cache entire pages, and thus, for each entry required, tens to hundreds of other vocabulary entries are cached unnecessarily.

We propose an approach to caching for search engines that allows the cache to handle heterogeneous query-time components, through use of a rich cost model that quantifies the cost of recomputing each item, and thus the loss should it be discarded and then required again. Previous work has considered caching only a limited subset of these components. The cost model makes use of detailed measurements of hardware system characteristics, allowing customization to different hardware environments.

We compare the benefits of caching a variety of query time components, both individually and in combination. We find that a well managed cache of many kinds of data item is far superior to a cache that only considers a single data type, with reductions in query time disk access of up to 80%. We compare a heterogeneous cache with a segmented cache that manages each data type independently. Our findings show that a heterogeneous approach with an effective replacement policy is comparable to the segmented approach, and is free of the need to tune individual cache sizes.

### 6.2 Related Work

To efficiently service user queries, a text search engine makes use of several components [Arasu et al., 2001; Brin and Page, 1998; Zobel and Moffat, 2006], in particular a vocabulary structure, inverted lists, a document map, and documents. The vocabulary structure contains the terms that appear in the indexed collection of documents, and maintains statistics about each term such as the number of documents in which it appears. For each term in the vocabulary, an inverted list of postings is stored, identifying the locations where that term appears in the collection. The document map holds, for each document, statistics such as document length, the number of distinct terms in the document, and the location of the document on disk.
At search time, a similarity metric makes use of statistics from the vocabulary structure, inverted lists, and document map to compute a score between a query and the documents in the collection. Result pages comprised of the top-scoring documents are presented to the user once a query has been processed; typically a summary or snippet of the document is presented with the results [Tombros and Sanderson, 1998]. Each of these items must be either held in memory or fetched from disk.

For small collections, the vocabulary structure could plausibly be held in main memory, but, as collection size increases, the rate at which new terms are encountered remains more or less constant [Williams and Zobel, 2005]. Thus for large collections the greater part of the vocabulary must reside on disk. Inverted lists form the largest part of an index, and typically exceed memory size by orders of magnitude. The lists for individual terms vary in size, with the lists for common terms such as “the”, “and”, and “as” comprising a significant part of the total index. The document map, with a few bytes per document, is directly proportional to the number of documents in the collection and can reside on either disk or in main memory. Finally, generation of result summaries of matching documents requires access to the document collection itself. The document collection is significantly larger than the index and must be stored on disk [Brin and Page, 1998].

Regardless of the similarity metric employed, query evaluation involves the following stages. First, for each query term the associated entry is retrieved from the vocabulary. Second, the term’s inverted list is loaded from disk. Third, for each posting in the inverted list a partial similarity score is calculated between the document to which the posting refers and the query. If the document that the posting refers to has not been previously encountered, an accumulator is created, initialized with the partial score for the document. If the document has been previously encountered, its accumulator is updated with the partial score. Fourth, after all query terms have been processed the accumulator values are adjusted using document based statistics that reside in memory. Finally, the documents are partially sorted by accumulator value, and the top $R$ documents are accessed through the document map, summarized and returned to the user. The search engine query evaluation process is discussed further in Section 2.5 (page 24).

Table 6.1 shows the estimated disk cost per query for the components, averaged over our test collections, and described in further detail in Section 6.4. The cost of accessing disk can be broken up into two distinct components. First, the disk head must seek to the location of the data to be read from the disk. Second, the data must be physically read from the disk. Yiannis [2005], in work on external sorting optimizations, presents an overview of the costs.
6.2. RELATED WORK

Table 6.1: The sequence of components accessed during query evaluation. Estimated disk costs are in average bytes read and average disk seeks per query for our test collections and query logs. Times are in milliseconds, using disk specifications from our test machine. We assume ten documents per results set.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk seeks</td>
<td>2.03</td>
<td>2.03</td>
<td>n/a</td>
<td>10.00</td>
<td>10.00</td>
<td></td>
</tr>
<tr>
<td>Bytes read</td>
<td>89.10</td>
<td>1,687,667</td>
<td>n/a</td>
<td>440.00</td>
<td>114,517</td>
<td></td>
</tr>
<tr>
<td>Seek time</td>
<td>8.42</td>
<td>8.42</td>
<td>n/a</td>
<td>41.60</td>
<td>41.60</td>
<td></td>
</tr>
<tr>
<td>Read time</td>
<td>$2.83 \times 10^{-3}$</td>
<td>5.36</td>
<td>n/a</td>
<td>$0.14 \times 10^{-3}$</td>
<td>0.36</td>
<td></td>
</tr>
</tbody>
</table>

associated with disk access in a retrieval environment. These figures suggest that disk seek time during query evaluation is dominated by access to the documents during summarisation, while disk read time is dominated by access to the inverted lists.

We speculate that, in a distributed environment, the situation does not differ significantly. Seek time is akin to establishing a connection with, or initiating a request from a peer or server, while read time is bound by the bandwidth of the network. The size of inverted lists and documents remain the same, and access to each is limited by their distribution within the network.

As discussed in Section 2.11 (page 43), various works have explored search engine caching strategies. Some have focused on the caching of inverted lists [Brown et al., 1994], others on result pages [Markatos, 2001; Lempel and Moran, 2003], and others on combinations of the two [Saraiva et al., 2001; Long and Suel, 2005]. However, to date, no work has considered a single heterogeneous cache capable of dealing with all the components utilised during the query evaluation process. Further, there has been no exploration of the costs associated with maintaining such a cache.

In this chapter we present a structured approach to search engine caching. We examine the effects of caching the various components used throughout the query evaluation process and provide cost models that can be adjusted to the underlying system architecture to estimate the savings that can be gained by effective caching.
CHAPTER 6. QUERY CACHING

6.3 Caching Strategies

The first choice that must be made when caching the data used to answer search engine queries is the set of data elements to cache. In other research, it is proposed that there be one cache for inverted lists, another for results sets, and perhaps even a third for computed information such as accumulators. Caching these components individually, and some in combination, has lead to performance gains [Saraiva et al., 2001; Baeza-Yates and Saint-Jean, 2003; Long and Suel, 2005]. It also seems reasonable that caching vocabulary entries and documents will lead to performance gain.

When caching results sets, a further decision must be made about the number of results to cache. The default number of results presented to a user on most current search engines is 10, and so it seems that this would be a minimum number of results to store for a query. We refer to this unit of results as a results page. If the user clicks on the “next” button to get results 11 through 20, the next results page, it would also be advantageous to have these results in cache. If these results are not cached, but the set of accumulators that were developed for the query are, then the next page can be derived from the cached accumulators, rather than requiring the inverted lists to be re-processed. In our approach we cache vocabulary entries, inverted lists, accumulator sets, and result pages of length 10.

Once the set of data items to be cached has been decided upon, an eviction policy must be designed. This is a strategy for choosing items to remove from a full cache to make way for items that are required but are not already in the cache. In previous work, the eviction strategy has often been LRU: least recently used items are removed from the cache until there is room for the new item. We adopt this as our baseline eviction strategy. For comparison, we also use a random eviction policy where items are chosen at random to be removed from the cache when space is required, along with a no-replacement policy in which items are never removed from the cache after it has been filled.

The final choice in cache design is deciding how much of the available cache space to allocate to each data item. In previous work, a fixed amount of cache has been allocated to each type of data item, and the allocation is tuned manually. In order to avoid explicit allocation decisions of this kind, we opt to keep all items in a single, heterogeneous cache.

We must then address several questions to make the approach practical. In such a cache, should each data item have its own eviction strategy? For example, should longer inverted lists, those for the most common terms, be preferred in the cache, or should shorter lists be kept for a longer duration as is suggested in web-page caching literature [Abrams et al.,
6.3. CACHING STRATEGIES

Keeping shorter lists in cache allows for a greater range of terms to match, but the caching of common terms may result in less reading from disk as the lists are larger and they may be requested more often. Also, how many result pages per query should be cached? Given that only 27% of users look beyond the first page of results [Jansen and Spink, 2006], is there any benefit in caching pages other than the first? Other components also offer interesting problems to solve. Is it worth caching items such as accumulator sets and vocabulary entries? How much can be gained by doing so? If accumulator sets are to be cached, how many results should be stored? If vocabulary entries are kept, how should terms be chosen? We address all of these questions by assigning a cost to each data item in the cache that reflects the disk time required to re-create that item and the probability that the item is about to be re-used. The lowest-cost item is then evicted from the cache when space is required. We now detail the cost model.

6.3.1 General Cost Model

We are concerned with reducing the overall disk I/O associated with query evaluation. We must first establish the costs of accessing the individual data items used in the query evaluation process from disk. As outlined in Section 6.2, first the vocabulary entry must be fetched from disk. The cost of accessing the vocabulary entry $v_t$ for term $t$ is:

$$C(v_t) = s + rB(v_t),$$

where $s$ is the average seek time required to position the disk head in milliseconds; $r$ is the average transfer rate when reading from the disk in milliseconds per byte; and $B(x)$ is a function that gives the size of component $x$ in bytes. Similarly, the cost of fetching inverted list $\ell_t$ for term $t$ is:

$$C(\ell_t) = s + rB(\ell_t).$$

Generation of an accumulator set requires access to inverted lists and vocabulary entries for the query terms. By assuming any required document based statistics are in memory, the disk cost to generate an accumulator set $A_q$ for query $q$ is:

$$C(A_q) = \sum_{t \in q} (C(v_t) + C(\ell_t)).$$

Finally, generation of a result set $R_q$ for query $q$ requires the generation of an accumulator set and snippets for each document in the results page. Generating a snippet involves reading
the document map entry, \( m_d \), and then fetching the document, \( d \), from disk. Hence, the cost of generating a results page is:

\[
C(R_q) = C(A_q) + \sum_{d \in R_q} \left( 2s + rB(m_d) + rB(d) \right).
\]

Given a cost \( C \) of accessing each data item, we estimate the disk access time saved to build a results set, \( T(R_q) \), for some query \( q \). Based on our assumptions, the time in milliseconds saved when processing an inverted list for term \( t \) is:

\[
T(\ell_t) = \begin{cases} 
0 & \text{if the list } \ell_t \text{ is on disk} \\
C(\ell_t) & \text{if the list } \ell_t \text{ is in cache.}
\end{cases}
\]

Similarly, the time saved when processing a vocabulary entry for term \( t \) is:

\[
T(v_t) = \begin{cases} 
0 & \text{if the vocabulary entry } v_t \text{ is on disk} \\
C(v_t) & \text{if the vocabulary entry } v_t \text{ is in cache,}
\end{cases}
\]

and the time saved in deriving accumulators is:

\[
T(A_q) = \sum_{t \in q} (T(\ell_t) + T(v_t)).
\]

Finally, the time saved in creating \( R_q \) is:

\[
T(R_q) = \begin{cases} 
T(A_q) & \text{if } R_q \text{ is not in cache} \\
C(R_q) & \text{if } R_q \text{ is in cache.}
\end{cases}
\]

This model of costs has several benefits. It is easily adapted to different hardware by simply changing the values of the disk seek and read timings, and it is easily extended to include factors such as CPU processing costs and network latency costs for remote disk access.

Due to experimental constraints, in this work we choose to not cache documents. However, adoption of cached documents into the \( T(R_q) \) cost formulation requires a trivial change where the cost of generating the results set is reduced by the number of documents that are present in the cache. We leave the examination of the document cache component as an item for future work.

### 6.3.2 Cache Management

Our general cost model provides a means by which we can calculate the expected time to process a query. In the case of a search engine cache we are interested in storing the
components of a query that cost the most to access. Therefore we must trade between the
cost of component access, and the amount of space that will be consumed by the items placed
in the cache.

A simple ratio of cost-per-byte (CPB) can be employed to value the cost of any item in
the cache:

\[ CPB(x) = \frac{C(x)}{B(x)}. \]

This ratio provides a method by which we can value any data type cached, allowing a het-
erogeneous cache replacement policy. However, the ratio does not account for the temporal
locality of queries. A search engine cache should value the components of queries that have oc-
curred recently with higher weight, as they have a greater likelihood of re-occurring. To allow
for such locality we apply a time decay function to items in the cache, giving the revised ratio:

\[ CPB'(x) = \frac{CPB(x)}{E(x)}, \]

where \( E(x) \) is the elapsed time, in number of queries processed, since item \( x \) was last used in
the cache. This decayed CPB ratio devalues items in the cache as time passes, hence allowing
for temporal locality within the cache.

A difficulty with the CPB approach is that, at each cache update, the CPB of every item
in the cache must be calculated to determine which item should be removed. Calculation
of these values may outweigh any savings in the disk access costs achieved. To simplify the
task of cache management we propose the following.

Vocabulary entries are typically close to a fixed size. The only variation in \( CPB'(v_t) \),
therefore, is the denominator \( E(v_t) \), hence the vocabulary entry with the largest \( E(v_t) \) will
have the smallest cost \( CPB'(v_t) \). This allows the cache management of vocabulary entries to
be simplified into a queue of elements with the oldest (highest \( E(v_t) \)) at the front. The exact
CPB can be calculated for this item and compared with the other data items to determine
the minimum cost item for eviction, without the need to keep exact costs for all vocabulary
entries. With the other data types in separate queues as described below, the cost of cache
eviction then reduces to a search amongst the front items in a small number of queues.

Other simplifications reduce the complexity for other data types. Generation of an accu-
mulator set requires access to both the inverted lists and the vocabulary entries of a query.
By substituting the fixed vocabulary entry size and assuming that inverted lists are of a fixed
size equal to the average inverted list size, we reduce the cost function for accumulator sets to:

\[ C(A_q) = |q| \times \sum_{v_t} \left( C(v_t) + C(\ell_t) \right)/n, \]
where \( n \) is the number of distinct terms in the collection, which is a constant multiplied by the query length, \(|q|\). Again, cache management can be reduced to a single queue by substituting an average query length for \(|q|\), or it can be reduced into several queues of differing query lengths. In our simulations below we use a single queue and assume an average query length.

Extending these simplifications, result set cost calculation can be simplified by assuming \( B(d) \), the size of a document, is constant at some average value. As all \( CPB(x) \) are now constant for vocabulary entries, accumulators, and result sets, LRU queues can be used for each of these data types.

Although a similar approach could in principle be taken for inverted lists, we choose to not simplify \( CPB(\ell_t) \) in the cache in this way, as inverted lists can vary in size by a factor of a million or more. The variation in inverted list size over our two test collections is shown in Table 6.2. The key problem is that, without representing the size accurately, there is the likelihood of large items remaining in the cache for an extended period of time. In this case, the decayed CPB for each inverted list item in the cache must be calculated at each cache update interval. We discuss methods to further optimize this requirement in Chapter 8.

With the above simplifications, we can reduce the effort of determining which item to remove from the cache to checking the CPB of the head of an LRU queue for each data type, as well as checking the cached set of inverted list entries.

### 6.4 Experiments

Our experiments use a simulator that records disk accesses at each stage of query evaluation, a common methodology in previous caching work [Markatos, 2001; Lempel and Moran, 2003]. The decision to use a simulator is based on the need to calculate actual accesses to the file system and to accurately control for a range of variables. Although we considered implementing the cache in a live search system, there are several arguments against this option.
It is difficult to control the internal caching of the operating system, and it is not possible to prevent hardware from making use of various layers of caching, such as caching by the disk controller and caching by the CPU. As a result, timings from a live system would include savings that are not due to our caching approach, and the results would be restricted to the architecture, memory capacity, and operating system on which the experiments were run. By studying the scheme with simulations (but real data) we get results that are independent of one particular computing system. Furthermore, we can vary parameters of the simulator to reflect different environments.

Our model for heterogeneous caching uses several search system values. In the results presented below we make the following assumptions:

- A vocabulary entry has a fixed size of 44 bytes;
- A result page presents the user with 10 ranked results and occupies 4 KB; and
- An accumulator is composed of a document identifier and similarity score requiring 12 bytes per accumulator. As an accumulator set is composed of 100 accumulators, each set is 1,200 bytes. The top 100 accumulators allow generation of up to the first 10 pages of query results.

In this work disk costs are based on the Seagate Barracuda 7200.9 series disk drives [Seagate]. These disks have an average seek time of \( s = 4.2 \) milliseconds and a maximum transfer rate of \( r = 300 \) MB/sec.

We make use of two collections in this work. The first is an external crawl of the domain anon.edu.au and a query log from the on-site search feature. We refer to this as the Anon collection. The collection contains 283,869 documents. The query log contains 576,649 queries as submitted to the site. As this log does not contain session information, query sessions were simulated by treating all repeat queries within a sliding window of 5 queries as sequential requests: that is, a request for the next page of results, with a limit of 5 repeat queries per sequence also imposed.

The second collection is the WT10g web collection from TREC. This collection contains 1.7 million documents gathered from the web and is a subset of the VLC2 collection. We use the 1997 Excite query log, as it represents web queries from about the same time period as when the document collection was gathered. We used the first 500,000 queries in the log. The TREC collections are outlined in further detail in Section 2.2.4 (page 18), while the Excite collection is detailed in Section 2.1.1 (page 10).
Figure 6.1 shows the frequency with which result pages are requested using the logs. We can see that both logs share similar properties, however the sliding window approach used for the Anon queries prevents requests for results beyond the fifth page.

For all collections, indexes were built using the open-source Zettair [Zettair] search engine using light stemming. The inverted lists sizes were extracted from each index and used in the simulations.

6.5 Results

First we present results where each component is cached separately, then present results on our single heterogeneous cache which contains all data elements.

6.5.1 Individual Component Caching

Figure 6.2 shows the predicted disk access time savings when caching different query time components with an LRU replacement policy. In all runs a maximum of 5 pages of re-
Figure 6.2: These plots show the savings in time achieved by caching different query time components with a LRU replacement policy for (a) the Anon collection and (b) the WT10g collection.
results were allowed per query in the cache. This value is based on experiments presented below.

For both test collections vocabulary caching provides a small saving. Note that, for the Anon collection after 512 KB of cache, and in the WT10g collection after 4 MB of cache, no further savings in disk access are achieved. For the vocabulary component, the upper limit can be attributed to the minimal space requirements of vocabulary entries within a cache, and the variance in when the limit is reached is due to the greater diversity of query terms in the Excite query log used for the WT10g collection.

As for caching of a vocabulary entry, caching the top 100 accumulators for a query requires a relatively small amount of memory. In our experiments, an upper limit for accumulator caching was reached at 64 MB and 256 MB for the Anon and WT10g collections respectively. Again, the diversity of the Excite query log accounts for the larger amount of cache required by the WT10g run. It should be noted that, for small cache sizes, accumulator caching can produce the largest savings, as shown in Figure 6.2 for the Anon collection between 8 KB and 64 KB caches, and for the WT10g collection for caches between 1 MB and 64 MB.

Caching of inverted lists is only effective with a substantial amount of cache space. For both collections, we see continued improvement with increased cache size. For the Anon collection, with a cache size of 256 MB, all requested query term lists can populate the cache with no removal required. In this run of the 955,476 lists requested, 97.8% are found in the cache, yet a saving of only 8.7% in disk access during query evaluation is achieved. This suggests that, even though the bulk of data read from disk at query time is due to inverted lists, the benefit of inverted list caching is limited by the small amount of disk seeking involved.

Note that, for small caches, a new list for a common term may require the cache to be almost completely flushed. This results in the poor performance of inverted list caching observed at cache sizes smaller than 64 MB. A possible solution to this would be to use stopping of common query terms, thus eliminating the need for long inverted lists.

Result pages produced the highest savings for the caching of an individual query-time component. This is consistent with previous work on search engine caching research in recent years, as discussed in Section 6.2.

For the Anon collection, result page caching is exceptionally effective. This is due to the skew of queries in the log, where the 50 most frequent queries account for 15% of the query log, as opposed to the Excite log where the 50 most frequent queries account for less than 2% of the query log.
Figure 6.3: Comparison of time savings when caching multiple data types with different cache replacement policies.

For both collections, the performance of result page caching improves as the cache size increases, with the cache fitting all query result pages in the log by 256 MB for the Anon collection, saving 82% of the query time disk access. Our largest cache for the WT10g collection of 2 GB was not sufficient to cache all requested result pages. However, a saving of 23% is observed, which is a large improvement over the 14% observed when caching accumulators only.

For the WT10g collection, result page caching outperforms all other forms of caching after the cache grows beyond 128 MB in size, and in the Anon collection after 64 KB.

6.5.2 Combined Caches

We have seen that caching the individual structures that are accessed from disk at query time can reduce disk costs. We next combine these individual components into a single cache and present savings using different cache management approaches.

Figure 6.3 shows the time saved using a single cache that is approximately 10% of the index size under different caching strategies. For the Anon index we used a cache size of 32 MB, and for the WT10g collection we used 256 MB. For each collection, the figure shows
the performance of a no-replacement policy (static-all), where, once full, no new items are added to the cache; a random replacement policy (random-all), where items are randomly selected from the cache to make space for new items; a global LRU replacement policy (lru-all); and the CPB approach outlined in Section 6.3 (cpb-all). Finally, for comparison, the results of an LRU cache of result pages only from Figure 6.2 is presented (lru-results).

We can see that for both collections the no-replacement, random, and LRU cache policies all perform poorly. In particular, the random policy is comparable to the LRU policy for the WT10g collection, and is noticeably better for the Anon collection, showing that a simple LRU policy is not capable of managing such a complex cache. Also worth noting is that the CPB approach is comparable to the result-only cache for the Anon collection, even though, as explained in Section 6.5.1, the collection and query log are highly skew. Further, for the WT10g collection our CPB approach provides a 75% improvement over result-only LRU caching. In other experiments not reported here, similar improvements were observed for other cache sizes.

Figure 6.3 also shows, for each approach, the breakdown in savings by component. We can see that, for WT10g, the CPB approach allows more savings to accrue through the use of the vocabulary and inverted list components than the other methods. As the Anon query access pattern is skewed, we see that more time savings are accrued through the caching of results sets on this collection. CPB has adapted the cache appropriately for each collection so that it is as good as a result-page-only approach on a skew access pattern, and, for a more typical pattern of query access, significantly better.

**Cost-per-byte and separate LRU cache**  Figure 6.3 shows that a global LRU policy is ineffective for a heterogeneous cache. We contrast the heterogeneous LRU approach with a segmented cache where four independent LRU cache components are maintained (one for each data type cached). Under this scheme, each component cache is allocated a fixed amount of space, and items are replaced when needed using an LRU policy.

However, we are unaware of any method that optimally partitions a fixed amount of memory amongst cache components. To overcome this difficulty we established a training set of 10,000 queries for an 8 MB cache, and ran all combinations of cache partitioning at 10% size intervals. Our exhaustive run resulted in the best performance occurring with partitioning of the cache into 10% for the vocabulary entries, 40% for the inverted lists, 10% for the accumulator sets, and 40% for the result pages for the Anon collection; and a partitioning of 10% for the vocabulary entries, 20% for the inverted lists, 20% for the
accumulator sets, and 50% for the result pages for the WT10g collection. The training process to establish cache partitions ran for several hours. It is far from clear that such brute force training is feasible in practice for larger-scale systems.

Figure 6.4 compares our CPB policy with separate LRU, and also shows LRU with result-page-only caching. The results show that the separate cache is potentially more efficient than CPB and generally better than LRU result-page-only caching, with the exception of large cache sizes with the Anon collection. For the Anon collection, separate LRU outperforms CPB by up to 9% for cache sizes from 4 KB to 4 MB, and performs well at 8 MB. For the WT10g collection, separate LRU outperforms CPB from cache sizes of 16 KB to 128 MB, however performance between the two is almost indistinguishable from cache sizes above 64 MB. It is possible that the performance of separate LRU can improve at greater cache sizes with the adequate tuning of cache proportions, however it is not clear how these proportions can reasonably be obtained.

The differences between CPB and separate LRU shows that gains are achievable with effective cache management of more than just result pages.

**Scaling result page costs** The effectiveness of the CPB approach is dependent on the balance of cost values between the various components in the cache. To test the sensitivity of our estimates we adjusted the CPB of result pages by up to a factor of 100. Lowering the CPB of result pages increases the likelihood that a result page be evicted from the cache; by increasing the CPB, result pages remain in the cache longer.

Figure 6.5 shows the predicted savings with adjustments to the CPB of result pages. For both collections, reducing the CPB of result pages results in a decrease in performance. As the scaling factor increases, for the Anon collection we can see an improvement in query evaluation time. This is not surprising, given the performance of result-page-only caching on this data set, as this adjustment likens the CPB policy to the LRU result-page-only approach. Indeed, when increasing the CPB by a factor of ten, the number of result pages evicted from the cache reduces by over 14%.

For the WT10g collection, we see a slight improvement with a doubling of the costs, and then a slight decrease in effectiveness beyond that. This result shows the importance of the balance in the CPB policy, and that, if result pages are over-valued, then the effectiveness of the cache will decline.
Figure 6.4: Comparison of CPB and LRU separate with proportions of cache allocated to different structures based on 10,000 training queries and an 8 MB cache.
Figure 6.5: Effects of modifying the CPB value of result pages for the Anon and WT10g collections. The x-axis shows the factor by which result page cost is scaled, and the y-axis shows estimated time savings.

**Result pages per query to cache** The effect of varying the number of pages stored in the cache is shown in Figure 6.6. Due to the sliding window approach adopted to simulate sessions, only values for up to the fifth page of results are presented for the Anon collection.

The results vary. For the Anon collection it is clear that caching up to the fifth page of results provides a substantial decrease in query costs. However, it is not clear to what degree the simulated session approach has on these results, and to what degree the results are due to the query log itself.

For the WT10g collection there is no clear improvement when caching a greater number of pages, with few runs showing improvement for 2 to 5 pages of results cached. This result is in agreement with that of Jansen and Spink [2005], which suggests that few users make use of results beyond the first presented page. Based on these results we allow up to the fifth page of results throughout this chapter.
Figure 6.6: The effect of varying the number of result pages cached per query on an LRU result-page-only cache.
Table 6.3: Number of cache hits as a percentage of items requested and found in the cache by item type (for Excite runs).

<table>
<thead>
<tr>
<th>Cache scheme</th>
<th>Cache size</th>
<th>Voc. entries found (%)</th>
<th>Inv. lists found (%)</th>
<th>Accum. sets found (%)</th>
<th>Result pages found (%)</th>
<th>Total time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No caching n/a</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>55,023</td>
</tr>
<tr>
<td>vocab-only lru</td>
<td>32 MB</td>
<td>94.71</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>50,460</td>
</tr>
<tr>
<td>invlist-only lru</td>
<td>32 MB</td>
<td>-</td>
<td>4.91</td>
<td>-</td>
<td>-</td>
<td>53,812</td>
</tr>
<tr>
<td>accum-only lru</td>
<td>32 MB</td>
<td>-</td>
<td>-</td>
<td>57.41</td>
<td>-</td>
<td>47,807</td>
</tr>
<tr>
<td>resset-only lru</td>
<td>32 MB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9.41</td>
<td>50,467</td>
</tr>
<tr>
<td>all-caches lru</td>
<td>32 MB</td>
<td>3.65</td>
<td>3.81</td>
<td>1.19</td>
<td>0.05</td>
<td>53,566</td>
</tr>
<tr>
<td>all-caches sep. lru</td>
<td>32 MB</td>
<td>90.85</td>
<td>2.53</td>
<td>40.51</td>
<td>6.71</td>
<td>43,725</td>
</tr>
<tr>
<td>all-caches cpb</td>
<td>32 MB</td>
<td>67.53</td>
<td>10.88</td>
<td>32.57</td>
<td>3.21</td>
<td>46,134</td>
</tr>
<tr>
<td>vocab-only lru</td>
<td>256 MB</td>
<td>94.71</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>50,460</td>
</tr>
<tr>
<td>invlist-only lru</td>
<td>256 MB</td>
<td>-</td>
<td>37.42</td>
<td>-</td>
<td>-</td>
<td>49,278</td>
</tr>
<tr>
<td>accum-only lru</td>
<td>256 MB</td>
<td>-</td>
<td>-</td>
<td>62.86</td>
<td>-</td>
<td>47,290</td>
</tr>
<tr>
<td>resset-only lru</td>
<td>256 MB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>20.67</td>
<td>44,998</td>
</tr>
<tr>
<td>all-caches lru</td>
<td>256 MB</td>
<td>25.65</td>
<td>25.67</td>
<td>16.50</td>
<td>1.11</td>
<td>46,978</td>
</tr>
<tr>
<td>all-caches sep. lru</td>
<td>256 MB</td>
<td>89.20</td>
<td>7.64</td>
<td>46.96</td>
<td>16.62</td>
<td>37,724</td>
</tr>
<tr>
<td>all-caches cpb</td>
<td>256 MB</td>
<td>87.85</td>
<td>34.32</td>
<td>45.95</td>
<td>15.72</td>
<td>37,476</td>
</tr>
<tr>
<td>vocab-only lru</td>
<td>1 GB</td>
<td>94.71</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>50,460</td>
</tr>
<tr>
<td>invlist-only lru</td>
<td>1 GB</td>
<td>-</td>
<td>86.21</td>
<td>-</td>
<td>-</td>
<td>46,219</td>
</tr>
<tr>
<td>accum-only lru</td>
<td>1 GB</td>
<td>-</td>
<td>-</td>
<td>62.86</td>
<td>-</td>
<td>47,290</td>
</tr>
<tr>
<td>resset-only lru</td>
<td>1 GB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>26.56</td>
<td>42,079</td>
</tr>
<tr>
<td>all-caches lru</td>
<td>1 GB</td>
<td>71.77</td>
<td>71.77</td>
<td>44.19</td>
<td>10.67</td>
<td>37,847</td>
</tr>
<tr>
<td>all-caches sep. lru</td>
<td>1 GB</td>
<td>88.20</td>
<td>32.73</td>
<td>49.00</td>
<td>24.34</td>
<td>32,820</td>
</tr>
<tr>
<td>all-caches cpb</td>
<td>1 GB</td>
<td>89.03</td>
<td>80.34</td>
<td>46.85</td>
<td>18.62</td>
<td>33,964</td>
</tr>
<tr>
<td>vocab-only lru</td>
<td>2 GB</td>
<td>94.71</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>50,460</td>
</tr>
<tr>
<td>invlist-only lru</td>
<td>2 GB</td>
<td>-</td>
<td>94.71</td>
<td>-</td>
<td>-</td>
<td>45,785</td>
</tr>
<tr>
<td>accum-only lru</td>
<td>2 GB</td>
<td>-</td>
<td>-</td>
<td>62.86</td>
<td>-</td>
<td>47,290</td>
</tr>
<tr>
<td>resset-only lru</td>
<td>2 GB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>26.68</td>
<td>42,020</td>
</tr>
<tr>
<td>all-caches lru</td>
<td>2 GB</td>
<td>86.54</td>
<td>86.54</td>
<td>48.10</td>
<td>23.86</td>
<td>31,592</td>
</tr>
<tr>
<td>all-caches sep. lru</td>
<td>2 GB</td>
<td>87.90</td>
<td>54.99</td>
<td>49.43</td>
<td>26.56</td>
<td>31,203</td>
</tr>
<tr>
<td>all-caches cpb</td>
<td>2 GB</td>
<td>88.14</td>
<td>88.14</td>
<td>50.71</td>
<td>22.36</td>
<td>32,119</td>
</tr>
</tbody>
</table>
Hit ratios  Table 6.3 shows the fraction of items that are found in the cache when requested for various runs over differing cache sizes. The table also shows the total time spent accessing disk. All values are based on runs of the Excite query log with the WT10g collection.

The table illustrates, for the component-only caches, the growth in hit rates for items requested as the cache grows. By a cache size of 2 GB, almost no gains are possible with increased cache size as all items for a single data type fit into cache memory (with the exception of result pages for the WT10g collection).

For the schemes that combine all data types, we can see the difference in hit rate between the various data types. At all cache sizes, the CPB approach closely matches or outperforms the simple LRU approach hit rate for all data types. For the separate LRU scheme, the results are closely matched, but we see that the CPB approach achieves significantly more hits with inverted lists. This can be attributed to the costing-based replacement policy, where shorter inverted lists are likely to stay in the cache for a greater duration than longer lists.

By a cache size of 2 GB, most requested items are available in the cache. This is illustrated by the similarity in performance of the various schemes that cache multiple data types.

6.6  Summary

We have explored several aspects of gaining efficiency in query evaluation through caching. Our conclusion is that ad-hoc approaches based on separate caches are not an ideal solution, and that some caching of all of the structures used during query evaluation can be of value. Underpinning this work, we have presented a model for determining the savings in disk access to be gained from caching. This model is adaptable to different search systems and underlying hardware, and provides a framework within which different caching policies can be fairly compared.

We have explored the benefits of separate caching of the different query evaluation time components. We found that, for larger cache sizes, result-page caching works best, but that caching of accumulator sets and vocabulary entries can also provide effective caching if cache space is limited.

Our main contribution is a new heterogeneous cache management policy, used to simultaneously manage the different query time components of query evaluation. Our cost-per-byte policy is based on the principle of maintaining the items within the cache that are most expensive to recreate, and is capable of dealing with any data structure provided that its cost model can be defined.
We found that caching of multiple data types provides significant savings over conventional approaches where only result pages or inverted lists are cached. A naive approach of having separate caches for each type creates the issue of tuning the separate cache sizes, a difficult problem with no obvious solution in a dynamic environment. In contrast, our principled approach based on a single heterogeneous cache adapts to data and queries with no intervention, and can lead to dramatic reductions in query-time costs.

The results presented in this chapter focus on single server search systems and are applicable to enterprise scale search. However, the model is adaptable for large scale search systems, where the index components are distributed throughout a network. We leave the adaption of the model for such distributed architectures as a topic for further study.
Chapter 7

Access Counts and Query Accuracy

In previous chapters we have explored the skew nature of document access by a search engine for the purposes of efficiency. In this chapter we extend the use of access counts and explore the applicability of this information to improve the effectiveness of search results, by giving preference to documents with a higher access score. Further, we examine the use of document access counts as predictors of system accuracy.

7.1 Access Counts as Query-Independent Evidence

In previous chapters we have utilised access counts to reorganise inverted lists with the goal of reduced query evaluation time. Our access-order based indexes depend on the ability of either dynamic (Chapters 3 and 4) or static (Chapter 5) pruning schemes to heuristically terminate the processing of inverted lists, while maintaining an acceptably high level of accuracy. In an access-ordered index, regardless of the pruning strategy employed, query time processing favours those documents with a high access count. The rationale is that those documents that appear in the results most frequently have a higher probability of doing so again.

We have shown that access-ordering with pruning produces results that are comparable in accuracy to that of full list evaluation. A reason for these favourable results is the apparent correlation between access-ordering and relevance, as shown in Figure 3.2 (page 59). Therefore, an alternative view, of access counts, is that they provide a measure of the likelihood of a document being relevant to a query.

The relationship between access counts and relevance raises several interesting questions. First, to what extent do access counts reflect a document’s likelihood of relevance, and can
such a relationship be quantified? Second, given such a relationship, can this likelihood be exploited to improve the accuracy of search results?

The utilisation of *query-independent evidence* in information retrieval is not new. Lafferty and Zhai [2001] speculate that document bias based on link evidence or high level document collection knowledge may result in improvements to language model based metrics. Simple features such as document length have been used to improve ranking for web search [Singhal et al., 1996b], while more specific features such as URL length, PageRank, and click distance have been used for specialised retrieval tasks [Upstill and Robertson, 2004; Craswell et al., 2005b].

In practice, it has been difficult to improve accuracy for ad-hoc querying using query-independent evidence. For example, although Westerveld et al. [2001] found that document length bias improved accuracy for some language modelling variants, similar gains were not possible across different language model smoothing functions. Further, Upstill et al. [2003] conclude that due to the limited size of research test collections, document bias based on link evidence is unlikely to result in significant accuracy improvements. Section 2.13 (page 53) outlines applications of query-independent evidence to ranking functions with the goal of improved accuracy.

In this chapter, we propose utilising access counts as a form of query-independent evidence for the purpose of improved query accuracy. In Section 7.1.1 we analyse the predictive power of this form of evidence and suggest techniques to adapt the feature into the query evaluation process.

### 7.1.1 Rationale

As we observed in Chapter 3, those documents that are tagged as relevant to TREC topics are often also those that have a high frequency of document access. To illustrate the relationship between access counts and document relevance we partitioned the WT10g collection into one hundred blocks ordered by decreasing access count, and counted the number of known relevant documents in each block. Relevant documents are those that have been tagged as relevant for any of the TREC-9 topics 451–500. Figure 7.1 shows the distribution of relevant documents within the collection.

The figure shows a clear trend that favours relevant documents early in the ordering, while few occurrences of relevant documents can be found towards the end of the ordering. The distribution is indeed skew. A document is 25 times more likely to be relevant if it
7.1. ACCESS COUNTS AS QUERY-INDEPENDENT EVIDENCE

It could be argued that as TREC judgements are based on the pooled submissions of participants, the set of marked relevant documents for a topic set, could be biased towards the documents retrieved by the ranking functions typically employed by TREC participants. Such bias would suggest that the skew in Figure 7.1 is artificial. However, in analysing the TREC evaluation process, Zobel [1998] shows that the judgments obtained through pooling are reliable with 50%-70% coverage of the relevant documents found, where most of the relevant documents not identified belong to those topics that have a high proportion of relevant documents in the collection.

7.1.2 Methods of Incorporating Access Counts into the Ranking Function

Various techniques have been proposed to combine external evidence with the results of a similarity ranking function. Language models explicitly allow the inclusion of a document...
prior. In language modelling, documents are ranked by the likelihood that they were generated by the same underlying language model that produced query. This likelihood, \( P(d|q) \), can be rearranged through Bayesian inversion into:

\[
P(d|q) = \frac{P(q|d)P(d)}{P(q)},
\]

where \( P(q|d) \) is the likelihood of observing the query given the language model of the document. The value of \( P(q) \) is the likelihood of observing the query, but is ignored in query evaluation as its inclusion does not influence the ranking of the documents. The value of \( P(d) \) is the prior probability of the document. A key problem in effectively utilising the document prior is determining an appropriate value for each document. As such, \( P(d) \) is typically set to a constant value, thus assuming a uniform prior probability.

In their work on entry page search, Westerveld et al. [2001] propose the use of linear interpolation to combine external evidence with document ranking metrics. Using this approach, query-independent evidence is incorporated into the ranking function as:

\[
S'_{q,d} = \alpha S_{q,d} + (1 - \alpha)E_d,
\]

where \( S_{q,d} \) is the score assigned to document \( d \) for query \( q \) using metric \( S \), the value \( E_d \) is the external evidence of document \( d \), and \( \alpha \) is a tuning parameter used to balance the effect of the query-independent evidence.

The two approaches described above depend on careful selection of a document prior distribution and \( \alpha \) parameter to successfully combine the external evidence with the selected ranking function. To avoid the problems associated with such combinations, Upstill et al. [2003] propose filtering the documents returned in response to a query by document evidence values. Such an approach has the benefit of selecting those documents that are estimated to be most similar to the query, and further refining this set by retaining those that have a strong query-independent evidence.

Upstill et al. show that their approach produces favourable results for home-page finding. Part of their success is due to the fact that their document evidence features and thresholds significantly reduce the number of documents considered at query time, while retaining a high proportion of home-pages. Although such an effect is desirable in a home-page task, it is unlikely to benefit ad-hoc search. In Chapter 5 we explored access-pruning where we retain only those postings that have an access count greater than a specified threshold in the index. Such an approach is similar to that of Upstill et al. as documents are filtered from the collection, with the exception that in our approach they are removed at indexing.
time instead of query time, and that the decision is based on access count evidence. In a less restrictive approach, Savoy and Rasolofo [2001] re-rank the documents returned for each query by document prior value.

In this chapter, we make use of access counts to influence the ranking function. As we have seen in Section 3.1 (page 58), access counts follow a power-law distribution, where most documents in the collection have an extremely low access count, while very few documents are allocated significantly high values. Given such a distribution, it is likely that document priors based on this property will require a non-linear transformation to be combined with the similarity metric.

To explore the application of access counts as query-independent evidence, we propose to test its effect on accuracy in three ways. First, we combine access counts with the result of a similarity metric using linear interpolation as suggested by Westerveld et al. [2001]. Such an approach can be tested on both the Okapi BM25 and language model ranking functions. As these ranking functions return values of differing magnitudes, similarity scores must be normalised for each query over the result set to a range of 0 to 1. Second, we incorporate access count priors directly into a Dirichlet language model, substituting $\Pr(d|a)$ with our access count based prior. Finally, we explore the approach of Savoy and Rasolofo and re-rank the top documents returned for each query by descending access count.

A factor of significant importance in this exploration is the assignment of document evidence values. We propose the following methods to assign documents weights:

- **NORM-AC**: A simple approach to incorporating access counts into the similarity metric is to add them directly to the estimated similarity of a document. However, it is necessary to normalise such values as their magnitude is directly proportional to the number of queries used in the training phase. As such, we propose using a normalised access count based on largest observed value where:

  $$ P(d|a) = \frac{a_d}{A}, $$

  where $a_d$ is the access count of document $d$, and $A$ is the largest observed access count.

- **LOG-AC**: As the distribution of access counts is highly skew, the impact of a single document may be overstated. Taking the logarithm of the access counts reduces the over-rating of highly accessed documents, while relatively increasing the impact of those documents that are accessed less frequently:

  $$ P(d|a) = \frac{\log a_d}{\log A}. $$
• **RANK-AC**: As the distribution of access counts is skew towards the most accessed documents, it is possible that those documents with median ranks will be under-valued. As such, we propose weighting the document by its ranking in the access count ordering:

\[
P(d|a) = 1 - \frac{r_{d,a}}{N},
\]

where \(r_{d,a}\) is the access count based rank of the document in the collection, and \(N\) is the size of the collection.

### 7.1.3 Experimental Environment

To examine the degree of predictive power that access count based priors have, we worked with TREC-9 topics 451–500 and the WT10g collection. The title field of each topic is used as a query. Details of the WT10g collection and related topics are in Section 2.2.4 (page 18). For each document, access counts were calculated using the fixed counting scheme considering the top 1,000 results per query using the Excite’97 and ’99 query logs. This constitutes approximately 2 million queries.

Query accuracy is reported using the standard measure of mean average precision (MAP). Further, given the web oriented nature of this task, we report precision values at 10 and 100 returned documents to represent those users who only consider the first few pages of ranked results. Significance testing was conducted using the Wilcoxon signed rank test [Sheskin, 1997], and we report p-values for each significance test performed. Confidence values above 95%, that is p-values below 0.05, are considered significant. Finally, document-to-query similarity is estimated using either Dirichlet language modelling or Okapi BM25 as implemented in the open source Zettair search engine [Zettair]. Each is distinguished in the presented results.

### 7.1.4 Results

Table 7.1 presents the results of NORM-AC weights when combined with the Okapi BM25 metric and a Dirichlet smoothed language model using linear interpolation. The results show a significant deterioration in accuracy over all examined \(\alpha\) parameters. Indeed, the impact of the NORM-AC weight on document similarity scores is so great, that even with \(\alpha\) parameters that highly favour the original values, significant deterioration is observed.

The negative outcome of this approach is related to the distribution of access counts. As few documents are allocated high access count values, normalisation of the counts by the
7.1. ACCESS COUNTS AS QUERY-INDEPENDENT EVIDENCE

Table 7.1: Effect of NORM-AC weights with linear interpolation on accuracy for TREC topics 451–500 and WT10g collection. Significance test p-values reported in brackets below each accuracy measure.

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>Okapi BM25</th>
<th></th>
<th></th>
<th>Dirichlet</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MAP</td>
<td>P@10</td>
<td>P@100</td>
<td>MAP</td>
<td>P@10</td>
<td>P@100</td>
</tr>
<tr>
<td>0.1</td>
<td>0.0243</td>
<td>0.0396</td>
<td>0.0269</td>
<td>0.0289</td>
<td>0.0479</td>
<td>0.0294</td>
</tr>
<tr>
<td></td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
</tr>
<tr>
<td>0.3</td>
<td>0.0271</td>
<td>0.0458</td>
<td>0.0281</td>
<td>0.0317</td>
<td>0.0521</td>
<td>0.0302</td>
</tr>
<tr>
<td></td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
</tr>
<tr>
<td>0.5</td>
<td>0.0311</td>
<td>0.0687</td>
<td>0.0298</td>
<td>0.0347</td>
<td>0.0771</td>
<td>0.0321</td>
</tr>
<tr>
<td></td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
</tr>
<tr>
<td>0.7</td>
<td>0.0446</td>
<td>0.1021</td>
<td>0.0325</td>
<td>0.0448</td>
<td>0.1062</td>
<td>0.0365</td>
</tr>
<tr>
<td></td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
</tr>
<tr>
<td>0.9</td>
<td>0.0878</td>
<td>0.1396</td>
<td>0.0462</td>
<td>0.0834</td>
<td>0.1521</td>
<td>0.0521</td>
</tr>
<tr>
<td></td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
</tr>
<tr>
<td>0.99</td>
<td>0.1617</td>
<td>0.2083</td>
<td>0.0975</td>
<td>0.1907</td>
<td>0.2354</td>
<td>0.1117</td>
</tr>
<tr>
<td></td>
<td>(0.002)</td>
<td>(0.131)</td>
<td>(0.044)</td>
<td>(&lt;0.001)</td>
<td>(0.059)</td>
<td>(0.007)</td>
</tr>
<tr>
<td>n/a</td>
<td>0.1827</td>
<td>0.2396</td>
<td>0.1023</td>
<td>0.2096</td>
<td>0.2562</td>
<td>0.1204</td>
</tr>
</tbody>
</table>

highest observed value results in an almost insignificant contribution for most documents. Conversely, the few documents that are assigned a high score receive disproportionately large boosts to their estimated similarity.

Results for LOG-AC weights are presented in Table 7.2. The logarithmic normalisation of the access count weights both reduces the range of values allocated, and increases the contributions for mid and low scoring documents. As such, the linearly interpolated score of those documents that are highly penalised by the NORM-AC prior are less affected.

As with the other schemes, assigning a high \( \alpha \) value to the LOG-AC component in the interpolation results in a significantly negative impact on accuracy. However, unlike the NORM-AC prior, for some values an improvement is be observed. Unfortunately, where improvements are observed, they are either statistically insignificant, or significant, but without a large absolute improvement. Further, while improvements were observed for the Okapi BM25 metric, no such improvements were observed for the Dirichlet metric, for which the baseline accuracy outperformed most of the improved Okapi BM25 results.
Table 7.2: Effect of LOG-AC weights with linear interpolation on accuracy for TREC topics 451-500 and WT10g collection. Significance test p-values reported in brackets below each accuracy measure.

<table>
<thead>
<tr>
<th>α</th>
<th>Okapi BM25</th>
<th>Dirichlet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MAP</td>
<td>P@10</td>
</tr>
<tr>
<td>0.1</td>
<td>0.1827</td>
<td>0.2396</td>
</tr>
<tr>
<td>0.2</td>
<td>0.1798</td>
<td>0.2396</td>
</tr>
<tr>
<td>0.3</td>
<td>0.1829</td>
<td>0.2687</td>
</tr>
<tr>
<td>0.4</td>
<td>0.1858</td>
<td>0.2604</td>
</tr>
<tr>
<td>0.5</td>
<td>0.1865</td>
<td>0.2562</td>
</tr>
<tr>
<td>0.6</td>
<td>0.1858</td>
<td>0.2458</td>
</tr>
<tr>
<td>0.7</td>
<td>0.1889</td>
<td>0.2500</td>
</tr>
<tr>
<td>0.8</td>
<td>0.1868</td>
<td>0.2500</td>
</tr>
</tbody>
</table>

Table 7.3 shows the effect of the RANK-AC weights on accuracy when combined to document scores using linear interpolation. As this variation only utilises the access count to establish its position in a global ordering of documents, the distribution of assigned weights is uniform. The effect of such a distribution is that while highly accessed documents retain a high document weight, the decay in the value of weights assigned as document accesses reduce is linear with the size of the collection, and thus even those documents that are accessed rarely, are likely to have a weight that can impact the similarity function scores.

As with the LOG-AC weights, the effect of RANK-AC is negative for high values of α where too much emphasis is placed on the access-order component of the equation. For
values of $\alpha$ above 0.8 the effect is not statistically worse for either ranking function, and as with LOG-AC, a significant improvement in MAP is observed when using the Okapi BM25 metric. Further a significant improvement in precision at 10 is also observed over the baseline Okapi metric for $\alpha = 0.8$. However, as with the other techniques, this improvement is not significant when compared to the baseline Dirichlet measure.

Finally Table 7.4 presents the results of language model prior combination with our access count based prior — that is, a direct multiplication of our prior with the result of the language model metric measure — and re-ranking the top ranked documents based on access count.

### Table 7.3: Effect of RANK-AC weights with linear interpolation on accuracy for TREC topics 451–500 and WT10g collection. Significance test p-values reported in brackets below each accuracy measure.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>Okapi BM25</th>
<th>Dirichlet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MAP P@10 P@100</td>
<td>MAP P@10 P@100</td>
</tr>
<tr>
<td>0.1</td>
<td>0.0824 0.1354 0.0573</td>
<td>0.0869 0.1396 0.0612</td>
</tr>
<tr>
<td></td>
<td>(&lt;0.001) (&lt;0.001) (&lt;0.001)</td>
<td>(&lt;0.001) (&lt;0.001) (&lt;0.001)</td>
</tr>
<tr>
<td>0.2</td>
<td>0.1213 0.1958 0.0769</td>
<td>0.1194 0.1937 0.0796</td>
</tr>
<tr>
<td></td>
<td>(0.002) (0.026) (0.013)</td>
<td>(&lt;0.001) (&lt;0.001) (&lt;0.001)</td>
</tr>
<tr>
<td>0.3</td>
<td>0.1567 0.2313 0.0883</td>
<td>0.1559 0.2167 0.0927</td>
</tr>
<tr>
<td></td>
<td>(0.031) (0.430) (0.140)</td>
<td>(&lt;0.001) (0.001) (&lt;0.001)</td>
</tr>
<tr>
<td>0.4</td>
<td>0.1709 0.2479 0.0962</td>
<td>0.1737 0.2354 0.1019</td>
</tr>
<tr>
<td></td>
<td>(0.258) (0.834) (0.535)</td>
<td>(&lt;0.001) (0.029) (0.002)</td>
</tr>
<tr>
<td>0.5</td>
<td>0.1827 0.2542 0.1010</td>
<td>0.1913 0.2625 0.1083</td>
</tr>
<tr>
<td></td>
<td>(0.897) (0.555) (0.550)</td>
<td>(0.008) (0.667) (0.012)</td>
</tr>
<tr>
<td>0.6</td>
<td>0.1870 0.2729 0.1050</td>
<td>0.1955 0.2562 0.1100</td>
</tr>
<tr>
<td></td>
<td>(0.139) (0.147) (0.373)</td>
<td>(0.010) (0.535) (0.018)</td>
</tr>
<tr>
<td>0.7</td>
<td>0.1885 0.2750 0.1042</td>
<td>0.2033 0.2687 0.1148</td>
</tr>
<tr>
<td></td>
<td>(0.030) (0.067) (0.466)</td>
<td>(0.034) (0.222) (0.129)</td>
</tr>
<tr>
<td>0.8</td>
<td>0.1889 0.2646 0.1046</td>
<td>0.2055 0.2604 0.1175</td>
</tr>
<tr>
<td></td>
<td>(0.006) (0.023) (0.238)</td>
<td>(0.085) (0.802) (0.144)</td>
</tr>
<tr>
<td>0.9</td>
<td>0.1894 0.2563 0.1035</td>
<td>0.2078 0.2625 0.1181</td>
</tr>
<tr>
<td></td>
<td>(&lt;0.001) (0.093) (0.568)</td>
<td>(0.211) (0.465) (0.010)</td>
</tr>
</tbody>
</table>

$n/a$ 0.1827 0.2396 0.1023 0.2096 0.2562 0.1204
Table 7.4: Effect of language model priors and access count based re-ranking on accuracy for TREC topics 451–500 and WT10g collection. Significance test p-values reported in brackets below each accuracy measure.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Okapi BM25</th>
<th>Dirichlet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MAP</td>
<td>P@10</td>
</tr>
<tr>
<td>NORM-AC</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG-AC</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANK-AC</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RERANK-10</td>
<td>0.1526</td>
<td>0.2396</td>
</tr>
<tr>
<td></td>
<td>(0.197)</td>
<td>(0.317)</td>
</tr>
<tr>
<td>RERANK-100</td>
<td>0.0849</td>
<td>0.1312</td>
</tr>
<tr>
<td></td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
</tr>
<tr>
<td>RERANK-1000</td>
<td>0.0419</td>
<td>0.0396</td>
</tr>
<tr>
<td></td>
<td>(&lt;0.001)</td>
<td>(&lt;0.001)</td>
</tr>
<tr>
<td>Baseline</td>
<td>0.1827</td>
<td>0.2396</td>
</tr>
</tbody>
</table>

While no approach results in improved accuracy, it is interesting to note that, as with linear interpolation, the effect of a prior based on NORM-AC is significantly worse than priors based on LOG-AC and RANK-AC. This suggests that the power-law distribution of access counts is not suitable for incorporation into the similarity function without normalisation of some kind.

Further, the performance of the re-ranking schemes clearly show that access counts alone cannot be used to rank documents with respect to a query. Even when applying re-ranking to only the top 10 documents per query, the impact of the query-independent reordering has a significantly negative effect on accuracy.

7.1.5 Analysis

Our results show limited increase in accuracy over the Okapi BM25 ranking function with the RANK-AC and LOG-AC priors. Although the distribution of known relevant documents in the collection is promising when ordered by access count, the likelihood of encountering a
relevant document appears to be too low to justify boosting the ranking of those documents that have a high access count. While our results in Chapters 3, 4, and 5 show that it is effective to prune lists based on access count, the benefits of manipulating rankings by this property are unclear.

While the results show some improvement in accuracy using the Okapi BM25 ranking function, the improvements are not encouraging, and often less effective than the baseline values using the Dirichlet language modelling ranking function. Further, no significant improvements over the Dirichlet language modelling ranking function were observed.

Although Singhal et al. [1996b] note an improvement in ad-hoc style queries with their document length pivoting, their improvement is over a cosine ranking function baseline. As the Dirichlet smoothed language model and Okapi BM25 ranking functions implicitly bias towards longer documents, such an improvement is unlikely to be achievable with these metrics. This view is supported by Westerveld et al. [2001], who find a Dirichlet smoothed language model produces results of higher accuracy than that of a Jelinek-Mercer smoothed language model with explicit use of a document length prior. In results not reported here, we experimented with document length evidence. Our results found that document length ordering produces a distribution of relevant documents that is similar to that of the access count based distribution in Figure 7.1. However, application of priors based on this evidence lead to mixed results akin to those of the access count based priors reported above.

Craswell et al. [2005b] suggest that a problem with incorporating query-independent evidence in similarity metrics is that their is redundancy between those documents that are retrieved by the similarity metric, and those documents ranked highly by the query-independent evidence. They propose techniques to reduce such redundancy that may be applicable here. While we have not explored this option, it is likely that access counts reflect the bias of the ranking function. As such, it is highly likely that some overlap occurs between those documents selected by the ranking function and those documents favoured by access count.

Finally, the results presented here test the application of access count based priors on a single test collection. In results not presented we examined the effect of these priors on the TREC Aquaint collection with results reflecting performance reported above. Due to unsuccessful application of access counts as document priors we did not pursue this work further.
7.2 Access Counting for Difficulty Prediction

A query is considered difficult when the accuracy of the search results are poor. Query difficulty prediction is the task of determining, without user feedback, how likely a retrieval system is to satisfy a user query. As discussed in Section 2.12 (page 46), predictions allow for the possibility of improved search results by providing retrieval systems the opportunity to determine how a query should be treated. For example, prediction of a difficult query may suggest that the query should be modified using techniques such as query expansion.

To date, the most successful work on difficulty prediction has focused on query features, combined with machine-learned weights [Kwok et al., 2004; 2005; Jensen et al., 2005; Kwok, 2005; Grivolla et al., 2005]; and language model divergence between the estimated query language model and the collection [Cronen-Townsend and Croft, 2002; Cronen-Townsend et al., 2002]. In this section, we examine the use of access counts as a form of evidence in determining query difficulty.

7.2.1 Determining Query Difficulty Using Access-Order

An access-ordering is based on the frequency of document occurrence in the search results. As we have seen (Figure 3.1 page 58), there is a non-uniform likelihood of access for the documents in a collection over a set of user queries. Using access counts, for each document we can establish a probability that indicates the likelihood of seeing that document in the top ranked results of a random future query. Given such a set of document probabilities, an absolute ordering of documents from most to least likely to be retrieved, indicates a default ranking for the documents in the collection. Taking this reasoning one step further, the ordering offers an approximation of the expected ranking of documents for any query independent of the query terms. That is, if a document appears frequently in the result sets, then it has a greater chance of being ranked highly.

Document ranking functions order documents with respect to the user query. As seen in Section 2.6 (page 27), for each document a score is obtained that is typically a combination of three primary components; the length of the document, the frequency of the query term in the document, and the uniqueness of the query term in the collection. Of these, the uniqueness of the query term in the collection, establishes the impact that the term will have relative to other terms in the query, while the frequency of the term in document, and document length, establish the impact the term will have on the score of that document.
Figure 7.2: Proposed query difficulty prediction approach. Query 1 produces a result set in which the documents are ranked in an order that does not differ greatly from the collection-wide access-order. Such a query is expected to be difficult to resolve. Conversely, query 2, produces a result set for which the ranking of the documents varies significantly from that of the access-order. Such a query has high discriminatory power and is predicted to be simple to resolve.
We speculate that, for a difficult query, the documents in the results set will be ranked in much the same order as those in the access-ordering. That is, the query terms will not have a strong enough impact on the document scores to significantly alter the ordering of documents from the access-order. Conversely, a query that produces a result set that significantly differs in order to the absolute access-ordering is considered to have high discriminatory power, and therefore is considered a simple query to resolve. Figure 7.2 illustrates the principle. In the figure, query 1 produces a result set for which the documents appear in a ranked order that closely resembles the collection-wide access-order. As such, it is assumed that such a query has a low influence on the overall ranking of documents, and therefore the query is predicted “difficult” to resolve. Query 2 produces a result set in which the ranking of the documents differs significantly from that of the collection-wide access-order. Such a query has a strong influence on the document ranking, and is therefore predicted “simple” to resolve.

Our proposed difficulty predictor is based on measuring the difference in the ordering of documents, between the ranked result set of a query, and the collection access-order. One technique to measure such distinction is Kendall’s \( \tau \) rank correlation co-efficient. Kendall’s \( \tau \) measures the agreement between two rankings by comparing the number of pairwise element changes required to convert one set of ranked data to the other [Sheskin, 1997]. The \( \tau \) value is reported between \(-1\) and \(+1\), where \(+1\) is perfect agreement between the two rankings, \(-1\) is perfect disagreement between the rankings, and \(0\) where the rankings are independent.

Using Kendall’s \( \tau \), we can measure the correlation between the collection access-order and the ranked results of each query. In doing so, we can evaluate the potential of access counts as a predictor of query difficulty. Should our hypothesis hold, low accuracy queries will be those with \( \tau \) correlations approaching \(+1\), while higher accuracy queries have \( \tau \) values closer to \(-1\).

Unlike other difficulty predictors, our proposed approach is novel for several reasons. While the idea of a default ranking of documents in the collection is not new [Page et al., 1999; Kleinberg, 1999], to our knowledge it has not previously been considered in query difficulty prediction. Although we propose utilising this technique based on the ordering obtained by access counts, it could potentially be applied to other forms of document priors. Further, other difficulty prediction approaches have sought external information to determine query difficulty. Specifically, Swen et al. [2004] have considered sources such as Wordnet to determine the ambiguity of a query term. This work presents an alternative means by which to measure the discriminatory power of an entire query with respect to the search engine.
7.2. ACCESS COUNTING FOR DIFFICULTY PREDICTION

Figure 7.3: Comparison of the access-order based predictor to the average precision of the TREC-9 topics on the WT10g collection. Each point represents an individual topic. The x-axis shows the Kendall’s $\tau$ correlation of the top 1,000 results returned for a topic, with the collection access-order. The y-axis shows the average precision of the topic.

7.2.2 Evaluation of Query Prediction Performance

To evaluate the potential for access counts as a predictor of query difficulty, we make use of the WT10g collection and the TREC-9 ad-hoc topics numbered 451–500. We also use the Gov collection and the named page topics NP1–150. For each topic, we retrieved the top 1,000 ranked results per query, using the Okapi BM25 ranking function. Using the ranked results accuracy was measured using average precision and precision at 10 for the ad-hoc topics, while reciprocal rank was used for the named page topics.

For the WT10g collection, an access-ordering was established by evaluating the Excite’97 and ’99 query logs. For the Gov collection, access counts were established using the MS-Gov query log. In both cases, access counts were calculated using the Static counting scheme, considering the top 1,000 results per query. Further details of the query logs can be found in Section 2.1.1 (page 10).
For each topic, a $\tau$ value was calculated by comparing the ranked order of the top 1,000 retrieved documents with that of the collection wide access-order. Figure 7.3 plots the average precision of each of the fifty TREC-9 topics in the WT10g collection against their respective $\tau$ value. The results do not support our hypothesis. Of the 30 topics awarded a $\tau$ correlation between $-0.1$ and $+0.1$, average precision varies from 0% to 85%. Further, of the 20 topics with the lowest average precision, 15 occur in this region. Were our hypothesis to hold, we would have expected higher accuracy in these low correlation $\tau$ regions of the graph. Figure 7.4 compares Kendall’s $\tau$ to P@10 over the TREC-9 topics on the WT10g collection. The figure shows a similar trend to that of average precision.

Along with the correlation between rankings, Kendall’s $\tau$ reports the significance of the correlation. Unreliable correlation estimates may skew the results. In results not reported here, we removed topics for which the confidence level less than 95%, but found no change in to trends reflected in the figures.
Figure 7.5: Comparison of the access-order based predictor to reciprocal rank for 150 named-paged topics on the Gov collection. Each point represents an individual topic. The $x$-axis shows the Kendall’s $\tau$ correlation of the top 1,000 results returned for a topic, with the collection access-order. The $y$-axis shows the reciprocal rank for the results of the topic.

Results for the Gov collection are shown in Figure 7.5. For these topics, the clustering of topics near low levels of correlation is greater than that in the WT10g collection. Of the 150 topics, 103 have a $\tau$ correlation between $-0.1$ and $+0.1$, with reciprocal ranks ranging from 0 to 1. Further, of the 25 topics with the lowest reciprocal rank, 21 occur within this range; while of the 67 topics with a reciprocal rank of 1, 49 also occur in this region. We conclude that access-ordering is not effective as a query difficulty predictor.

7.2.3 Analysis of Access-Order Based Prediction

The scattered distribution of accuracy over the topics suggests that access-ordering is not a useful indicator of query difficulty. We found that over both the ad-hoc and named page tasks, the collection-wide ordering had little correlation with the topic ranked results. At best, a low correlation of $-42\%$ was observed, while the majority of topics exhibited correlations
between $-10\%$ and $+10\%$. As such, differentiating between topics based on this criteria is difficult.

Kendall's $\tau$, used to measure the correlation between ranked results and the collection access-order, considers any variation in the ranked orderings equally. That is, differences in the rankings towards the end of the results set, are weighted equally to differences earlier in the rankings. A possible avenue for further exploration is to compare the order of the ranked results and access counts with higher weight assigned to those documents towards the start of the results. Such an approach reflects the importance of the high ranked documents, as they are the ones likely to be viewed by the search system user.

While our results show that access-ordering has limited use as a predictor, our model of prediction is conceptually appealing. The comparison of a result set to a collection wide ordering based on document prior is not computationally expensive as it requires access to a single numeric value for each retrieved result. Selecting queries by their ability to produce results sets that significantly differ to other queries is novel. However, our approach is limited by the fact that the effectiveness of the training queries – those used to establish access counts – is unknown. Perhaps a set of document priors trained on poorly performing queries would produce an ordering by which future difficult queries could be selected. We leave this open as an area for future work.

### 7.3 Summary

In this chapter, we have explored two possible applications of access count's for the purposes of improved query evaluation accuracy. First, we incorporated access counts as document priors into the similarity ranking function. Based on related work, we proposed several methods by which to combine priors based on access counts to the similarity rank function document scores. We found that due to the skew distribution of access counts, normalisation is required to combine access counts with ranking scores. Our results showed limited improvements in accuracy over an Okapi BM25 baseline. However, when compared to a Dirichlet language model ranking function, the access count priors did not significantly improve accuracy.

We then proposed a novel approach to query difficulty prediction based on the difference in the rankings of the documents in the collection-wide access-order, when compared to the ranked results returned in response to a query. The motivation behind this approach is that access counts provide an indication of the likelihood of observing a document in the top ranked results for future queries. We employed Kendall’s $\tau$ to compute the correlation
between the two rankings. Over two TREC collections and topic sets, our experiments produced a wide range of accuracy values that did not correlate with the predicted difficulty estimate. Further, for most test topics, the range of prediction values was not diverse enough to categorise the topics.

The effectiveness of the two applications explored here are dependent on the access count weights obtained during a training phase where past queries are processed. The counting scheme used considers a fixed number of top ranked results returned in response to each query. While results in previous chapters have shown that this scheme is sufficient for query evaluation with list pruning, it is possible that more refined weights are necessary for document prior and difficulty prediction tasks. An alternative weighting function that was not explored here, and remains open for further exploration is the assignment of access counts based on the accumulation of normalised similarity function scores.
Chapter 8

Conclusions

We have presented several techniques that take advantage of the repetition in query logs to improve query efficiency. The genesis of this work was the observation of non-uniform document occurrence in the top ranked results composed by search engines in response to a set of user queries. In Chapter 3 we proposed three counting schemes to measure the frequency of document access by the search system. Each proposed scheme allocated varying weights to the documents retrieved in response to a query. Our experiments showed that, after processing a large set of queries, the counting schemes produced a set of document weights that ranked those documents known to be relevant higher than most other documents in the collection. Although varied in technique, the counting schemes all produced similar results, with a high proportion of known relevant documents ranked highly in the ordering; suggesting that, on average, past queries are an effective means of identifying useful documents in the collection.

Based on the observed relationship between the frequently accessed and known-relevant documents, we proposed techniques to reduce query evaluation time. Exploitation on the frequency of access of an item is often achieved though cache implementations that allow rapid access to the subset of frequently requested items. However, for search engine query evaluation, the inverted list of any given term is likely to contain postings for both frequently and infrequently retrieved documents. While it is feasible for a cache to retain a segment of an inverted list, no previous work has considered dynamically caching individual postings from the inverted lists as they are too small and numerous to viably maintain separately in a cache.

An alternative solution is to rearrange the inverted lists so that those postings that are likely to have an impact on the ranking function are clustered towards the front of each
inverted list. During query evaluation the “important” postings at the head of the list can be processed, while other postings that are less likely to affect the ranked order of results are optionally processed. Persin et al. [1996] and Anh and Moffat [2002b] propose inverted list organisations that order inverted lists by the impact of an individual posting on the ranking function. In their frequency- and impact-ordered lists, inverted list postings are ordered by descending impact on the ranking function. In contrast, our lists were ordered by the frequency of occurrence of a document in the results of past queries. While other list orderings have an explicit relationship with the ranking function, our approach has an indirect relationship. Documents that achieve high similarity scores are ranked highly in response to past queries, hence the postings for those documents are moved to the front of the lists.

We applied various pruning heuristics to the access-ordered lists and showed that, on average, processing as little as 10% of the query term inverted list postings produced results that were as accurate as full list evaluation. While our ordering and pruning schemes proved more efficient than full evaluation (with query processing time reductions of around 50%), the results showed that impact-ordering produces greater query evaluation time savings. The difference can be attributed to the pruning techniques of impact-ordering, where some lists can be skipped entirely during query evaluation, while for access-ordering, all lists are partially evaluated.

A second important distinction between our access-ordered approach and other orderings is that an access-ordering is a collection-wide ordering of documents. In Chapter 4 we reordered the collection to reflect document access. While the focus of past collection reordering work has been a reduction in index size [Blandford and Blelloch, 2002; Shieh et al., 2003; Silvestri et al., 2004a], our proposal retains an index of size equivalent to a non-reordered index while allowing dynamic query time pruning. Our exploration of collection reordering encompassed an analysis of access-order stability. The results showed that an ordering established on as little as 100,000 queries was sufficiently stable to produce similar results as that of an index trained on 20 million queries. That is, a relatively small amount of training can obtain an ordering that accurately produces results while pruning for a large number of future queries, therefore suggesting that index rebuilds need not be frequent.

Inverted list pruning heuristics take two broad forms. The first is dynamic pruning at query time, where a heuristic is applied to determine when to terminate the processing of an inverted list. The second is static index pruning, where postings are removed from the inverted lists at index construction time. The benefit of the former is that dynamic pruning
provides the option of increasing the number of postings examined per list. For a “difficult” query it may be desirable to process a greater proportion of each inverted list. The latter reduces the on-disk size of an index allowing improved possibilities of caching, and reduced disk access costs. However, a large proportion of the term lists in a typical index contain very few postings. Our experiments in Chapter 5 show that the index size reduction that can be obtained through static index pruning is low, around 19%. The results reflect those of Anh and Moffat [2002b] who, through removing low impact postings from the inverted list, manage to reduce index size by around 25%, while retaining query effectiveness. An alternative solution is to prune selected terms from the index, or to use a combination of pruning both postings and terms. In related work, Büttcher and Clarke [2005b] have shown that such an approach is capable of producing an index small enough to reside in memory. In combination with an unpruned on-disk index, they show significant reductions in query evaluation times. We have not investigated such term-based pruning approaches in combination with access counts and leave this open as an area of future work.

Our exploration of static pruning made use of a pre-retrieval difficulty predictor. We applied an inverse document frequency (IDF) predictor to a training set of topics from TREC, and found an imbalance in the predicted difficulty of the TREC topics and web-log queries. While our analysis was not focused on this matter, it raises the question of how representative TREC test topics are of real-world web queries.

Throughout this work, we contrast our collection-oriented global access-ordering to the more individualised term-based orderings of the impact- and frequency-based approaches. The primary advantage of a collection-wide ordering is the ability to rearrange the collection, allowing the use of well-known index compaction techniques to compress the index. However, term-based ordering schemes are able to establish an individualised posting order for each list, and therefore are able to achieve higher accuracy results at a higher level of pruning than our approach. We consider our access-ordered indexes a compromise between the naïve simplicity of a document-ordered index, and the ranking function oriented impact-ordered approach.

Our observations of repeated document access during the evaluation process led to an examination of other frequently accessed search engine data structures. Given the frequency of access to documents in the search results, repetitive access patterns were likely for other items required during the query evaluation process. In Chapter 6 we proposed a cost model for disk access during query evaluation. We examined the frequency of access of the components in a search engine and proposed measures to estimate the benefits of caching. Our proposed
cost model can easily be adjusted to different search engine implementations and hardware configurations. The results showed that the implementation of a cache at various levels of the query evaluation process can yield substantial savings in disk activity. On our data, reductions in disk activity of over 30% were achieved with a cache that is 10% the size of the index. Indeed, a small cache for vocabulary entries alone resulted in estimated savings of up to 14% in disk activity.

Our exploration of search engine component caching was limited to full inverted lists. We believe that, for alternative list orderings such as impact-, frequency-, or access-ordered, an investigation of caching list fragments will result in further gains. A limitation of our cost-per-byte (CPB) eviction policy was the need for rapid access to the inverted-lists component of the cache. There is a need to explore techniques for the selection of an item to evict from the cache. A simple approach to this may be to quantize inverted lists into a limited range of sizes, and then, for each quantized size, maintain an LRU queue. Alternatively, heuristics may be used to limit the number of cache elements updated per query, but to also ensure that, within a given confidence level, the inverted list removed from the cache is the one with the lowest CPB.

Our cost analysis also confirms the limited benefits of static pruning. As the majority of disk access costs are related to disk seek times, and as static pruning only reduces list size, the benefits of reducing the size of the inverted list to read is minimal. In contrast, our experiments show that a cache of frequently accessed lists can result in significant query time savings, supporting the results of Büttcher and Clarke [2005b].

As access counts provide evidence in support of document relevance we attempted to exploit them to improve the accuracy of retrieval. We explored the application of access counts as a form of document prior in the ranking function, increasing the estimated similarity of the documents retrieved in response to a query by an amount proportional to the access count of each document. We conclude that, while effective for pruning heuristics, feedback of access counts into the ranking functions is ineffective. We suspect that the ineffectiveness is due to the dependency access counts have on the ranking function used to generate such counts. Similarly, when we utilised the collection-wide access-ordering as a default ranking of documents for the task of difficulty prediction our results were not promising. Our difficulty prediction model was based on determining the difference between the ranked results returned for a query, and the expected ordering of documents for a generic “difficult” query. As relevance judgements are not available for the training queries, we suspect that an access-count-based ordering is not representative of the ranked results of a generic “difficult” query.
8.1 DIRECTIONS FOR FUTURE RESEARCH

A possible (future) solution is to use click-through information to filter “difficult” queries. For example, generating access counts on only those queries for which users do not click on a result.

8.1 Directions for Future Research

Several directions for future work are possible. Parallels can be drawn between access counts and other document weighting schemes [Page et al., 1999; Kleinberg, 1999], hence whether index organisations based on such schemes — such as PageRank — could also be used for effective list pruning remains open.

Another area open to exploration is the establishment of the access-ordering. While our exploration of counting schemes provided enough evidence to show that past queries are able to select “useful” documents, we limited our techniques to including only a fixed number of top-ranked results per query. An exhaustive exploration of counting techniques that allow for dynamic query-dependent inclusion of documents in the counted results remains unexplored. Further, counting schemes that award increments relative to the estimated similarity of the document to the query may prove more accurate than our rank-based techniques. However, some pilot work investigating alternative counting schemes did not suggest significantly different orderings.

Finally, our difficulty prediction technique is based on measuring the degree of disorder in the ranked documents retrieved in response to a query, compared to the “default” ranking of documents in the collection for a generic query. Our work makes the assumption that a generic query is a “poor” or “difficult” one, and as such, the access-order of documents over a set of queries establishes a ranking that can be used to measure the disorder. The concept of a “default” collection ranking is based on the idea that there is ranking bias towards certain documents irrespective of the query. As such, an ideal “default” ranking would be based on the bias that a ranking measure exhibits towards each document in the collection. A direction of further exploration is an analysis of ranking functions to confirm that such bias is present, and to determine the factors that lead to such bias.

8.2 Final Remarks

In conclusion, we have investigated the use of past queries to establish an ordering of documents in the collection. Our access-ordering is an efficient index organisation. In combination with list pruning, query evaluation requires on average, the processing of 10% of the postings
in the inverted lists. Further, no reduction in accuracy is observed. We hypothesised that similar skews of access were likely for data used at query evaluation time. We proposed a search engine cache architecture that maintains several components of the search process, and showed significant benefits over simple caches that consider items such as inverted lists or result pages.
Bibliography


S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In

0163-5840.

E. W. Brown. Fast evaluation of structured queries for information retrieval. In E. A. Fox,
P. Ingwersen, and R. Fidel, editors, Proc. 18th ACM SIGIR conf. on Research and De-

E. W. Brown, J. P. Callan, W. B. Croft, and J. E. B. Moss. Supporting full-text information
retrieval with a persistent object store. In M. Jarke, J. A. Bubenko Jr., and K. G. Jeffery,
editors, Proc. 4th Conference on Extending Database Technology EDBT, pages 365–378,

GIR conf. on Research and Development in Information Retrieval, pages 97–110, Montreal,

S. Büttcher and C. L. A. Clarke. Indexing time vs. query time: trade-offs in dynamic infor-
mation retrieval systems. In O. Herzog, H. Schek, N. Fuhr, A. Chowdhury, and W. Teiken,
editors, Proc. 14th ACM Conference on Information and Knowledge Management, pages

S. Büttcher and C. L. A. Clarke. Efficiency vs. effectiveness in terabyte-scale information
Conference TREC, Gaithersburg, Maryland, 2005b. National Institute of Standards and
Technology (NIST). NIST Special Publication 500-266.

F. Cacheda and A. Vina. Experiences retrieving information in the world wide web. In

D. Carmel, E. Amitay, M. Herscovici, Y. Maarek, Y. Petruschka, and A. Soffer. Juru at
TREC-10: Experiments with index pruning. In E. M. Voorhees and D. K. Harman,
editors, Proc. The 10th Text Retrieval Conference TREC, pages 228–236, Gaithersburg,


DMOZ. Dmoz open directory project. http://dmoz.org/, 2006. The Open Directory Project is a human-edited directory of the Web that is maintained by volunteer editors.


N. Fuhr, M. Lalmas, S. Malik, and G. Kazai, editors. *Advances in XML information retrieval and evaluation, 4th international workshop of the INInitiative for the Evaluation of XML
BIBLIOGRAPHY


lemurproject.org/, 2006. Language Technologies Institute CMU and Center for Intelligent Information Retrieval UMASS.


Lycos. Lycos homepage Germany. http://www.lycos.de/, 2006. The Lycos German Internet Web portal hosts a range of Web services including a search facility.


BIBLIOGRAPHY


