Multi-User File System Search

by

Stefan Büttcher

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2007

Abstract

Information retrieval research usually deals with globally visible, static document collections. Practical applications, in contrast, like file system search and enterprise search, have to cope with highly dynamic text collections and have to take into account user-specific access permissions when generating the results to a search query.

The goal of this thesis is to close the gap between information retrieval research and the requirements exacted by these real-life applications. The algorithms and data structures presented in this thesis can be used to implement a file system search engine that is able to react to changes in the file system by updating its index data in real time. File changes (insertions, deletions, or modifications) are reflected by the search results within a few seconds, even under a very high system workload. The search engine exhibits a low main memory consumption. By integrating security restrictions into the query processing logic, as opposed to applying them in a postprocessing step, it produces search results that are guaranteed to be consistent with the access permissions defined by the file system.

The techniques proposed in this thesis are evaluated theoretically, based on a Zipfian model of term distribution, and through a large number of experiments, involving text collections of non-trivial size — varying between a few gigabytes and a few hundred gigabytes.

Acknowledgements

First and foremost, I would like to thank my supervisor, Charlie Clarke, who did a tremendous job over the past few years, offering guidance whenever I needed it, providing advice whenever I sought it, and allowing me the academic freedom to pursue the research that I was most interested in. I owe you big-time.

Thanks to the members of the IR/PLG lab, who created a most enjoyable working atmosphere. In particular, I would like to thank Richard Bilson, Ashif Harji, Maheedhar Kolla, Roy Krischer, Ian MacKinnon, Brad Lushman, Thomas Lynam, and Peter Yeung.

Thanks to Gordon Cormack, who taught me by example that it is possible to have a successful career without ever growing up. Thanks also to the School of Computer Science for supporting me through the David R. Cheriton graduate student scholarship, and to Microsoft for supplying us with little toys that ensured a certain level of distraction at all times.

Finally, thanks to the members of my committee, Charlie Clarke, Gordon Cormack, Alistair Moffat, Frank Tompa, and Olga Vechtomova, who provided valuable feedback that greatly helped improve this thesis.

Bibliographical Notes

Preliminary versions of the material presented in some parts of this thesis have appeared in the following publications:

- Stefan Büttcher and Charles L. A. Clarke. Indexing time vs. query-time trade-offs in dynamic information retrieval systems. In *Proceedings of the 14th ACM Conference on Information and Knowledge Management* (CIKM 2005). Bremen, Germany, November 2005. (Chapter 6)

- Stefan Büttcher and Charles L. A. Clarke. A security model for full-text file system search in multi-user environments. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies* (FAST 2005). San Francisco, USA, December 2005. (Chapter 5)

- Stefan Büttcher and Charles L. A. Clarke. A hybrid approach to index maintenance in dynamic text retrieval systems. In *Proceedings of the 28th European Conference on Information Retrieval* (ECIR 2006). London, UK, April 2006. (Chapter 6)

- Stefan Büttcher and Charles L. A. Clarke. Adding full-text file system search to Linux. *;login: The USENIX Magazine*, 31(3):28–33. Berkeley, USA, June 2006. (Chapter 7)

- Stefan Büttcher, Charles L. A. Clarke, and Brad Lushman. Hybrid index maintenance for growing text collections. In *Proceedings of the 29th ACM SIGIR Conference on Research and Development in Information Retrieval* (SIGIR 2006). Seattle, USA, August 2006. (Chapter 6)

I would like to thank my co-authors for their assistance and the anonymous reviewers for their helpful feedback on the issues discussed in these papers.

to my parents

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Over the last decade, Web search engines like Altavista[1] and Google[2] have dramatically changed the way in which we access information. Billions of documents published on the Internet are now accessible to a searcher within a matter of seconds — thanks to vast numbers of high-performance index servers operating in the data centers of the major Web search companies, and thanks to elaborate ranking functions that ensure that the searcher is only presented those documents that are most likely to match his or her information need.

Surprisingly, for the contents of a user's own file system, the situation has been quite different for a long time. While a random document on the World Wide Web could be found through a short sequence of key-strokes and mouse clicks, finding a misplaced document on one's private desktop computer could be a frustrating and time-consuming endeavor. Existing interfaces allowing to search the contents of the local file system usually did not allow the user to specify search constraints regarding the contents of a matching file, but only regarding some outward criteria (meta-data), such as the file name. Search interfaces that did allow the user to specify content-related criteria, on the other hand, like UNIX's `grep` utility (or the traditional Windows search mechanism), were unbearably slow and easily required several minutes to locate the file in question — unless the file was stored in a special format, such as PDF[3], in which case the search was condemned to be unsuccessful from the very beginning.

To some extent, the situation has changed with the recent advent of various desktop search engines, such as Copernic/Google/MSN/Yahoo! desktop search or Apple's Spotlight system. These programs build a full-text index for the contents of the computer's file system, employing special input filters for file formats like PDF. The index is kept up-to-date, reflecting

[1]http://www.altavista.com/ (accessed 2007-03-01)
[2]http://www.google.com/ (accessed 2007-03-01)
[3]http://www.adobe.com/products/acrobat/adobepdf.html (accessed 2007-08-01)

the contents of the ever-changing file system. It allows a user to find a misplaced file within seconds instead of minutes, thus closing the gap between Web search and local desktop search.

Unfortunately, virtually nothing has been published about the internals of desktop search engines. Moreover, they are usually targeting a very specific scenario: a single-user desktop computer system containing a very limited amount of indexable data, perhaps a few hundred megabytes. It is unclear whether they can be extended to be used in multi-user environments or for file systems that contain substantial amounts of indexable data.

The problems posed by file system search may seem trivial compared to those present in Web search, as the amounts of data that need to be dealt with are so much smaller. This impression, however, is not entirely accurate. The main challenge of Web search does in fact stem from the vast amounts of data that need to be processed by the search engine. The difficulty of file system search, on the other hand, stems from the scarcity of computational resources available to manage the task at hand. While a Web search engine typically runs in parallel on thousands of computers dedicated to just this specific purpose, a file system search engine only runs on a single machine. Moreover, it has to share all its resources – CPU, main memory, and hard drive bandwidth – with other processes running on the same hardware. Under these circumstances, building and maintaining an index for a multi-gigabyte text collection becomes a challenging task.

In this thesis, I propose a set of algorithms and data structures that can be used to implement a highly scalable file system search engine — scaling from a single-user desktop computer to a multi-user search environment, and scaling from a small collection of private e-mail messages and personal notes to a document collection comprising several gigabytes of text. The search engine can update its index in real time. It enables users to perform low-latency search tasks, identifying and returning to the user a set of matching documents in under a second — for a text collection that can be regarded representative of the contents of a medium-size file system. The techniques proposed for constructing and maintaining a full-text index for a dynamic text collection scale even further and can easily handle hundreds of gigabytes of text. At that point, however, sub-second search operations become difficult.

The methods presented and examined are part of the publicly available Wumpus[4] search engine. Wumpus is an experimental file system search engine and multi-user information retrieval system that serves as a proof of concept for the methods put forward in this thesis.

While most design decisions made are motivated by the specific requirements of file system search, the results obtained are applicable to other applications as well. Real-time index updates, for example, are a desirable feature for any search engine that needs to deal with a

[4]http://www.wumpus-search.org/ (accessed 2007-07-01)

dynamic text collection. Multi-user security restrictions, on the other hand, are not only useful in file system search, but are essential in collaborative work environments that are targeted by typical enterprise search engines (e.g., Microsoft SharePoint[5], Google Search Appliance[6], and FAST Enterprise Search[7]).

1.1 What is File System Search?

I view file system search from the UNIX[8] perspective and, more specifically, from the GNU/Linux[9,10] point of view. As such, it is more general than desktop search. While desktop search is only concerned with the data owned by a single user, UNIX file system search needs to explicitly take into account the existence of multiple users, dozens or even hundreds, as supported by every UNIX installation. While desktop search usually focuses on certain parts of the data stored in the file system, such as a user's e-mail, recently accessed Web documents, and the contents of the *My Documents* folder in Windows, file system search needs to cover a broader spectrum, providing efficient access also to `man` pages, header files, kernel sources, etc.. In essence, a file system search engine ought to be able to index all data in the file system that allow a meaningful textual representation.

Moreover, in contrast to a desktop search engine, a file system search engine should allow other applications to make use of its search functionalities. For example, if the search engine enables the user to search her collection of e-mails, then it no longer makes sense for an e-mail client to maintain its own search infrastructure. Instead, the mail client should provide the user with a search interface that, in the background, contacts the file system search engine and utilizes its index structures in order to produce the search results. The actual presentation of the search results may be application-specific, but the underlying search operation is performed in a centralized way that can be leveraged by many different applications. Ideally, file system search should be an operating system service, similar to opening a file or sending a message across a network (although not necessarily realized in the kernel).

This point of view is similar in spirit to the path adopted by Windows Vista[11] and Apple's Mac OS X Tiger[12]. In both systems, full-text search is viewed as an integral component of the user interface that should be available in every possible context.

[5]http://www.microsoft.com/sharepoint/ (accessed 2007-03-02)
[6]http://www.google.com/enterprise/ (accessed 2007-03-02)
[7]http://www.fastsearch.com/ (accessed 2007-03-15)
[8]http://www.unix.org/ (accessed 2007-03-02)
[9]http://www.gnu.org/ (accessed 2007-03-02)
[10]http://www.linux.org/ (accessed 2007-03-02)
[11]http://www.microsoft.com/windows/products/windowsvista/ (accessed 2007-03-01)
[12]http://www.apple.com/macosx/tiger/ (accessed 2007-03-02)

Requirements of File System Search

The general search scenario outlined above leads to a set of five fundamental requirements that should be met by any file system search engine and that are addressed in this thesis.

Requirement 1: Low Latency Query Processing
Whenever a user submits a query to the search engine, the search results should be returned within at most a few seconds, preferably less, similar to what users of Web search engines are accustomed to.

Requirement 2: Query Flexibility
The set of query operations provided by the search engine should be flexible enough for the search engine to be accessed by the user indirectly, through a third-party application, such as an e-mail client. In order for this to be possible, the query language provided by the search engine needs to be as flexible and expressive as possible without sacrificing query performance beyond reason.

Requirement 3: Index Freshness
Whenever possible, the current version of the index, and thus the results to a search query, should reflect the contents of the file system. Changes in the file system – file deletions, creations, and modifications – need to be propagated to the search engine's index structures in real time.

Requirement 4: Non-Interference
The search engine should not interfere with the normal operation of the computer system. For example, neither excessive disk space usage nor excessive main memory usage are acceptable. In addition, the search engine should only exhibit significant CPU or disk activity if this activity does not derogate the performance of other processes running in the system.

Requirement 5: Data Privacy
If the computer system is accessed by multiple users, then the search engine must respect the security restrictions defined by the file system. For example, if user A submits a search query, then the results to that query may only contain information about a file F if A has read permission for F.

Obviously, some of these requirements work against each other. A more general query language makes it possible to specify search operations that are more expensive to carry out. Overly tight constraints on the freshness of the search engine's index will force the index to interfere with other processes running on the same machine. As always, the challenge lies in finding the “right” point in each such trade-off. That point, however, may very well be different from system to system and even from user to user.

Main Research Question

The main research question underlying this thesis is how to meet the basic requirements outlined above in a highly dynamic, multi-user search environment.

How can we guarantee a certain degree of index freshness if the contents of the file system are continuously changing? How can we ensure that the search results to a query submitted by user A can never depend on the contents of user B's private files? In what ways does the interaction between the five different requirements affect the decisions that must be made when designing a multi-user file system search engine?

1.2 Thesis Organization

The next chapter provides an overview of related work in the area of file system search and in the greater context of information retrieval. I present a summary of commonly used search operations, index structures, and index maintenance strategies that can be used to keep the index synchronized with a changing text collection.

Chapter 3 provides the experimental and methodological framework adhered to in my experiments. It describes the hardware and software configuration as well as the text collections used in the experiments.

In Chapter 4, I discuss how exactly the search engine's index structures should be integrated into the file system. I propose to use device-specific schema-independent inverted files as the basic data structures of the search engine. I then present a statistical model, based on Zipf's law [117] that can be used to approximate certain properties of inverted files. This model is made use of extensively throughout Chapter 4 and Chapter 6 and is shown to yield very accurate estimates of the actual characteristics of inverted files.

The material in Chapter 4 can be used to build a search engine for a single-user file system whose index structures are updated periodically, maybe once per day. The two main contributions of the chapter are related to memory management aspects of the search engine. To reduce the memory consumption caused by the search engine's dictionary, containing an entry for every keyword found in the text collection, I propose an interleaving scheme that keeps most dictionary entries in the on-disk index and only holds a very small number of entries in main memory. To improve memory utilization during index construction, I propose a novel way of arranging postings in the extensible posting lists used in most state-of-the-art in-memory indexing methods. The method is a combination of linked lists and controlled pre-allocation.

In Chapter 5, the results from Chapter 4 are extended into the domain of multi-user search environments. I present a discussion of security issues that arise if the same index is shared by multiple users with different file access privileges. After giving a proper definition of what it means for a file to be *searchable* in the context of multi-user file system search, I show that the naïve way to enforce security restrictions – as a postprocessing step, performed after an otherwise unaltered query processing procedure – can be exploited by a user to obtain information about files for which she does not have search permission. This shortcoming is rectified by a modified query processing strategy that is guaranteed to produce secure search results and that is only marginally slower than the postprocessing approach. In fact, if the user's visibility of the file system (i.e., the relative number of files searchable or readable) is small, then the new query processing strategy may even lead to higher query performance than the postprocessing approach.

In Chapter 6 is concerned with efficient, real-time index updates. Focusing first on incremental text collections, allowing the insertion of new files into the collection, but not the deletion or modification of existing files, I propose LOGARITHMIC MERGE, an index maintenance strategy that partitions the on-disk index into sub-indices of exponentially increasing size. LOGARITHMIC MERGE offers indexing performance close to that of the index construction method for static collections evaluated in Chapter 4, while providing query performance close to that of more costly index update mechanisms. This method is then extended by combining it with an in-place update strategy for long lists, resulting in a *hybrid index maintenance* strategy with asymptotically optimal (i.e., linear) index maintenance disk complexity.

I show how the security restrictions described in the previous chapter can almost effortlessly be used by the search engine to support file deletions, by combining them with a lazy garbage collection strategy that conveniently integrates into the methods proposed for incremental index updates. I also discuss how the index structures need to be changed if the search engine has to offer efficient support for file append operations, indexing only the new data that have been appended to an existing file, and not re-indexing the entire file. If the file system contains large amounts of data in append-only files, such as UNIX system log files (`/var/log/*`), then this is an essential performance aspect of the index maintenance process.

The chapter concludes with a list of changes that need to be made to the search engine to turn its *amortized* index maintenance complexity into actual real-time update performance. With these modifications in place, the search engine is able to react to every file update event within a few seconds, even when the update load is very high.

Chapter 7 puts the index update strategies proposed in Chapter 6 into the proper context of file system search. I examine existing file system notification mechanisms available in Linux, detailing what events must trigger notifications to the search engine and what specific information the operating system needs to convey to the indexing process. I point out some deficiencies of the existing implementations (`dnotify` and `inotify`) and present an alternative solution, `fschange`. I also discuss how the specific properties of file system search affect some decisions regarding the index structures chosen and the index update strategies employed by the search engine.

Conclusions, as well as possible directions for future work, are given in the final chapter of the thesis.

Chapter 2

Background and Related Work

In order to be able to search a text collection, such as the contents of a file system, in an efficient and convenient manner, a set of tools must be available that allow the user to express a search query and find all parts of the text collection matching the search constraints specified by the query. Depending on the size of the collection and the performance requirements defined by the user or the application, special data structures might or might not be necessary to realize search operations on a given text collection.

This chapter provides a brief summary of the traditional UNIX search paradigm (Section 2.1), followed by a discussion of what search operations are commonly supported by search applications and information retrieval systems and might be useful and desirable in the context of file system search (Section 2.2). The discussion covers Boolean queries and several types of ranked queries, including query processing strategies for each query type.

Naturally, the type of search operation to be supported by the search engine defines what index data structure should be used. After a short discussion of what search data structures are best suited for different kinds of search operations (Section 2.3), I give an overview of inverted indices, the de-facto standard data structure for keyword-based search operations (Section 2.4). This includes a summary of related work in the areas of index construction and index compression.

In Section 2.5, I review known update strategies for inverted indices, covering the three different update operations that can be witnessed in dynamic search environments: document insertions, document deletions, and document modifications.

Finally, I provide a summary of existing file system search engines and point out similarities and dissimilarities between file system search and two closely related fields: desktop search and enterprise search (Section 2.6).

2.1 Traditional File System Search in UNIX

Traditional file system search in UNIX is based on the `find`/`grep` paradigm:

- The `find` utility, part of the GNU findutils package[1], can be used to obtain a list of files or directories that match certain external criteria (meta-data), like file name, size, or modification time.
- The `grep`[2] utility can be used to match the contents of a file (or multiple files) against a regular expression.

In combination, `find` and `grep` can be used to specify highly complex search criteria. For example, a list of all files in or below the user's home directory whose name ends with ".pdf" can be obtained by issuing the command:

```
find $HOME -type f -name \*.pdf
```

A list of all Java source files that are dealing with I/O exceptions and that have been accessed within the last 24 hours can be obtained by executing the command:

```
find $HOME -type f -atime 0 -name \*.java | xargs grep -l IOException
```

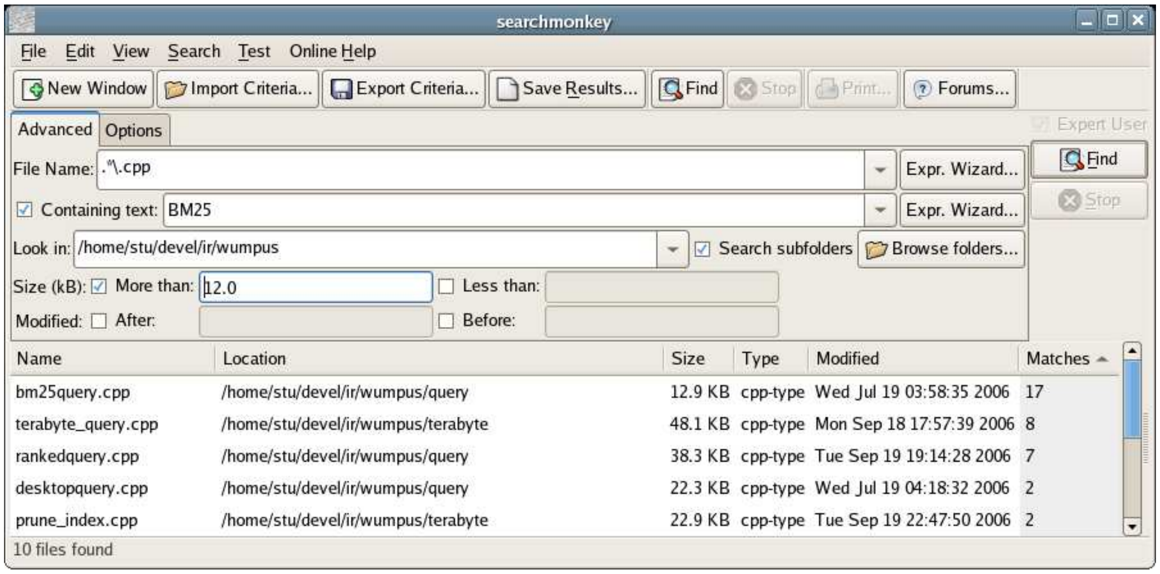
(here, the `xargs` utility, also part of the GNU findutils package, transforms its input to command-line parameters sent to `grep`).

A more convenient and user-friendly way of searching the contents of files in the file system is provided by Searchmonkey[3], a graphical application that mostly offers the same functionality as `find`/`grep` (see Figure 2.1), but presents the results to a search query in a way that makes it easier to browse them, especially for less experienced users.

Both the command-line version and the graphical version of the traditional UNIX search paradigm share the same two shortcomings:

1. Because no index data structure is maintained, it might be necessary to scan the entire file system in order to obtain a list of files matching a given set of external criteria. Depending on the size of the file system, this procedure can easily take several minutes.
2. Furthermore, in order to obtain a list of files matching a certain content-related criterion (in the example in Figure 2.1, the requirement that a file contains the string "BM25"), all candidate files need to be read from disk. Depending on the number and the sizes of the candidate files, this procedure may very well take several hours to complete.

[1]http://www.gnu.org/software/findutils/ (accessed 2007-05-15)
[2]http://www.gnu.org/software/grep/ (accessed 2007-05-15)
[3]http://searchmonkey.sourceforge.net/ (accessed 2007-05-15)



File Name: .*\.cpp
☑ Containing text: BM25
Look in: /home/stu/devel/ir/wumpus ☑ Search subfolders
Size (kB): ☑ More than: 12.0 ☐ Less than:
Modified: ☐ After: ☐ Before:

Name	Location	Size	Type	Modified	Matches
bm25query.cpp	/home/stu/devel/ir/wumpus/query	12.9 KB	cpp-type	Wed Jul 19 03:58:35 2006	17
terabyte_query.cpp	/home/stu/devel/ir/wumpus/terabyte	48.1 KB	cpp-type	Mon Sep 18 17:57:39 2006	8
rankedquery.cpp	/home/stu/devel/ir/wumpus/query	38.3 KB	cpp-type	Tue Sep 19 19:14:28 2006	7
desktopquery.cpp	/home/stu/devel/ir/wumpus/query	22.3 KB	cpp-type	Wed Jul 19 04:18:32 2006	2
prune_index.cpp	/home/stu/devel/ir/wumpus/terabyte	22.9 KB	cpp-type	Tue Sep 19 22:47:50 2006	2

10 files found

Figure 2.1: Searchmonkey, a graphical alternative to `find`/`grep`.

The first shortcoming is addressed by the `locate` utility. Like `find`, `locate` is part of the GNU findutils package. `locate` periodically scans the entire file system (running an `updatedb` process, usually once per day) and builds a database containing the names of all files in the file system. By using this database, a file whose name matches a certain criterion (e.g., "$*$.cpp") can be found much faster than by performing an exhaustive scan of the entire file system.

Unfortunately, `locate` does not take a user's file permissions into account when it processes a search query. This allows an arbitrary user to obtain a list of files found in a different user's directory — a deficiency that is remedied by the `slocate`[4] utility. `slocate` is a secure version of `locate` that offers the same functionality, but takes user permissions into account when processing a query. After generating a list of files matching the query, `slocate` removes from this list all files that cannot be seen by the user, according to the standard UNIX file system security model.

The second shortcoming of the `find`/`grep` paradigm, having to read the actual contents of all candidate files from disk, is not addressed by any of the traditional UNIX search utilities. The main reason for this deficiency seems to be that, when `find`/`grep` came into existence, reading file contents from disk during a search operation was not as grave a performance problem as it is nowadays. The throughput/capacity ratio was in favor of throughput, and reading the entire content of a hard disk was an operation that could be performed within a matter of minutes. With modern hard drives, this is no longer the case.

[4]http://slocate.trakker.ca/ (accessed 2007-01-11)

Hard Drive Characteristics: Capacity vs. Speed

The throughput of a hard disk drive is essentially determined by two factors: its storage density and its rotational velocity. The higher the storage density, measured in bits per square inch, and the higher the rotational velocity, measured in revolutions per minute, the greater the hard drive's data throughput (other factors, such as the average radius of a disk track, also play a role, but seem to have remained more or less constant over the past decade).

Obviously, there are some limits to the maximum rotational velocity at which a hard drive may run. Lee et al. [64], for instance, discuss how increasing the rotational velocity of a hard drive is related to a higher generation of contamination particles on the disk platter. Consequently, the rotational speed of a typical consumer hard drive has only increased by about a factor 2 (from 4,200 to 7,200 rpm) within the last 10 years. Instead of raising the rotational velocity, hard drive manufacturers have concentrated on increasing the density with which data can be stored on a hard disk, from 3 gigabits per square inch in 1995[5] to a remarkable 179 gigabits per square inch in 2005[6]. New technologies, such as perpendicular recording, will allow storage densities beyond the terabit-per-square-inch mark [73].

Increasing the storage density of a hard disk has two immediate effects: it increases the hard drive's storage capacity, and it increases its throughput. These two aspects of the hard drive's performance, however, are not affected in the same way. Because data are stored in two dimensions on a disk platter, but are read and written in only one dimension, an increase in storage density by a factor 4 approximately decreases the throughput of the hard drive by a factor 2 — relative to its storage capacity.

As a result, while the capacity of a typical consumer hard drive has grown by about a factor 50 over the last 10 years, the sequential read/write performance has only increased by about a factor 10. Hence, reading the entire content of a hard drive in a long sequential read operation now takes 5 times as long as 10 years ago, usually several hours. For random access, the ratio is even less favourable: The duration of a random disk seek could be reduced by only 50% over the last 10 years, implying that random disk access (e.g., opening and reading all files that potentially match a search query) now is more expensive than ever.

A detailed overview of different performance aspects of hard disk drives is given by Ruemmler and Wilkes [96] and also by Ng [82]. What is important here, however, is the insight that a search infrastructure which requires an exhaustive scan of the file system in order to process a search query no longer represents a feasible solution. In order to realize efficient full-text search, a content index needs to be built that contains information about

[5]http://www.littletechshoppe.com/ns1625/winchest.html (accessed 2007-01-25)
[6]http://en.wikipedia.org/wiki/Computer_storage_density (accessed 2007-01-25)

all occurrences of potential search terms within the file system, reducing (i) the amount of data read from disk and (ii) the number of random disk access operations performed while processing a query.

Before I can move on to discuss how such an index could look like and what data structures have been proposed for this purpose in the past, I first need to examine what types of search operations have to be supported by the index, because this directly affects which specific data structure should be chosen. For now, let us assume that an index simply provides the search engine with fast access to the list of all occurrences of a given term. The exact information contained in such a list, and the way in which the information is encoded, will be discussed later.

2.2 Search Queries

As we have seen in the previous section, regular expressions can effectively be used to search the contents of a file system. Indeed, regular expression search is older than UNIX itself and was first proposed by Ken Thompson, one of the fathers of UNIX, in 1968 [106]. However, while regular expressions are undoubtedly a powerful search tool and are very useful for certain applications, such as code search[7], they might actually be more powerful than necessary for the purposes of file system search. In addition, regular expressions do not provide a satisfying mechanism for ranking search results by predicted relevance.

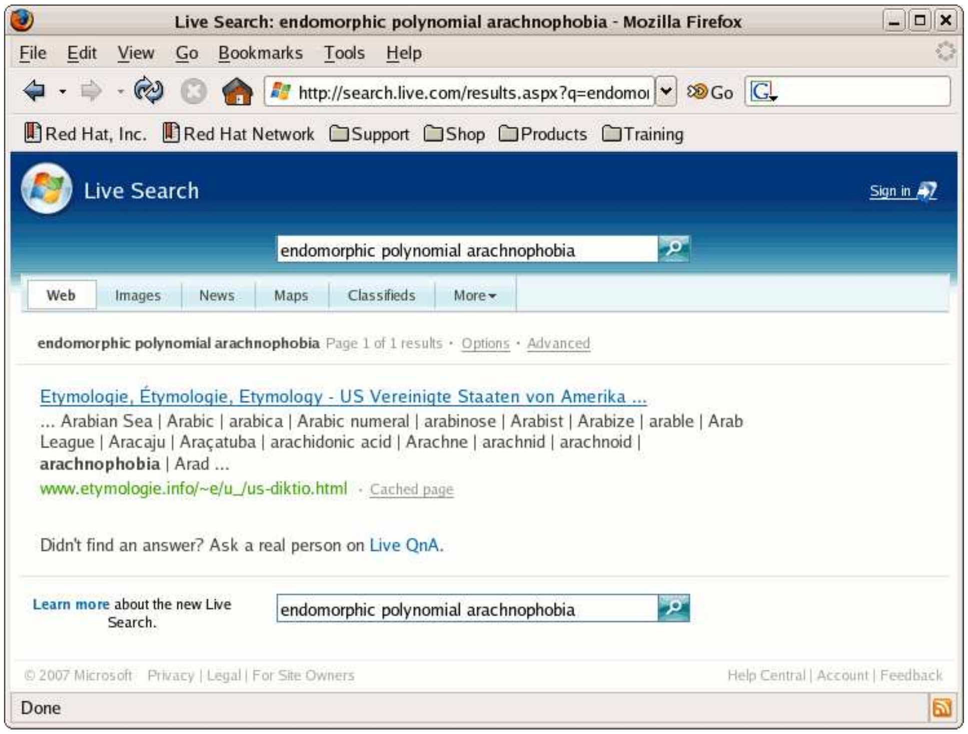
The predominant type of search query in many applications is the *keyword query*. In a keyword query, the search constraints are not represented by a regular expression, but by a set of search terms, the *keywords*. The user has a set of words in mind that best describe the document(s) she is looking for, and she sends these words to the search engine. The search engine then processes the given query by matching it against a set of documents or files. The keywords in a given search query can be combined in a number of different ways, for example according to the rules defined by Boolean algebra [14].

2.2.1 Boolean Queries

A Boolean search query [46] allows the user to combine keywords using one of the operators "$\wedge$" (AND), "$\vee$" (OR), and "$\neg$" (NOT). For example, the search query

$$\text{“keyword”} \wedge (\text{“query”} \vee \text{“search”}) \wedge (\neg\ \text{“retrieval”})$$

[7]http://www.google.com/codesearch (accessed 2007-01-25)


Live Search: endomorphic polynomial arachnophobia - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://search.live.com/results.aspx?q=endomo
Go
Red Hat, Inc.
Red Hat Network
Support
Shop
Products
Training
Live Search
Sign in
endomorphic polynomial arachnophobia
Web
Images
News
Maps
Classifieds
More
Done

endomorphic polynomial arachnophobia Page 1 of 1 results · Options · Advanced

Etymologie, Étymologie, Etymology - US Vereinigte Staaten von Amerika ...
... Arabian Sea | Arabic | arabica | Arabic numeral | arabinose | Arabist | Arabize | arable | Arab League | Aracaju | Araçatuba | arachidonic acid | Arachne | arachnid | arachnoid | **arachnophobia** | Arad ...
www.etymologie.info/~e/u_/us-diktio.html · Cached page

Didn't find an answer? Ask a real person on Live QnA.

Learn more about the new Live Search.

© 2007 Microsoft Privacy | Legal | For Site Owners Help Central | Account | Feedback

Figure 2.2: Boolean search query with Microsoft Live Search. Clearly, there are many documents in the index that contain one or two of the three query terms, only a single document that contains all three: "endomorphic", "polynomial", and "arachnophobia" (implicit Boolean AND).

matches all documents containing the term "keyword" and one of the terms "query" and "search" (or both), but not the term "retrieval". The Boolean operator that is most commonly used in keyword search is the AND operator, building the conjunction of two or more query terms. In fact, this operator is so common that, if a query just consists of a set of keywords, without any explicit operators, many search engines (and basically all Web search engines) automatically assume that these keywords are meant to be combined via the AND operator (see Figure 2.2 for an example).

The original version of Boolean document retrieval, with its three basic operators, has experienced many extensions, most notably the introduction of proximity operators [78].

Processing Boolean Queries

Because of the high prevalence of conjunctive Boolean queries in search engines, it is important to provide efficient support for this type of query.

Suppose the search engine needs to process a conjunctive Boolean query $\mathcal{Q}$, consisting of two query terms Q_1 and Q_2. Further suppose that it has already obtained two lists of document identifiers $\mathcal{D}_{Q_1}$ and $\mathcal{D}_{Q_2}$ from its index, representing the documents containing Q_1 and Q_2, respectively. $\mathcal{D}_{Q_1}$ and $\mathcal{D}_{Q_2}$ contain document identifiers in an arbitrary order, not necessarily sorted. Let $m := |\mathcal{D}_{Q_1}|$ and $n := |\mathcal{D}_{Q_2}|$, $m \leq n$ w.l.o.g.. Then the intersection of the two lists, denoted as $\mathcal{D}_{Q_1 \wedge Q_2}$, can be computed with

$$\Theta(m \cdot \log(m) + n \cdot \log(m))$$

comparison operations, by first sorting $\mathcal{D}_{Q_1}$ and then performing a binary search for every element in $\mathcal{D}_{Q_2}$ on the now-sorted list $\mathcal{D}_{Q_1}$. The time complexity of this procedure can be improved by performing the intersection through hash table lookup instead of sort-based comparison, leading to a total expected complexity of $\Theta(m + n)$. However, the fundamental limitation of this approach remains the same: Every element in each list has to be inspected at least once, implying a lower bound of $\Omega(m + n)$.

Now suppose that the elements of $\mathcal{D}_{Q_1}$ and $\mathcal{D}_{Q_2}$ are stored in the search engine's index in ascending order. Then their intersection can be computed more efficiently. A $\Theta(m + n)$ algorithm can be realized trivially, by simultaneous linear probing of both input lists ("co-sequential processing"), similar to the procedure employed in the merge step of the mergesort algorithm. If m is much smaller than n, then a faster solution is possible. Like before, for each element in $\mathcal{D}_{Q_1}$ a binary search on $\mathcal{D}_{Q_2}$ is performed, but now the search engine can make use of the information that $\mathcal{D}_{Q_1}[i] < \mathcal{D}_{Q_1}[i+1]$ when selecting the interval from $\mathcal{D}_{Q_2}$ on which the binary search for $\mathcal{D}_{Q_1}[i+1]$ takes place. The resulting algorithm, called `SvS`, is discussed by Demaine et al. [38]. `SvS` has a worst-case complexity of $\Theta(m \cdot \log(n))$. Thus, if $m \in o\left(\frac{n}{\log(n)}\right)$, then `SvS` is faster than linear probing. If $m \in \omega\left(\frac{n}{\log(n)}\right)$, and in particular if $m \in \Omega(n)$, it is slower.

SvS can be improved by choosing the sub-interval of $\mathcal{D}_{Q_2}$ dynamically by performing a local *galloping* search (iteratively doubling the size of the search interval before each binary search operation). This algorithm is discussed by Demaine et al. [37]. It is very similar to a solution to the unbounded search problem presented by Bentley and Yao [11]. The application of galloping search to the problem of intersecting two sorted lists is shown in Algorithm 1. The algorithm computes the intersection of two lists of length m and n, $m \leq n$, using $\Theta\left(m \cdot \log(\frac{n}{m})\right)$ comparisons (worst-case complexity). For all m, this is at least as fast as linear probing with its $\Omega(m + n)$ time complexity. It is strictly faster than linear probing if $m \in o(n)$.

Algorithm 1 Compute the intersection of two lists L_1 and L_2 by galloping search. The result is stored in the output parameter *resultList*.

```
Require: $L_1.length > 0 \wedge L_2.length > 0$
Require: $L_1$ and $L_2$ are sorted in increasing order
let $L_1[i] := \infty$ for $i > L_1.length$, $L_2[i] := \infty$ for $i > L_2.length$ by definition
$L_1.position \leftarrow 1$, $L_2.position \leftarrow 1$   // set position in both lists to beginning of list
$candidate \leftarrow L_1[L_1.position]$
$L_1.position \leftarrow L_1.position + 1$
$currentList \leftarrow L_2$   // start off by searching $L_2$ for an occurrence of $candidate$
while $candidate < \infty$ do
  if $currentList[currentList.position] \leq candidate$ then
    // try to find a range $lower \ldots (lower + delta)$ such that
    // $currentList[lower] \leq candidate \leq currentList[lower + delta]$
    $lower \leftarrow currentList.position$
    $delta \leftarrow 1$
    while $currentList[lower + delta] < candidate$ do
      $delta \leftarrow delta \times 2$
    end while

    // perform a binary search for $candidate$ on $currentList[lower \ldots upper]$
    $upper \leftarrow lower + delta$
    while $lower < upper$ do
      if $currentList[\lfloor (lower + upper)/2 \rfloor] < candidate$ then
        $lower \leftarrow \lfloor (lower + upper)/2 \rfloor + 1$
      else
        $upper \leftarrow \lfloor (lower + upper)/2 \rfloor$
      end if
    end while
    if $currentList[lower] = candidate$ then
      $resultList.append(candidate)$
      $currentList.position \leftarrow lower + 1$
    else
      $currentList.position \leftarrow lower$
    end if
  end if

  // swap the roles of the two lists $L_1$ and $L_2$
  $candidate \leftarrow currentList[currentList.position]$
  $currentList.position \leftarrow currentList.position + 1$
  if $currentList = L_1$ then
    $currentList \leftarrow L_2$
  else
    $currentList \leftarrow L_1$
  end if
end while
```

Regardless of the actual algorithm employed, performing a binary search on the sorted list $\mathcal{D}_{Q_2}$ usually only gives an improvement over the linear algorithm if the input lists are stored in main memory. If the lists are larger than the available amount of main memory, binary search in many cases will not offer a substantial speed advantage. In that case, however, it is even more important that the elements of both lists are sorted in some predefined order. Every other arrangement would necessarily lead to a non-sequential disk access pattern, jumping back and forth on disk, and probably slowing down the whole query processing procedure dramatically.

Even though the discussion presented so far has been limited to the case of intersecting lists of document identifiers in the context of Boolean retrieval, the same techniques can be used to resolve phrase queries if the search engine has access to an index that contains fully positional information about all term occurrences in the collection.

It is clear from the discussion that, if we want the search engine to process Boolean queries efficiently, then it is crucial that the list of all occurrences for a given term be stored in the index pre-sorted, preferably in the same order in which they appear in the text collection. In fact, this kind of index layout is not only required for Boolean queries, but also for phrase queries, proximity operators, structural query constraints, and for query optimizations in the context of ranked queries. This will have to be taken into account when choosing the underlying data structure of the search engine.

2.2.2 Ranked Queries

Pure Boolean queries perform remarkably well in some applications, such as legal search [108] and search operations on small text collections, but tend to be of limited use when the number of search results matching a given query becomes large. In this situation, the search engine must provide a way of ranking matching documents according to their predicted relevance to the given query. Lee [65] discusses how extended Boolean models, such as fuzzy set theory, can be used to rank the results to a search query. Clarke and Cormack [28] apply their shortest-substring retrieval framework to rank matching documents based on the length of the shortest passage that matches a given Boolean query.

As an alternative to the above approaches, Boolean query constraints and result ranking techniques may be viewed as two logically independent components of the search process. In a first step, Boolean search is used to find an initial set of candidate documents. In a second step, all matches found in the first step are ranked according to their predicted relevance to the query.

Salton et al. [98] [99] propose to use a *vector space model* to rank search results based on their similarity to the query. In their approach, the search query and all documents in the collection are treated as vectors from an n-dimensional vector space, where n is the number of different terms in the text collection. The similarity between a document and the search query is usually measured by their cosine distance, that is, the angle between the two vectors in the n-dimensional space:

$$\cos(\sphericalangle(\vec{x},\vec{y})) = \frac{\vec{x}^T\vec{y}}{||\vec{x}|| \cdot ||\vec{y}||}. \tag{2.1}$$

Thus, if both the query and the document are represented by unit vectors $\vec{x}$ and $\vec{y}$, then the score of a document is simply the value of the inner product: $\vec{x}^T\vec{y}$.

When the vector space model is used to rank search results, it is usually (though not necessarily) assumed that the Boolean query is a disjunction of all query terms. That is, every document containing at least one of the query terms is a potential candidate for retrieval.

Probabilistic Retrieval

The basic vector space model does not provide a theoretically satisfying way of assigning document scores based on how frequent a given query term is in the collection and how often it occurs in a given document. For example, in the original formulation of the vector space model, a document containing two occurrences of a query term Q_1 would receive the same score as a document that contains one occurrence of Q_1 and one occurrence of Q_2 (assuming both query terms are assigned the same weight). This shortcoming is rectified by the probabilistic approach to document retrieval [93]. Motivated by Robertson's probability ranking principle [92], which states that documents should be returned in order of probability of relevance to the query, probabilistic retrieval methods try to estimate this probability for each document, based on statistics available from the text corpus.

The predominant ranking function in the family of probabilistic retrieval methods, and the only probabilistic retrieval method used in this thesis, is Okapi BM25 [95] [94]. Given a query $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$, the BM25 score of a document D is:

$$\text{score}(D, \mathcal{Q}) = \sum_{Q \in \mathcal{Q}} \log\left(\frac{|\mathcal{D}|}{|\mathcal{D}_Q|}\right) \cdot \frac{f_D(Q) \cdot (k_1 + 1)}{f_D(Q) + k_1 \cdot (1 - b + b \cdot |D|/avgdl)}, \tag{2.2}$$

where $\mathcal{D}$ is the set of all documents, $\mathcal{D}_Q$ is the set of documents containing the term Q, $f_D(Q)$ is the number of occurrences of the query term Q in the document D, $|D|$ is the length of the document D (number of tokens), and *avgdl* is the average length of a document in the collection.

The parameter k_1 determines the fan-out of the within-document term frequency; b specifies the weight of the document length normalization component in BM25. The two parameters are usually set to their default values $(k_1, b) = (1.2, 0.75)$.

The first factor in Equation 2.2 is referred to as the *inverse document frequency* (IDF) component, while the second factor is referred to as the *term frequency* (TF) component. Consequently, BM25 is sometimes called a *TF-IDF* retrieval method.

Equation 2.2 is only one instance of a large class of functions covered under the name BM25. An in-depth analysis of this general class of ranking functions is provided by Spärck Jones et al. [59].

Language Modeling

A language model $\mathcal{M}$ is a probability distribution on a set of lexical objects, such as terms or term n-grams. The most common form is the unigram language model, defining a probability distribution on all terms over a given alphabet, e.g., Unicode:

$$\mathcal{M}: \quad \mathcal{V} \to [0,1], \quad T \mapsto \Pr(T \mid \mathcal{M}) \ ; \quad \sum_{T \in \mathcal{V}} (\Pr(T \mid \mathcal{M})) = 1, \tag{2.3}$$

where $\mathcal{V}$ is the set of all white-space-free Unicode strings. If built from a text collection $\mathcal{C}$ according to the maximum likelihood estimate, the probability of each term is simply its relative frequency in the collection:

$$\Pr(T \mid \mathcal{M}) = \frac{f_{\mathcal{C}}(T)}{\sum_x f_{\mathcal{C}}(x)}, \tag{2.4}$$

where $f_{\mathcal{C}}(T)$ is T's collection frequency — the number of times it occurs in $\mathcal{C}$.

The language modeling approach to information retrieval, as first proposed by Ponte and Croft [87], interprets each document D as a representative of a unigram language model $\mathcal{M}_D$ and assigns a relevance value score$(D, \mathcal{Q})$ to the document, depending on how likely it is for the search query Q to be generated from the language model $\mathcal{M}_D$.

Several variations of the language modeling approach exist, including but not limited to multiple-Bernoulli models [77] and multinomial models [105]. Within the scope of this thesis, I limit myself to the case of multinomial language models employing *Bayesian smoothing with Dirichlet priors*, analyzed by Zhai and Lafferty [116]. In this particular method, the probability of a term T according to the document model $\mathcal{M}_D$ is defined as:

$$\Pr_{\mu}(T \mid \mathcal{M}_D) = \frac{f_D(T) + \mu \cdot \Pr(T \mid \mathcal{C})}{\sum_x f_D(x) + \mu}, \tag{2.5}$$

where $f_D(T)$ is the number of times T appears in the document D, and $\mathcal{C}$ is the language model of the entire text collection (e.g., all files in the file system), usually computed according to the maximum likelihood heuristic. μ is a free parameter, realizing implicit document length normalization, and is normally chosen as $\mu = 2000$.

Given a query $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$, the relevance score of a document D then is:

$$\text{score}(D, \mathcal{Q}) = \prod_{Q \in Q} \Pr_\mu(Q \mid \mathcal{M}_D) = \prod_{Q \in Q} \frac{f_D(Q) + \mu \cdot \Pr(Q \mid \mathcal{C})}{\sum_x f_D(x) + \mu}. \tag{2.6}$$

The higher the probability of the query being generated from the language model $\mathcal{M}_D$, the greater the score that the document receives.

Proximity Ranking and Passage Retrieval

The ranking methods described above have in common that they rank documents purely based on the number of occurrences of each query term in a document and in the entire collection, without taking into account the relationship between different query terms, such as the distance between two query term occurrences in the same document. For this reason, they are sometimes referred to as *bag-of-words* retrieval models.

One essential limitation of bag-of-words retrieval functions is that they can only rank and return entire documents. For example, if a user is searching the contents of a set of PDF documents, then the ranking function is always applied on an entire PDF document. This approach has two shortcomings. First, the document that a user is looking for might not be relevant to the search query in its entirety, but only partially. This can lead to a sub-optimal ranking of the search results. Second, a matching document might be very long, maybe a few hundred pages. In that case, reporting the entire document back to the user is probably not going to be of much help. Therefore, it is desirable to use a retrieval method that, in addition to ranking the matching documents, also returns information about the exact part of a matching document that best represents the information need expressed by the query.

One such method is the QAP algorithm presented by Clarke et al. [33]. Given a query $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$ and a text passage P of length l, containing one or more query terms, P's relevance score is determined based on its probability of containing the query term(s) by chance. If $\mathcal{Q}' \subseteq \mathcal{Q}$ is the set of query terms that appear in P, then the probability of their occurring by pure chance is:

$$\Pr(\mathcal{Q}' \mid \text{length}(P) = l) = \prod_{Q \in \mathcal{Q}'} (1 - (1 - \Pr(Q \mid \mathcal{C}))^l) \;\approx\; \prod_{Q \in \mathcal{Q}'} l \cdot \Pr(Q \mid \mathcal{C}), \tag{2.7}$$

where $\Pr(Q \mid \mathcal{C})$ is Q's probability of occurrence according to the collection-wide unigram language model, built according to the maximum likelihood estimate: $\Pr(Q \mid \mathcal{C}) = \frac{f_{\mathcal{C}}(Q)}{|\mathcal{C}|}$. The score of the passage P is usually computed as the query-specific information content of the passage:

$$\text{score}(P, \mathcal{Q}) = - \sum_{Q \in (\mathcal{Q} \cap P)} \left(\log \left(\frac{f_{\mathcal{C}}(Q)}{|\mathcal{C}|} \right) + \log(l) \right). \tag{2.8}$$

QAP finds the highest-scoring passage in each matching document and returns it to the user (where a *passage* is an arbitrary-length sequence of tokens). Documents may be ranked according to the score of their highest-scoring passage, but may also be ranked according to some other criterion. In the latter case, QAP is run as a postprocessing step on a small number of documents (the top k search results) returned by some other ranking function (e.g., BM25) in order to find passages that can be used to construct query-specific document summaries (*snippets*).

As an alternative to Clarke's method, Büttcher et al. [22] and Rasolofo and Savoy [89] discuss how term proximity features can be integrated directly into the BM25 scoring function. Without further modifications, however, these methods cannot be used for passage retrieval. Other alternatives are explored by de Kretser and Moffat [36].

Query-Independent Ranking

Sometimes it can be useful to combine the query-specific score of a document with a query-independent score component that represents the general importance of the document. Such query-independent evidence of a document's relevance can be integrated fairly easily into the language modeling approach, by defining prior probabilities for document relevance (usually referred to as *document priors*) [63]. It is known that query-independent score components are essential for effective navigational Web search, such as homepage finding [62]. They are assumed to form the basis of most commercial Web search engines (e.g., Kleinberg's HITS algorithm [61] and Google's PageRank [85]).

Inspired by the success of PageRank, Bhagwat and Polyzotis [12] propose FileRank, a method that – similar to PageRank, but to be used in the context of file system search instead of Web search – estimates the query-independent importance of a file. The file rank of a given file F depends on three components: the content that F shares with other files ("content overlap link"), the overlap between F's name and the name of other files ("name overlap link"), and the overlap between F's name and the content of other files ("name reference link"). Unfortunately, Bhagwat and Polyzotis neither give a sound theoretical justification for their particular ranking function, nor do they provide experimental support showing that their method is actually able to produce high-quality search results. It seems appropriate to remain suspicious.

What appears more promising is to include the type of a matching file as a feature in the ranking function. Similar to the method proposed by Yeung et al. [115], the search engine could exploit user interaction, such as click-through, to build a model of file types that a user is more likely or less likely to be interested in. This information can then be used at query time to adjust the score that a particular file receives.

Processing Ranked Queries

When processing a ranked search query, it is reasonable to assume that the user is not interested in a complete ranking of all matching documents, but only in the top k documents, for some small value of k, say $k = 20$. For most bag-of-words ranking functions, such as Okapi BM25 and multinomial language models, a search query can then be processed in two different ways: following a document-at-a-time approach, or following a term-at-a-time approach.

A term-at-a-time query processing strategy fetches each query term's posting list (i.e., the list of all its occurrences) from the index, one list at a time, starting with the least-frequent term (which also carries the greatest weight), and computes the contribution of each list element to the score of the document it refers to. This results in a set of partial document scores, stored in in-memory *accumulators*. If the number of matching documents – documents with non-zero accumulators – is small, then this method works well without any further modifications. However, if the number of matching documents exceeds the amount of memory that can be allocated for in-memory accumulators, then accumulator pruning strategies need to be applied.

Moffat and Zobel [81] present two strategies (Quit and Continue) that can rank documents using a limited, pre-defined amount of space for in-memory score accumulators, usually between 1% and 5% of the number of documents in the collection. The Quit strategy stops execution as soon as the accumulator limit is reached. The Continue strategy allows existing accumulators to be updated after the limit is reached, but does not allow new accumulators to be created.

Lester et al. [67] notice that the Continue strategy suffers from *accumulator bursts* that can result in a number of in-memory accumulators that exceed the pre-defined limit by a large margin. They propose an adaptive pruning technique that does not exhibit this shortcoming and that offers a better efficiency-vs.-effectiveness trade-off than Continue.

If each posting list is stored in a contiguous region of the hard disk, then the term-at-a-time approach leads to high query processing performance, since it only requires a single random disk access (i.e., disk seek) per query term. Unfortunately, if the scoring function makes use of term proximity features, then the score of a document is no longer just the sum of each query term's contribution. Thus, the query cannot be processed in a term-at-a-time fashion. The search engine has to resort to a document-at-a-time strategy.

A document-at-a-time query processing strategy loads all lists into memory and arranges them in a priority queue, such as a MIN-HEAP, sorted by the position of the next term occurrence in each list (alternative approaches are explored by Vo and Moffat [4]). By repeatedly removing the first element from the posting list that is currently at the beginning of the priority queue, the postings for all query terms can be inspected in the order in which they occur in the collection, and exact document scores for all documents containing at least one query term can be computed on the fly. In order for this to be possible, however, the postings in each list need to be stored in the index pre-sorted in collection order.

If the query terms' posting lists are small enough to be loaded entirely into main memory, then document-at-a-time query processing requires the same number of disk seeks as the term-at-a-time approach. If, however, lists are too long to be loaded into RAM completely, the query processor can only load partial posting lists into memory and needs to switch back and forth between the different lists in order to fetch more postings for the respective query term. In that case, term-at-a-time methods are usually more efficient than the document-at-a-time approach.

The most important query optimization technique for document-at-a-time query processors is the MAXSCORE heuristic due to Turtle and Flood [109]. The main assumption behind MAXSCORE is that the search engine does not report all matching documents back to the user, but only the k top-scoring documents. Suppose a set of documents is to be ranked according to their BM25 scores for the query {"information", "retrieval"}. Further suppose that the IDF weights of the two query terms are

$$w(\text{“information”}) = 2.06 \quad \text{and} \quad w(\text{“retrieval”}) = 5.51$$

and that, while processing the query, the kth-best document seen so far has received the BM25 score $s_k = 4.6$. If BM25 is executed with default parameters $(k_1, b) = (1.2, 0.75)$, then the maximum score that can be achieved by a document that only contains the term "information" is at most

$$s_{\max}(\text{“information”}) = w(\text{“information”}) \cdot (k_1 + 1) = 2.06 \cdot 2.2 = 4.532 \qquad (2.9)$$

(cf. Equation 2.2). Therefore, the query processor no longer needs to consider documents that only contain the term “information”, but not the term “retrieval”. For longer queries, this technique can be generalized to eliminate the n least-weight query terms instead of just a single one.

MaxScore can be applied to most ranking functions, including all three functions described above (BM25, QAP, LM with Dirichlet smoothing). In the experiments reported on by Turtle and Flood, MaxScore can save up to 50% of the total query execution cost when the search engine only needs to return the top 10 documents to the user.

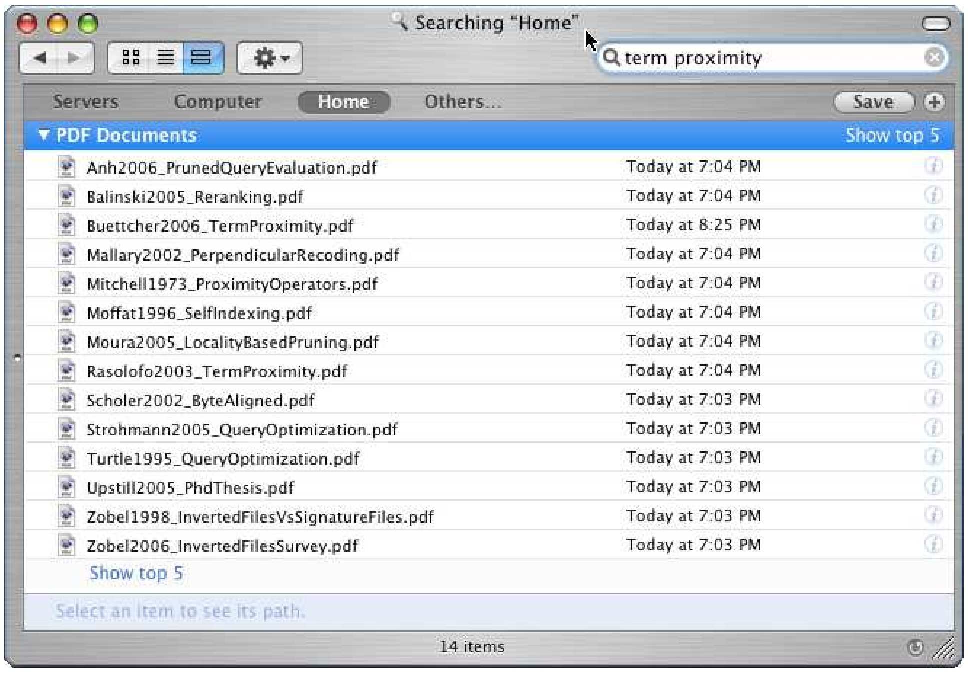
In order for MaxScore to be truly advantageous, the search engine has to be able to skip parts of a posting list and jump directly to the next posting that refers to a potentially top-scoring document. This is usually realized in a fashion that is very similar to the galloping search described when discussing Boolean queries. At the index level, it can be supported by inserting synchronization points into each posting list, as suggested by Moffat and Zobel [81]. Like in the case of Boolean queries, this optimization requires that the elements of a given posting list be stored in the index in the order in which they appear in the collection.

2.2.3 Boolean vs. Ranked Retrieval in File System Search

Appropriate query types and ranking techniques in the context of desktop and file system search have not been studied very well yet. It is therefore not entirely clear which retrieval strategy – Boolean retrieval or relevance ranking – yields the best results in this new context. Preliminary results do, however, indicate that pure Boolean retrieval, not taking any statistical information into account when ranking a set of matching documents, leads to low-quality search. For their *Stuff I’ve Seen* desktop search engine, Dumais et al. [39] report that the majority of the users chose to order search results either by their creation/modification date (recent files first) or by their relevance score (using a variant of Okapi BM25).

Among the commercial desktop search engines, there seems to be disagreement regarding the best way to rank search results. While Google’s and Microsoft’s products perform some sort of statistical relevance ranking, Apple’s Spotlight simply presents the search results in alphabetical order (see Figure 2.3 for an example).

I simulated a search task in which a user is trying to locate a document that she had read some time ago and that had been misplaced somewhere in the vastness of her file system. The document in question is a research paper on the use of term proximity features in document retrieval, and she remembers that it contains the two keywords “term” and “proximity”. Figure 2.3 shows the results to her search query, as produced by Spotlight (applying a Boolean AND and ranking by file name). In contrast, Figure 2.4 shows the search results produced by the Wumpus search engine (applying a Boolean OR and ranking documents according



Anh2006_PrunedQueryEvaluation.pdf	Today at 7:04 PM
Balinski2005_Reranking.pdf	Today at 7:04 PM
Buettcher2006_TermProximity.pdf	Today at 8:25 PM
Mallary2002_PerpendicularRecoding.pdf	Today at 7:04 PM
Mitchell1973_ProximityOperators.pdf	Today at 7:04 PM
Moffat1996_SelfIndexing.pdf	Today at 7:04 PM
Moura2005_LocalityBasedPruning.pdf	Today at 7:04 PM
Rasolofo2003_TermProximity.pdf	Today at 7:04 PM
Scholer2002_ByteAligned.pdf	Today at 7:03 PM
Strohmann2005_QueryOptimization.pdf	Today at 7:03 PM
Turtle1995_QueryOptimization.pdf	Today at 7:03 PM
Upstill2005_PhdThesis.pdf	Today at 7:03 PM
Zobel1998_InvertedFilesVsSignatureFiles.pdf	Today at 7:03 PM
Zobel2006_InvertedFilesSurvey.pdf	Today at 7:03 PM

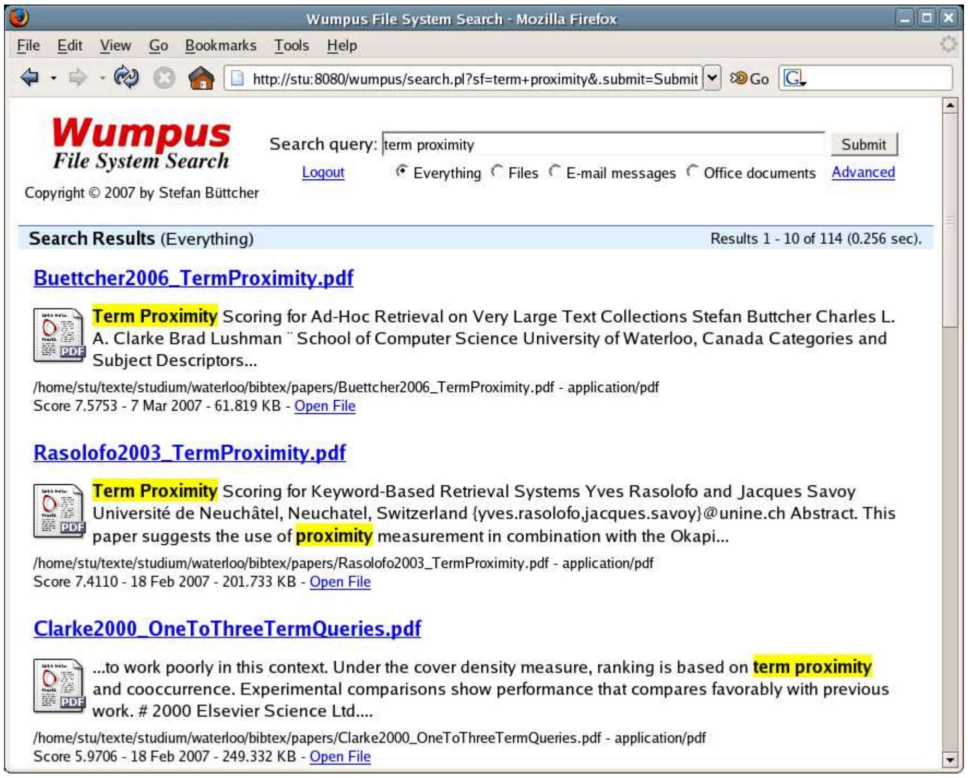
Figure 2.3: File system search with Apple's Spotlight. Search query: "term proximity". Search results are ranked alphabetically, without taking their relevance into account.

to their BM25TP [22] score, a variant of BM25). By comparing the two figures, it is clear that, without relevance ranking, the user might need to spend some time browsing the search results until she finally finds the paper she was interested in. With relevance ranking, on the other hand, it is likely that the document in question is one of the top three documents shown in Figure 2.4.

Although this example can only be considered anecdotal evidence of the necessity of relevance ranking techniques, it does indicate that there are in fact some search scenarios in which the application of statistical ranking techniques can improve the quality of the search results dramatically. Since relevance ranking in general is rather unlikely to degrade the quality of the search results, compared to a pure Boolean approach, it seems appropriate for any file system search engine to support it by default.

2.2.4 Structural Constraints

Sometimes, it is useful to be able to specify certain structural constraints that a document needs to satisfy in order to match the given query. For example, a user might be interested in e-mails coming from a University of Waterloo e-mail address that are not concerned with


Wumpus File System Search - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://stu:8080/wumpus/search.pl?sf=term+proximity&.submit=Submit
Go
Wumpus
File System Search
Copyright © 2007 by Stefan Büttcher
Submit
Logout
Advanced

Search query: term proximity

Everything [✓] Files E-mail messages Office documents

Search Results (Everything) Results 1 - 10 of 114 (0.256 sec).

Buettcher2006_TermProximity.pdf

Term Proximity Scoring for Ad-Hoc Retrieval on Very Large Text Collections Stefan Buttcher Charles L. A. Clarke Brad Lushman ¨ School of Computer Science University of Waterloo, Canada Categories and Subject Descriptors...

/home/stu/texte/studium/waterloo/bibtex/papers/Buettcher2006_TermProximity.pdf - application/pdf
Score 7.5753 - 7 Mar 2007 - 61.819 KB - Open File

Rasolofo2003_TermProximity.pdf

Term Proximity Scoring for Keyword-Based Retrieval Systems Yves Rasolofo and Jacques Savoy Université de Neuchâtel, Neuchatel, Switzerland {yves.rasolofo,jacques.savoy}@unine.ch Abstract. This paper suggests the use of proximity measurement in combination with the Okapi...

/home/stu/texte/studium/waterloo/bibtex/papers/Rasolofo2003_TermProximity.pdf - application/pdf
Score 7.4110 - 18 Feb 2007 - 201.733 KB - Open File

Clarke2000_OneToThreeTermQueries.pdf

...to work poorly in this context. Under the cover density measure, ranking is based on term proximity and cooccurrence. Experimental comparisons show performance that compares favorably with previous work. # 2000 Elsevier Science Ltd....

/home/stu/texte/studium/waterloo/bibtex/papers/Clarke2000_OneToThreeTermQueries.pdf - application/pdf
Score 5.9706 - 18 Feb 2007 - 249.332 KB - Open File

Figure 2.4: File system search with Wumpus. Search query: "term proximity". Search results are ranked by their predicted relevance to the query (using BM25TP [22]).

fee payment:

$$
\begin{aligned}
& ((\text{“}\langle\text{email}\rangle\text{”} \cdots \text{“}\langle/\text{email}\rangle\text{”}) \rhd ((\text{“}\langle\text{from}\rangle\text{”} \cdots \text{“}\langle/\text{from}\rangle\text{”}) \rhd \text{“uwaterloo.ca”})) \\
\not\rhd\ & ((\text{“}\langle\text{subject}\rangle\text{”} \cdots \text{“}\langle/\text{subject}\rangle) \rhd \text{“fee*”})
\end{aligned}
$$

(where "$\rhd$" denotes *containment* relationship, and "$\cdots$" denotes *followed-by*). Many such requirements can be expressed using XML-based query languages, such as XPath[8] and XQuery[9]. A more light-weight way of enforcing structural constraints was introduced by the Pat algebra [97] and improved upon by Clarke et al. [30] [31]. Their GCL query language is based on the shortest-substring paradigm, stating that a result to a search query (including intermediate results) must never contain a substring that also matches the query. GCL supports operators for containment, followed-by, Boolean AND, and Boolean OR.

In this thesis, a set of seven GCL operators is made use of. Let A and B be GCL expressions, and let $\mathcal{I}_A$ and $\mathcal{I}_B$ denote the set of intervals (text passages) matching A and B,

[8]http://www.w3.org/TR/xpath (accessed 2007-02-14)
[9]http://www.w3.org/TR/xquery (accessed 2007-02-14)

respectively. Then:

- $A \cdots B$ matches all intervals starting with some $I_A \in \mathcal{I}_A$ and ending with some $I_B \in \mathcal{I}_B$;
- $A \vee B$ matches every interval $I \in (\mathcal{I}_A \cup \mathcal{I}_B)$;
- $A \wedge B$ matches every interval I that contains two sub-intervals $I_A \in \mathcal{I}_A$ and $I_B \in \mathcal{I}_B$;
- $A \rhd B$ matches every interval $I_A \in \mathcal{I}_A$ that contains an interval $I_B \in \mathcal{I}_B$;
- $A \not\rhd B$ matches every interval $I_A \in \mathcal{I}_A$ that does not contain an interval $I_B \in \mathcal{I}_B$;
- $A \lhd B$ matches every interval $I_A \in \mathcal{I}_A$ that is contained in an interval $I_B \in \mathcal{I}_B$;
- $A \not\lhd B$ matches every interval $I_A \in \mathcal{I}_A$ that is not contained in an interval $I_B \in \mathcal{I}_B$.

For details on these operators, see Clarke et al. [31]. Note that the notation used here is slightly different from the one chosen by Clarke.

For flexibility reasons, implementations of the GCL text retrieval framework are usually, though not necessarily, based on a schema-independent index (cf. Section 2.4). Because the structure of the text collection – explicitly encoded in a schema-dependent index – can easily be reconstructed at query time by using a combination of GCL operators, using a schema-independent index results in more powerful query semantics.

The result of a GCL query is a list of *index extents* (i.e., passages of arbitrary length in the indexed text collection), without any explicit ranking. If the application employing the GCL framework requires a ranked list of matches (e.g., the top k documents), then the ranking step is performed outside the GCL retrieval framework, according to explicitly specified rules, such as probabilistic retrieval [95] [94] or Clarke's shortest-substring ranking method [28] that unifies Boolean queries and ranked retrieval within the GCL framework.

GCL forms the basis of the security-aware query processor described in Chapter 5. Like in the case of Boolean queries and ranked queries, the performance of the GCL operators relies on the fact that all postings for a given term are stored in the index in collection order.

2.3 Index Data Structures

Over the last 30 years, various data structures for efficient full-text search have been proposed:

- The **inverted index** [53] is the dominant data structure used to realize keyword-based search operations. For every term in a given text collection, the inverted index contains a list of all its occurrences, sorted by their position in the collection. Depending on the granularity of the index, it might either contain lists of exact word positions, or just lists of document identifiers, representing the documents in which a given term appears.

- A **suffix tree** [110] for a given text collection is a search tree with the special property that every path from the tree's root to one of its leaves corresponds to a unique suffix of the text collection. Hence, every path from the tree's root to one of its nodes corresponds to a substring of the collection. Suffix trees, therefore, can be used to realize efficient phrase queries and in this respect are more flexible than inverted files, which only support single-term lookups.

  If used for keyword search, however, suffix trees have two essential limitations. First, they are usually larger than inverted indices and less efficient when stored on disk. Second, they produce result lists that are sorted in lexicographical order and not in the order in which they appear in the text collection. While the first limitation to some extent can be addressed through more complicated encodings [60] [72] [27], the second limitation remains, and renders suffix trees impractical for Boolean search queries (cf. Section 2.2.1).

- A **suffix array** [52] [75] is an array of pointers into the text collection. Each pointer corresponds to a unique suffix of the collection. The pointers are sorted in lexicographical order of their respective suffixes. Hence, phrase queries can be supported via a binary search operation on the list of suffixes. When used for keyword search applications, suffix arrays essentially have the same limitations as suffix trees.

- In a **signature file** [42], every term T is represented by a vector $s(T) \in \{0,1\}^n$, for some $n \in \mathbb{N}$. Likewise, each document in the text collection is represented by a vector $s(D) \in \{0,1\}^n$, its *signature*:

  $$s(D)[i] = \max_{T \in D}\{s(T)[i]\}. \tag{2.10}$$

  That is, $s(D)$ is the Boolean OR of the signatures of all terms found in the document. Searching for a document containing a term T^* is realized by searching for a document signature $s(D^*)$, such that

  $$s(D^*)[i] \geq s(T^*)[i] \quad \text{for } 1 \leq i \leq n. \tag{2.11}$$

  This operation can be realized fairly efficiently by means of bit slicing [41].

  Similar to a Bloom filter [13], a signature file cannot be used to produce definite matches, but only to rule out non-matching documents. That is, there is always the chance of false positives that need to be eliminated by inspecting the actual document reported back as a potential match. Moreover, because it does not contain any positional information, a signature file cannot be used for phrase searches or search operations that take the proximity of query terms into account.

Zobel et al. [120] argue that signature files offer inferior performance compared to inverted files and that inverted files are the data structure of choice. Carterette and Can [24] argue in the opposite direction, pointing out applications in which signature files might be appropriate.

Given that suffix trees and suffix arrays do not store the list of occurrences of a given term in the order in which they appear in the text collection (and in fact *cannot* do so without substantially increasing the amount of redundancy in the index), it is impossible to use the query processing strategies presented in Section 2.2 if the search engine is based on a suffix tree or suffix array. Therefore, under the assumption that the main focus in the context of file system search is on efficient support for keyword queries, including phrase queries and Boolean queries, the indexable contents of the file system are best represented by an inverted index. A discussion of whether this assumption actually holds or whether other types of queries need to be supported as well, is given in Section 4.1.

2.4 Inverted Indices

The *inverted index* [53] [119] (also referred to as *inverted file* or *postings file*) is the fundamental data structure employed by most text search engines. The inverted index for a given text collection realizes a mapping from each term T to a list of T's occurrences within the collection. The list of T's occurrences in the collection is called the term's *inverted list* (or its *posting list*); each element of such a list is called a *posting*.

Based on the amount of information stored in an inverted index, and based on the structure of the index, we can distinguish between four different types of inverted files:

- In a **docid index**, each posting is a simple integer — a numeric document identifier representing a single document.
- A **frequency index** contains postings of the form *(docid, tf)*, the document that a term appears in and the number of times it appears in that document (the *term frequency*).
- Postings in a **document-centric positional index** are of the form

$$(\textit{docid},\ \textit{tf},\ \langle\ p_1, \ldots, p_{tf}\ \rangle),$$

where the p_i are the word positions of all occurrences of the given term within the document referred to by *docid*.

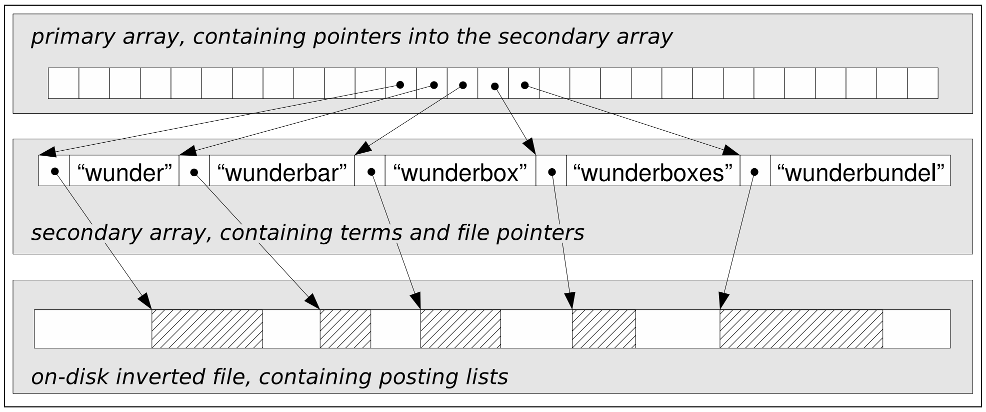

primary array, containing pointers into the secondary array
"wunder"
"wunderbar"
"wunderbox"
"wunderboxes"
"wunderbundel"
secondary array, containing terms and file pointers
on-disk inverted file, containing posting lists

Figure 2.5: Sort-based dictionary with secondary array to reduce internal fragmentation. The per-term pointer overhead of the additional level of indirection is between 3 and 7 bytes (32-bit/64-bit pointers).

- A **schema-independent index** contains word-level occurrence information, just like a positional index, but does not make explicit reference to documents in the collection. Instead, each posting is a simple integer, encoding the word position of a term occurrence, relative to the very beginning of the text collection.

For the same reasons that already ruled out signature files (no support for phrases; no support for proximity queries), docid indices and frequency index are not appropriate for use in file system search. The question of whether a document-centric positional index or a schema-independent index is the better data structure for full-text file system search – both support phrase queries and proximity search operations – shall be postponed until Chapter 4. For now, I assume that we are dealing with schema-independent indices exclusively. Thus, whenever I refer to an inverted index, I mean a schema-independent inverted file.

In addition to the data stored in the posting lists, an inverted index consists of a dictionary, providing a mapping from each term to the position of its posting list in the index, and a document map that can be used to translate the contents of the posting lists back into their original context, that is, into textual document identifiers, file names, or URLs.

2.4.1 Dictionary

The mapping from terms to the position of their respective posting list is realized through the *dictionary.* If the dictionary is stored in memory, it is usually implemented as a hash table or a sort-based lookup data structure, such as a binary search tree or a sorted array.

If the dictionary is implemented as a sorted array of all terms, then it is essential that all terms occupy an equal amount of space in the search array (otherwise, binary search operations may become quite complicated). Because terms are of different lengths, this may lead to internal fragmentation (a term uses less space than allocated). This problem is usually dealt with by introducing one level of indirection, as shown in Figure 2.5. The search array does not contain the dictionary terms themselves, but fixed-size (32-bit or 64-bit) pointers into a secondary array that contains the actual term strings. If each pointer consumes 4 bytes, then the per-term overhead of this method is 3 bytes (adding 4 bytes for the pointer, but also saving 1 byte per term because explicit term delimiters are no longer necessary). On the other hand, internal fragmentation is eliminated entirely.

This way of storing terms in the dictionary is sometimes referred to as the *dictionary-as-a-string* approach [114, Section 4.1], because all terms are stored as one long string, possibly interleaved by file pointers (as shown in the figure).

It is not uncommon for a search engine to maintain two different dictionaries, one for index construction the other for query processing. In that case, the indexing-time dictionary, needed to perform a large number of extremely low-latency dictionary lookups, is usually based on a hash table implementation, while the query-time dictionary is sort-based.

2.4.2 Posting Lists

Posting lists may be stored either in memory or on disk. Within the context of file system search, it is unreasonable to assume that all postings can be stored in the computer's main memory. Instead, the majority of all postings data will typically be stored on disk. In order to minimize disk access latency, it is then advisable to store each list in a contiguous region of the hard disk.

Random access into a term's on-disk posting list, useful for query optimizations like galloping search and MaxScore (cf. Section 2.2), is facilitated by maintaining a local index for the contents of the list. This can be realized either by arranging each posting list in a static B-tree, or by employing a simple two-level search data structure that contains the value and the file position of every j-th posting in the list (typically, $2^8 \leq j \leq 2^{16}$) [81]. Because these entries into a posting list are normally used in conjunction with index compression techniques, they are also referred to as *synchronization points*.

2.4.3 Index Construction

A text collection can be thought of as a sequence of term occurrences — tuples of the form

$$(term, position),$$

Input text:

How much wood would a woodchuck chuck if a woodchuck could chuck wood?

Token sequence (in collection order):

(“how”, 1), (“much”, 2), (“wood”, 3), (“would”, 4), (“a”, 5), (“woodchuck”, 6), (“chuck”, 7), (“if”, 8), (“a”, 9), (“woodchuck”, 10), (“could”, 11), (“chuck”, 12), (“wood”, 13)

Token sequence (in index order):

(“a”, 5), (“a”, 9) (“chuck”, 7), (“chuck”, 12), (“could”, 11), (“how”, 1), (“if”, 8), (“much”, 2), (“wood”, 3), (“wood”, 13), (“woodchuck”, 6), (“woodchuck”, 10), (“would”, 4)

Figure 2.6: The main task of an index construction process is to rearrange the token sequence that constitutes the text collection — to transform it from collection order into index order (here for a schema-independent representation of the text collection, without any explicit structure).

where *position* is the word count from the beginning of the collection, and *term* is the term that occurs at that position. Figure 2.6 shows an example text collection under this interpretation. Naturally, when a text collection is read in a sequential manner, these tuples are ordered by their second component, their position in the collection. The task of constructing an index is to change the ordering of the tuples so that they are sorted by their first component, the term they refer to (ties are broken by the second component). Once the new order has been established, and postings are stored on disk or in memory in this new order, creating a proper index (adding the additional data structures described in the previous sections) is a comparatively easy task.

Index construction methods can be divided into two categories: in-memory index construction, and disk-based index construction. In-memory index construction methods build an index for a given text collection entirely in main memory and can therefore only be used if the text collection is small relative to the amount of available main memory. They do, however, form the basis for more sophisticated, disk-based index construction methods. Disk-based methods can be used to build indices for very large collections, much larger than the available amount of main memory.

Algorithm 2 In-memory index construction algorithm, making use of two abstract data types: in-memory dictionary and extensible postings lists.

```
1: $currentPosting \leftarrow 1$
2: while there are more tokens to be processed do
3:     $T \leftarrow$ next input token
4:     obtain dictionary entry for $T$; create new entry if necessary
5:     append new posting $currentPosting$ to the posting list for term $T$
6:     $currentPosting \leftarrow currentPosting + 1$
7: end while
8: sort dictionary entries in lexicographical order
9: for each term $T$ in the dictionary do
10:     write $T$'s inverted list to disk
11: end for
12: write the dictionary to disk
```

In-Memory Index Construction

An in-memory index construction algorithm relies on the availability of two fundamental data structures:

- an in-memory dictionary that can be used to efficiently find the memory address of the posting list for a given term;
- an extensible list data structure that provides efficient support for append operations and that is used to accumulate the postings for a given term in memory.

If these two data structures are available, then the index construction process is straightforward (cf. Algorithm 2).

The dictionary can be realized by any data structure that supports efficient lookup and insertion operations, such as a binary tree or a hash table. However, Zobel et al. [118] make a strong case for hash tables with linear chaining and *move-to-front heuristic*. The move-to-front heuristic moves a dictionary entry to the front of its respective chain whenever it is accessed, thus ensuring that frequent terms (like "the") stay close to the front of their chain. According to the experiments discussed in [118], such a hash table is between 3 and 5 times faster than a binary search tree, even if the number of slots in the table is considerably smaller than the number of terms in the dictionary.

Unfortunately, hash tables have the disadvantage that vocabulary terms are not stored in lexicographical order. This complicates the processing of prefix queries, such as "inform$*$", matching all words starting with "inform". Thus, if the same dictionary is used to construct the index and to process search queries, a hash table might not be the best choice. Heinz et al. [56] therefore suggest to use a *burst trie* instead of a hash table. A burst trie is a hybrid data structure, consisting of a number of binary search trees, arranged in a trie. All terms in the same binary tree share a common prefix, as defined by the path from the trie's root node to the root of the binary tree. A binary tree is *burst* into two or more different trees when it becomes too large (hence the name of the data structure). Because, in a burst trie, terms are arranged in lexicographical order, prefix search operations can be supported efficiently. Regarding their performance in an index construction task, Heinz et al. [56] report that burst tries are about 25% slower than move-to-front hash tables and have very similar memory requirements.

The second aspect of in-memory index construction, the extensible in-memory posting lists required by Algorithm 2 can be realized in various different ways. Fox [44], for example, proposes not to use extensible lists at all, but to employ a two-pass indexing strategy. The first pass is used to gather collection statistics and to determine the size of each term's posting list. In the second pass, the space allocated for each list is filled with postings data. The disadvantage of this two-pass indexing method is that the collection needs to be read twice, adversely affecting indexing performance and rendering the method unusable in environments where real-time (or near-real-time) index updates are required.

In a single-pass index construction method, the final size of the inverted list for a given term is unknown, and each list actually has to be extensible. One possibility then is to use a linked list data structure (cf. Moffat and Bell [79]). However, this increases the overall memory requirements of the indexing process by about 100%. As an alternative, an initial buffer might be allocated for each list. Whenever the capacity of the buffer is reached, its size is increased by relocating the data in the buffer to a new memory location, where there is more space for the term's postings. This is the strategy employed by Heinz and Zobel [55]. In their implementation, which uses `realloc`[10] to relocate in-memory posting lists[11], a user-defined pre-allocation factor k is set such that, whenever a list is relocated, the new buffer is k times as large as the old buffer (Heinz and Zobel use $k = 2$). This guarantees that the total number of relocations per list is logarithmic in the total size of the list, and the number of bytes copied during relocation operations is linear in the size of the list.

Sort-Based Index Construction

The main limitation of the in-memory index construction method outlined above is that it cannot be used to create an index that is larger than the available amount of main memory.

[10] http://www.gnu.org/software/libc/manual/html_node/Changing-Block-Size.html (accessed 2007-02-22)
[11] Justin Zobel, private communication (August 2005)

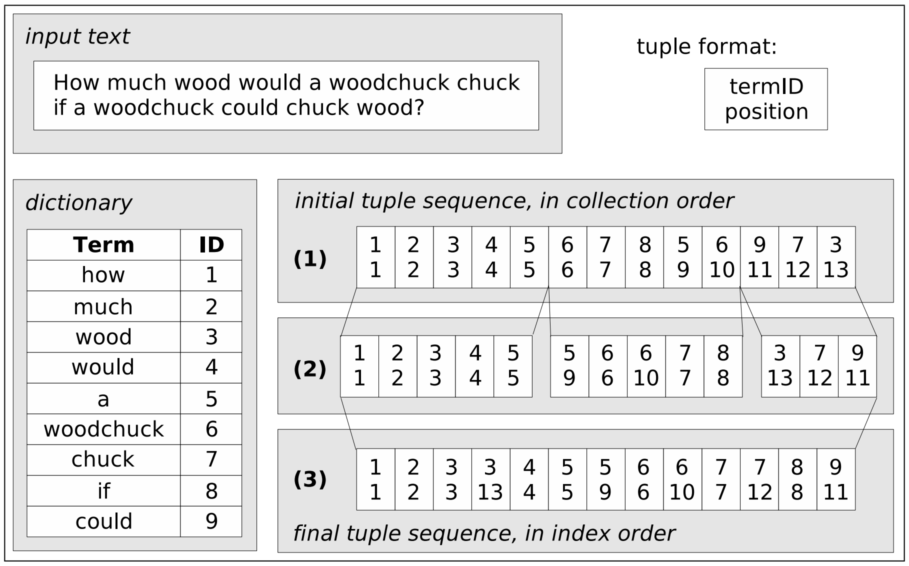


Figure 2.7: Sort-based index construction with global term IDs. Main memory is large enough to hold 5 (*termID*, *position*) tuples at a time. (1)→(2): sorting blocks of size ≤ 5 in memory, one at a time. (2)→(3): merging sorted blocks into bigger block.

With sort-based index construction, the size of the index is not limited by the available amount of main memory. Conceptually, sort-based indexing solves the problem of creating an inverted file in the simplest way possible: for every incoming tuple of the form

$$(term, position),$$

a new tuple of the form

$$(termID, position)$$

is written to disk. An in-memory dictionary is used to translate each term into a unique numerical termID value. After the entire collection has been read from disk, a disk-based sorting algorithm is used to re-arrange the on-disk tuples from their original collection order into their target index order. This is done by loading up to M tuples into memory at a time (where M is defined by the amount of main memory available to the indexing process) and sorting them in memory. After all $\lceil N/M \rceil$ such blocks have been sorted, a final sequence of sorted tuples is produced by merging the $\lceil N/M \rceil$ blocks in a multi-way merge operation. This procedure is shown in Figure 2.7.

After the (*termID*, *position*) tuples that represent the text collection have been brought from their original collection order into their new index order, creating a proper index, adding a dictionary and some other auxiliary data structures, is straightforward.

The sort-based index construction described above has several shortcomings, such as its excessive disk space requirements. Most of these shortcomings can be (and have been) addressed. For example, Moffat and Zobel [80] employ compression techniques to reduce the amount of disk space consumed by the M-tuple blocks. Moffat and Bell [79] describe an in-situ merge mechanism that can be used in the last phase (i.e., (2)$\rightarrow$(3) in the figure) to create the final index using only a constant amount of temporary disk space, reducing space requirements even further.

The only limitation that seems unsolvable is that, in order to transform each incoming *term* into a *termID*, the complete vocabulary of the text collection has to be kept in memory. For a collection comprising many millions of different terms, this can easily exceed the amount of memory that the indexing process is allowed to use.

Merge-Based Index Construction

Merge-based indexing methods avoid the high memory requirements seen in sort-based index construction by not using globally unique term identifiers at all. Each vocabulary term is its own identifier. When building an index for a given text collection, a merge-based indexing algorithm starts like the in-memory indexing method described above, constructing an index in main memory. When it runs out of main memory, however, the indexing process transfers the entire in-memory index to disk and releases all memory resources it had allocated to build the index. It then continues processing input data, building another in-memory index. This procedure is repeated several times, until the entire collection has been indexed, and the indexing process has created a set of independent on-disk indices, each representing a small part of the whole collection. After the collection has been indexed, all these sub-indices are combined through a multi-way merge operation, resulting in the final index for the collection. Because the posting lists in each sub-index are sorted in lexicographical order (cf. Algorithm 2), this operation is straightforward. Like in the case of sort-based indexing, the in-situ merge method described by Moffat and Bell [79] can be used to virtually eliminate temporary storage space requirements while building the index.

Heinz and Zobel [55] report that their merge-based index construction method is about 50% faster than a carefully tuned sort-based indexing algorithm. In addition, merge-based index construction can build an index using an arbitrarily small amount of main memory, because it does not need to maintain a global in-memory dictionary. Moreover, with in-memory index construction (and thus with merge-based index construction), it is possible to compress incoming postings on-the-fly, increasing memory efficiency and reducing the number of sub-indices created during the index construction process.

Collection	**Postings (uncompressed)**	**Dictionary (uncompressed)**
SBFS	7.8 GB (91.5%)	738.7 MB (8.5%)
TREC	6.1 GB (99.2%)	52.9 MB (0.8%)
GOV2	324.6 GB (99.6%)	1288.0 MB (0.4%)

Table 2.1: Space requirements of the two main components of an inverted index – dictionary and posting lists – for three different text collections: SBFS (contents of a file system; 1.0 billion tokens, 28.7 million distinct terms), TREC (newswire articles; 824 million tokens, 2.1 million distinct terms) and GOV2 (Web pages; 43.6 billion tokens, 49.5 million distinct terms). Assuming 64-bit integers for all references (postings and pointers).

2.4.4 Index Compression

A main advantage of inverted indices is that they are highly compressible. This is especially important in the context of file system search, where all data structures should be as lightweight as possible so that they do not interfere with the normal usage of the computer system.

As explained above, an inverted file consists of two main components: the dictionary and the posting lists. Table 2.1 gives an impression of the relative size of the two components in indices created for three different text collections. It can be seen that, for all three collections, the combined size of the uncompressed posting lists is over 90% of the total size of the (uncompressed) index. Index compression techniques therefore usually focus on the compression of the posting lists, ignoring the comparatively small dictionary.

Compressing the data found in the index's posting lists serves two different purposes. First, it decreases the storage requirements of the index. Second, it reduces the I/O overhead at query time and thus potentially increases the search engine's query processing performance.

Compressing Posting Lists

In an inverted index – be it a docid index, a positional index, or a schema-independent index – the elements of a given list are usually stored in their original collection order (possibly with the exception of index layouts that are explicitly tailored towards efficient processing of ranked queries [86], although, even in those cases, counterexamples exist [3]). For example, the beginning of the posting list for the term "the" in the TREC collection is:

$$70, 80, 98, 700, 738, 751, 758, 773, 781, 791, 814, \ldots$$

Because postings are stored in strictly increasing order, this sequence can be transformed into an equivalent sequence of positive integers:

$$70, 10, 18, 602, 38, 13, 7, 15, 8, 10, 23, \ldots$$

This new representation makes it possible to apply general integer encoding techniques to the problem of index compression [40] [45]. [51] [101]

Integer encoding schemes can be divided into two groups: parametric and non-parametric. Elias's γ code [40] is an example of a non-parametric coding scheme. The γ code for a positive integer n consists of two components: a *selector* $\lambda(n) = \lceil \log_2(n) + 1 \rceil$, indicating the length of the integer n as a binary number, encoded in unary, and the *body* — n itself, encoded as a λ-bit binary number. Because the most-significant bit of the body is implicitly encoded in the selector, it may be omitted in the body. Thus, the γ code for a positive integer n is of size $2 \cdot \lfloor \log_2(n) \rfloor + 1$ bits.

Golomb codes [51] are examples of parametric codes. They are based on the assumption that the integers to be encoded (that is, the gaps in a posting list) roughly follow a geometric distribution:

$$\Pr(d = k) \approx p \cdot (1-p)^{k-1}, \quad p \in (0, 1).$$

Within the context of information retrieval, this is equivalent to the assumption that the occurrences of a given term are independent of each other and are scattered across the text collection according to a uniform distribution. A Golomb code has a single parameter b. The code word for a positive integer n consists of two components: $\lceil n/b \rceil$, encoded as a unary number, followed by n mod b, encoded as a $\lceil \log_2(b) \rceil$-bit or $\lceil \log_2(b) - 1 \rceil$-bit binary number. Choosing $b = \left\lceil -\frac{\log(2-p)}{\log(1-p)} \right\rceil$ minimizes the expected code length [48].

The LLRUN method described by Fraenkel and Klein [45] is another example of a parametric encoding scheme. LLRUN is very similar to Elias's γ method, except that the selector λ is not stored in unary, but using a minimum-redundancy Huffman code [57].

All three methods – γ, Golomb, and LLRUN – have in common that they produce unaligned codes. That is, the code word for a given posting may commence at an arbitrary bit position and does not necessarily coincide with a byte boundary. This may require more complicated decoding routines and thus may have an adverse effect on the search engine's query processing performance. Williams and Zobel [112] and Scholer et al. [100] examine a very simple byte-aligned encoding scheme, referred to as *vByte*. Each byte of a vByte code word contains 7 data bits and a continuation flag. The latter indicates whether this byte is the last byte of the current code word or whether there are more to follow.

Algorithm 3 Decoding a sequence of vByte-encoded postings. Input parameters: *compressed*, an array of byte values, and *length*, the number of postings in the decoded list. Output parameter: *uncompressed*, a buffer to hold the uncompressed postings.

```
1: let $uncompressed[-1] := 0$, by definition
2: $inputPosition \leftarrow 0$
3: $outputPosition \leftarrow 0$
4: while $outputPosition < length$ do
5:    $delta \leftarrow 0$
6:    repeat
7:       $delta \leftarrow (delta \times 128) + (compressed[inputPosition] \bmod 128)$
8:       $inputPosition \leftarrow inputPosition + 1$
9:    until $compressed[inputPosition - 1] < 128$
10:   $uncompressed[outputPosition] \leftarrow uncompressed[outputPosition] + delta$
11:   $outputPosition \leftarrow outputPosition + 1$
12: end while
```

vByte can be decoded very efficiently, as shown in Algorithm 3. Williams and Zobel [112] demonstrate that byte-aligned postings compression, compared to an uncompressed index organization, usually improves the storage requirements as well as the query processing performance of the search engine.

More recently, Vo and Moffat [1] [2] have discussed word-aligned codes that lead to both better compression effectiveness and higher decoding throughput than vByte.

Dictionary Compression

Even though the vast majority of all data in an inverted index are postings data, it is sometimes still beneficial to apply compression techniques to the dictionary as well — either because the number of distinct terms in the collection is very large, or because the memory resources available for the dictionary are very limited.

Consider again Figure 2.5, showing a sort-based in-memory dictionary organization. For this type of dictionary, there are three points at which space can be saved: the memory references in the primary search array, the file pointers in the secondary array, and the term strings themselves.

The pointer overhead in the primary array can be reduced by combining adjacent terms into groups of k, for a reasonable value of k (e.g., $1 < k \leq 256$). This decreases the pointer overhead by a factor $\frac{k-1}{k}$. On the other hand, dictionary entries now can no longer be accessed through a pure binary search operation. Instead, a term is found by performing an initial binary search, followed by a linear scan of up to k dictionary entries.

The space occupied by the file pointers in the secondary array can be reduced by applying

Group size	**No compression**	**Front coding**	**Front coding + vByte**
1	42.4 MB / 2.6 μs		
4	36.4 MB / 2.4 μs	27.7 MB / 2.3 μs	17.3 MB / 2.2 μs
16	34.9 MB / 2.7 μs	24.0 MB / 2.7 μs	11.0 MB / 2.7 μs
64	34.5 MB / 4.9 μs	23.1 MB / 4.3 μs	9.4 MB / 5.5 μs

Table 2.2: Total dictionary size and average dictionary lookup latency for different dictionary compression methods (using data from the TREC collection). Combining terms into groups, compressing term strings by front coding, and compressing file pointers using vByte can reduce the total memory requirements by up to 78%.

any of the integer coding techniques describe above. Assuming that on-disk posting lists are stored in the index according to the lexicographical ordering of their respective terms, the distance between two lexicographically consecutive posting lists is usually small and can be represented using far less than 64 bits.

Finally, the space consumption of the term strings themselves can be decreases by applying a technique known as *front coding* [16]. Front coding exploits the fact that terms are stored in lexicographical order and thus are likely to share a non-trivial prefix. A term can then be encoded by storing the length of the prefix the term has in common with the previous term, followed by the term's suffix. For example, the term sequence

"wunder", "wunderbar", "wunderbox", "wunderboxes", "wunderbundel"

from Figure 2.5 would be represented as:

(0, 6, "wunder"), (6, 3, "bar"), (7, 2, "ox"), (9, 2, "es"), (7, 5, "undel").

Usually, the first two components of each tuple (i.e., length of prefix and length of suffix) are stored in 4 bits each. The combination (0,0) can serve as an escape sequence to deal with terms whose suffix is longer than 15 bytes.

Table 2.2 shows that a combination of front coding (for the term strings) and vByte (for the file pointers) can decrease the dictionary's memory consumption by up to 78%. At the same time, the average lookup latency only exhibits a slight increase — 114%, due to the additional decoding operations and the sequential search necessary within each block. While this increase in lookup latency would have a disastrous effect on indexing performance, it is negligible for query performance, as the per-query-term lookup time of 5.5 μs only accounts for a tiny fraction of the total time it takes to process a search query. This further motivates the use of two different dictionaries, one for indexing, the other for query processing.

Compressed In-Memory Index Construction

Index compression cannot only be applied to static inverted lists, but also to the extensible posting lists used during the index construction process. Integrating compression into the indexing process is relatively easy. For each term T in the search engine's dictionary, an integer field $T.\mathit{lastPosting}$ is maintained, containing the value of the last posting for the term T. Whenever a new posting P has to be added to T's extensible posting list, it is not stored in its raw form, but as a delta value: $\Delta(P) = P - T.\mathit{lastPosting}$, using one of the integer coding schemes outlined above.

Because the number of distinct terms is small compared to the number of postings, increasing the per-term storage overhead by a small amount (8 bytes) is worth the effort and pays off in terms of greatly reduced storage requirements for the in-memory index.

2.5 Index Maintenance

Unlike some other text collections, the contents of a file system are not static. For example, new files are being created and existing files removed from the file system almost continually. This has the effect that the index, after it has been created, soon becomes obsolete, because it does not properly reflect the contents of the file system any more.

The task of keeping an existing index structure synchronized with the changing contents of a text collection is referred to as *index maintenance*. It usually requires the search engine to address the following three aspects of a dynamic text collection:

document insertions – a new document is added to the collection and needs to be indexed by the search engine;

document deletions – a document is removed from the collection, and the search engine has to take measures to ensure that the deleted document is not reported back to the user as a search result;

document modifications – the contents of a document are changed, either completely or partially, and the search engine needs to make sure that it does not return the old version of the document in response to a search query.

2.5.1 Incremental Index Updates

The vast majority of all published work in the area of dynamic text collections focuses on incremental text collections, supporting document insertions, but neither deletions nor modifications. When new documents enter the collection, they have to be indexed, and the resulting

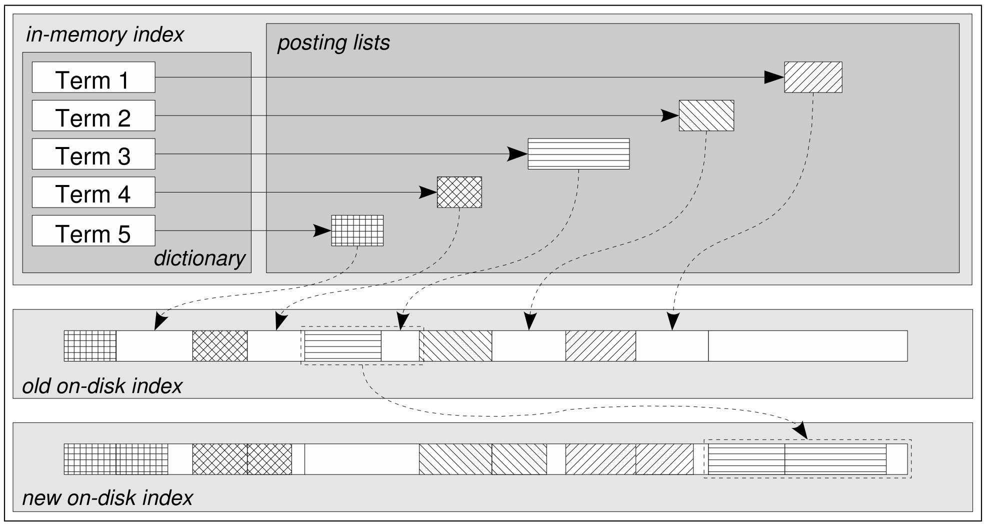


Figure 2.8: In-place index update with pre-allocation. At the end of the on-disk list for term 3, there is not enough room for the new postings. The list has to be relocated.

postings data have to be appended to the appropriate inverted lists in the existing (on-disk) index.

Cutting and Pedersen [35] were among the first who studied this problem in-depth. Starting from an index layout based on a B-tree [8] [9], they find that it leads to better update performance if the available main memory is spent on buffering incoming postings rather than on caching pages of the on-disk B-tree. That is, instead of applying updates immediately to the on-disk index, postings are buffered in memory and are added to the on-disk index as a whole as soon as a predefined memory utilization threshold is reached. This way, the ratio between the number of postings and the number of distinct terms involved in the physical index update ($\frac{\#postings}{\#terms}$), which – according to Zipf [117] – grows with $\Theta(\log(\#postings))$, is improved. Expensive list updates can therefore be amortized over a larger number of postings.

Since Cutting's seminal paper in 1990 (and possibly even before that), virtually every index maintenance algorithm has followed the same general strategy — buffering postings in memory for a while and eventually adding them to the existing index in one big physical update operation [15] [25] [29] [69] [68] [70]. [103] [107] Regarding the exact way in which buffered postings are combined with the existing index, one can distinguish between two different families of update strategies: in-place and merge-based.

In-Place Index Update

An in-place index update strategy usually stores each inverted list in a contiguous region of the disk and leaves some free space at the end of each list. Whenever postings accumulated in memory have to be combined with the existing on-disk inverted index, they are simply appended at the end of the respective list. Only if there is not enough space left for the new postings, the entire list is relocated to a new place, where there is enough space for the incoming postings. This general procedure is shown in Figure 2.8.

Tomasic et al. [107] discuss list allocation policies that determine how much space at the end of an on-disk inverted list should be reserved for future updates and under what circumstances a list should be relocated. Viewing the parameters of their in-place update strategies as a trade-off between index maintenance performance and query processing performance, Tomasic et al. conclude that inverted lists do not necessarily have to be kept in a contiguous region of the disk. However, discontiguous update strategies should only be used if the search engine's query load is very small. If query performance is important, on-disk posting lists should be kept contiguous, and space should be allocated employing a proportional pre-allocation strategy that reserves at least $k \cdot n$ bytes of index space for an on-disk posting list occupying n bytes. In the long run, this guarantees that a list $\mathcal{L}$ is relocated $\mathcal{O}(\log(|\mathcal{L}|))$ times. Tomasic et al. recommend $k = 1.1$.

Shieh and Chung [102] follow a statistical approach to predict the future growth of an inverted list based on behavior observed in the past. Their method leads to a number of list relocations close to that caused by proportional pre-allocation with $k = 2$, but the space utilization of the inverted file is similar to the utilization achieved by $k = 1.2$.

The main limitation of in-place index maintenance is the large number of disk seeks associated with list relocations. Lester et al. [70] discuss various optimizations to the original in-place strategy, including a very thorough discussion of how to manage the free space in an in-place-managed index. They recommend that very short lists be kept directly in the search engine's dictionary (realized through an on-disk B^{+}-tree) instead of the postings file. By updating the on-disk dictionary in a sequential fashion whenever the system runs out of memory, short lists can be updated without requiring additional disk seeks.

Chiueh and Huang [25] present an interesting alternative to the above techniques. In their method, an on-disk inverted lists can either be updated when the system runs out of memory or when it is accessed during query processing. This kind of *piggybacking* has the advantage that it can save up to 1 disk seek per list update. However, it is not clear whether in practice it actually makes a difference. Chiueh and Huang's experimental results indicate that it does not.

Merge-Based Index Update

In contrast to the in-place update strategies described so far, Cutting's original method [35] is merge-based. Whenever their system runs out of memory, the in-memory index data are combined with the existing on-disk index in a sequential manner, resulting in a new on-disk index that supersedes the old one. By reading the old index and writing the new index strictly sequentially, merge update avoids costly disk seeks and can be performed comparatively quickly.

Cutting and Pedersen use a B-tree to store dictionary entries and postings. The general method, however, also works for other index organizations. Merge-based index maintenance, therefore, is very similar to the merge-based index construction method discussed in Section 2.4.3. The difference is that the in-memory index is merged immediately with the existing on-disk index whenever the system runs out of memory. I refer to this basic version of merge-based update as Immediate Merge.

The main shortcoming of Immediate Merge is that it requires the entire on-disk index to be read from and written back to disk every time the system runs out of memory. Hence, when indexing a text collection containing N tokens with a memory limit of M postings, the total number of postings written to disk during index construction/maintenance is approximately

$$\sum_{k=1}^{\lceil \frac{N}{M} \rceil} k \cdot M \;\in\; \Theta\left(\frac{N^2}{M}\right). \tag{2.12}$$

This is an essential limitation, especially in the context of file system search, where M is usually rather small.

As an alternative to Immediate Merge, the merge-based index construction method from Section 2.4.3 could be treated as if it were an on-line index maintenance method. As pointed out before, the temporary in-memory index created by hash-based in-memory inversion is immediately queriable. Therefore, queries can be processed even during the index construction process, by concatenating the posting list found in the sub-indices stored on disk and the in-memory index that is currently being built.

This method, which I refer to as the No Merge update strategy, is shown in Figure 2.9. It cures Immediate Merge's quadratic update complexity, but potentially introduces a very large number of disk seeks during query processing. Each query term's posting list needs to be assembled by concatenating the sub-lists found in the $\frac{N}{M}$ independent sub-indices.

Lester et al. [68] propose a set of update strategies that represent less extreme points in this trade-off between indexing performance and query processing performance. Within their

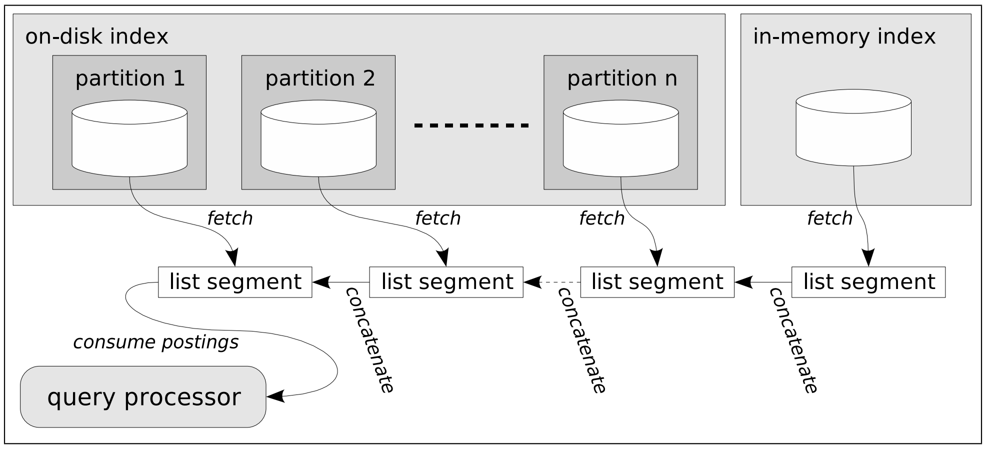


Figure 2.9: Transforming off-line merge-based index construction into an on-line index maintenance strategy. The posting list for each query term is fetched from the in-memory index being currently built and from the on-disk sub-indices created earlier. The final inverted list is constructed by concatenating the sub-lists.

general framework of *geometric partitioning*, they describe two different ways to limit the number of on-disk index partitions, and thus the amount of fragmentation in each on-disk posting list.

In their first method, they require that the number of index partitions not exceed a predefined partition threshold p. Similar to the partitioning scheme found in a constant-height B-tree T (where the amount of data found at depth k is $N^{(k+1)/(h+1)}$, $h = \text{height}(T)$, and N is the amount of data found at depth h, i.e., T's leaf nodes), the index maintenance complexity is optimized if the relative size of the p on-disk sub-indices is kept close to a geometric progression (hence the name of their method):

$$\frac{\text{size}(I_k)}{M} \approx \text{size}(I_{k-1}) \cdot \left(\frac{N}{M}\right)^{1/p} \quad (\text{for } 1 < k \leq p) \tag{2.13}$$

$$\text{size}(I_1) \approx \left(\frac{N}{M}\right)^{1/p}, \tag{2.14}$$

where $\text{size}(I_k)$ denotes the number of postings stored in index I_k. The resulting disk complexity then is:

$$\Theta\left(N \cdot \left(\frac{N}{M}\right)^{1/p}\right).$$

In their second method, Lester et al. do not impose an explicit bound on the number of index partitions, but require that index partition i either be empty or contain between

$$M \cdot r^{i-1} \ \text{ and } \ M \cdot (r-1) \cdot r^{i-1} \tag{2.15}$$

postings ($r \geq 2$). For a collection comprising N tokens, this leads to a maximum number of on-disk index partitions no bigger than

$$\left\lceil \log_r \left(\frac{N}{M} \right) \right\rceil .$$

Merging on-disk indices in such a way that requirement 2.15 is always satisfied leads to a total index maintenance complexity of

$$\Theta \left(N \cdot \log \left(\frac{N}{M} \right) \right) .$$

Derivations of these bounds can be found in [68].

The case where $r = 2$ has independently been described and analyzed in [18]. It is referred to as LOGARITHMIC MERGE, in analogy to the *logarithmic method* described by Bentley and Saxe [10] (and because of its asymptotical complexity). Accordingly, the case where the number of partitions is limited to $p = 2$ is referred to as SQRT MERGE.

Although unpublished, LOGARITHMIC MERGE reportedly has been used in the Altavista[12] Web search engine since the mid-1990s[13]. It is also employed by the open-source search engine Lucene[14].

In-Place vs. Merge Update

Intuitively, it seems that in-place update with proportional pre-allocation leads to better results than merge-based index maintenance. Assuming a pre-allocation factor $k = 2$, each posting in a list of length l is relocated at most $\lceil \log_2(l) \rceil$ times, and the average number of postings relocated per list is at most

$$\sum_{i=1}^{\lceil \log_2(l) \rceil} 2^i \ = \ 2^{\lceil \log_2(l) \rceil + 1} - 1 \ \leq \ 4 \cdot l. \tag{2.16}$$

This implies a linear disk complexity, which is only rivalled by the NO MERGE strategy. However, proportional in-place update keeps on-disk lists contiguous and thus leads to dra-

[12]http://www.altavista.com/ (accessed 2007-02-22)
[13]Anonymous reviewer, private communication (July 2005)
[14]http://lucene.apache.org/ (accessed 2007-02-22)

matically better query performance than No Merge.

Unfortunately, in recent years, nobody has actually been able to come up with an implementation of in-place update that performs better than merge update. In fact, nobody has even found an implementation that, for a realistic buffer size (memory limit), leads to better update performance than Immediate Merge with its quadratic disk complexity (cf. Lester et al. [69] [70]).

Hence, merge update, because it minimizes the number of disk seeks during index maintenance, appears to lead to strictly better performance than in-place update, at least if on-disk posting lists are required to be stored in a contiguous fashion.

This is not an absolute result, however. The relative performance of in-place and merge-based update strategies entirely depends on the operating characteristics of the hard drive(s) used to store the index. For example, for the experiments that Shoens et al. [103] conducted in 1993, they report an average random access disk latency of 20 ms and a sustained sequential read transfer rate of 1 MB/sec. In comparison, hard drives used in the experiments described in this thesis, exhibit a random access latency of about 8 ms and a sequential read throughput of 50 MB/sec. That is, while the performance of sequential read operations has increased by a factor 50, random access performance has only improved by a factor 2.5. Hence, in relative terms, a disk seek in 2007 is 20 times as expensive as it used to be in 1993. This has dramatic consequences on the performance of disk-seek-intensive in-place update strategies and just by itself already gives sufficient motivation to thoroughly re-evaluate all results regarding the relative efficiency of in-place and merge-based update obtained more than 10 years ago.

2.5.2 Document Deletions

The second update operation seen in dynamic text collections is the deletion of existing documents. In a file system, document deletions are triggered by file deletions. Within the search engine, deletions are usually dealt with by maintaining an invalidation list, containing an entry for every deleted document, and applying a postprocessing step to the search results before returning them to the user [29] [25]. Like in the case of document insertions, deletions may also be handled by piggybacking them on top of query operations: Whenever a list is accessed during query processing, all elements referring to documents that no longer exist are removed from the list, and the list is written back to disk [25].

Ultimately, however, the search engine needs to employ some sort of a garbage collection policy. Otherwise, not only will query processing become very expensive, but the index will also consume far more disk space than necessary, potentially interfering with other applications storing their data on the same hard drive. Apart from my own work [18] [19], I am not aware of any published results on the efficiency of garbage collection strategies for dynamic text retrieval systems.

2.5.3 Document Modifications

Document modifications constitute the third type of update operations in the context of a dynamic search environment. A modification to an existing document is usually treated as as if deleting the old version of the document and then inserting its new version [25] [18]. While this approach seems appropriate if the document is relatively small or if the amount of changes made to the document is large compared to its total size, it may cause substantial performance problems if applied to a large document that only experienced a minor change. As an extreme example, suppose that the entire text collection consists of only a single document. In that case, the delete-insert policy will necessarily lead to a quadratic index maintenance complexity and render the optimizations discussed in the previous section (e.g., Logarithmic Merge) essentially useless.

While this extreme case is unlikely to be seen in practice, less extreme, but equally problematic cases are ubiquitous in file system search. The most prominent examples are e-mail folders in the `mbox` file format and system log files. Both file types should be searchable by a file system search engine, and both file types usually experience very small updates that, however, accumulate over time.

Lim et al. [71] describe an update strategy that takes partial document modifications into account. Instead of storing postings of the form (*docId*, *position*) in the index, their postings are of the form (*docId*, *landmarkId*, *position*), where *landmarkId* refers to a landmark in the document. This can either be a certain type of HTML/XML tag (e.g., `<p>`, `<br>`), or just a random word, for instance, every 32nd word in the document. When parts of a document are changed, postings corresponding to unchanged parts do not need to be updated, because their position relative to their respective landmark remains unchanged. Lim et al. report that their method leads to a factor-2-to-3 improvement over a baseline that re-indexes the entire document.

However, the algorithm does not make updates immediately available for search. Instead, updates are buffered and applied to the existing index in a manner very similar to Immediate Merge, implying a quadratic update complexity despite the savings achieved by the landmark approach. In addition, it requires the search engine to keep the list of landmarks in memory (or to load it into memory when processing a query). For a large collection, the combined size of the landmarks can be quite substantial and can easily exceed the amount of memory available to the search engine.

2.6 Applications

Desktop and File System Search

Desktop search is similar to file system search, but is limited to a single user, as it does not take security restrictions, such as file permissions, into account when producing the search results to a given query. A desktop search engine can therefore only be used on a computer system that is accessed by a very small number of users, preferably a single user.

One of the first publicly available desktop search engines is GLIMPSE[15] developed by Manber et al. [76] [74] in the early 1990s. Although originally referred to as *file system search engine*, it has to be considered a *desktop search engine* in the context of this thesis, since it does not take into account the possibility of multiple users sharing the data in the file system.

GLIMPSE is different from most other search engines in that it does not maintain a positional index. Instead, it follows a *two-level search* paradigm. Within this framework, the index does not contain exact information about all term occurrences, but only pointers to file system regions where term occurrences may be found. Such a region typically contains multiple files, with a total limit of 256 regions in Manber's original implementation [76].

The first phase in GLIMPSE's two-level process, based on the incomplete information found in the index, is followed by a sequential scan of all matching file system regions, similar to the retrieval procedure employed when using signature files for full-text retrieval (cf. Section 2.3). The main motivation behind this approach was to reduce the storage requirements of the search engine at the cost of a reduced query processing performance. While this was a sensible design decision in the early 1990s, it is not any more, mainly because disk access latency (in particular random access, which is necessary for the second phase in the retrieval process) is a more limiting factor of modern hard drives than their storage capacity.

In the past 3 years, search engine companies have discovered the desktop search market, and a large number of commercial desktop search systems have followed GLIMPSE:

- October 2004: Copernic Desktop Search (Windows);
- October 2004: Google Desktop Search (Windows);
- December 2004: Ask Jeeves Desktop Search (Windows);
- December 2004: MSN Desktop Search (Windows);
- January 2005: Yahoo!/X1 Desktop Search (Windows);
- early 2005: Beagle Desktop Search (Linux).

[15]http://webglimpse.net/ (accessed 2007-03-28)

In contrast to GLIMPSE, these systems all maintain positional indices, leading to lower query response times than the two-level search procedure outlined above. Most of them share a number of properties that are also required from a file system search engine: continuous (real-time) index updates, low memory footprint, low storage requirements, and non-interference with other processes.

Just like GLIMPSE, however, none of these search systems provides explicit support for multiple users. Every user has her own independent index, containing information about all files readable by her. Files readable by multiple users are indexed several times, once for each user — obviously not a very satisfying solution, neither in terms of system performance nor in terms of memory and storage requirements.

A true multi-user file system search engine is Apple's Spotlight[16] search system, part of Apple's Mac OS X Tiger (and later versions). Spotlight integrates into the Mac OS file system, updating its index structures as files are added to or removed from the file system, and provides support for multi-user security restrictions. Every file is only indexed once. All index data are stored in one central index, and security restrictions are applied to a set of search results before returning them to the user.

Enterprise Search

Enterprise search is another information retrieval application that is similar to file system search. In a typical enterprise search scenario, a variety of users with different access privileges submit queries to the search server(s). The search results returned by the engine usually refer to files created and owned by users different from the one who submitted the query, but should not refer to any files that are not accessible by her.

These security requirements are usually addressed by a postprocessing step that takes place after the actual search operation [5]. After the search engine has determined a set of matching documents for the given query, it checks for each document whether the user's access privileges allow her to access that document. If not, the document is removed from the search results. The postprocessing approach is followed by a large number of enterprise search systems, such as Coveo Enterprise Search[17], Google Search Appliance[18] and Microsoft SharePoint[19] [5]. In Chapter 5, I show that it should only be used if the search engine does not rank matching documents by their statistical relevance to the query. Otherwise, a user may obtain information about files for which she does not have read permission.

[16]http://www.apple.com/macosx/features/spotlight/ (accessed 2007-03-28)
[17]http://www.coveo.com/ (accessed 2007-03-28)
[18]http://www.google.com/enterprise/ (accessed 2007-03-28)
[19]http://www.microsoft.com/sharepoint/ (accessed 2007-03-28)

Chapter 3

Evaluation Methodology

3.1 Data Sets

The main focus of this thesis is on performance aspects of full-text file system search. Whenever possible, I try to validate any theoretical considerations made about the performance of file system search engines by actual experiments on realistic text collections. For this purpose, I use the following three collections:

SBFS is the text collection that comprises the contents of all indexable files in Stefan Büttcher's file system (as of 2007-01-22). The collection consists of about 300,000 files, including non-plain-text files, such as PDF and MP3.

TREC is a combination of TREC disks 1-5, a set of collections used to evaluate search systems in the Text REtrieval Conference (TREC)[1]. The TREC collection contains a total of 1.5 million documents.

GOV2 is the result of a Web crawl of the .gov domain, conducted by Clarke [34] in early 2004. It contains a total of 25.2 million documents.

A statistical summary of all three collections is shown in Table 3.1. An additional, more detailed, summary of SBFS is given by Table 3.2.

SBFS serves as the primary text collection, used in my experiments whenever appropriate. Unfortunately, it has some limitations: First of all, SBFS is not large enough to show that the methods discussed here do in fact scale to large collections. For example, a compressed positional index for SBFS consumes less than 3 GB of disk space, not much more than the amount of main memory found in a typical consumer PC these days (1 GB). Second,

[1]http://trec.nist.gov/

	SBFS collection	**TREC collection**	**GOV2 collection**
Size of collection	14.8 GB	4.8 GB	426.4 GB
Number of files	310,459	15,449	27,204
Number of documents	query-dependent	1.5 million	25.2 million
Number of tokens	1.04 billion	823.7 million	43.6 billion
Number of distinct terms	28.7 million	2.1 million	49.5 million
Size of compressed index	2.7 GB	1.4 GB	62.2 GB
Query set	n/a	TREC 1–500	TREC TB06 Eff.
Terms per query (avg.)	n/a	3.7	3.5
Computer system	Sempron 2800+	Sempron 2800+	Athlon64 3500+

Table 3.1: Overview of the three text collections used in my experiments. The SBFS collection is meant to be representative for file system search. GOV2's main purpose is to show the scalability of the methods presented in this thesis.

SBFS is not a plain-text collection. Building an index for it requires substantially more time than building an index for text-only collections, such as TREC and GOV2, because some of the input files first need to be parsed and converted into plain text before they can be indexed. Third, because SBFS is not publicly available, it seems appropriate to also use two publicly-available standard collections, so as to allow other researchers to reproduce the results presented in this thesis.

GOV2's main purpose is to demonstrate the scalability of the proposed methods. TREC's main purpose is to serve as a convenient substitute for SBFS, being similar in size.

Query performance on the different text collections is measured using a set of standard search queries. For TREC, search queries are created from the title fields of TREC topics 1–500. For GOV2, a set of 27,204 queries is used, randomly selected from the 100,000 search queries employed in the efficiency task of the TREC 2006 Terabyte track [23]. The reason for not using the entire query set is that the GOV2 collection is distributed in 27,204 files. Using the same number of search queries makes it easier to conduct experiments in which each file insertion is followed by a search query.

In both cases, TREC and GOV2, queries are processed by the search engine after removing all stopwords ("the", "and", "of", ...). Removing stopwords from queries before sending them to the query processor is quite common in information retrieval and is widely assumed not to harm search quality. However, although stopwords are not used for query processing purposes, the search engine's index – in all experiments described in this thesis – still contains full information about all terms, giving equal treatment to both stopwords and non-stopwords.

3.2 Hardware Configuration & Performance Measurements

In my experiments, I use two different computer systems:

- A desktop computer equipped with an AMD Sempron 2800+ processor (32-bit, 1.6 GHz), 768 MB RAM, and two 80 GB SATA hard drive (7,200 rpm).
- A workstation based on an AMD Athlon64 3500+ processor (64-bit, 2.2 GHz) with 2GB RAM and several 300 GB SATA hard drives (7,200 rpm).

Using two different computers for the experiments, although potentially introducing an additional element of confusion, has the advantage that it allows me to demonstrate the scalability of the methods presented in two directions: scalability to large text collections (essential for general applicability), and scalability to low-memory computer systems (essential for applicability in file system search). The latter is covered by using the desktop computer with its comparatively small main memory; the former is covered by conducting experiments with the rather large GOV2 collection which, unfortunately, is too large for the desktop's hard disk and thus requires the experiments to be performed on the more powerful 64-bit workstation.

All experiments discussed in this thesis that are related to SBFS or the TREC collection were conducted on the 32-bit AMD Sempron desktop PC. Experiments related to the larger GOV2 collection were conducted on the 64-bit AMD Athlon64 workstation.

Implementation and Measurements

Unless explicitly stated otherwise, all performance results presented were obtained using the Wumpus[2] search engine, an experimental file system search engine and multi-user information retrieval system. Wumpus is freely available for download under the terms of the GNU General Public License (GPL). Because most of Wumpus's internal computations, such as index pointer arithmetic and document score computations, are realized using 64-bit integer arithmetic, the performance difference between the 32-bit desktop PC and the 64-bit workstation are larger than what can be expected from their raw CPU frequency numbers.

Search queries are processed by Wumpus in a document-at-a-time fashion, employing an aggressive pre-fetching strategy. Whenever the search engine detects sequential access to the contents of a given posting list, it automatically loads the next few megabytes of that list into main memory. In many cases, this means that the list is loaded into memory in its entirety, thus reducing the potential savings achieved by query optimization strategies. Despite this

[2]http://www.wumpus-search.org/ (accessed 2007-04-30)

Total number of files	310,459
Plain text	141,597
HTML	110,644
XML	26,268
PDF	1,585
Postscript	1,656
`man` pages	5,771
E-mail messages	22,307

Table 3.2: Overview of the contents of the SBFS collection.

problem, the policy leads to excellent results in practice because it helps keep the number of disk seeks, switching back and forth between posting lists, very low.

Aggregate performance statistics, such as the total time to build an index for collection X, are usually obtained through the UNIX `time` command. Finer-grained performance figures are realized by Wumpus's built-in timing mechanisms, utilizing the `gettimeofday` and `times` system calls to keep track of wall-clock time and CPU time, respectively.

Accessing the Text Collections

When indexing non-plain-text files from the SBFS collection, Wumpus uses external conversion programs to extract the text from the input files:

- For Adobe's PDF[3] format, text extraction is realized by the `pdftotext` utility that is part of the Xpdf package[4].
- Postscript files are converted to text by transforming them to PDF first (using the `ps2pdf` utility that comes with Ghostscript[5]) and then extracting the text from the PDF file via `pdftotext`.
- `man` pages are rendered into plain text by GNU Troff[6] before being sent to the index.

All other file formats (plain text, HTML, XML, MP3) are recognized and parsed by Wumpus's built-in filters that do not require the execution of external helper programs.

[3]http://www.adobe.com/products/acrobat/adobepdf.html (accessed 2007-01-31)
[4]http://www.foolabs.com/xpdf/ (accessed 2007-01-31)
[5]http://www.ghostscript.com/ (accessed 2007-01-31)
[6]http://www.gnu.org/software/groff/ (accessed 2007-01-31)

The two plain-text collections TREC and GOV2 are split up into small files of manageable size before indexing. The 25.2 million documents in GOV2 are accessed as distributed, divided into 27,204 files in 273 directories. The 1.5 million documents in the TREC collection are divided into 15,449 files containing 100 documents each. When an index is built for TREC or GOV2, the files for the respective collection are parsed and processed in random order.

Index Storage

In most of my experiments, I assume that the index for a given text collection is too large to be completely stored in primary memory and that the majority of the index data need to be stored in secondary memory. My experimental setup is based on the assumption that secondary storage is realized by some number of hard disk drives and that these hard drives exhibit the same general performance characteristics already pointed out in Section 2.1. That is, I assume that hard drives are not true random access devices, but that reading two consecutive blocks can be performed much faster than reading two blocks that are located in different regions of the disk.

While this certainly holds for all types of hard disks, it may not necessarily hold for future storage devices, such as solid-state drives[7] which exhibit virtually no seek latency. Some of the methods proposed in this thesis will need to be re-evaluated if a major shift takes place from currently used hard drives to flash-memory-based solid-state drives. Such a shift, however, is unlikely to happen very soon, as the price/capacity ratio for solid-state drives is still about 50 times higher than for traditional magnetic hard disk drives (as of May 2007).

For practicality reasons (available storage space), text data and index data are stored on different hard drives in all my experiments.

Operating System Support

Whenever I refer to an on-disk inverted index, I silently assume that the index is stored as an ordinary file in the file system. The file system implementation used in the experiments is `ext3`[8], as implemented in the Linux 2.6.* operating system available from http://www.kernel.org/. For some experiments, I also use the older `ext2` file system (essentially `ext3` without journalling) and Hans Reiser's Reiser4[9] file system, as implemented in Linux 2.6.20-rc6-mm3.

[7]http://en.wikipedia.org/wiki/Solid_state_drive (accessed 2007-03-15)
[8]http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html (accessed 2007-02-01)
[9]http://www.namesys.com/v4/v4.html (accessed 2007-03-01)

Storing the index as a regular file in the file system has the disadvantage that the amount of file fragmentation found in the index cannot be controlled by the search engine, but is entirely up to the operating system. The contiguity of on-disk posting lists will play a major role in the discussion of Chapter 6.

The reader should be advised that, whenever I claim that a list is stored in a contiguous fashion, this claim is based on the assumption that the file system implementation stores each file in a contiguous region of the hard disk, i.e., that there is no difference between logical and physical contiguity. Obviously, this assumption is wrong. However, it seems reasonable to assume that most modern file system implementations employ their own strategies to reduce file fragmentation and that these strategies are sufficiently effective to make the error introduced by the incorrect file contiguity assumption negligible. Moreover, the alternative – storing all index data in raw disk partitions directly managed by the search engine – would complicate index access and index maintenance beyond reason and would also make the results obtained less applicable to real-life file system search (who keeps a spare disk partition around, just for the purpose of storing an index on it?). I therefore decided to utilize the existing file system implementation as the low-level storage infrastructure.

Chapter 4

Index Structures

In this chapter, I investigate what index structures are appropriate for efficient file system search. I discuss how the index is best integrated into the file system and what type of index should be used (Section 4.1). I argue that a file system search engine should maintain multiple independent indices, one for each mount point in the file system, instead of a single index structure covering all storage media connected to the computer.

Based on the assumption that the term distribution in the file system is approximately Zipfian [117] (Section 4.2), I then examine some statistical properties of inverted indices, the preferred data structure for keyword search. The results obtained form the foundation for statistical performance analyses carried out later in this chapter and in Chapter 6.

I explore what strategies lead to the best results when building the content index for a file system (Section 4.3) and discuss how to keep the memory consumption of the search engine low so that it does not interfere with the normal operation of the computer system (Section 4.4). Finally, I present a comparison of schema-independent and document-centric positional indices in terms of query performance and storage requirements, showing that a document-centric positional index usually leads to higher query processing performance than a schema-independent one, but that the performance difference is highly dependent on the nature of the search queries to be processed (Section 4.5).

This chapter is not concerned with real-time index updates (which are the topic of Chapter 6). Instead, it provides a comparative baseline for the methods discussed in Chapter 6. Taken together with Chapter 2, the techniques described in this chapter can be used to build a file system search engine that maintains its index structures by periodically scanning the file system and re-building the index from scratch, similar to the behavior exhibited by `locate`[1] and its `updatedb` indexing process.

[1] http://www.gnu.org/software/findutils/ (accessed 2007-02-28)

		Device axis	
		local	**global**
User axis	**local**	a separate index for each user on each storage device	per-user indices; each per-user index covers all storage devices
	global	device-specific indices, each containing data for all users	a single, system-wide index covering all users and all storage devices

Table 4.1: The two main locality axes in multi-user multi-device file system search.

4.1 General Considerations

Before we can start delving into the details of how to build an index for the contents of a file system, two basic questions need to be answered:

- How should the index be embedded into the file system?
- Exactly what type of index should be used to facilitate search operations?

Let me start with the first question. In a typical UNIX installation, the file system contains files belonging to more than just a single user. Similarly, the file system usually spans across multiple mount points, representing different, independent storage devices. These two aspects define two independent *locality axes* (the user axis and the device axis, as shown in Table 4.1). For each axis, it needs to be decided whether data should be stored locally or globally along that axis.

The user and the device axis are not the only locality axes in file system search. Other axes, such as the time axis, exist and do play a role. The user and the device axis, however, are the most important ones.

The User Axis: A Single, Global Index to Be Accessed by All Users

Most existing desktop search tools maintain per-user indices. While this is obviously acceptable in single-user search environments, in pure desktop search environments (i.e., without the option to search the entire file system), and in environments with a small number of users and very little shared content (this is the case in a typical Windows system), it is not a good idea in a true multi-user file system search environment. Maintaining per-user indices, where each index only contains information about files that may be searched by the respective user, leads to two types of problems:

1. **Redundancy:** Many files (such as `man` pages and other documentation files) can be accessed by all users in the system. All these files have to be independently indexed for each user in the system, leading to a massive storage overhead in systems with more than a handful of users.

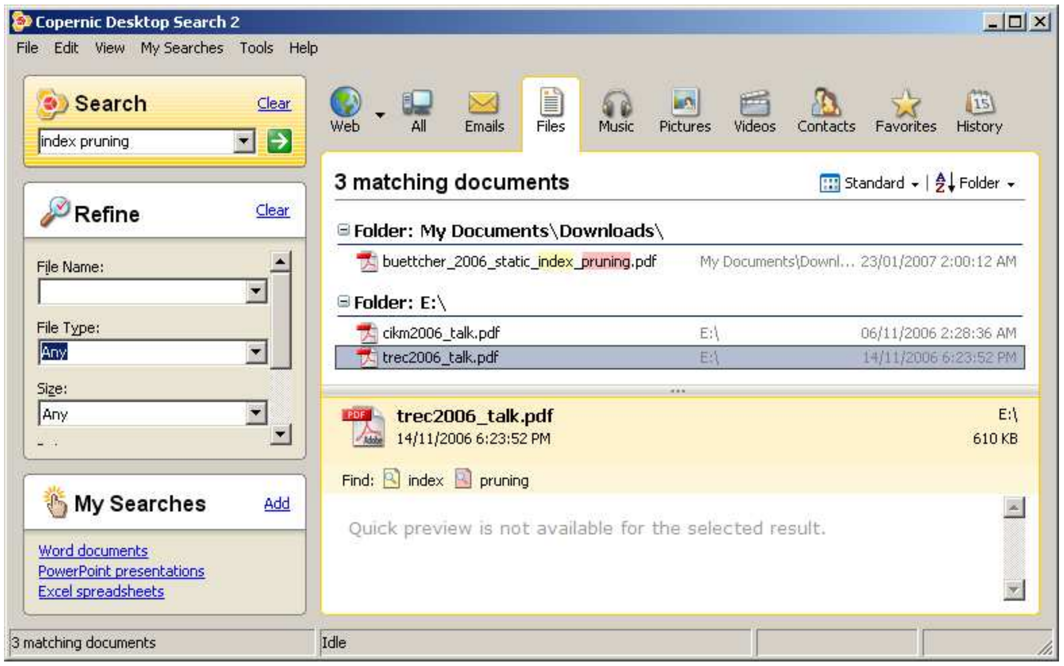
2. **Performance:** If per-user indices are used, then even a single `chmod` or `chown` operation, changing file permissions or file ownership, can trigger a large number of disk operations because the respective file needs to be completely re-indexed (or data need to be copied from one user's index to another user's index) each time a user executes `chown`. If real-time index updates are required, then even in a system with only two users this can potentially be exploited to realize a denial-of-service attack on the indexing service.

Together, these two problems violate Requirements 3 (index freshness) and 4 (non-interference) from Chapter 1. The only way to eliminate them is to use a single index that is shared by all users in the system, instead of maintaining a large number of per-user indices. This global index is maintained by a process with superuser rights that can add new information to the index when new files are created and remove data from the index when files are deleted. `chmod` and `chown` operations can then be dealt with by simply updating index meta-data, without the need to re-index the file content.

Of course, to guarantee data privacy (Requirement 5), the global index, because it contains information about all indexable files in the system, may never be accessed directly by a user. Instead, whenever a user submits a search query, it is sent to the indexing service (running with superuser rights). The indexing service then processes the query, fetching all necessary data from the index, and returns the search results to the user. It applies all security restrictions that are necessary to make the search results consistent with the user's view of the file system, not revealing any information about files that may not be accessed by the user who submitted the query. The problem of applying user-specific security restrictions to the search results is non-trivial, but can be solved (doing so is the topic of Chapter 5).

The Device Axis: Local, Per-Device Indices

When experimenting with various desktop search systems for Windows, I noticed that most of them had problems with removable media. They either refused to index data on removable media altogether, or they added information about files on removable media to the index, but removing the medium from the system later on was not reflected by the index. Search results still referred to files on a USB stick, for example, even after the device had been unplugged. An example is shown in Figure 4.1.



3 matching documents

Folder: My Documents\Downloads\
buettcher_2006_static_index_pruning.pdf My Documents\Downl... 23/01/2007 2:00:12 AM

Folder: E:\
cikm2006_talk.pdf E:\ 06/11/2006 2:28:36 AM
trec2006_talk.pdf E:\ 14/11/2006 6:23:52 PM

trec2006_talk.pdf E:\
14/11/2006 6:23:52 PM 610 KB

Quick preview is not available for the selected result.

3 matching documents Idle

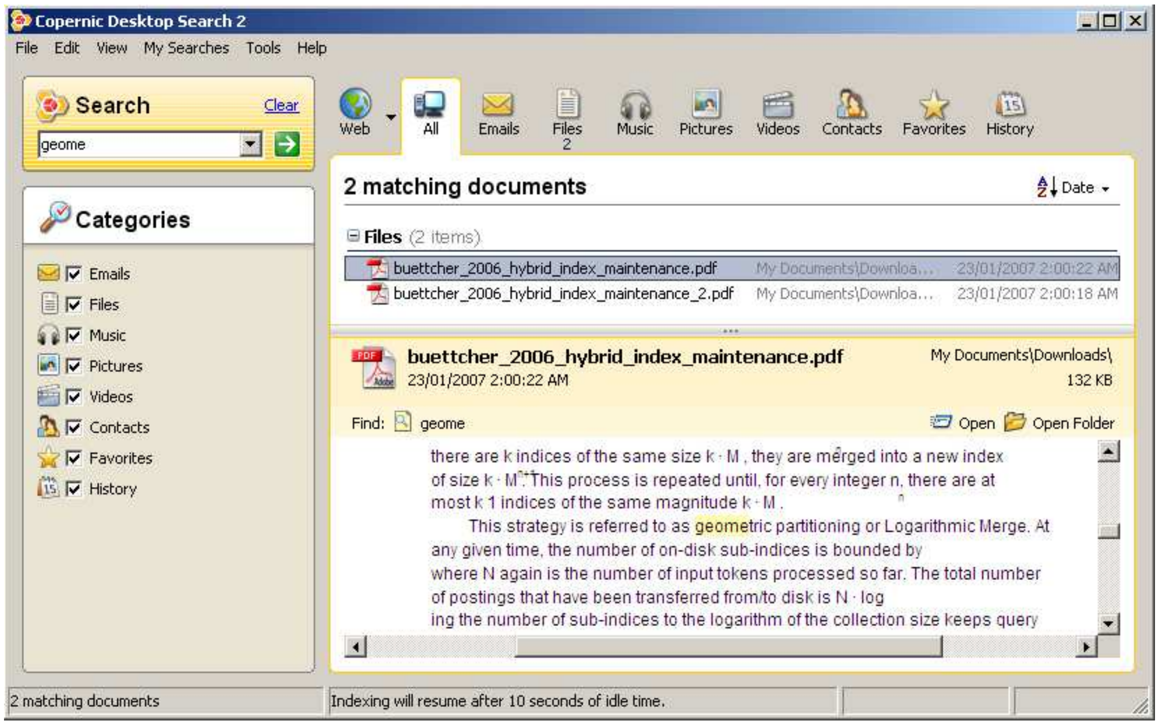
Figure 4.1: Copernic Desktop Search, returning documents matching the query "index pruning". Two of the three files matching the query are stored on a USB stick that has been removed from the system: "Quick preview is not available."

If index data are stored in a global, system-wide index, it is not clear how to deal with removable media. Should the index data be removed from the index immediately after the medium is removed from the system? If not, how long should the indexing service wait until it removes the data? Should external hard drives be treated as removable media?

The only feasible solution is to maintain per-device indices. In Linux, for instance, this means that each storage device (`/dev/hda`, `/dev/hdb`, etc.) will get its own local index that only contains information about files on that particular device. Whenever a device is removed from the file system, the indexing process associated with that device is terminated. Whenever a device is added to the file system, a new indexing process is started for the new device (or not, depending on parameter settings). Search queries are processed by combining the information found in the individual per-device indices and returning the search results, which may refer to several different devices, to the user.

For network file systems, such as NFS[2] mounts, this means that the index is not kept on the client side, but on the server that contains physical storage device. This requires additional communication between the NFS server and the client when processing a search query. It thus represents a potential bottleneck in situations where an NFS server is accessed

[2]http://nfs.sourceforge.net/ (accessed 2007-08-01)



2 matching documents

Files (2 items)

buettcher_2006_hybrid_index_maintenance.pdf	My Documents\Downloa...	23/01/2007 2:00:22 AM
buettcher_2006_hybrid_index_maintenance_2.pdf	My Documents\Downloa...	23/01/2007 2:00:18 AM

buettcher_2006_hybrid_index_maintenance.pdf
23/01/2007 2:00:22 AM
My Documents\Downloads\
132 KB

there are k indices of the same size k · M , they are merged into a new index
of size k · M. This process is repeated until, for every integer n, there are at
most k 1 indices of the same magnitude k · M .
This strategy is referred to as geometric partitioning or Logarithmic Merge. At
any given time, the number of on-disk sub-indices is bounded by
where N again is the number of input tokens processed so far. The total number
of postings that have been transferred from/to disk is N · log
ing the number of sub-indices to the logarithm of the collection size keeps query

2 matching documents

Indexing will resume after 10 seconds of idle time.

Figure 4.2: Search-as-you-type prefix query in Copernic Desktop Search. The term "geometric" matches the search query "geome".

by a large number of clients and where many users want to search for data on the server. On the other hand, this is the only way to allow the index for the NFS mount to be updated in real-time, as it is impossible for an NFS client to be informed of all changes that take place in a remote file system. In addition, keeping index data for an NFS mount on the client-side may cause substantial index data duplication, violating Requirement 4 from Chapter 1 ("non-interference").

Keyword Queries vs. Search-as-You-Type Queries

The discussion in Section 2.3 has made it clear that an inverted index is the preferred index data structure for keyword queries. However, is the typical query in file system search really a keyword query? The opinions on this matter seem to vary. While the desktop search engines released by Google and Microsoft are keyword-based, Apple's Spotlight, Yahoo!/X1, and Copernic Desktop Search support search-as-you-type queries in which the query can match part of a word in a document (see Figure 4.2 for an example).

While search-as-you-type queries seem to gain popularity (they are, for example, also supported by the Firefox Web browser when searching the contents of a Web page), it is not clear whether they actually add any important functionality. Furthermore, they require slightly more complicated index data structures and/or query processing strategies than keyword queries.

Conceptually, search-as-you-type is no different from prefix queries, with a new prefix query sent off to the search engine whenever a new character is typed into the search box, perhaps after a short period of inactivity. If the search engine is based on an inverted index, then a prefix query requires the engine to fetch from the index the posting lists for all terms starting with the given prefix and to merge them into a single list. This operation can be rather costly, depending on the number of terms matching the query and the length of their posting lists. If realized through a multi-way merge operation with a heap, the worst-case complexity of the merge operation is:

$$\Theta\left(\log(n) \cdot \sum_{i=1}^{n} L_i\right),$$

where n is the number of terms matching the prefix query, and L_i is the length of the posting list for the i-th matching term.

Therefore, if search-as-you-type queries need to be supported, then the inverted index might not be the optimal data structure; a suffix tree might lead to better results. Alternatively, a slight variation of the inverted index could be employed. One such variant is proposed by Bast et al. [6] [7]. In their index, the terms in the search engine's dictionary are split into groups. All terms in the same group share a common prefix. They also share a list in the inverted index, corresponding to the list of all occurrences by any of the terms in a given group. Additional information is added to each such list that enables the search engine at query time to decide which posting in the combined list corresponds to which term in the dictionary. Bast's index structure can also be used to realize an auto-completion feature for partially specified query terms.

This variation of the original inverted index is sufficiently close to the original version so that I can ignore the problem of prefix queries and focus on pure keyword queries instead. Most of the issues associated with building and updating an inverted index translate directly to Bast's version of the inverted index.

4.2 Statistical Properties of Inverted Files

Throughout Chapters 4 and 6 of my thesis, I make extensive use of statistical properties of inverted files. In particular, I propose that partitioning the terms in the search engine's vocabulary into frequent terms and infrequent terms in many cases is a good idea, as it can reduce the storage requirements of the dictionary and also improve the performance of the search engine's index maintenance process. In this section, I lay the statistical baseplate for the analyses carried out later in this thesis.

4.2.1 Generalized Zipfian Distributions

It is widely assumed (e.g., [35], [49], [104]) that terms in a collection of text in a natural language roughly follow a generalized Zipfian distribution [117]:

$$f(T_i) \approx \left[\frac{c}{i^\alpha} \right], \tag{4.1}$$

where T_i is the i-th most frequent term, $f(T_i)$ is the collection frequency of T_i, i.e., the number of times the term appears in the collection ("$[\cdots]$" denotes rounding to the nearest integer); c and α are collection-specific constants. In his original work, Zipf [117] postulated $\alpha = 1$.

The assumption that terms in a text collection follow a Zipfian distribution is quite common in information retrieval, especially when modeling the number and the length of posting lists in an inverted index. In the context of index maintenance, for instance, Cutting and Pedersen [35] make the same assumption when discussing whether the available main memory should be used as a cache for an on-disk B-tree data structure or as temporary buffer for incoming postings.

Equation 4.1 is equivalent to the assumption that the probability of an occurrence of the term T_i, according to a unigram language model, is

$$\Pr(T_i) = \frac{c}{N \cdot i^\alpha}, \tag{4.2}$$

where N is the total number of tokens in the text collection. In order for this probability distribution to be well-defined, the following equation needs to hold:

$$\sum_{i=1}^{\infty} \Pr(T_i) = \frac{c}{N \cdot i^\alpha} = 1. \tag{4.3}$$

Because N and c are constants, this implies the convergence of the sum

$$\sum_{i=1}^{\infty} \frac{1}{i^\alpha}$$

and thus requires that α be greater than 1. If $\alpha > 1$, then the sum's value is given by

Riemann's Zeta function $\zeta(\alpha)$ and can be approximated in the following way:

$$\sum_{i=1}^{\infty} \frac{1}{i^\alpha} \approx \gamma + \int_1^\infty \frac{1}{x^\alpha} dx = \gamma + \frac{1}{\alpha - 1}, \tag{4.4}$$

where

$$\gamma = -\lim_{n \to \infty} \left(\ln(n) - \sum_{i=1}^{n} \frac{1}{i} \right) \approx 0.577216 \tag{4.5}$$

is the Euler-Mascheroni constant [111]. The quality of the approximation in (4.4) depends on the value of the collection-specific parameter α. For realistic values ($\alpha < 1.4$), the approximation error is less than 1%. The integral representation of Riemann's Zeta function is far more accessible to analysis than its representation as a sum. I therefore exclusively use the former when modeling the term distribution in a given text collection.

As a notational simplification, I refer to the term $\gamma + \frac{1}{\alpha-1}$ as γ^*_α. From this definition, it follows that the value of the constant c in equations 4.1 ff. is:

$$c = \frac{N}{\gamma^*_\alpha} \text{ and thus } f(T_i) = \frac{N}{\gamma^*_\alpha \cdot i^\alpha}, \tag{4.6}$$

where N again is the number of tokens in the text collection.

Estimating the Size of the Active Vocabulary

The above definition of the term frequency function f assumes an infinite vocabulary. Obviously, a finite text collection only contains terms from a finite subset of this infinite vocabulary. Sometimes, it is useful to obtain an estimate of the number of terms that actually appear in a text collection of size N, that is, the size of the collection's *active vocabulary*. One such estimate is provided by Heaps's law [54], which states that $V(N)$, the size of the active vocabulary for a text collection with N tokens, is:

$$V(N) = k \cdot N^\beta. \tag{4.7}$$

Unfortunately, it is not clear whether Heaps's law is consistent with the generalized form of Zipf's law. That is, it is possible that there are no parameters α, β and k, such that

$$\sum_{i=1}^{k \cdot N^\beta} \frac{N}{\gamma^*_\alpha \cdot i^\alpha} \equiv N, \tag{4.8}$$

where N is a free variable. Moreover, there is no theoretical justification for equation 4.7.

To make sure that Heaps and Zipf play well together, I will try to derive $V(N)$ directly from the definition of the Zipfian term distribution. Modeling the number of distinct terms in the active vocabulary of a text collection is straightforward. The probability that the i-th most frequent term in the collection's infinite vocabulary actually appears in the collection is:

$$\Pr[T_i \in \mathcal{V}(N)] = 1 - \left(1 - \frac{1}{\gamma^*_\alpha \cdot i^\alpha}\right)^N. \tag{4.9}$$

Hence, the expected size of the collection's active vocabulary is:

$$\mathrm{E}[V(N)] = \sum_{i=1}^{\infty} 1 - \left(1 - \frac{1}{\gamma^*_\alpha \cdot i^\alpha}\right)^N. \tag{4.10}$$

Unfortunately, this sum does not appear to have a closed-form solution. Thus, it might be difficult to use this representation for further computations.

Fortunately, it is possible to obtain an approximation of $V(N)$ by not considering all terms in the universe, but only the set of terms for which Zipf's rule (rounded to the nearest integer) predicts a non-zero number of occurrences. That is, the set of terms T_i with $f(T_i) \geq 0.5$:

$$f(T_i) \geq 0.5 \quad \Leftrightarrow \quad \frac{N}{\gamma^*_\alpha \cdot i^\alpha} \geq 0.5 \quad \Leftrightarrow \quad i \leq \left(\frac{2N}{\gamma^*_\alpha}\right)^{1/\alpha}. \tag{4.11}$$

This leads to the following approximation of the number of terms in the active vocabulary of a collection of size N:

$$V'(N) = \left\lfloor \left(\frac{2N}{\gamma^*_\alpha}\right)^{1/\alpha} \right\rfloor. \tag{4.12}$$

It turns out that this is exactly as predicted by Heaps's law, with parameters

$$k = \left(\frac{2}{\gamma^*_\alpha}\right)^{1/\alpha} \quad \text{and} \quad \beta = \frac{1}{\alpha}.$$

Hence, assuming a generalized Zipfian distribution for the terms in a given text collection is consistent with estimating the size of the collection's vocabulary according to Heaps's law. If it is in fact valid to model term occurrences according to a Zipfian distribution, then this implies that the size of a text collection's vocabulary does not grow linearly with the size of the collection, but considerably slower:

$$V(N) \;\in\; \Theta(N^{1/\alpha}). \tag{4.13}$$

	SBFS collection	**TREC collection**	**GOV2 collection**
Zipf parameter	$\alpha = 1.12$	$\alpha = 1.22$	$\alpha = 1.34$
Predicted size of vocabulary	30.1 million	9.4 million	57.1 million
Actual size of vocabulary	28.7 million	2.1 million	49.5 million

Table 4.2: Using Zipf's law to predict the size of the active vocabulary for three different text collections. For SBFS and GOV2, the prediction is very accurate (error $< 16\%$).

Validation

It is not completely undisputed whether the distribution of terms in a text collection can be accurately modeled by a Zipfian distribution. Williams and Zobel [113], for example, argue (and provide experimental support) that the active vocabulary of a collection grows linearly with the number of tokens in the collection, contradicting equation 4.13. I therefore experimentally validate the assumption that a text collection can in fact be accurately modeled by a Zipfian distribution.

Figure 4.3 shows the rank/frequency curves in log/log-scale for the three collections used in my experiments and the curves for the corresponding Zipfian distributions. In each case, the distribution parameter α was chosen in such a way that the area between the two curves (after the log-log transformation) is minimized. As can be seen from the figure, both SBFS and GOV2 can be represented rather accurately by a Zipfian distribution. For the TREC collection, the approximation is not quite as good, mainly because the long tail on the right-hand side of the plot deviates from the distribution predicted by Zipf.

Table 4.2 shows the size of the active vocabulary for all three collections, comparing the actual size of the vocabulary with the size predicted by modeling it according to a Zipfian distribution with appropriate parameter α. For SBFS and GOV2, the estimate is very accurate (relative error for SBFS: 5%; GOV2: 15%). For TREC, however, the model overestimates the number of unique terms in the collection dramatically, by about 450%. The reason for this could be that the documents in the TREC collection are newswire articles. These articles have a controlled vocabulary, missing colloquial language and exhibiting a small number of typos.

Finally, Figure 4.4 shows the size of the active vocabulary for 1-gigatoken, 2-gigatoken, ... prefixes of the GOV2 collection (files sorted in lexicographical order). It can be seen that the Zipfian distribution (here with parameter $\alpha = 1.34$) more or less accurately predicts the number of distinct terms in each prefix of the collection. The reason why the curve for GOV2 is not convex is that the collection is quite heterogeneous. While the beginning of GOV2, when

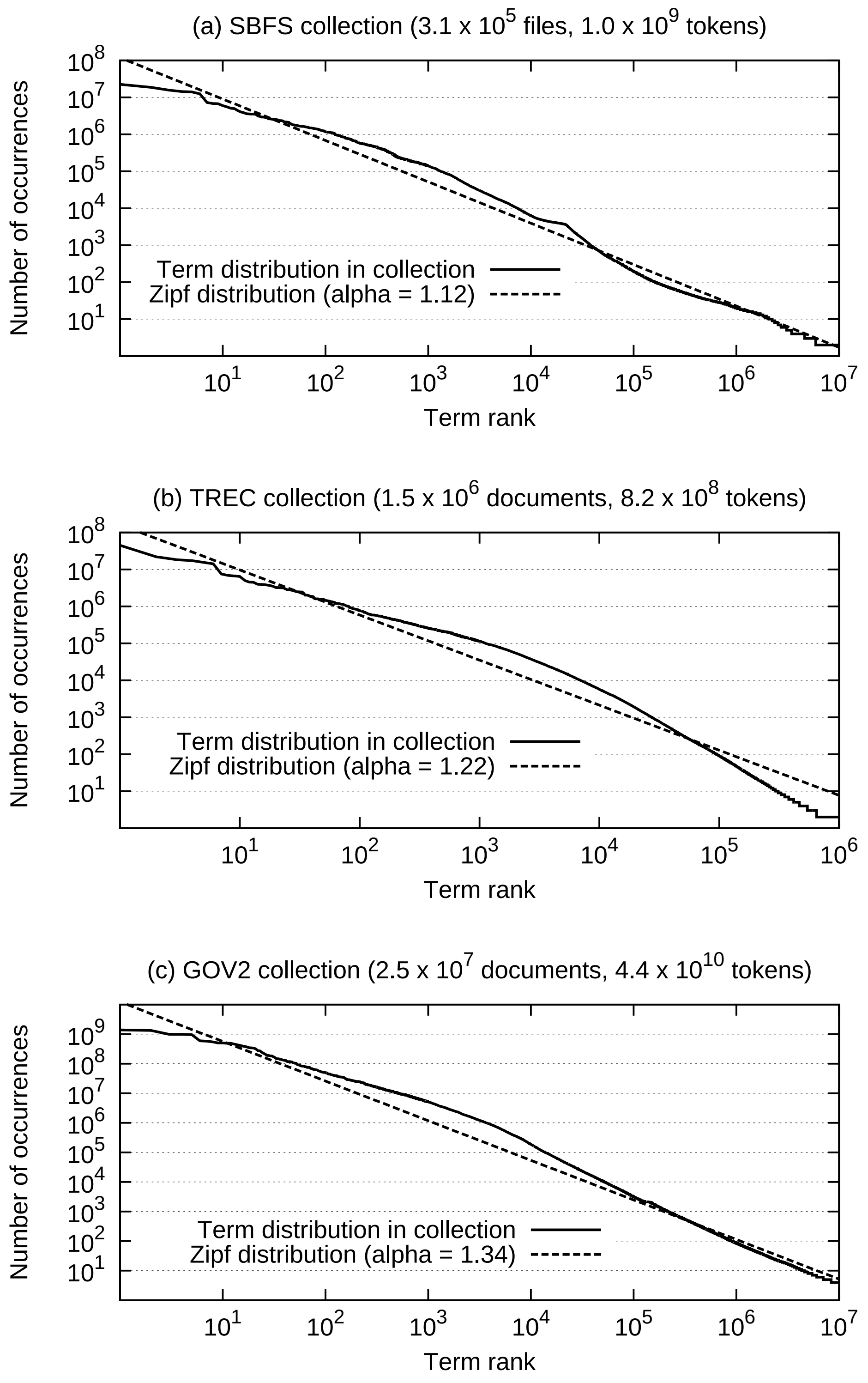


Figure 4.3: Validating Zipf's law experimentally. For each of the three text collections, a Zipf parameter α is determined that minimizes the area between the Zipf curve (after the log/log transformation) and the actual term distribution curve, as defined by the respective collection.

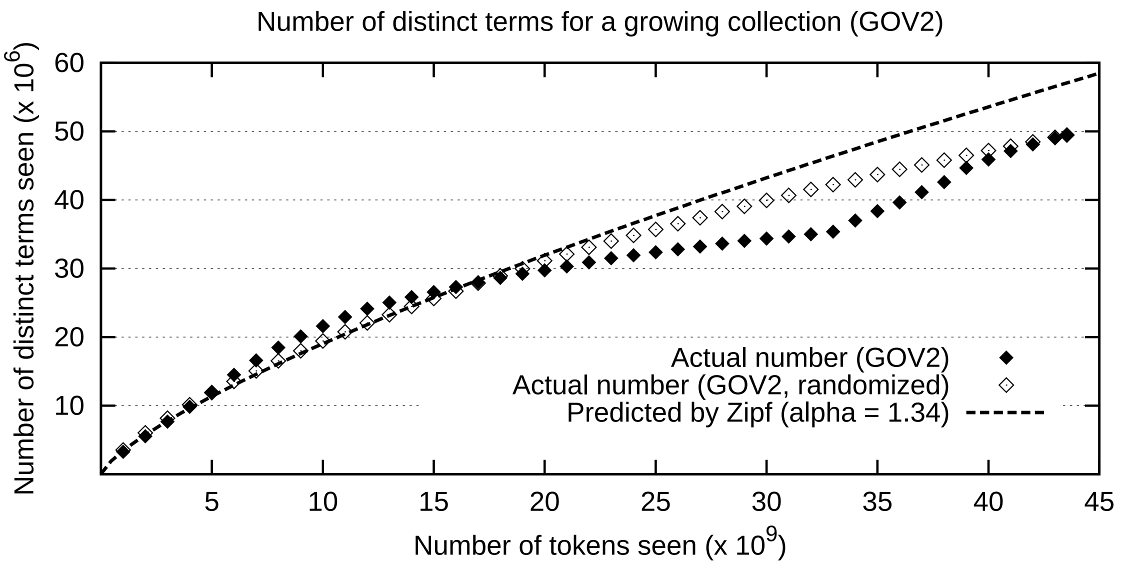


Figure 4.4: Using Zipf to predict the number of distinct terms seen in various prefixes of the GOV2 collection, simulating the scenario of a growing text collection. As can be seen, modeling the collection by Zipfian distribution leads to a very accurate approximation of the number of distinct terms.

processing the collection in crawl order, is dominated by HTML files, documents towards the end ($>$ 34 billion tokens seen) tend to be PDF and PostScript files. These file types, however, tend to contain more noise than HTML documents, mainly due to text extraction problems. If documents are indexed in random order, this artifact disappears.

Combining the evidence presented above, I conclude that term occurrences in a text collection do in fact follow a generalized Zipfian distribution with some collection-specific parameter $\alpha > 1$. While the TREC collection is a bit of an outlier, both SBFS and GOV2 allow an accurate approximation of various collection characteristics.

4.2.2 Long Lists and Short Lists

For many aspects of creating and maintaining an inverted index, it is useful to distinguish between *frequent terms* and *infrequent terms* and thus between *long posting lists* and *short posting lists*. This is done by choosing a threshold value, ϑ. Then, by definition, all posting lists containing more than ϑ entries are *long*, while all lists containing fewer than ϑ entries are *short*.

Under the assumption that the term distribution in a text collection can be modeled by a generalized Zipfian distribution, I now derive an approximation of the number of terms in a text collection of size N whose posting list contains more than ϑ entries, denoted as $L(N, \vartheta)$.

From the definition of the Zipfian term frequency function $f(T_i)$ in equation 4.6, we know:

$$f(T_i) > \vartheta \;\;\Leftrightarrow\;\; i^\alpha < \frac{N}{\gamma^*_\alpha \cdot \vartheta} \;\;\Leftrightarrow\;\; i < \left(\frac{N}{\gamma^*_\alpha \cdot \vartheta}\right)^{1/\alpha}.$$

Hence, we have

$$L(N,\vartheta) = \left\lfloor \left(\frac{N}{\gamma^*_\alpha \cdot \vartheta}\right)^{1/\alpha} \right\rfloor \;\in\; \Theta\left(\frac{N^{1/\alpha}}{\vartheta^{1/\alpha}}\right). \qquad (4.14)$$

The total number of postings found in these lists, denoted as $\hat{L}(N,\vartheta)$, then is

$$\begin{aligned}
\sum_{i=1}^{\lfloor L(N,\vartheta)\rfloor} f(T_i) &= \frac{N}{\gamma^*_\alpha} \cdot \sum_{i=1}^{\lfloor L(N,\vartheta)\rfloor} \frac{1}{i^\alpha} \\
&\approx \frac{N}{\gamma^*_\alpha} \cdot \left(\gamma + \int_1^{L(N,\vartheta)} x^{-\alpha}\, dx\right) \\
&= \frac{N}{\gamma^*_\alpha} \cdot \left(\gamma + \left(\frac{(L(N,\vartheta))^{1-\alpha}}{1-\alpha} - \frac{1}{1-\alpha}\right)\right) \\
&= N - \frac{N}{\gamma^*_\alpha \cdot (\alpha - 1)} \cdot \left(\frac{N}{\gamma^*_\alpha \cdot \vartheta}\right)^{\frac{1-\alpha}{\alpha}} \\
&= N - N^{1/\alpha} \cdot \frac{\vartheta^{(1-1/\alpha)}}{(\alpha-1)(\gamma^*_\alpha)^{1/\alpha}}. \qquad (4.15)
\end{aligned}$$

It immediately follows that the total number of postings found in short lists (lists containing no more than ϑ entries), denoted as $\hat{S}(N,\vartheta)$, is

$$\begin{aligned}
\hat{S}(N,\vartheta) &= N - \hat{L}(N,\vartheta) \\
&= \frac{N^{1/\alpha} \cdot \vartheta^{(1-1/\alpha)}}{(\alpha-1)(\gamma^*_\alpha)^{1/\alpha}} \;\in\; \Theta\left(\vartheta^{(1-1/\alpha)} \cdot N^{1/\alpha}\right). \qquad (4.16)
\end{aligned}$$

In other words, the relative number of postings found in short lists converges to zero, as N marches towards infinity. The speed of convergence depends on the Zipf parameter α.

4.2.3 Notation

Later in this thesis, I will make use of the results obtained above. I therefore introduce the following notation to refer to various statistical properties of an inverted index:

- $V(N)$, the size of the active vocabulary of a text collection containing N tokens:

$$V(N) \approx \left(\frac{2N}{\gamma^*_\alpha}\right)^{1/\alpha} \;\in\; \Theta\left(N^{1/\alpha}\right).$$

- $L(N,\vartheta)$, the number of long lists ($> \vartheta$ entries) in an index for a collection of size N:

$$L(N,\vartheta) \approx \left(\frac{N}{\gamma_\alpha^* \cdot \vartheta}\right)^{1/\alpha} \in \Theta\left(\left(\frac{N}{\vartheta}\right)^{1/\alpha}\right).$$

- $S(N,\vartheta)$, the number of short lists ($\leq \vartheta$ entries) in an index for a collection of size N:

$$\begin{aligned} S(N,\vartheta) &= V(N) - L(N,\vartheta) \\ &\approx \left(\frac{2N}{\gamma_\alpha^*}\right)^{1/\alpha} - \left(\frac{N}{\gamma_\alpha^* \cdot \vartheta}\right)^{1/\alpha} \in \Theta\left(N^{1/\alpha}\right). \end{aligned}$$

- $\hat{L}(N,T)$, the total number of postings found in long lists:

$$\hat{L}(N,\vartheta) \approx N - \frac{N^{1/\alpha} \cdot \vartheta^{(1-1/\alpha)}}{(\alpha-1)(\gamma_\alpha^*)^{1/\alpha}} \in \Theta(N).$$

- $\hat{S}(N,\vartheta)$, the total number of postings found in short lists:

$$\begin{aligned} \hat{S}(N,\vartheta) &= N - \hat{L}(N,\vartheta) \\ &\approx \frac{N^{1/\alpha} \cdot \vartheta^{(1-1/\alpha)}}{(\alpha-1)(\gamma_\alpha^*)^{1/\alpha}} \in \Theta\left(\vartheta^{(1-1/\alpha)} \cdot N^{1/\alpha}\right). \end{aligned}$$

This notation will be used in the following section to estimate the amount of internal fragmentation in the extensible posting lists used in hash-based in-memory construction, and in Section 4.4 to obtain an approximation of the memory requirements of the search engine's in-memory dictionary.

4.3 Index Construction

In Section 2.4.3, I have provided a general discussion of different index construction methods, coming to the conclusion that a merge-based approach, in combination with hash-based in-memory inversion, leads to the best indexing performance. I now investigate two particular aspects of this process. First, I examine possible implementations of the extensible in-memory posting lists used in in-memory index construction. Second, I discuss how the final merge phase in merge-based index construction can be sped up by merging compressed lists, eliminating the need to decompress posting lists during the merge operation.

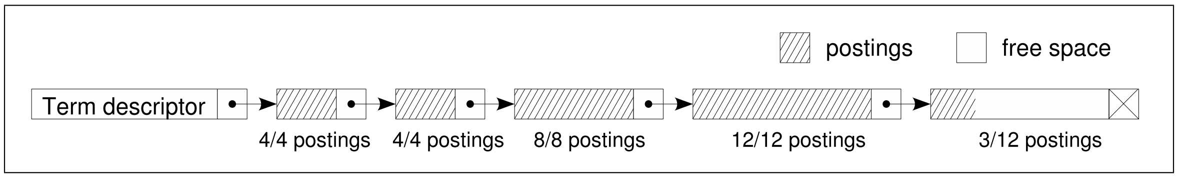


Figure 4.5: Grouping consecutive postings when maintaining extensible in-memory posting lists, here with parameters $(\mathit{init}, k, \mathit{limit}) = (4, 2, 12)$.

4.3.1 In-Memory Index Construction: Extensible Posting Lists

From Table 3.1 in Chapter 3, it is clear that the amount of indexable data in a file system easily exceeds the available main memory and *a fortiori* the main memory that can reasonably be allocated to the indexing process. Hence, pure in-memory index construction techniques cannot be applied when building a content index for the file system. Nonetheless, because in-memory indexing methods form the basis of every merge-based index construction algorithm (cf. Section 2.4.3), it is worth discussing possible optimizations.

Dictionary organization for in-memory index construction have been thoroughly discussed by Heinz et al. [56] [118]. A last remaining question is how to organize the in-memory posting lists. Two competing strategies are (a) to use linked lists, adding a pointer for every posting, and (b) to use resizable arrays (realized through `realloc` or a similar mechanism) [55]. If resizable arrays are used, they are normally managed according to a proportional pre-allocation strategy. Initially, space is reserved for a small number of postings, denoted as *init* (e.g., $\mathit{init} = 4$). Whenever the free space in a given list is exhausted, new space is allocated:

$$s_{\text{new}} = \max\{s_{\text{old}} + \mathit{init}, \lceil k \cdot s_{\text{old}} \rceil\}, \tag{4.17}$$

where s_{old} is the old size of the array, and s_{new} is its new size, after reallocation.

The linked-list approach increases the memory consumption of the indexing process by 50%-100%, depending on the relative size of postings and pointers. The `realloc` approach, on the other hand, suffers from internal fragmentation (if the pre-allocation factor is too large) or decreased indexing performance because of costly list relocations (if the pre-allocation factor is chosen too small). For a pre-allocation factor k, the expected amount of internal fragmentation is approximately $\frac{k-1}{2k}$. Choosing $k = 1.2$, for instance, is expected to lead to an internal fragmentation of 8.3% ($k = 1.3$: 11.5%).

As an alternative to linked lists and `realloc`, it is possible to combine the two strategies, resulting in a technique referred to as *grouping* [20]. Like in the case of `realloc`, an initial buffer holding up to *init* postings is reserved for each list. However, whenever the buffer space

	Realloc			**Grouping**		
$k =$	1.1	1.2	1.3	1.1	1.2	1.3
Indexing time (total)	27.6 sec	25.8 sec	25.2 sec	21.5 sec	22.4 sec	21.9 sec
Indexing time (CPU)	23.7 sec	21.5 sec	20.7 sec	18.0 sec	18.0 sec	18.1 sec
Internal fragmentation	5.2%	8.8%	12.1%	4.4%	4.4%	4.5%

Table 4.3: Relative performance of extensible in-memory posting lists, realized through `realloc` (*init* = 4) and through linked lists with grouping (*init* = 4, *limit* = 128). Because of its simplistic memory allocation procedure, grouping is between 15% and 30% faster than `realloc`. Text collection being indexed: 100,000 random documents from TREC (45 million postings in total).

is exhausted, the list is not relocated. Instead, a new buffer is allocated, according to the rule

$$s_{\text{new}} = \min\{\mathit{limit}, \max\{\mathit{init}, \lceil (k-1) \cdot s_{\text{total}} \rceil\}\}, \tag{4.18}$$

where *limit* is a pre-defined upper bound (e.g., *limit* = 128), and s_{total} is the total number of postings accumulated so far for the given term. Different posting buffers allocated for the same term are organized in a linked list. That is, each buffer holds a pointer to the next buffer for the given term. This procedure is visualized by Figure 4.5. By keeping the value of the *limit* parameter small, internal fragmentation in long lists can be virtually eliminated. According to Zipf's law, this means that the average internal fragmentation of all lists is kept low as well. Compared to the initial linked-list approach (which is equivalent to the parameter setting *init* = *limit* = 1), the relative pointer overhead is drastically reduced, to $\frac{1}{\mathit{init}}$ for short lists and $\frac{1}{\mathit{limit}}$ for long lists.

The relative amount of internal fragmentation in the extensible posting lists can be estimated via Zipf's law:

$$\begin{aligned} \mathit{relFrag} &\approx \frac{1}{N} \sum_{i=1}^{V(N)} \frac{\max\{\mathit{init}, \min\{\mathit{limit}, f(T_i) \cdot (k-1)\}\}}{2} && (4.19) \\ &= \frac{1}{N} \sum_{i=1}^{(2N/\gamma_\alpha^*)^{1/\alpha}} \frac{\max\{\mathit{init}, \min\{\mathit{limit}, N/(\gamma_\alpha^* \cdot i^\alpha) \cdot (k-1)\}\}}{2} && (4.20) \end{aligned}$$

(assuming that, on average, 50% of the pre-allocated space is unused). For $\alpha = 1.22$ (TREC), $N = 45$ million, *init* = 4, *limit* = 128, and $k = 1.2$, the estimated amount of fragmentation is 4.8%. For the same value of k, the expected amount of fragmentation in the case of `realloc` would be 8.3% ($= \frac{k-1}{2k}$).

Table 4.3 shows the experimental results obtained for the two memory management strategies, `realloc` and grouping, by indexing 100,000 random documents from the TREC collection. As expected, `realloc` leads to higher internal fragmentation than grouping. In order for the former to achieve a fragmentation level close to that of the latter, the pre-allocation parameter k needs to be reduced so far that reallocations and copy operations become a major bottleneck, increasing `realloc`'s CPU time by up to 32% compared to grouping ($k = 1.1$; 23.7 sec vs. 18.0 sec). The data points in the row labeled "Internal fragmentation" include unused list space as well as the pointer overhead caused by the pointers in the grouping approach.

Both pre-allocation strategies, grouping and `realloc`, can be combined with on-the-fly index compression, encoding postings as delta values before they are appended to the respective list. If on-the-fly index compression is applied, however, then the encoding performance of the compression algorithm employed may become a bottleneck of the index construction process. Simple byte-aligned methods, such as vByte [100], seem to be an appropriate choice.

4.3.2 Merge-Based Index Construction: The Final Merge Operation

After a set of on-disk sub-indices for the given text collection has been created by repeatedly invoking an in-memory index construction algorithm, they need to be merged into the final index for the collection. Because posting lists are stored in the sub-indices according to some predefined, static ordering, usually the lexicographical ordering of their respective terms, this merge operation can be performed very efficiently, as shown in Algorithm 4.

Because of the way in which merge-based indexing partitions the text collection when creating the set of sub-indices, we know that for every term T:

$$\mathit{inputIndices}[j].\mathit{list}(T).\mathit{lastPosting} < \mathit{inputIndices}[k].\mathit{list}(T).\mathit{firstPosting} \ \ \forall\ j < k \tag{4.21}$$

(assuming $\mathit{inputIndices}[i].\mathit{list}(T).\mathit{lastPosting} = -\infty$, and $\mathit{inputIndices}[i].\mathit{list}(T).\mathit{firstPosting} = +\infty$ if T does not appear in $\mathit{inputIndices}[i]$). That is, the list segments found in different sub-indices are well-ordered and non-overlapping. Therefore, they can be merged by simply concatenating them, as shown in the algorithm.

In order to minimize disk I/O activity, the sub-indices created in the first phase of merge-based index compression are stored on disk in compressed form. In particular, all posting lists are stored in compressed form. Algorithm 4 seems to suggest that, in order to perform the final merge operation, the individual list segments needs to be decompressed before they can be concatenated. However, this is not necessary. Suppose posting lists are compressed using one of the gap compression techniques described in Section 2.4.4. Conceptually, a

Algorithm 4 Final merge procedure in merge-based index construction. Input parameters: *inputIndices*[1..*n*], an array of *n* index iterators, returning lists in the respective index, one at a time. Output parameter: *outputIndex*.

```
 1: sort $inputIndices[1..n]$ by $inputIndices[i].currentTerm$,
      using $inputIndices[i].currentList.firstPosting$ to break ties
 2: // $inputIndices[1..n]$ has heap property now
 3: $currentTerm \leftarrow nil$
 4: $outputList \leftarrow \emptyset$
 5: while $inputIndices[1].currentTerm \neq nil$ do
 6:   if $inputIndices[1].currentTerm \neq currentTerm$ then
 7:     $outputIndex.addPostings(currentTerm, outputList)$
 8:     $outputList.clear()$
 9:     $currentTerm \leftarrow inputIndices[1].currentTerm$
10:   end if
11:   assert($outputList.length = 0 \ \vee \ outputList.last < inputIndices[1].currentList.first$)
12:   $outputList.appendList(inputIndices[1].currentList)$
13:   $inputIndices[1].advanceToNextTerm()$
14:   restore heap property by moving $inputIndices[1]$ down the tree
15: end while
16: $outputIndex.addPostings(currentTerm, outputList)$
```

gap-compressed posting list L, comprising $|L|$ elements, can be thought of as a tuple

$$L = (\ |L|, L_1, \langle \Delta_L(2), \ldots, \Delta_L(|L|) \rangle\ ),$$

where $\Delta_L(i) = L_i - L_{i-1}$. By sacrificing a little bit of compression and encoding $L_{|L|}$ directly (or through the difference $L_{|L|} - L_1$) instead of storing $\Delta_L(|L|)$, it is possible to provide efficient access to the first and the last element of the compressed list:

$$L = (\ |L|, L_1, L_{|L|}, \langle \Delta_L(2), \ldots, \Delta_L(|L|-1) \rangle\ ).$$

Using this new representation of compressed posting lists, it is possible to concatenate two lists P and Q ($P_{|P|} < Q_1$) without the need to decompress them first:

$$P \circ Q = (\ |P| + |Q|, P_1, Q_{|Q|}, \langle \Delta_P(2), \ldots, \Delta_P(|P|), Q_1 - P_{|P|}, \Delta_Q(2), \ldots, \Delta_Q(|Q|-1) \rangle\ ).$$

This makes the final merge operation in merge-based index construction very light-weight, as partial posting lists no longer need to be decompressed, but may simply be copied from the input indices to the output index. In order for this to be as efficient as possible, however, all lists must be encoded using a byte-aligned compression method, such as vByte [100].

Collection	**Merging decompressed lists**		**Merging compressed lists**	
	Total time	CPU time	Total time	CPU time
SBFS	279.5 .. 282.3 sec	168.8 .. 171.0 sec	241.7 .. 243.7 sec	116.8 .. 118.0 sec
TREC	104.4 .. 106.1 sec	62.7 .. 62.9 sec	84.0 .. 88.7 sec	27.7 .. 28.3 sec
GOV2	96.1 .. 97.6 min	28.3 .. 28.6 min	80.2 .. 83.8 min	14.4 .. 14.6 min

Table 4.4: Performance improvement achieved by merging lists without prior decompression. Posting lists are stored on disk in compressed form (vByte). All performance figures represent 95% confidence intervals, obtained by running each experiment 6 times.

Table 4.4 shows the time consumed by the final merge operation when indexing the three different text collections, assuming that all on-disk posting lists are stored in vByte-compressed form. By merging posting lists in their compressed form, without prior decompression, CPU time can be decreased by between 31% (SBFS) and 61% (TREC). As a result, the CPU utilization during the final merge operation is below 50% for all three collections.

The relative CPU time, compared to the total time of the merge operation, is lowest for SBFS and highest for GOV2, suggesting that there might be a relationship between the CPU utilization and the collection's Zipf parameter α. And indeed, this is the case. A smaller α value leads to a greater number of unique terms and thus a higher CPU utilization. Comparing term strings is more costly than copying compressed postings.

4.3.3 Performance Baseline

Based on the optimizations achieved for the in-memory indexing part and the final merge part of merge-based index construction, I can now present an index construction performance baseline. Table 4.5 shows the indexing performance for all three text collections achieved by a merge-based index construction process, assuming that:

- the in-memory indexing process employs a move-to-front hash table for dictionary lookups and linked lists with grouping for extensible posting lists;
- incoming postings are compressed using vByte [100] before being appended to the respective posting list (on-the-fly compression);
- all on-disk posting lists (intermediate indices and final index) are stored in compressed form, using vByte;
- in the final merge operation, lists are concatenated without prior decompression.

It is interesting to see that, while reading and parsing the input files takes between 37% and 54% of the total time for GOV2 and TREC, respectively, the indexing process spends

Text collection	**Reading/parsing input**	**Indexing**	**Merging**	**Total**
SBFS	69.2 min	13.0 min	4.2 min	86.3 min
TREC	8.5 min	5.9 min	1.4 min	15.8 min
GOV2	139.1 min	148.7 min	82.9 min	371.7 min

Table 4.5: Index construction baseline, following a merge-based approach extending upon hash-based in-memory inversion. Memory limit: 32 MB for SBFS and TREC; 256 MB for GOV2.

over 80% of its time reading input files when building an index for SBFS. The reason for this is the large number of PDF and PS files in the SBFS collection. These files require external conversion tools, and the associated parsing routines can be very costly. Thus, when indexing the contents of the file system, the bottleneck does not seem to be the actual index construction process, but the external conversion programs that are responsible for transforming the input files into a format that can be processed by the indexer.

For all three text collections, however, it seems computationally feasible to update the index by scanning the collection once a day and building a new index for it. As long as the computer system is idling for at least a few hours every day, this approach appears to constitute a valid update strategy. Its deficiency, of course, is that it allows the index to be inconsistent with the contents of the changing text collection for up to 24 hours.

4.4 Memory Requirements

After the index has been created, it needs to be accessed whenever a user submits a search query. Every index access has to go through the search engine's dictionary, which translates query terms into addresses of on-disk posting lists (cf. Section 2.4.1). In order for this procedure to be as efficient as possible, it is desirable to keep the dictionary in main memory at all times.

Unlike in enterprise search and Web search, however, a file system search engine does not run on a dedicated search server. Instead, it has to share all its resources with user applications that run on the same machine as the search engine. This implies that the search engine's memory requirements should be kept as low as possible (Requirement 4 from Chapter 1). It then seems sensible to reduce the size of the search engine's dictionary as far as possible, so as to minimize the impact that the file system search engine has on other processes running in the system.

Compression method	**SBFS**	**TREC**	**GOV2**
None	598.5 MB	42.4 MB	1047.7 MB
Front coding + vByte (group size: 256)	157.1 MB	9.1 MB	243.5 MB
Front coding + vByte + LZ (group size: 256)	116.6 MB	5.6 MB	163.8 MB

Table 4.6: Total size of a sort-based in-memory dictionary when dividing dictionary entries into groups of 256 terms each and applying dictionary compression techniques to each group.

4.4.1 Dictionary Compression

One way to decrease the memory requirements of the search engine's dictionary is to apply dictionary compression techniques, as discussed in Section 2.4.4. Table 4.6 summarizes the savings that can be achieved by storing the in-memory dictionary in compressed form, following the dictionary-as-a-string approach, compressing terms by front coding and storing file pointers (positions of on-disk posting lists) in vByte-encoded form. Both techniques, front coding and vByte, are applied to groups of dictionary entries, comprising 256 terms each.

One shortcoming of the combination of front coding and vByte is that they do not take the bigger context of the data being compressed into account, but only encode a dictionary entry by giving reference to its immediate predecessor. This can be remedied by running a standard, general-purpose compression algorithm over each of the 256-term dictionary blocks after they have been compressed by the combination of front coding and vByte. The row labeled "Front coding + vByte + LZ" demonstrates the additional savings that can be obtained by following this path (utilizing the LZ implementation that is part of the zlib[3] compression library). For the TREC collection, for instance, the total memory requirement of the dictionary can be reduced by 38% over the compression achieved by front coding + vByte, leading to a total saving of 87% compared to an uncompressed dictionary.

Unfortunately, these savings are insufficient to keep the search engine's dictionary in main memory. For both SBFS and GOV2, with their large number of unique terms (SBFS: 28.7 million, GOV2: 49.5 million), the compressed dictionary still exhibits a memory consumption in excess of 100 MB. Presumably, few users are willing to permanently dedicate 150 MB (in-memory dictionary + buffers for the indexing process) of their computer's memory resources to the search engine.

A possible solution to this problem is to store the dictionary on disk instead of in main memory. However, at query time this causes an additional disk seek for each query term, because now the term's dictionary entry as well as its posting list need to be fetched from disk. Since we do not know how many query terms there are or how costly a disk seek is compared

[3]http://www.zlib.net/ (accessed 2007-08-01)

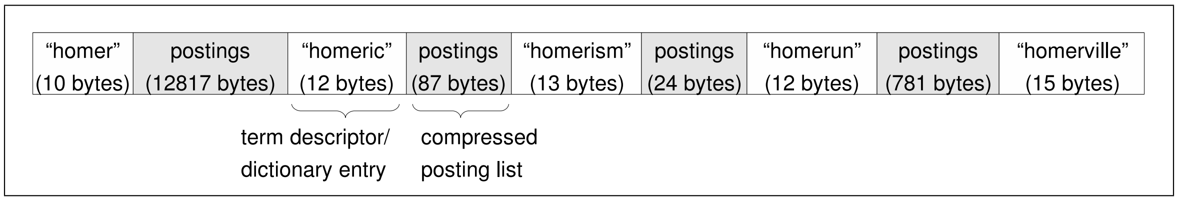


Figure 4.6: Interleaving posting lists and dictionary entries in the on-disk index. By keeping a term's dictionary entry close to its posting list, only a single disk seek is necessary to access both.

to the length of each query term's posting list (cf. Requirement 2 from Chapter 1), this extra disk seek might or might not have a disastrous impact on query processing performance.

4.4.2 Interleaving Posting Lists and Dictionary Entries

Fortunately, the extra disk seek per query term can be avoided by storing each term's dictionary entry on disk, right before the term's posting list. This way, both can be accessed in one sequential read operation, without the need for an additional disk seek. However, how is the search engine going to find a term's dictionary entry under these new circumstances? The idea is to still keep *some* dictionary entries in memory, but not all of them.

Consider the on-disk index layout shown in Figure 4.6, representing a fragment of the inverted file for a hypothetical text collection. For each term in the collection, the index contains a term descriptor and a posting list. The term descriptor contains the term itself, and also some meta-data that are useful for accessing its posting list (e.g., a set of synchronization points for the posting list). It is stored in the on-disk index, immediately preceding the postings.

Because all terms and posting lists in the inverted file are sorted in lexicographical order, it is not necessary to keep the dictionary entry for every term in main memory. Suppose a user submits the query "homerism" to the search engine. The search engine might not know the exact index position of the posting list for "homerism", but maybe it knows the index position of "homeric" and "homerville". Then it can quickly find the posting list for "homerism", by performing a linear scan of the index data between "homeric" and "homerville" (929 bytes in the example).

As long as the distance between two terms for which the search engine has an in-memory dictionary entry is not too large, this operation can be performed very efficiently and only adds a negligible overhead to the total cost of processing a search query. For example, if the hard drive can deliver 50 MB/sec in a sequential read operation, and the distance between the on-disk data for two consecutive in-memory dictionary entries is guaranteed to be less than

64 KB, then the overhead per query term is less than 1.3 ms — a substantial improvement over the 8-10 ms that, on average, would be caused by an additional disk seek.

The incomplete in-memory dictionary that is necessary to provide efficient index access in the manner outlined above can be constructed incrementally, while the search engine is creating the inverted index. A new entry is added to the dictionary whenever the on-disk distance between the new term being added to the index and the term referred to by the last in-memory dictionary entry exceeds a predefined threshold B (e.g., $B = 64$ KB).

In the following, I refer to the index data stored between the two points in the on-disk inverted file defined by two consecutive in-memory dictionary entries as an *index segment.* Let B denote the block size of the index, i.e., the minimum size of each index segment. Based on the rules described above, it is clear that each index segment contains at least B bytes. Some segments, however, might contain more than B bytes, for example whenever the segment contains a posting list that consumes more than B bytes of index space.

Because each segment contains at least B bytes, the total number of in-memory dictionary entries for an inverted file consuming size($\mathcal{I}$) bytes of disk space cannot be more than $\frac{\text{size}(\mathcal{I})}{B}$. If $B = 2^{16}$, then this leads to no more than 2.69 GB / 64 KB = 44,095 dictionary entries for the SBFS collection — a dramatic improvement over the memory requirements exhibited by a complete in-memory dictionary, containing entries for all 28.7 million terms in the collection.

However, this upper bound is not very tight. Using Zipf's law, a tighter bound on the number of in-memory dictionary entries may be obtained. Suppose postings are stored in the on-disk index in compressed form, with each posting consuming p bytes on average. Suppose further that there is a per-term storage overhead of o bytes, caused by the term string itself and some term-specific meta-data. Now note that for each in-memory dictionary entry there can be at most one posting list with more than $\frac{B}{p}$ entries in the index segment referred to by the dictionary entry. Employing the notation established in Section 4.2, this results in $L(N, \frac{B}{p})$ entries for these long lists. I refer to this set of dictionary entries as $\mathcal{L}$:

$$|\mathcal{L}| \;=\; L(N, \frac{B}{p}) \;\in\; \mathcal{O}\left(\left(\frac{N}{B}\right)^{1/\alpha}\right). \tag{4.22}$$

$\mathcal{L}$ covers a major part of the on-disk index. There are, however, some remaining segments that do not contain any of the terms from $\mathcal{L}$. To estimate the combined amount of data stored in those index segments, I need to model the average size of their postings, based on the compression scheme employed. Suppose the posting lists in the index are compressed using vByte. Then the byte-size of the posting list for the i-th most frequent term, T_i, including term-specific meta-data, can be approximated as

$$\begin{aligned}\text{size}(T_i) &= o + f(T_i) \cdot \frac{\log_2(N/f(T_i))}{7} && (4.23)\\ &= o + \frac{N}{\gamma_\alpha^* \cdot i^\alpha} \cdot \frac{\log_2(\gamma_\alpha^* \cdot i^\alpha)}{7}, && \text{// using the definition of } f(T_i) \quad (4.24)\end{aligned}$$

because each gap δ in T_i's posting list requires about $\log_2(\delta)$ bits to be represented as a binary number, and vByte uses 7 bits per byte to store this information. With the additional definition

$$q := \left(\frac{N \cdot p}{\gamma_\alpha^* \cdot B}\right)^{1/\alpha} \qquad (4.25)$$

(i.e., $f(T_q) = \frac{B}{p}$), the total amount of data stored in on-disk index segments not covered by $\mathcal{L}$ is approximately

$$\begin{aligned} & \sum_{i=q}^{V(N)} o + \frac{N}{\gamma_\alpha^* \cdot i^\alpha} \cdot \frac{\log_2(\gamma_\alpha^* \cdot i^\alpha)}{7} && (4.26)\\ \leq\ & \frac{\log_2(2 \cdot N)}{7} \cdot \sum_{i=q}^{V(N)} o + \frac{N}{\gamma_\alpha^* \cdot i^\alpha} \qquad \text{// using (4.12)} && (4.27)\\ =\ & \frac{\log_2(2 \cdot N)}{7} \cdot \left(o \cdot S\left(N, \frac{B}{p}\right) + \hat{S}\left(N, \frac{B}{p}\right)\right) \qquad \text{// definition of } S \text{ and } \hat{S} && (4.28)\\ \in\ & \mathcal{O}\left(\log(N) \cdot \left(N^{1/\alpha} + N^{1/\alpha} \cdot B^{1-1/\alpha}\right)\right) \qquad \text{// since } o \text{ and } p \text{ are constant} && (4.29)\\ =\ & \mathcal{O}\left(\log(N) \cdot N^{1/\alpha} \cdot B^{1-1/\alpha}\right). && (4.30)\end{aligned}$$

The number of in-memory dictionary entries for those data then is at most:

$$\mathcal{O}\left(\frac{\log(N) \cdot N^{1/\alpha} \cdot B^{1-1/\alpha}}{B}\right) = \mathcal{O}\left(\log(N) \cdot \left(\frac{N}{B}\right)^{1/\alpha}\right) \subseteq \mathcal{O}\left(\left(\frac{N}{B}\right)^{1/\alpha+\varepsilon}\right). \qquad (4.31)$$

Combining (4.22) and (4.31), the total number of in-memory dictionary entries, for the whole index, is bounded in the following way:

$$\#\text{ dictionary entries} \in \mathcal{O}\left(\left(\frac{N}{B}\right)^{1/\alpha}\right) + \mathcal{O}\left(\left(\frac{N}{B}\right)^{1/\alpha+\varepsilon}\right) = \mathcal{O}\left(\left(\frac{N}{B}\right)^{1/\alpha+\varepsilon}\right). \qquad (4.32)$$

That is, the main memory requirements of the search engine's dictionary are strictly sub-linear in the size of the collection. The magnitude of the difference depends on the value of the Zipf parameter α. For small α (e.g., SBFS), the difference between linear growth

Collection	**Number of terms**	**Index size**	**Number of dictionary entries**		
			proportional	Zipf approx.	actual
SBFS	28.7 million	2.69 GB	44,095	32,601	30,351
TREC	2.1 million	1.39 GB	22,749	12,625	8,492
GOV2	49.5 million	62.18 GB	1,018,741	125,958	112,227

Table 4.7: Reducing memory consumption by using an incomplete dictionary, containing one entry per index segment. Size of each index segment: $\geq$ 64 KB.

and actual growth will be relatively small. For larger α (e.g., GOV2), the difference will be greater.

Table 4.7 shows the size of the resulting in-memory dictionaries for the three text collections, using equation 4.26 to estimate the combined size of the short posting lists, and assuming a segment size of 64 KB (resulting in a query-time overhead of at most 1-2 ms per query term). The column labeled "proportional" refers to the simplistic approximation of the number of dictionary entries obtained by dividing the size of the index by the segment size. The column labeled "Zipf approx." refers to the approximation obtained by applying Zipf's law. The column labeled "actual" refers to the actual number of in-memory dictionary entries for the given segment size (B = 64 KB).

It can be seen that the proposed dictionary organization leads to a very small memory footprint. Even for the GOV2 collection with its 49.5 million distinct terms, the incomplete in-memory dictionary does not consume more than a few megabytes. The size of the incomplete dictionary can be decreased even further, by using the same compression techniques as for the complete dictionary. This way, the size of the incomplete dictionary for GOV2 can be reduced to 927 KB (front coding + vByte; group size: 256 terms).

The approximation of the number of dictionary entries in the incomplete dictionary obtained through the application of Zipf's law is remarkably accurate for SBFS (error: 7.4%) and GOV2 (error: 12.2%). Only for the TREC collection, the number of dictionary entries is greatly overestimated; the error stems from the fact that TREC's active vocabulary is much smaller than predicted by Zipf (cf. Table 4.2).

4.5 Document-Centric vs. Schema-Independent Inverted Files

Thus far, I have assumed that the index being created for a given text collection is a schema-independent one. The results obtained, however, both about statistical properties of inverted files and about performance optimizations in the index construction process, can just as easily be applied to document-centric positional indices, containing explicit reference to the inherent structure of the text collection.

4.5.1 Query Flexibility

One of the main requirements in file system search is query flexibility (Requirement 2 from Chapter 1). We do not know the users' exact search needs, and we do not know what kind of search operation a third-party application might want to carry out, utilizing the infrastructure provided by the search engine. Hence, a content index with predefined document boundaries, such as a document-centric positional index, that is targeting a very specific type of retrieval operation (document retrieval) might be too restrictive in the context of general-purpose file system search.

Consider, for instance the `mbox` file format that is used by many e-mail clients to store their messages. An `mbox` file contains a sequence of e-mail messages. Normally, the messages in such a file would be treated as independent documents. It is, however, possible that a user is looking for the `mbox` file itself, or a thread of messages within a file, maybe remembering some of the keywords that the file contains — in different e-mails. This perfectly legitimate search operation would be very difficult to realize with a document-centric index, because the definition of what constitutes a document, and thus can be retrieved by the search engine, may not be changed after the index has been created.

A schema-independent index makes it possible to redefine the concept of a *document* on a per-query basis. For some queries, a document might be a file; for others it might be an e-mail message; for even others, it might be something completely different, maybe a passage in a PDF document. Because a schema-independent index makes no assumptions about what the unit of retrieval is going to be, it offers the greatest flexibility at query time. For this reason, I choose to exclusively use schema-independent indices in all experiments described in this thesis.

The great versatility of the schema-independent approach to text retrieval does, however, carry an unwanted piece of baggage: reduced query performance. If the unit of retrieval targeted by a search query is in fact always the same for all queries (i.e., the text collection is a *document collection* in a very strong sense), then a document-centric index, explicitly taking into account the special structure of the text collection, may lead to greatly reduced query response times.

4.5.2 Query Processing Performance

Consider again the relevance ranking functions listed in Section 2.2.2: Okapi BM25 [94], language modeling [116], and others. All these ranking functions are statistical. They require access to the text collection's global term statistics in order to rank a set of matching search results according to their relevance to the search query. BM25, for instance, needs access to the number of documents containing each query term. In a document-centric index, this information can be precomputed and be stored in the term's header data in the on-disk index, or maybe even in the in-memory dictionary. With a schema-independent index, because any reference to "documents" is meaningless until the actual query is received by the search engine, this information cannot be precomputed. It has to be calculated on-the-fly after the query has been received. These calculations can be rather costly.

Recall from Section 2.2.2 the calculation of the BM25 score for a document D relative to a keyword query $\mathcal{Q}$:

$$\text{score}_{\text{BM25}}(D, \mathcal{Q}) = \sum_{Q \in \mathcal{Q}} \log\left(\frac{|\mathcal{D}|}{|\mathcal{D}_Q|}\right) \cdot \frac{f_D(Q) \cdot (k_1 + 1)}{f_D(Q) + k_1 \cdot (1 - b + b \cdot |D| / \mathit{avgdl})}. \tag{4.33}$$

Before $\text{score}_{\text{BM25}}(D, \mathcal{Q})$ can be computed, the query processor needs to obtain two pieces of information: $|\mathcal{D}_Q|$, the number of documents containing Q (for each query term Q), and *avgdl*, the average length of a document in the collection. Assuming documents are delimited by XML-style markup of the form

`<doc> ··· </doc>`,

the value of *avgdl* can be obtained by evaluating all text passages matching the GCL expression

$$\text{“}\langle\text{doc}\rangle\text{”} \cdots \text{“}\langle/\text{doc}\rangle\text{”}$$

and computing their average length. Similarly, $|\mathcal{D}_Q|$ can be computed via the GCL expression

$$(\text{“}\langle\text{doc}\rangle\text{”} \cdots \text{“}\langle/\text{doc}\rangle\text{”}) \rhd Q.$$

For the large number of documents found in the text collections used in my experiments (TREC: 1.5 million documents; GOV2: 25.2 million documents), these operations can be very CPU-intensive. They also diminish the effect of query optimization strategies like MaxScore [109], because now all documents (or, rather, all postings corresponding to the document delimiters "⟨doc⟩" and "⟨/doc⟩") need to be visited no matter what, just to compute *avgdl*.

Table 4.8 shows performance results obtained for query processing with a schema-independent index for each of the three collections. It compares the pure schema-independent approach with a semi-document-centric approach in which all statistics (term weights and

	Collection	**Query set**	**Stat's pre-computed**	**On-the-fly**
QAP	TREC	TREC topics 1-500	151 ms/query	285 ms/query
	GOV2	TREC TB 2006 efficiency	3,022 ms/query	3,982 ms/query
BM25	TREC	TREC topics 1-500	97 ms/query	577 ms/query
	GOV2	TREC TB 2006 efficiency	1,876 ms/query	4,923 ms/query

Table 4.8: Query processing performance for schema-independent indices. The performance varies greatly, depending on the amount of query-relevant information precomputed and stored in the index.

average document length) are pre-computed, and the results to the GCL query "⟨doc⟩" ··· "⟨/doc⟩" are kept in an in-memory cache for fast access, avoiding the costly re-combination of "⟨doc⟩" and "⟨/doc⟩" for every query. The underlying index structure is still schema-independent, but is enriched with extensive caching of information that is shared among queries. While a true document-centric index can be expected to achieve a performance level that is even a little higher than that of the semi-document-centric approach, the differences are likely to be small.

It can be seen from the table that the performance gap between document-centric and schema-independent retrieval varies greatly, depending on the type of query executed by the search engine. While, for QAP queries, the slowdown is not excessively dramatic (using a schema-independent index decreases query performance by about 100%) on the TREC collection, the document-centric approach is almost 6 times as fast as the schema-independent one if BM25 queries are processed by the search engine. The statistics used by QAP (and also by ranking functions based on the language modeling approach) are less expensive to compute than for BM25.

Therefore, if search operations and ranking functions are chosen wisely, the slowdown caused by the overhead of computing term statistics and document boundaries on-the-fly may be acceptable. Moreover, there is no real alternative to a schema-independent index. A document-centric index – as pointed out above – does not offer the degree of flexibility that is desirable for file system search.

4.6 Summary

This chapter was mainly concerned with static inverted indices and how to integrate them into the file system. Taken by itself, its applicability to file system search is limited. However, it provides the foundation for the following two chapters on secure file system search (Chapter 5) and real-time index updates (Chapter 6).

In addition to the optimizations for hash-based and merge-based index construction laid out in Section 4.3, the main points of this chapter are:

- In a multi-user environment, it is essential to have multiple users share the same index. Otherwise, indexing overhead and storage requirements of the search engine are no longer bounded by the total size of the indexable data in the file system. This requirement will raise certain security issues in Chapter 5.
- The dictionary interleaving technique from Section 4.4.2 can be used to decrease the memory requirements of an inverted index to under a megabyte, even for very large text collections containing millions of different terms. Dictionary interleaving leads to a minor slowdown at query time, because slightly more data need to be read from disk. However, this slowdown is negligible compared to the other costs that accrue during query processing.

  Interleaving dictionary entries with on-disk posting lists is only possible if all posting lists are stored in some predefined order (e.g., lexicographical). This has implications for the competitiveness of index update strategies that allow on-disk lists to be stored in random order (in-place update, Section 6.3).
- The derivation of some statistical properties of inverted files (Section 4.2) shows that the size of the vocabulary and the total number of postings found in short posting lists both are sub-linear in the total size of the collection. This result has already been made use of in Section 4.4.2 and will be made use of again in the analysis of hybrid index maintenance in Chapter 6.

Chapter 5

Secure File System Search

In the previous chapter, I have argued that, for performance and storage efficiency reasons, it is essential to have multiple users share the same index. This way, a file only needs to be indexed once, even if it can be accessed by multiple users. I now discuss the security issues that arise under such circumstances.

If multiple users store files in the same file system, and all index data are stored in one central index that is shared by all users, then clearly there must be mechanisms in the search engine that ensure that the results to a search query submitted by user A do not contain any information about files that may only be searched by user B. In order to address this problem in a proper way, it first needs to be defined what it means for a file to be *searchable* by a user. Basic security restrictions, in the form of file access permissions, are already present in every UNIX file system implementation, but it is not entirely clear how these access permissions can be extended to the domain of file system search.

After a brief review of the UNIX security model (Section 5.1), I describe a file system search security model that is based on the UNIX security model and that defines under what exact circumstances a given file is *searchable* by a given user (Section 5.2). I present several possible implementations of the security model, some based on a postprocessing approach, others making use of structural constraints expressible within the GCL query framework. I then show that, under the assumption that the search engine ranks matching documents by their statistical relevance to the query, the implementations based on the postprocessing approach are insecure, allowing an arbitrary user to obtain information about the contents of files for which she does not have read permission (Section 5.3). A GCL-based implementation, which does not exhibit this insecurity, is discussed and evaluated in Sections 5.4 and 5.5. Opportunities for performance optimizations are identified and examined in Section 5.6.

	owner	*group*	*others*
Read	x	x	
Write	x		
e**X**ecute	x		x

Figure 5.1: File permissions in UNIX (example). *Owner* permissions override *group* permissions; *group* permissions override *others*. Access is either granted or denied explicitly (in the example, a member of the group would not be allowed to execute the file, even though everybody else is).

5.1 The UNIX Security Model

The UNIX security model [91] [90] is an ownership-based model with discretionary access control that has been adopted by many operating systems. Every file is owned by a certain user. This user (the file *owner*) can associate the file with a certain *group* (a set of users) and grant access permissions to all members of that group. She can also grant access permissions to the group of all users in the system (*others*). Privileges granted to other users can be revoked by the owner later on.

5.1.1 File Permissions

UNIX file permissions can be represented as a 3×3 matrix, as shown in Figure 5.1. When a user wants to access a file, the operating system searches from left to right for an applicable permission set. If the user is the owner of the file, the leftmost column is used to check whether the user is allowed to access the file. If the user is not the owner, but member of the group associated with the file, the second column is taken. Otherwise, the third column is taken. This can, for instance, be used to grant read access to all users in the system except for those who belong to a certain group.

File access privileges in UNIX fall into three different categories: **R**ead, **W**rite, and e**X**ecute. The semantics of these privileges are different depending on whether they are granted for a file or a directory. For files,

- the read privilege entitles a user to read the contents of a file;
- the write privilege entitles a user to change the contents of the file or to append data at the end of the file;
- the execute privilege entitles a user to run the file as a program.

For directories,

- the read privilege allows a user to read the contents of the directory, which includes file names and attributes of all files and subdirectories within that directory;
- the write privilege allows a user to add files to the directory, to remove files from it, and to change the names of files in the directory;
- the execute privilege allows a user to access files and subdirectories within the directory.

The UNIX security model makes it possible to grant a group of users $\mathcal{U}$ read access to a specific file without revealing any information about other files in the same directory, by granting execute permission for the directory and letting the members of $\mathcal{U}$ know the name of the file. Because the directory is not readable, only users who know the name of the file are able to access it (note that, even without read permission for a directory, a user can still access all files in it; she just cannot use `ls` to search for them).

Extensions to the basic UNIX security model, such as access control lists (ACLs) [43], have been implemented in various operating systems (e.g., Windows, Linux, MacOS). An ACL is a list of rules associated with a file or directory that allows the owner to define access permissions for specific users or groups of users. This overcomes the limitation of the traditional UNIX security model which only allows to distinguish between owner, group, and others. The maximum number of rules that may be defined for any given file depends on the file system implementation and varies between 32 (ext2) and several thousand (JFS).

Despite the advantages offered by ACLs, the simple owner/group/others model is still the dominant security paradigm. In many Linux distributions, for example, file system support for ACLs is disabled by default, and has to be explicitly activated for each mount point separately.

5.1.2 Traditional File System Search in UNIX

Regardless of whether file permissions are defined according to the basic UNIX security model or via access control lists, the end result is the same: a global, system-wide security matrix that defines for every user U and every object O in the file system (file or directory) the access privileges for O that have been granted to U — a subset of {R, W, X}. Access control lists, just like the traditional model, do not provide a mechanism to define whether a file is *searchable* by a user. Thus, the UNIX security model needs to be extended so that it takes *searchability* into account.

Recall from Section 2.1 the traditional UNIX `find`/`grep` search paradigm, using `find` to specify a set of external criteria, and using `grep` to define constraints regarding the contents of a matching file. This search paradigm implicitly defines a file system search security model. A file can only be searched by a user if:

1. the file can be found in the file system's directory tree and
2. the file's contents may be read by the user.

Let us assume that the base directory for a `find` operation is the file system's root directory. Then, in terms of file permissions, condition 1 means that there has to be a path from the root directory to the file in question, and the user needs to have both the read privilege and the execute privilege for every directory along this path. This is because, in order to find out that a file exists, the user needs to be able to read the directory listing of the file's parent directory; in order to find out that the parent directory exists, the user needs to be able to read the directory's parent directory; and so on. Finally, condition 2 requires the user to have the read privilege for the actual file.

Because UNIX allows a file to be linked from multiple directories (i.e., to have multiple hard links pointing to it), the traditional UNIX `find`/`grep` search paradigm implicitly defines the following security model.

Definition: File Searchability According to find/grep

A file F is *searchable* by a user U if and only if:

1. There is at least one path from the root directory to F, such that U has the *read* and *execute* privilege for all directories along this path.
2. U has been granted read privilege for F.

The same definition is also used by `slocate`[1], a security-aware version of the `locate`[2] utility program (`locate` and `slocate` can be thought of as fast versions of `find`, relying on a periodically updated index containing the names of all files and directories in the file system).

5.2 A File System Search Security Model

While the security model implicitly defined by `find`/`grep` seems to be appropriate in many scenarios, it has a significant shortcoming: It is not possible to grant search permission for a

[1]http://slocate.trakker.ca/ (accessed 2007-03-06)
[2]http://www.gnu.org/software/findutils/ (accessed 2007-03-06)

single file without revealing information about other files in the same directory. In order to make a file searchable to other users, the owner has to give them the read privilege for the file's parent directory; this reveals file names and attributes of all other files within the same directory. It is, however, desirable to be able to treat all files in the file system independently of each other when defining file searchability — just like the *read* privilege for a file F_1 is not contingent on the read privilege for a different file F_2. The `find`/`grep` security model does not meet this requirement.

This suggests a slight modification of the definition of *searchable*. Instead of insisting that there is a path for which the searching user has full read and execute privilege, we now only insist that there is a path from the file system's root directory to the file in question such that the user has the execution privilege for every directory along this path. Unfortunately, this new definition conflicts with the traditional use of the read and execution privileges, in which this constellation is usually used to give read permission to all users who know the exact file name of the file in question (cf. Section 5.1). While this is a less severe limitation than the make-the-whole-directory-visible problem discussed above, it still is somewhat unsatisfactory.

The only completely satisfying solution would be the introduction of an explicit fourth access privilege, the *search* privilege, in addition to the existing three. Since this is very unlikely to happen, as it would probably break most existing UNIX software, I decided to base my definition of *searchable* on a combination of *read* and *execute*, as motivated above.

Definition: File Searchability in Wumpus

A file F is searchable by a user U if and only if:

1. There is a path from the file system's root directory to F such that U has the *execute* privilege for all directories along this path.
2. U has been granted the *read* privilege for F.

Based on this definition, search permissions can be granted and revoked, just like the other permission types, by modifying the respective read and execute privileges.

The resulting file system search security model is not necessarily based on the basic owner/group/others UNIX security model. It can easily be applied to more complicated models, including, for example, access control lists, as long as the set of access privileges (R, X) stays the same. The model only requires a user to have certain privileges; it does not make any assumptions about where these privileges come from. Moreover, once it has been established what exactly it means for a file to be searchable by a user, the details of the definition become irrelevant. Thus, the possible implementations of the file system search security model discussed in the remainder of this chapter are applicable in any environment in which there are some files that may be searched by a given user and other files that may not. This includes enterprise search scenarios, such as the one described by Bailey et al. [5].

Any implementation of the file system search security model needs to fulfill two essential requirements:

- Every file that is searchable by a user U is a potential search result and must be returned to the user if it matches the query.
- The search engine's response to a search query submitted by U may not contain any information, either explicit or implicit, about files not searchable by U.

Several possible implementations of the security model are presented in the following sections. The first two implementations (based on a postprocessing approach) do not meet the requirements stated above; the following two implementations (based on query integration via structural query constraints) do meet the requirements.

5.3 Enforcing Security by Postprocessing

In Section 2.2.3, I have demonstrated by example that it is highly desirable for the search engine to provide some sort of relevance ranking and to present the search results to the user in decreasing order of predicted relevance to the query.

Suppose the search engine does rank search results by their expected relevance. The question then is how the statistical information necessary to perform the relevance ranking should be obtained by the search engine. Two fundamentally different approaches exist. The engine may either use global, system-wide term statistics extracted from all files in the system, or it may use term statistics obtained exclusively from the set of files searchable by the user who submitted the query.

The first approach has the advantage that term statistics can be obtained more efficiently and may even be cached by the search engine, improving query performance substantially (cf. Section 4.5). Precomputing user-specific versions of these statistics, separately for each user in the system, is not feasible, due to the potentially very large number of users. Unfortunately, using global term statistics, as I will show later in this chapter, leads to insecurities in the search engine that can be exploited by an arbitrary user to reach a very accurate estimate of the statistics used to rank the search results, allowing her to obtain information about the contents of files for which she does not have read permission.

If global term statistics are used to rank documents by their statistical relevance, then a possible implementation of the security model defined in the previous section is based on the well-known postprocessing approach (cf. Bailey et al. [5]). Whenever a query is submitted by a user U, it is processed as usual, without taking U's identity and access privileges into account. After the query has been processed, all search results referring to files that are not searchable by U (according to the file system search security model) are removed from the final list before the results are returned to the user.

The final postprocessing step in this general approach to security-aware query processing can be realized in two different ways.

Implementation 1: `stat`

The search engine can obtain information about security permissions by accessing each file referenced in the search results (for instance, by issuing a `stat` system call) and ensuring – at query time – that the file is searchable by the user who submitted the query. Depending on the number of files in the search results, and depending on the percentage of the files searchable by the user, this might or might not be a viable solution.

Recall the essence of my discussion of existing techniques for file system search in UNIX (Section 2.1). I pointed out that, compared to hard disk capacity and disk throughput for sequential read/write operations, random disk seeks have become more and more costly over the last decade. Therefore, the traditional `find`/`grep` paradigm, I concluded, no longer represents an attractive method of full-text search.

The same logic applies here. Individually accessing each matching file is an extremely expensive operation that, in total, can easily take several seconds. This is particularly problematic if the number of matching files that the user has the search privilege for is small compared to the total number of matching files. Suppose the user U who submitted a query to the search engine has the search privilege for 50% of all files in the file system. Suppose further the search engine does not return a complete ranking of all documents matching the query, but only the top 10, according to some relevance ranking function. Then, on average, the search engine needs to check access privileges for 20 files in order to find the top 10 files that are searchable by U. If, however, U may only search 5% of all files in the file system, then the postprocessing step may require hundreds of file access operations, just to find 10 files that may be searched by the user.

Implementation 2: Caching File Permissions

As an alternative to the `stat`-based implementation, the search engine can maintain cached information about file access privileges. This seems natural, since the search engine needs to maintain some file-specific information, such as file names, creation dates, etc., in any case.

Compared to the first implementation, the postprocessing step now can be performed much faster, because the security information for all indexed files is stored in a small region of the hard disk, requiring only a small number of disk seeks. If the total size of these security-related meta-data is small, it might even be possible to permanently cache them in main memory. Whether this is actually a sensible way of consuming precious memory resources depends on the characteristics of the file system. If basic UNIX security restrictions (i.e., owner/group/others) are used exclusively, it seems tolerable. The per-file overhead then is no more than 6 bytes (2 bytes for owner and group ID each, 2 bytes for R/W/X access privileges). However, if file owners make extensive use of ACLs with several thousand entries (as permitted by JFS), then permanently storing all security meta-data in main memory is certainly not feasible.

In the following, I show that the postprocessing approach, using global term statistics to produce and rank search results, is insecure, as it allows an arbitrary user to obtain information about files for which she does not have read permission.

5.3.1 Exploiting Relevance Scores

Let us assume that the search engine employs some sort of statistical relevance ranking (cf. Section 2.2) to sort search results before returning them to the user. Let us further assume that the search engine reports the exact relevance scores back to the user, and that these scores are based on global term statistics, i.e., on the contents of all files in the file system. Then a random user U will be able to deduce from the search results exact file-system-wide term statistics for an arbitrary search term. Depending on the actual ranking function employed by the search engine, this information may include the number of files that a term appears in or the number of times that the term occurs in the file system.

One might argue that revealing to an unauthorized user the number of files that contain a certain term is only a minor problem. I disagree and give two example scenarios in which the ability to infer term statistics is highly undesirable and can have disastrous effects on file system security:

- An industrial spy knows that the company she is spying on is developing a new chemical process. She starts monitoring term frequencies for certain chemical compounds that are likely to be involved in the process. After some time, this will have given her enough information to tell which chemicals are actually used in the process — without reading any files.

- The search system can be used as a covert channel to transfer information from one user account to another, circumventing security mechanisms like file access logging.

For all calculations performed in this section and the next, I assume that the number of files in the system is sufficiently large so that the creation or removal of a single file does not cause a substantial alteration of the collection statistics. This assumption is not necessary, but it simplifies some of the calculations.

BM25: Computing the Number of Files Containing a Given Term

Suppose the search system uses a system-wide index and implements Okapi BM25 to perform relevance ranking on files matching a search query. After all files matching a user's query have been ranked by BM25, files that may not be searched by the user are removed from the list. The remaining files, along with their relevance scores, are presented to the user.

Recall from Chapter 2 that, for a given ranked search query $\mathcal{Q} := \{Q_1, \ldots, Q_n\}$, a sequence of n query terms, the BM25 score for a file F is:

$$\text{score}_{\text{BM25}}(\mathcal{Q}, F) = \sum_{i=1}^{n} \log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{Q_i}|}\right) \cdot \frac{(1 + k_1) \cdot f_F(Q_i)}{f_F(Q_i) + k_1 \cdot \left((1 - b) + b \cdot \frac{|F|}{\mathit{avgfl}}\right)}, \tag{5.1}$$

where:

- $\mathcal{F}$ is the set of indexable files in the file system;
- $\mathcal{F}_{Q_i}$ is the set of files containing the term Q_i;
- $f_F(Q_i)$ is the number of times that Q_i occurs in F;
- $|F|$ is the length of the file F, measured in tokens;
- *avgfl* the the average file length in the file system, i.e., the average number of tokens found in an indexable file;
- k_1 and b are free parameters. For the purpose of the argument presented here, I assume that their values are known. This assumption is not necessary; by submitting just two additional search queries, it is possible to obtain the values of both k_1 and b.

I start by showing how an arbitrary user, called *Eve*[3], can compute the values of the unknown parameters $|\mathcal{F}|$ (the number of files in the file system) and *avgfl* (the average file length in tokens) in Equation 5.1. For this purpose, Eve generates two unique terms T_1 and T_2 (for example, with the help of a random number generator) that do not appear in any file. She then creates three new files F_1, F_2, and F_3:

- F_1 contains only the term T_1;
- F_2 consists of two occurrences of the term T_1;
- F_3 contains only the term T_2.

Now, Eve sends two queries to the search engine: $\{T_1\}$ and $\{T_2\}$. For the former, the engine returns F_1 and F_2; for the latter, it returns F_3. For the scores of F_1 and F_3, Eve knows that

$$\text{score}(F_1) = \log\left(\frac{|\mathcal{F}|}{2}\right) \cdot \frac{(1+k_1)}{1 + k_1 \cdot ((1-b) + \frac{b}{avgfl})}, \tag{5.2}$$

because T_1 occurs in two files, and because it occurs a single time in F_1, and

$$\text{score}(F_3) = \log\left(\frac{|\mathcal{F}|}{1}\right) \cdot \frac{(1+k_1)}{1 + k_1 \cdot ((1-b) + \frac{b}{avgfl})}, \tag{5.3}$$

because T_2 occurs in one file, and because it occurs a single time in F_3. Dividing (5.2) by (5.3) results in

$$\frac{\text{score}(F_1)}{\text{score}(F_3)} = \frac{\log\left(\frac{|\mathcal{F}|}{2}\right)}{\log\left(\frac{|\mathcal{F}|}{1}\right)} \quad\Leftrightarrow\quad \frac{\text{score}(F_1)}{\text{score}(F_3)} = 1 - \frac{1}{\log_2 |\mathcal{F}|} \tag{5.4}$$

and thus

$$|\mathcal{F}| = 2^{\left(\frac{\text{score}(F_3)}{\text{score}(F_3) - \text{score}(F_1)}\right)}. \tag{5.5}$$

Now that Eve knows $|\mathcal{F}|$, the number of files in the file system, she proceeds to compute the second unknown: *avgfl*, the average file length. Using equation (5.3), she obtains:

$$avgfl = \frac{b}{X - 1 + b}, \tag{5.6}$$

where

$$X = \frac{(1+k_1) \cdot \log(|\mathcal{F}|) - \text{score}(F_3)}{\text{score}(F_3) \cdot k_1}. \tag{5.7}$$

[3]In cryptography, Eve is the canonical name for an eavesdropper, a user that is sitting on the line and trying to get her hands on some data that are not meant to be read by her.

Since she now knows all parameters of the BM25 scoring function, Eve creates another file F_4, which contains the term T^* that she is interested in, and submits the query $\{T^*\}$. The search engine returns F_4 along with score(F_4). This information is used by Eve to construct the final equation

$$\text{score}(F_4) = \frac{(1+k_1) \cdot \log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|}\right)}{1 + k_1 \cdot \left((1-b) + \frac{b}{\mathit{avgfl}}\right)}, \tag{5.8}$$

in which $|\mathcal{F}_{T^*}|$ is the only unknown. Eve can therefore easily calculate its value. In the end, after submitting only three queries to the search engine, she knows the number of files containing T^*:

$$|\mathcal{F}_{T^*}| = |\mathcal{F}| \cdot 2^{-Y \cdot \text{score}(F_4)}, \tag{5.9}$$

where

$$Y = \frac{1 + k_1 \cdot ((1-b) + \frac{b}{\mathit{avgfl}})}{1 + k_1}. \tag{5.10}$$

To avoid small changes in $|\mathcal{F}|$ and *avgfl*, the file F_4 can be created before the first query is submitted to the system.

Language Modeling: Computing the Number of Occurrences of a Given Term

If the search engine's ranking function is based on a language modeling approach instead of BM25, then Eve cannot determine the number of files containing the term T^*. She can, however, find out how often T^* occurs in the entire file system.

Recall from Chapter 2 that, for a given ranked query $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$, containing n query terms, the relevance score of the file F, according to a language modeling approach with Dirichlet smoothing, is:

$$\text{score}_{\text{LMD}}(\mathcal{Q}, F) = \prod_{i=1}^{n} \frac{f_F(Q_i) + \mu \cdot \Pr(Q_i|\mathcal{F})}{|F| + \mu}, \tag{5.11}$$

where

- $f_F(Q_i)$ is the number of occurrences of Q_i within F;
- $|F|$ is the length of the file F, measured in tokens;
- $\Pr(Q_i|\mathcal{F})$ is the unigram probability of Q_i according to the language model defined by all files in the file system, estimated using the maximum likelihood heuristic;
- μ is a free parameter.

In order to determine the total number of occurrences of the term T^* in the file system, Eve generates a unique term T_1 and two new files F_1 and F_2:

- F_1 consists of a single occurrence of T_1;
- F_2 contains one occurrence of T_1 and T^* each.

She then submits a search query $\{T_1\}$, for which the search engine returns F_1 and F_2, with scores:

$$\text{score}(F_1) = \frac{1 + \mu \cdot Pr(T_1|\mathcal{F})}{1+\mu}, \tag{5.12}$$

$$\text{score}(F_2) = \frac{1 + \mu \cdot Pr(T_1|\mathcal{F})}{2+\mu}. \tag{5.13}$$

Dividing (5.12) by (5.13) results in

$$\mu = \frac{2 \cdot \text{score}(F_2) - \text{score}(F_1)}{\text{score}(F_1) - \text{score}(F_2)} \tag{5.14}$$

and thus

$$\Pr(T_1|\mathcal{F}) = \text{score}(F_1) + \frac{(\text{score}(F_1) - 1) \cdot (\text{score}(F_1) - \text{score}(F_2))}{2 \cdot \text{score}(F_2) - \text{score}(F_1)}. \tag{5.15}$$

Submitting the query $\{T^*\}$, Eve can obtain the value of $\Pr(T^*|\mathcal{F})$ in the same way. Combining the information, she then computes the total number of occurrences of T^* as:

$$f_{T^*} = 2 \cdot \frac{\Pr(T^*|\mathcal{F})}{\Pr(T_1|\mathcal{F})}. \tag{5.16}$$

If the search engine computes relevance scores using a different incarnation of the language modeling approach, for example, employing Jelinek-Mercer smoothing [58] (linear interpolation) instead of Dirichlet smoothing, Eve can proceed in essentially the same way, performing only slightly different calculations.

5.3.2 Exploiting Ranking Results

The above argument has made it clear that, if file system search security is enforced by following the postprocessing approach, then the relevance scores of the matching files must not be revealed to the user. Otherwise, the security of the system will be compromised, because the relevance scores, based on global term statistics, make it possible to reverse-engineer the scoring process and to compute exact term statistics.

I now show that, even if the relevance scores are not made available to the user (Eve) who submits a search query, she will still be able to obtain a very good approximation of the global term statistics, by carefully analyzing the order in which matching files are returned by the search engine. This implies that the postprocessing approach is inherently insecure and should not be used. The accuracy of the approximation that can be achieved by the procedure described here depends on the number of files $F_{\max}$ that Eve may create without drawing the attention of the system administrator. Let us assume $F_{\max} = 2000$.

My discussion of the method employed by Eve is limited to the case where BM25 [94] is used to score matching files, but the general idea can be applied to other scoring functions as well (e.g., language modeling or QAP).

Eve starts by creating a file F_0, containing only a single occurrence of T^*, the term she is interested in. She then generates a unique term T_1 and creates 1,000 files $F_1 \ldots F_{1000}$, each containing a single occurrence of T_1. After doing this, she submits the ranked query

$$\{T^*, T_1\}$$

to the search engine. Since BM25 performs a Boolean OR to determine the set of matching files, the search engine returns F_0 as well as $F_1 \ldots F_{1000}$, ranked according to their BM25 scores. If, in the response to Eve's query, the file F_0 appears before any of the other files ($F_1 \ldots F_{1000}$), then she knows that

$$\text{score}(F_0) \geq \text{score}(F_1) \quad \Rightarrow \quad \log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|}\right) \geq \log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T_1}|}\right) \tag{5.17}$$

(cf. equation 5.1) and thus:

$$|\mathcal{F}_{T^*}| \leq 1000 = |\mathcal{F}_{T_1}|. \tag{5.18}$$

That is, T^* occurs in at most 1,000 files. Eve can now perform a binary search, adjusting the number of files containing the term T_1, until she knows the exact value of $|\mathcal{F}_{T^*}|$.

If, on the other hand, F_0 appears after the other files ($F_1 \ldots F_{1000}$) in the ranking produced by the search engine, Eve knows that $|\mathcal{F}_{T^*}| \geq 1000$. It might be that this information is sufficient for her purposes. However, if she needs a better approximation, she can achieve that, too.

Eve first deletes all files that she has created so far. She then generates two more unique terms T_2 and T_3. She creates 1,000 files $F'_1 \ldots F'_{1000}$, each containing the two terms T_1 and T_2, and 999 files $F'_{1001} \ldots F'_{1999}$, each containing a single occurrence of the term T_3. She finally creates one last file, F'_0, containing one occurrence of T^* and T_3 each. After she has created

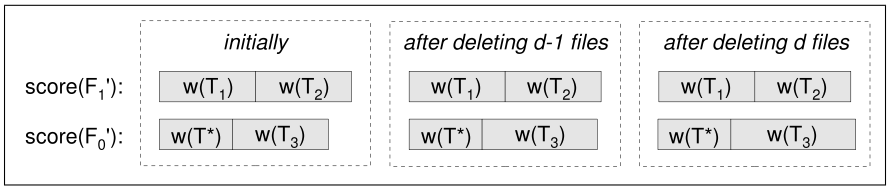


Figure 5.2: Relative BM25 score of F_0' (containing T^* and T_3) and F_1' (containing T_1 and T_2) as the number of files containing T_3 is reduced. Initially, score(F_0') is smaller than score(F_1'), because T^* appears in more files than the other terms and thus has smaller IDF weight. After d deletions, score(F_0') is greater than score(F_1').

these 2,000 files, Eve submits the query

$$\{T^*, T_1, T_2, T_3\}$$

to the search system. All 2,000 files she has created match the query. The relevance scores of the files $F_0' \ldots F_{1000}'$ are:

$$\text{score}(F_0') = C \cdot \left(\log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|}\right) + \log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T_3}|}\right) \right), \tag{5.19}$$

because F_0' contains T^* and T_3, and

$$\text{score}(F_i') = C \cdot \left(\log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T_1}|}\right) + \log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T_2}|}\right) \right) \tag{5.20}$$

(for $1 \leq i \leq 1000$), because all the F_i''s contain T_1 and T_2. The constant C, which is the same for all files created, is the BM25 length normalization component for a file of length 2:

$$C = \frac{1 + k_1}{1 + k_1 \cdot \left((1 - b) + \frac{2 \cdot b}{avgfl} \right)}. \tag{5.21}$$

Eve now subsequently deletes one of the files $F_{1001}' \ldots F_{1999}'$ at a time, starting with F_{1999}', and re-submits her query to the search engine, until F_0' appears before F_1' in the list of matching files (i.e., score(F_0') $\geq$ score(F_1')). Suppose this happens after d deletions. Because

$$\text{score}(F_1') = C \cdot 2 \cdot \log\left(\frac{|\mathcal{F}|}{1000}\right), \tag{5.22}$$

she then knows that

$$\log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|}\right) + \log\left(\frac{|\mathcal{F}|}{1000-d}\right) \geq 2\cdot\log\left(\frac{|\mathcal{F}|}{1000}\right) \geq \log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|}\right) + \log\left(\frac{|\mathcal{F}|}{1000-d+1}\right), \tag{5.23}$$

and thus

$$-\log(|\mathcal{F}_{T^*}|) - \log(1000-d) \geq -2\cdot\log(1000) \geq -\log(|\mathcal{F}_{T^*}|) - \log(1000-d+1), \tag{5.24}$$

which implies

$$\frac{1000^2}{1000-d} \geq |\mathcal{F}_{T^*}| \geq \frac{1000^2}{1000-d+1}. \tag{5.25}$$

The general procedure is visualized in Figure 5.2. Just like in the first case, where $|\mathcal{F}_{T^*}|$ was smaller than 1000, Eve can realize her quest for $|\mathcal{F}_{T^*}|$ through a binary search, reducing the number of search queries that need to be submitted to about 10.

Example

Suppose the term that Eve is interested in occurs in $|\mathcal{F}_{T^*}| = 9{,}000$ files. Throughout the process of approximating $|\mathcal{F}_{T^*}|$, the score of F_1' remains constant:

$$\begin{aligned}\text{score}(F_1') &= C\cdot 2\cdot\log(|\mathcal{F}|) - C\cdot\log(1000) - C\cdot\log(1000) && (5.26)\\ &\approx C\cdot(2\cdot\log(|\mathcal{F}|) - 13.8155). && (5.27)\end{aligned}$$

After deleting 888 files, F_0''s score is:

$$\begin{aligned}\text{score}(F_0') &= C\cdot 2\cdot\log(|\mathcal{F}|) - C\cdot\log(9000) - C\cdot\log(112) && (5.28)\\ &\approx C\cdot(2\cdot\log(|\mathcal{F}|) - 13.8235). && (5.29)\end{aligned}$$

After deleting 889 files, its score is:

$$\begin{aligned}\text{score}(F_0') &= C\cdot 2\cdot\log(|\mathcal{F}|) - C\cdot\log(9000) - C\cdot\log(111) && (5.30)\\ &\approx C\cdot(2\cdot\log(|\mathcal{F}|) - 13.8145). && (5.31)\end{aligned}$$

Hence, after deleting 888 files, F_0' is ranked lower than F_1'. After the 889th file is deleted, F_0' is ranked higher than F_1'. Therefore, Eve can conclude that

$$\log(|\mathcal{F}_{T^*}|) + \log(112) \geq 2\cdot\log(1000) \geq \log(|\mathcal{F}_{T^*}|) + \log(111). \tag{5.32}$$

She obtains the following bounds: $8929 \leq |\mathcal{F}_{T^*}| \leq 9009$. She also knows that the relative error of her approximation is at most $\frac{(9009-8929)/2}{8929} \approx 0.45\%$.

Generalization

The method described above provides an approximation of the following form:

$$\frac{(F_{\max}/2)^2}{(F_{\max}/2) - d} \geq |\mathcal{F}_{T^*}| \geq \frac{(F_{\max}/2)^2}{(F_{\max}/2) - d + 1}, \tag{5.33}$$

where $F_{\max}$ is the maximum number of files Eve may create without raising suspicion. Obviously, if $|\mathcal{F}_{T^*}|$ increases, then the quality of the approximation deteriorates. For example, if $|\mathcal{F}_{T^*}| = 9{,}000$ and $F_{\max} = 2{,}000$ (as before), then the approximation error is guaranteed to be at most 0.45%. However, if T^* is rather frequent and appears in, say, $|\mathcal{F}_{T^*}| = 160{,}000$ files, then the approximation error may be as large as 8.3%, making it difficult to obtain a reliable estimate.

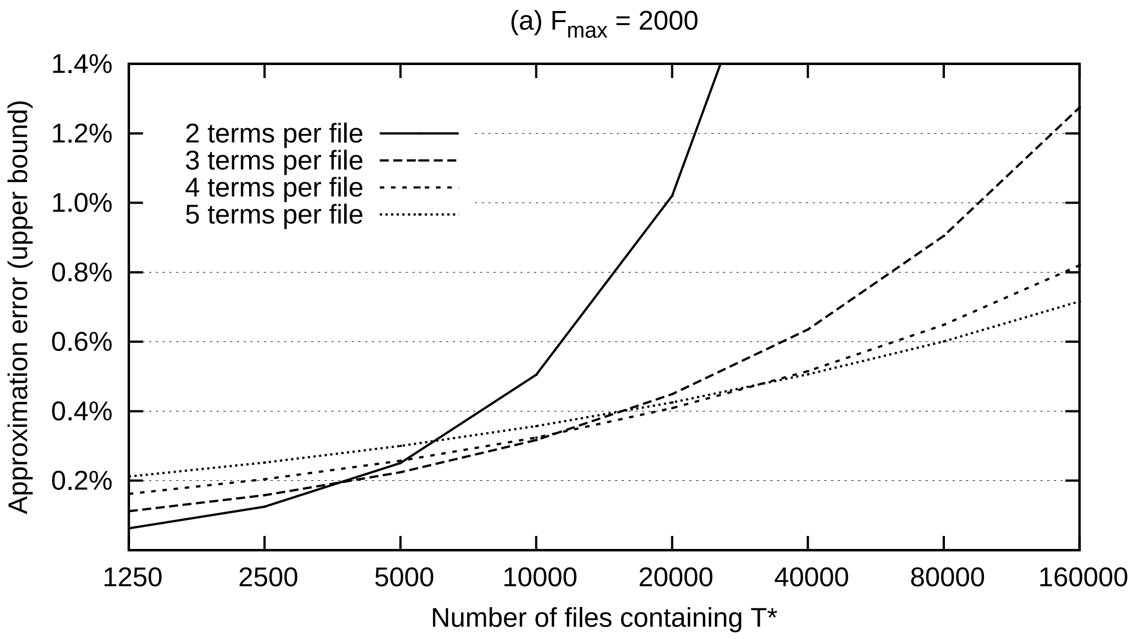
This problem can be addressed by generalizing the method and storing more than 2 terms in each file, as follows:

- Pick an integer t, the number of terms to store in each file.
- Generate $2t - 1$ unique terms $T_1, \ldots, T_{2t-1}$.
- Create $F_{\max}/2$ files, each containing the terms $T_1, \ldots, T_t$.
- Create $F_{\max}/2 - 1$ files, each containing the terms $T_{t+1}, \ldots, T_{2t-1}$.
- Create a single file containing one occurrence each of T^* and $T_{t+1}, \ldots, T_{2t-1}$.
- Proceed as before.

Depending on the values of $|\mathcal{F}_{T^*}|$, $F_{\max}$, and t, Eve can obtain an approximation of the form:

$$\frac{(F_{\max})^t}{(F_{\max} - d)^{t-1}} \geq |\mathcal{F}_{T^*}| \geq \frac{(F_{\max})^t}{(F_{\max} - d + 1)^{t-1}}. \tag{5.34}$$

Guaranteed approximation errors (upper bounds) for various values of t and $|\mathcal{F}_{T^*}|$ are shown in Figure 5.3. Using the generalized method, the approximation error stays well below 1% for $F_{\max} = 2{,}000$, even if T^* appears in a very large number of files (e.g., $|\mathcal{F}_{T^*}| = 100{,}000$).



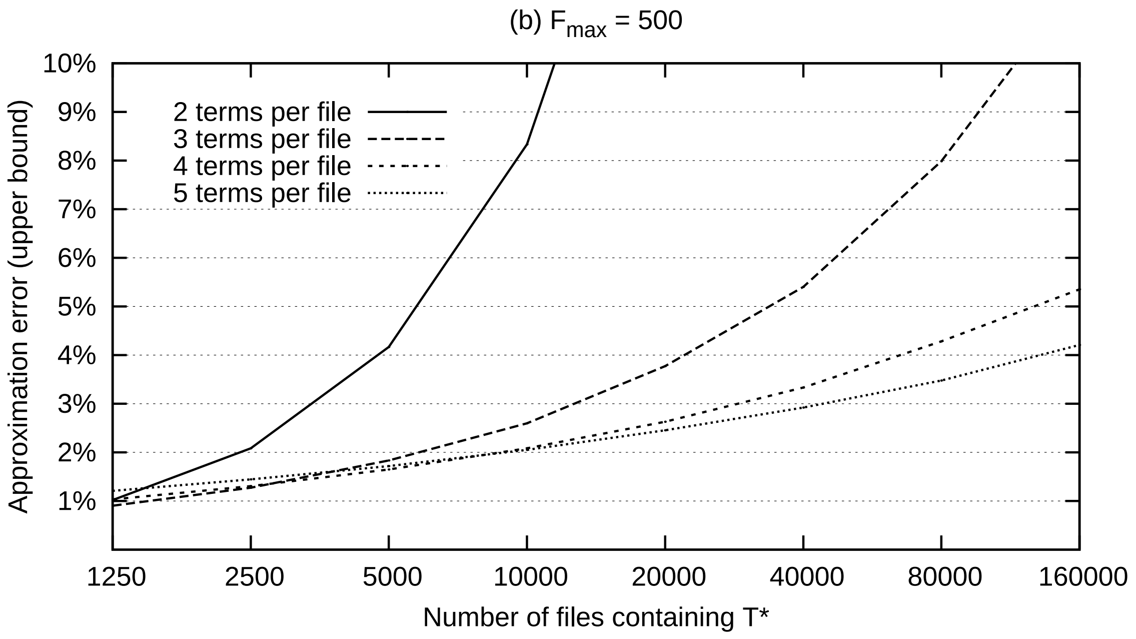


Figure 5.3: Guaranteed approximation error (upper bound) when calculating $|\mathcal{F}_{T^*}|$, the number of files containing a given term T^*; storing 2, 3, 4, or 5 terms per file (original method: 2 terms per file). The quality of the approximation depends on $F_{\max}$, the maximum number of files that may be created by a user without raising suspicion.

5.3.3 Exploiting Support for Structural Constraints

So far, we have only seen how to exploit ranking results in order to calculate (or approximate) the number of files that contain a particular term. While this already is undesirable, the situation is even worse if the search system allows relevance ranking for more complicated queries, such as Boolean queries, phrases, or structural queries, like those supported by GCL.

If the search engine supports GCL-type structural query constraints and enforces security restrictions by following the postprocessing approach, then it is comparatively trivial for a user to infer information about files that are not accessible by her. By creating a file of the form

$$T_1 \quad T_2 \quad T_3$$

(containing the three terms T_1, T_2, and T_3) and submitting the query

$$T_2 \lhd ((T_1 \cdots \text{“interesting phrase”}) \vee (\text{“interesting phrase”} \cdots T_3)),$$

it is possible to determine whether the “interesting phrase” appears somewhere in the file system. Since T_2 lies within a file that is searchable by the user, the security subsystem of the search engine – only invoked after the query has been processed – has no way of knowing that the search result depends on information that is stored outside that file.

But even if the search engine does not offer support for full-blown structural constraints, it is still possible to go beyond single-term statistics. If, for instance, the search system allows phrase queries of arbitrary length, then it might be possible to infer the entire content of a file — depending on how exactly ranking for phrases is realized in the search engine. Assume phrases are treated as if they were atomic terms, and assume we know that a certain file contains the phrase “$A\ B\ C$”. We then try all possible terms D and and calculate the number of files which contain “$A\ B\ C\ D$” until we have found a D that gives a non-zero result. We then continue with the next term E. This way, it is possible to construct the entire content of a file using a finite number of search queries (although this might take a long time). Some a-priori knowledge about the contents of the file in question, or even a simple n-gram language model [26] [47] can be used to predict the next word and thus increase the efficiency of this method.

In general, the more powerful the search engine's query language, and the more features it uses to rank the search results, the more detailed is the information that Eve can obtain by following a method similar to the one outlined above.

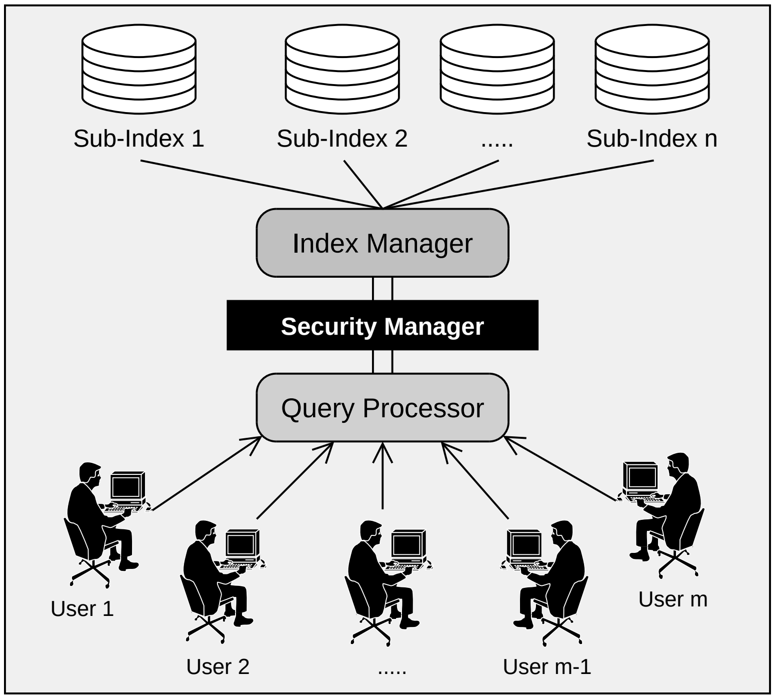


Figure 5.4: The three components of the search engine that are involved in secure query processing: index manager (providing access to the query terms' posting lists), security manager (applying security restrictions to all posting lists), and query processor (matching and scoring documents).

5.4 Integrating Security Restrictions into the Query Processor

Motivated by the shortcomings of the postprocessing approach described in the previous section, I now describe how GCL-type structural query constraints can be used to apply security restrictions to all search results in a reliable manner — by integrating the security restrictions into the query processing procedure instead of applying them as a postprocessing step.

When a search query is submitted by a user U, three main components of the search engine are involved in processing that query (as shown in Figure 5.4):

- The *index manager* provides access to a set of posting lists stored in the in-memory index and to the on-disk indices.
- The *security manager* applies security restrictions to all posting lists, making sure that they do not contain any information related to files that are not searchable by U.
- The *query processor* performs a sequence of relevance computations and orders the set of matching documents by their relevance to the given query.

GCL query:

$$(\texttt{<doc>} \cdots \texttt{</doc>}) \rhd ((\texttt{mad} \wedge \texttt{cow}) \lhd [3])$$

Operator tree:

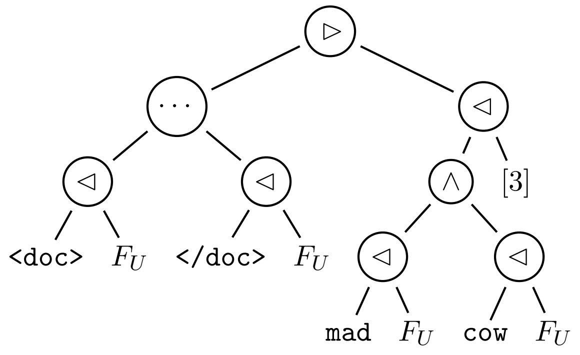


Figure 5.5: Integrating security restrictions into the query processing. GCL query and resulting operator tree with security restrictions applied to all posting lists (GCL containment operator "◁"), for textual query "return all documents in which 'mad' and 'cow' appear within 3 words".

In the course of processing a query, the query processor requests posting lists from the index manager and uses the information found in these lists to perform the actual search operation. However, every posting list that is sent back to the query processor has to pass the security manager, which applies user-specific restrictions to the list. As a result, the query processor only ever sees those parts of a posting list that lie within files that are searchable by the user who submitted the query. Since the query processor's response to a search query now is solely dependent on the parts of the posting lists that correspond to files searchable by the user, the results are guaranteed to be consistent with the user's view of the file system and cannot violate any file permissions.

Applying user-specific security restrictions can be realized in a lazy-evaluation fashion, by utilizing the operations provided by the GCL retrieval framework [31], as follows. The search engine's security manager maintains a list of index extents, corresponding to all files in the system, and represented by the GCL expression

$$\texttt{<file>} \cdots \texttt{</file>}$$

(the actual expression used within the search engine is slightly different, so as to avoid conflicts with XML documents containing `<file>` or `</file>` tags, but for readability reasons I use the above notation). Whenever the search engine receives a query from a user U, the security manager is asked to compute a list F_U of all index extents that correspond to files whose content is searchable by U (using the security model's definition of *searchable*). F_U is a sub-

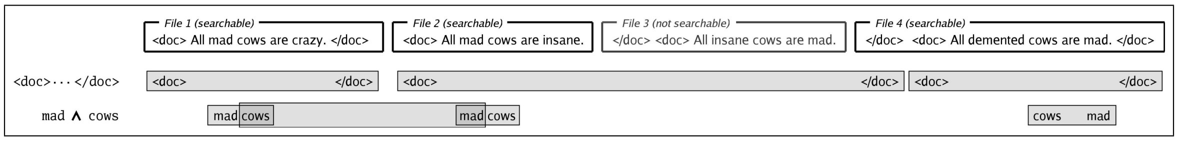


Figure 5.6: Query results for two example queries ("`<doc>` $\cdots$ `</doc>`" and "`mad` $\wedge$ `cows`") with security restrictions applied. Only postings from files that are searchable by the user are considered by the query processor.

list of the GC-list defined by "`<file>` $\cdots$ `</file>`" and represents the user's view of the file system at the moment when the search engine receives the query. Changes to the file system taking place while the query is being processed are ignored; the same list F_U is used to process the entire query. Constructing the list F_U does not involve any inverted index operations, as the GC-list F_U is generated from the security manager's internal data structures, and not from the index. It does, however, require a sequential scan of the internal file list of the security manager, containing one entry for every indexed file — an operation with time complexity $\Theta(|\mathcal{F}|)$, where $\mathcal{F}$ is the set of all indexed files.

The query processor, while it is processing the query, never contacts the index manager directly when requesting a posting list for a query term T. Instead, it sends its request to the security manager, which forwards it to the index. After receiving the posting list L_T from the index, and before returning it to the query processor, the security manager transforms the original, unrestricted list L_T into a security-restricted list $L_{T,U}$:

$$L_{T,U} \equiv (L_T \lhd F_U).$$

The GCL operator tree that results from adding security restrictions to a given GCL search query is shown in Figure 5.5. Their effect on query results is shown in Figure 5.6.[4]

The query processor never accesses L_T directly, but only the restricted version $L_{T,U}$. Hence, it is impossible for it to produce query results that depend on the content of files that are not searchable by U. If term statistics need to be obtained in order to rank the matching documents after a query has been processed, then these statistics are also computed by accessing $L_{T,U}$, and not L_T. Thus, the search results produced only depend on files searchable by U; they cannot be used to obtain any information about files not searchable by U.

Because all GCL operators support lazy evaluation, it is not necessary to apply the security restrictions to the entire posting list when only a small portion of the list is used to process a query. This is important for query processing performance (e.g., MaxScore [109]).

[4]At first sight, it might not make a lot of sense that the documents in Figure 5.6 span across multiple files. However, keep in mind that the search engine, based on a schema-independent retrieval framework, has no notion of *documents* whatsoever, and it is entirely up to each application (e-mail client, word processor, ...) how to organize its data.

Limitations

One drawback of the current implementation of the security model is that, in order to efficiently generate the list of index extents representing all files searchable by a given user (F_U), the security manager needs to keep some information about every indexed file in main memory. This information includes the file's i-node ID, its start and end address in the index address space, its owner, permissions, etc.; in total, it consumes 32 bytes per indexed file. For file systems with a few million indexable files, the amount of main memory required to keep this information in RAM may exceed the amount of RAM that the user or system administrator is willing to allocate to the search engine.

Alternatively, all security meta-data may be kept on disk. However, this is not a very satisfying solution, either, as it would make sub-second query response times difficult to achieve. At this point, keeping the meta-data in memory and applying compression techniques similar to the ones discussed in Section 2.4.4 seems to be the best solution.

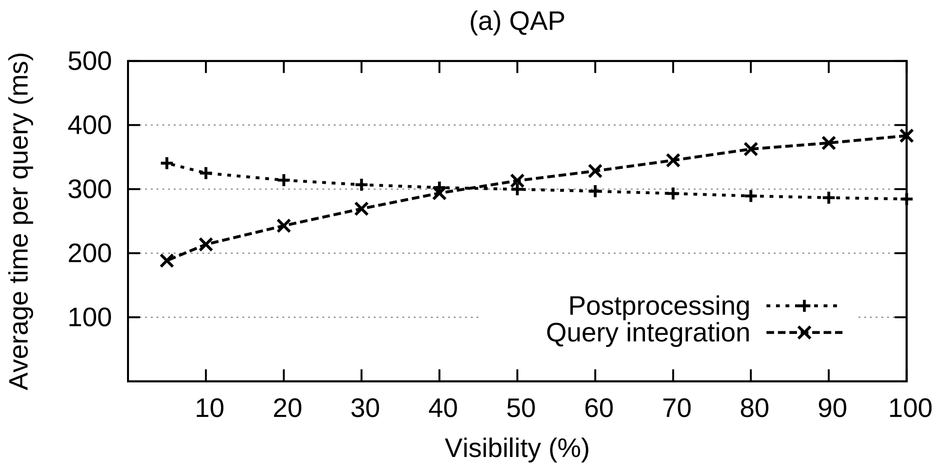
Applicability to Index Updates

It is worth pointing out that this implementation of the security model has the nice property that it automatically supports index update operations. When a file is deleted from the file system, this file system change has to be reflected by the search results. Without a security model, every file deletion would either require an expensive physical update of the internal index structures (updating the posting lists of all terms that appeared in the now-deleted file), or the application of a postprocessing step – similar to the one discussed in the previous section – in which all query results that refer to deleted files are removed from the final result list (cf. Chiueh and Huang [25]). The postprocessing approach would have the same problems as the one from Section 5.3: it would use term statistics that do not reflect the user's actual view of the file system.

By integrating the security restrictions into the core query processing logic, file deletions are immediately supported by the search engine, because the corresponding index address range, associated with the expression

$$\texttt{<file>} \cdots \texttt{</file>},$$

is automatically removed from the security manager's internal representation of the file system. This way, it is possible to keep the index up-to-date at minimal cost. Updates to the actual index data, which are very expensive, may be delayed and applied in batches. Doing so is the topic of Section 6.5.



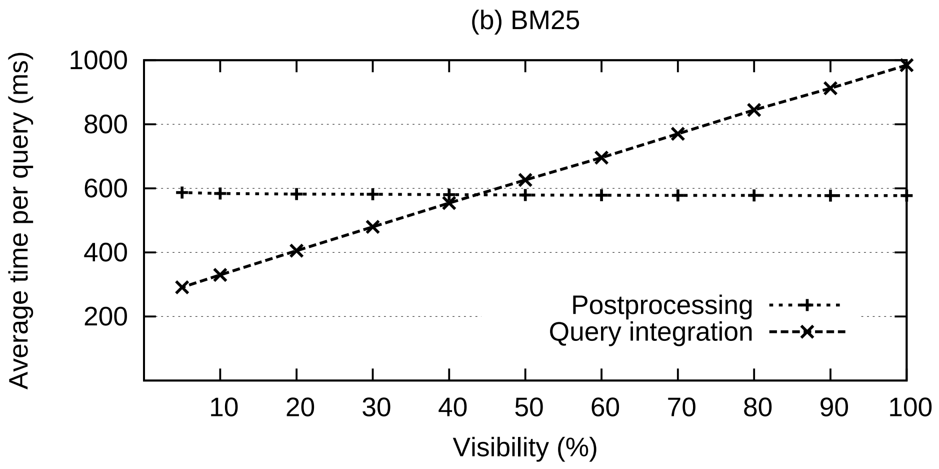


Figure 5.7: Performance comparison: Query integration using GCL operators vs. postprocessing approach. Time per query for the optimized integration is between 50% and 171%, compared to postprocessing. The actual time depends on the scoring function employed and on the relative number of files searchable by the user.

5.5 Performance Evaluation

I experimentally evaluated the performance of the query integration approach from the previous section and the second implementation of the postprocessing approach (keeping cached meta-data in the search engine), using the TREC corpus as the test collection. See Chapter 3 for a description of the corpus and the search queries used.

The 1.5 million documents in the collection were split into 15,449 files, and each of the 500 search queries was run against the index built for the collection (using QAP [32] and BM25 [94] to score and rank the set of matching documents). For both scoring functions, I conducted several experiments, with different percentages of searchable files in the file system.

Starting with the situation in which all 15,449 files are searchable by the user who submits the queries, collection visibility was gradually reduced, from 100% down to 5% (by marking a random fraction, p, of all files as readable). At each visibility level, all 500 queries were evaluated; between each consecutive pair of experiments, the operating system's disk cache was emptied, by reading large amounts of random data from the file system.

For each query, the search engine is asked to identify the top 20 documents according to the respective scoring function. When the GCL-based implementation is used, this is straightforward. For the postprocessing implementation, however, the situation is a bit more complicated. In order to be able to report the top 20 *searchable* results back to the user, the search engine needs to keep track of a large number of top-ranking documents, because not all of them will be searchable by the user. In the experiments, I configured the search engine so that it produced a set of k top-ranking documents, where k was chosen in such a way that the probability that at least 20 of out of the k documents were searchable by the user was at least 99%:

$$\Pr[\#\text{visible in top } k \geq 20] = 1 - \sum_{i=0}^{19} p^i \cdot (1-p)^{k-i} \cdot \binom{k}{i} \;\geq\; 0.99. \tag{5.35}$$

At visibility level $p = 0.2$, for instance, the search engine needs to keep track of the top $k = 153$ documents in order to meet the requirement.

This has consequences for query processing performance. First, maintaining a large set of top documents requires more complex book-keeping operations than maintaining a small set of top documents (updating the MIN-HEAP that contains the top k search results requires $\Theta(\log(k))$ operations on average). Second, the MAXSCORE [109] heuristic becomes less effective as k gets larger, since more document scores need to be computed before a query term can be removed from the priority queue used in the document-at-a-time query processing strategy employed by the search engine (cf. Section 2.2.2).

The performance results obtained under these circumstances are shown in Figure 5.7. As expected, if the percentage of searchable files is large, then the postprocessing approach yields better performance than query integration. For QAP and 100% visibility, query integration increases average response time by 35% compared to postprocessing (BM25: 71%). However, as the number of searchable files in the collection is decreased, query processing time drops for the query integration approach, since fewer documents have to be examined and fewer relevance scores have to be computed. For the postprocessing approach, on the other hand, query performance remains about constant or is even slightly increased, which is consistent with the expectation. For a visibility level of 5%, query integration reduces the average time per query by 45% (QAP) and 50% (BM25), respectively.

GCL query:

$$(\texttt{<doc>} \cdots \texttt{</doc>}) \rhd ((\texttt{mad} \wedge \texttt{cow}) \lhd [3])$$

Operator tree:

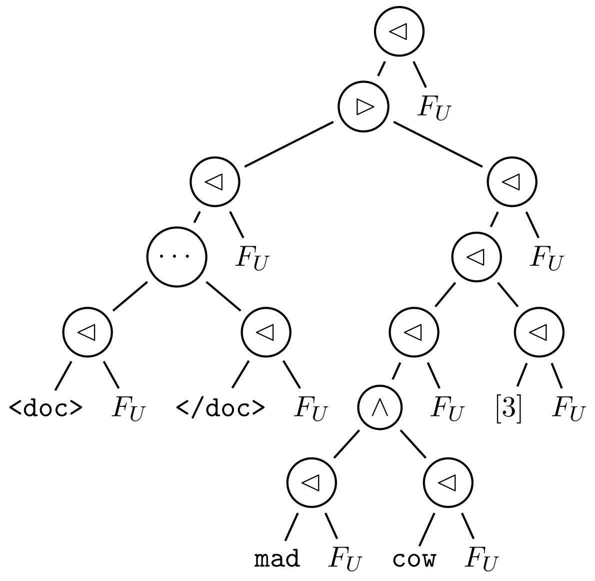


Figure 5.8: New operator tree, following the file independence assumption. The tree corresponds to the one shown in Figure 5.5, but with security restrictions applied to every tree node.

5.6 Query Optimization

Although the GCL-based implementation of the security model does not exhibit an excessively decreased performance, it is nevertheless slower than the postprocessing approach if more than 50% of the files can be searched by the user. The slowdown is caused by applying the security restrictions ($\ldots \lhd F_U$) not only to every query term, but also to the document delimiters (`<doc>` and `</doc>`) used in the TREC collection. These document delimiters, just like ordinary query terms, need to be fetched from the index and are therefore subject to the same security restrictions as the actual query terms.

Obviously, in order to guarantee consistent query results, it is only necessary to apply them to either the documents (in which case the scoring function will ignore all occurrences of query terms that lie outside searchable documents) or the query terms (in which case unsearchable documents would not contain any query terms and therefore receive a score of 0). However, this optimization would be very specific to the concrete query type (TF/IDF relevance ranking) used in the experiments described here.

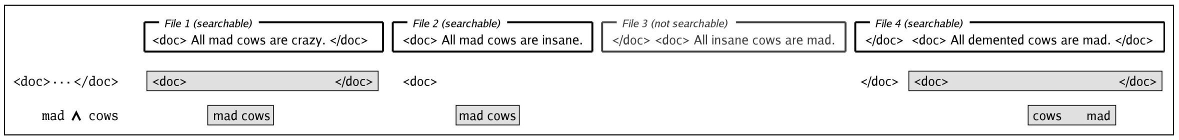


Figure 5.9: Query results for two example queries ("`<doc>` ··· `</doc>`" and "`mad` ∧ `cows`") with revised security restrictions applied. Even though `<doc>` in file 2 and `</doc>` in file 4 are visible to the query processor, they are not considered valid results, since they are in different files.

In order to address the problem of redundant security restrictions in a more general way, a new constraint is introduced: file independence. Assuming that files in the file system can be indexed by the search engine in any order, their relative order in the index address space may be a random permutation. Hence, issuing queries that produce GC-lists spanning across multiple files has no meaningful interpretation any longer, not even for intermediate search results.

It is therefore possible to adjust the secure GCL operator tree in such a way that security restrictions are applied to every node of the original, unsecure tree so that every search result (including every intermediate result), must reside completely within one file. The resulting operator tree is shown in Figure 5.8. The effect that the file independence constraint has on the search results is exemplified by Figure 5.9.

Obviously, applying security restrictions to every single node is overkill. However, the new tree can serve as a starting point for query optimization and can be transformed into a simpler, equivalent form. Recall the semantics of the seven GCL operators $\{\cdots, \vee, \wedge, \rhd, \not\rhd, \lhd, \not\lhd\}$ from Section 2.2.4. For two arbitrary GCL expressions E_1 and E_2, the following equivalences hold:

1. $((E_1 \lhd F_U) \cdots (E_2 \lhd F_U)) \lhd F_U \;\equiv\; (E_1 \cdots E_2) \lhd F_U$
2. $((E_1 \lhd F_U) \vee (E_2 \lhd F_U)) \lhd F_U \;\equiv\; (E_1 \vee E_2) \lhd F_U$
3. $((E_1 \lhd F_U) \wedge (E_2 \lhd F_U)) \lhd F_U \;\equiv\; (E_1 \wedge E_2) \lhd F_U$
4. $((E_1 \lhd F_U) \rhd (E_2 \lhd F_U)) \lhd F_U \;\equiv\; (E_1 \rhd E_2) \lhd F_U$
5. $((E_1 \lhd F_U) \not\rhd (E_2 \lhd F_U)) \lhd F_U \;\equiv\; (E_1 \not\rhd E_2) \lhd F_U$

However:

6. $((E_1 \lhd F_U) \lhd (E_2 \lhd F_U)) \lhd F_U \;\not\equiv\; (E_1 \lhd E_2) \lhd F_U$
7. $((E_1 \lhd F_U) \not\lhd (E_2 \lhd F_U)) \lhd F_U \;\not\equiv\; (E_1 \not\lhd E_2) \lhd F_U$

Algorithm 5 Constructing a secure GCL operator tree. Security restrictions are moved up in the tree as far as possible, following the file independence assumption. Input: A GCL expression E and a list of searchable files F_U. Output: A secure GCL expression E_U.

```
procedure makeSecure($E$, $F_U$):
 1: $E \leftarrow makeAlmostSecure(E, F_U)$
 2: return $(E \lhd F_U)$

procedure makeAlmostSecure($E$, $F_U$):
 3: if $E.leftChild = nil$ then
 4:    // this is a leaf node; by definition, it is almost secure
 5:    return $E$
 6: end if
 7: $E.leftChild \leftarrow makeAlmostSecure(E.leftChild, F_U)$
 8: $E.rightChild \leftarrow makeAlmostSecure(E.rightChild, F_U)$
 9: if $(E.operator = \lhd) \vee (E.operator = \not\lhd)$ then
10:    // in order to make this expression almost secure, restrictions need to be
11:    // applied to the container component of the containment expression
12:    $E.rightChild \leftarrow (E.rightChild \lhd F_U)$
13: end if
14: return $E$
```

Equivalences 1 through 5 hold because each of the five operators $\{\cdots, \vee, \wedge, \rhd, \not\rhd\}$ is monotonic. That is, if an index extent $\mathcal{E}$ matches the expression $E_1 \diamond E_2$ (where "$\diamond$" is one of the five operators), then the extents matching E_1 and E_2 from which $\mathcal{E}$ was formed by applying the GCL operator are always covered by $\mathcal{E}$. Therefore, applying the security restrictions exclusively to the top-most level of the respective operator tree does not affect the outcome of the search operation.

The operators $\{\lhd, \not\lhd\}$, however, are non-monotonic. For example, the GCL query

"term" $\lhd$ (`<doc>` $\cdots$ `</doc>`)

may evaluate to an index extent that is smaller than the input extent matching the subexpression `<doc>` $\cdots$ `</doc>`. This phenomenon has already been exploited in Section 5.3.3 to obtain information about arbitrary phrase occurrences if security restrictions are enforced by means of postprocessing.

It is unclear which strategy yields the best results for general GCL queries. However, for typical ranked queries, finding a set of matching documents by applying a Boolean OR (GCL operator "$\vee$"), good performance is achieved by moving the security restrictions as far up in the tree as possible. This procedure is formalized by Algorithm 5. The algorithm takes a GCL expression E and a user-specific restriction list F_U; it produces a security-restricted GCL expression E_U in a bottom-up fashion.

GCL query:

$$(\texttt{<doc>} \cdots \texttt{</doc>}) \triangleright ((\texttt{mad} \wedge \texttt{cow}) \triangleleft [3])$$

Operator tree:

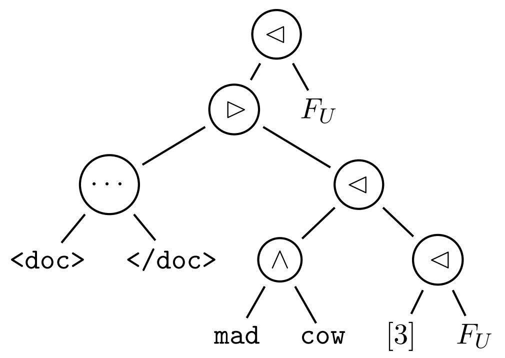


Figure 5.10: GCL query and resulting operator tree with optimized security restrictions applied to the operator tree.

Algorithm 5 makes use of the concept of an *almost secure* operator tree. A GCL tree is *almost secure* if it can be made secure by applying a single security restriction operator at the root node of the tree. A basic GCL expression (corresponding to a single term or a phrase) is always almost secure. A compound expression is almost secure if one of the following two requirements holds:

- the root node's operator is one of $\{\cdots, \vee, \wedge, \triangleright, \ntriangleright\}$ and both subtrees (left and right) are almost secure;
- the root node's operator is one of $\{\triangleleft, \ntriangleleft\}$, the left subtree (containee) is almost secure, and the right subtree (container) is secure.

The first requirement is enforced by lines 7–8. The second requirement is enforced by lines 9–12. The secure GCL operator tree produced by the algorithm, for the same search query as before, is shown in Figure 5.10. It contains two instances of the security restriction operator. The one at the root node of the operator tree is necessary to make the tree secure. The one applied to the subexpression "[3]" (matching all index extents of length 3) is not strictly necessary; it is an artifact of Algorithm 5.

I evaluated the performance of this new implementation of the secure query processing strategy, employing the move-to-top heuristic in order to reduce the total number of applications of the security restriction operator. The experimental results obtained are shown in Figure 5.11.

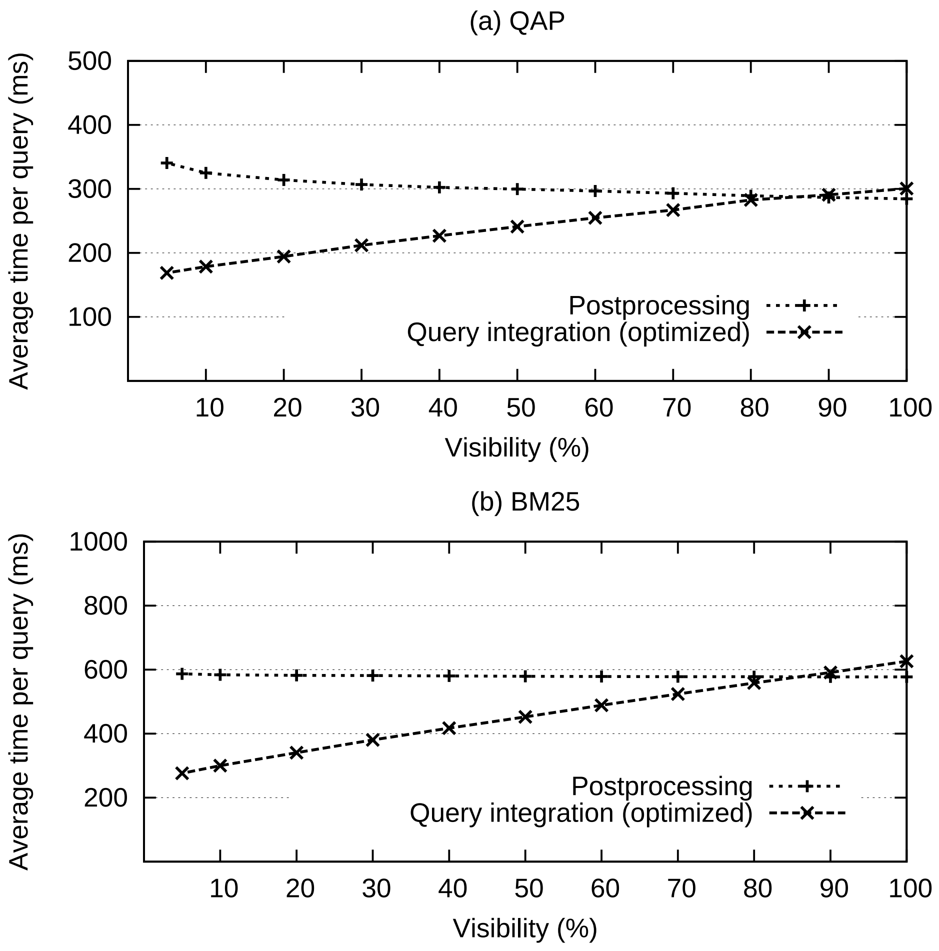


Figure 5.11: Performance comparison – query integration using GCL operators (optimized) vs. postprocessing approach. Compared to postprocessing, the average time per query for the optimized query integration is between 47% and 108%, depending on ranking function and visibility.

The security restrictions applied by the new implementation are very light-weight. Compared to the postprocessing approach, the slowdown for a file visibility level $p = 1$ is only 6% for QAP and 8% for BM25. At visibility level $p = 0.05$, the new implementation is more than twice as fast as postprocessing, for both QAP and BM25.

5.7 Discussion

I have investigated the security issues that arise in the context of multi-user file system search, when the search engine's index is shared and accessed by multiple users. I have shown that enforcing security restrictions by means of postprocessing leads to vulnerabilities that may be exploited by an arbitrary user to gain information about files she is not allowed to read. This problem applies not only to file system search, but also to related areas, such as enterprise search, where the same index is also queried by a variety of users with diverse access privileges. As most enterprise search engines employ a postprocessing step of some sort to remove results not accessible by the user who submitted the query [5], they are most likely amenable to attacks similar to the ones described in Section 5.3. However, this is only speculation, as I did not have access to any of these systems to verify my claim.

Contrary to what I have suggested in previous work [17], Apple's Spotlight search engine is not affected by the postprocessing problem, since it does not order the search results by their expected relevance to the query before returning them to the user (cf. Section 2.2.3). While this is good news from a security perspective, it also reduces the usability of their search engine.

The alternative implementation of the security model – integrating it into the query processing logic by utilizing the structural operators provided by the GCL retrieval framework – fixes the security problem. As a side-effect, it also leads to better query performance, because the search engine never even computes retrieval scores for documents not searchable by the user. Only at a very high visibility level, close to 100%, postprocessing results in lower query response times than query integration. For true multi-user file systems, however, such a high visibility level is not realistic.

The limitation of the query integration approach is that it requires the search engine to hold a copy of all security-related meta-data in memory. If the file system exclusively uses traditional UNIX file permissions (i.e., no ACLs), then this is not too unrealistic. Every indexed file will require approximately 10 bytes of memory (2 bytes each for owner, group, and permission vector; about 4 bytes for vByte-encoded [100] start and end address), which seems tolerable for a file system containing a few million files. For very large file systems, however, containing dozens of millions of files, the resulting memory requirements might exceed the limit that the users of the system are willing to sacrifice. Also, if users define more complicated access permissions, for example through access control lists, memory consumption may easily be greater than 10 bytes per file, as the upper limit on the number of rules in an ACL can be very large.

Even if all security-related meta-data may be kept in memory at all times, they still represent a performance bottleneck. Whenever the search engine receives a new query, it needs to determine group membership for the user U who submitted the query and construct the user-specific security list F_U, requiring time $\Omega(|F_U|)$. Holding F_U in a cache is certainly possible, but not very realistic in a true multi-user environment.

An alternative to the proposed method of enforcing security restrictions is to store the lists F_U in the index and to treat them like ordinary index terms. Unfortunately, this introduces a whole range of problems related to updating these lists. Not only would a single `chmod` operation need to be propagated to $|\mathcal{U}|$ different posting lists (where $\mathcal{U}$ is the set of all users in the system); the new implementation would also require the search engine to maintain an explicit invalidation list, containing information about obsolete index data and used for lazy deletion (discussed in more detail in Section 6.5). In the current implementation, this is realized through the security framework. If the security framework itself is based upon ordinary posting lists, then this is no longer possible.

All in all, despite the shortcomings pointed out above, the implementation of the security model suggested in this chapter seems to be the best trade-off. For medium-sized file systems, containing only a few million indexable files, it leads to reasonable results, in terms of both memory consumption and query performance.

Chapter 6

Real-Time Index Updates

In Chapter 4, I have described several performance optimizations for inverted files. Taken together, the techniques laid out allow a file system search engine to efficiently build, and periodically update, an index for the contents of the file system and to provide low-latency search operations to the users. The memory consumption of the search engine's index structures is very low, thanks to the interleaved dictionary representation proposed in Section 4.4.2. In this chapter, I extend the techniques from Chapter 4, discussing how the index, after it has been built, can be kept up-to-date and consistent with the continuously changing contents of the file system.

The first part of the chapter focuses on incremental updates, allowing new files to be added to the collection, but not allowing existing files to be deleted or modified. It compares merge-based (Section 6.2) and in-place (Section 6.3) index update strategies and quantify the trade-off between update performance and query response time that arises by dividing the on-disk index into several index partitions, each containing parts of the total index data. I propose a family of hybrid index maintenance strategies — a mixture of in-place and merge-based update, where long lists are updated in-place, while short lists are maintained using a merge strategy (Section 6.4). Hybrid index maintenance combines the advantages of in-place update (small number of postings transferred from/to disk) and merge-based update (largely sequential disk access pattern during update operations).

In the second part of the chapter, I attend to file deletions and file modifications. I study garbage collection strategies that, in conjunction with the secure query processing framework described in the previous chapter, can be used to realize support for deletion operations (Section 6.5). I then investigate possible mechanisms that can be integrated into the search engine to provide support for partial document modifications, in particular those of the Append class (Section 6.6).

The methods presented up to, and including, Section 6.6 do not actually meet any strict real-time requirements, as all update operations are only guaranteed to have low *amortized* complexity. In Section 6.7, I therefore describe a small set of modifications that can be made to the search engine in order to change the amortized nature of all index update operations into actual real-time index updates.

6.1 General Considerations and Terminology

When new documents are added to an existing text collection, the search engine's internal index structures need to be updated, so as to reflect the new situation. If the index is stored in main memory, then this is a relatively easy task (as demonstrated in Section 4.3). If the index is stored on a hard disk, however, realizing efficient index updates can be quite challenging. A file F that is added to the file system can be very small, yet at the same time contain hundreds of different terms. If an in-place update strategy is chosen, then updating the inverted lists for all terms appearing in F requires a large number of disk seeks. Their total time can easily take a hundred or even a thousand times longer than the initial event that triggered the index update (i.e., the creation of F). For this reason, it is absolutely crucial to buffer index information for incoming documents in main memory and to apply all updates to the on-disk index structures in a larger batch, as already pointed out in Section 2.5.

Accumulating updates in memory has two effects. First, it reduces the relative number of on-disk inverted lists that need to be updated. Two different files, although they might be independent of each other, are likely to share some common terms (Zipf's law). Therefore, the ratio

$$\frac{\#\text{lists to be updated}}{\#\text{postings to be added}}$$

shrinks as more updates are buffered in memory. For the three text collections used in my experiments, this effect is shown in Table 6.1.

Second, accumulating updates in memory increases the number of on-disk inverted lists that need to be updated and thus allows the index maintenance process to schedule on-disk updates in a more efficient way, reducing the seek distance between two on-disk list updates. The second effect can even be observed if the newly added documents do not share any common terms.

The drawback of buffering index data for incoming files in main memory is that these data might not be immediately accessible to the search engine's query processing routines and that the user has to wait until the next update of the on-disk index structures before new files become visible in the search results. Fortunately, this deficiency can be cured by employing

Text collection	SBFS			TREC			GOV2		
k (tokens seen) $\times 10^6$	0.1	1.0	10.0	0.1	1.0	10.0	0.1	1.0	10.0
Unique terms seen $\times 10^3$	22.6	134.2	270.6	10.8	34.3	90.4	8.0	33.3	147.6
Tokens per term	4.4	7.5	37.0	9.2	29.2	110.7	12.5	30.0	67.7

Table 6.1: Indexing the first $k = 10^5, 10^6, 10^7$ tokens from all three text collections. The ratio #terms seen / #tokens seen ("tokens per term") shrinks as k gets bigger.

the hash-based in-memory indexing method from Section 2.4.3 (and its variation in discussed in Section 4.3.1) to organize incoming postings data in main memory. The extensible posting lists employed by hash-based indexing can immediately be used for query processing purposes. By concatenating a term's on-disk list with the list found in memory, the search engine always has access to the most recent version of each term's posting list.

Because in-memory index construction can be performed very efficiently, the procedure outlined above allows for near-instantaneous index updates. Every now and then, however, the inverted lists constructed in memory do need to be combined with the existing on-disk index. This batch update operation is very costly in terms of disk activity. The main focus of the first part of this chapter, therefore, is to reduce the search engine's disk activity during these on-disk update cycles.

Every strategy for combining in-memory index data with the existing on-disk structures represents a performance trade-off. Search queries can be processed most efficiently if the on-disk inverted list for every term in the index is stored in a contiguous region of the hard disk (minimizing disk seeks). However, enforcing contiguity can be expensive. Sometimes, when appending in-memory postings to an on-disk inverted list, there is not enough free space at the list's current position on disk. In that case, the entire list needs to be read from disk and written to a new location. As the text collection, and therefore the on-disk index, grows, the performance degradation caused by such a copy operation becomes more and more grave, up to a point where it might no longer be feasible to guarantee the contiguity of on-disk posting lists.

Terminology

In some rare cases, when comparing two different index maintenance strategies A and B, it turns out that A leads to both better update performance as well as better query performance than B. I shall then say that A *dominates* B. This is an unusual situation, however. In most cases, two strategies A and B do in fact represent different trade-off points, where one strategy allows faster updates, while the other leads to more efficient processing of search queries.

```
@addfile trec/chunks/trec.04093
@rank[qap][count=20][id=215] "⟨doc⟩"··· "⟨/doc⟩" by "unexplained", "highway", "accidents"
@addfile trec/chunks/trec.08966
@rank[qap][count=20][id=8] "⟨doc⟩"··· "⟨/doc⟩" by "rail", "strikes"
@addfile trec/chunks/trec.14707
@rank[qap][count=20][id=104] "⟨doc⟩"··· "⟨/doc⟩" by "oil", "spills"
.....
```

Figure 6.1: Beginning of the combined insertion/query sequence for the TREC collection used in my experiments. The command sequence is meant to simulate a fully dynamic search environment, where index update operations are interleaved with search operations. It consists of 15,449 @addfile and 15,449 @rank commands (random queries selected from TREC topics 1-500).

6.2 Merge-Based Update

The Immediate Merge and No Merge strategies introduced in Section 2.5.1 represent two extreme points in the trade-off between index maintenance and query processing performance. Immediate Merge combines the in-memory index with the on-disk index by merging them in a sequential fashion, resulting in a new on-disk inverted file that replaces the old one. Because the new index is created from scratch, one list at a time, all its posting lists will be stored on disk in a contiguous fashion, optimizing query performance. The No Merge strategy is targeting the other end of the spectrum. By creating a separate on-disk inverted file every time the search engine's in-memory update buffers are exhausted, fragmentation is maximized, leading to a large number of disk seeks when processing a search query. Indexing performance, on the other hand, is optimal, because postings are never relocated after they have been written to disk.

I analyzed the performance of both methods, No Merge and Immediate Merge, in a dynamic search environment, where the search engine has to deal with file insertions and search operations at the same time. Such an environment is simulated by the command sequence shown in Figure 6.1. One after the other, each of the 15,449 files comprising the TREC collection is indexed. After every file insertion operation, a random search query from the TREC query set (cf. Chapter 3) is processed.

The performance results obtained through measurement are shown in Figure 6.2. Plot (a) depicts the per-file indexing time for No Merge and Immediate Merge. Each file added to the index contains 100 TREC documents. The vast majority of all files (94.5%) requires less than 100 ms to be indexed. A very small number of insertions, however, triggers a physical

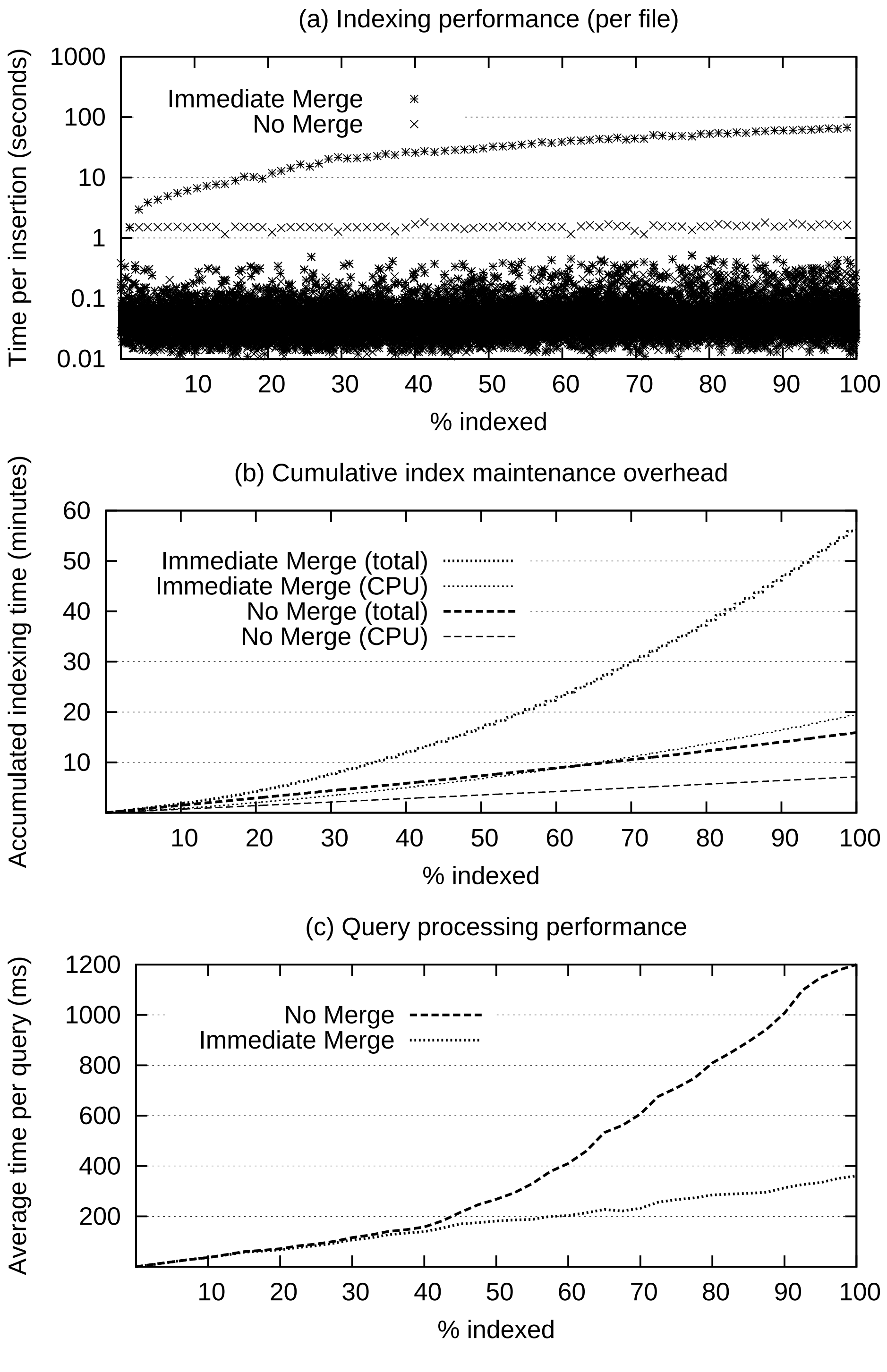


Figure 6.2: Index maintenance and query processing performance for TREC, measured as the collection grows (100% $\widehat{=}$ 1.5 million documents, 824 million tokens). Space for in-memory buffers: 32 MB. Hardware used: AMD Sempron 2800+ desktop PC (32-bit CPU, 768 MB RAM).

index update (search engine runs out of memory). For No Merge, this means that the in-memory posting lists have to be sorted in lexicographical order and written to disk. This operation takes about 1.6 seconds on average. For Immediate Merge, each physical index update also requires the search engine to combine the in-memory index data with the on-disk inverted file. The time required to carry out this re-merge task grows linearly with the size of the on-disk index (note that the vertical axis in Figure 6.2(a) is in logarithmic scale), starting at 3 seconds for the first merge operation, and gradually increasing to 67 seconds for the last one.

Plot (b), just like (a), shows the time spent on indexing operations, but cumulative instead of per file. While No Merge can build and update the index in linear time, Immediate Merge exhibits a quadratic time complexity, requiring a total of 56 minutes to index the collection (compared to a total of 16 minutes in the case of No Merge).

The quadratic complexity of Immediate Merge is caused by the fact that it needs to read and write the entire on-disk index whenever the search engine runs out of main memory. Let N denote the total size of the collection (number of postings in the index at the end of the experiment) and M the number of postings that can be accumulated in RAM before the in-memory buffers are exhausted. Then the total number of postings written to disk is

$$D(N,M) \;=\; \sum_{i=1}^{\lfloor N/M \rfloor} i \cdot M \;=\; M \cdot \frac{\lfloor N/M \rfloor \cdot \lfloor N/M + 1 \rfloor}{2} \;\in\; \Theta\left(\frac{N^2}{M}\right). \qquad (6.1)$$

Figure 6.2(b) also shows the cumulative CPU time consumed by both strategies. Thanks to the optimization from Section 4.3.2 (merging posting lists without prior decompression), Immediate Merge's CPU utilization is around 35%, proving that the bottleneck of this strategy is its disk activity.

It might be possible to slightly decrease disk activity by applying better index compression techniques than the vByte [100] method used here. However, this does not solve the problem in general and also introduces new intricacies due to the need of re-compressing posting lists during a merge operation. This indicates that there is a limit for Immediate Merge's update performance that cannot be overcome. If M is small compared to N, then there might be better update strategies than Immediate Merge.

Plot 6.2(c), finally, shows the query performance achieved by both strategies in the dynamic environment simulated by the given command sequence: time per search query, averaged over a sliding window covering 300 queries. Not surprisingly, No Merge performs much worse than Immediate Merge. Towards the end of the experiment, a search query takes 3 times longer to be processed with No Merge than with Immediate Merge. This is caused by the large number of inverted files maintained by No Merge. In total, 78 independent on-disk indices are created. For every query term, the search engine needs to visit each of these indices, resulting in 78 disk seeks per query term.

What is surprising here, however, is that the difference in query performance between the two update strategies only becomes apparent after 40% of the TREC collection have been indexed. Before that, the average time per query is essentially the same for both. This can be explained by the relative magnitude of the index and the main memory of the computer used to run the experiment. The final index for TREC consumes 1.4 GB; the computer used to conduct the experiment is equipped with 768 MB of RAM. Thus, up until the half-way point, most index operations required during query processing are cached by the operating system.

Nonetheless, if M is small compared to N, then the number of disk seeks performed during query processing will be very large. At the end of the experiment (100% indexed), an average query takes about 230% longer with No Merge than with Immediate Merge (1200 ms vs. 360 ms).

6.2.1 Index Partitioning

The general problem of updating an inverted file is very similar to the problem of providing efficient lookup operations for a growing set of integer numbers (or any other kind of totally ordered objects). That is, the implementation needs to support an *insert(x)* operation, adding a new integer x to the existing set, and a *find(x)* operation returning the smallest integer i in the set such than $i \geq x$.

Suppose the data structure used to maintain the set is a simple contiguous array, sufficiently large and stored in main memory. Then there are two extreme update strategies:

1. A new element is inserted into the set by appending it to the existing array, increasing the size of the array by 1. This can be realized in constant time, assuming sufficient space is available at the end of the array. However, because the elements of the set may appear in arbitrary order in the array, a search operation on this array has a worst-case time complexity of $\Theta(n)$.

2. The complexity of search operations can be reduced by ensuring that the elements of the set are always stored in the array in increasing order. An element of the set can then be found by a binary search operation, requiring $\Theta(\log(n))$ steps. Inserting a new element, however, requires the relocation of n array elements in the worst case and thus exhibits a time complexity of $\Theta(n)$.

Translated back into the context of text retrieval, strategy 1 corresponds to No Merge, while strategy 2 corresponds to Immediate Merge (assuming a buffer size $M = 1$ in both cases).

Bentley and Saxe [10] present an algorithm that achieves an amortized insertion complexity of $\Theta(\log(n))$ and a worst-case search complexity of $\mathcal{O}(\log^2(n))$, as follows. Instead of maintaining a single sorted array, as in Strategy 2, the algorithm maintains a family of arrays $A_0, A_1, A_2, \ldots$. Each array A_i is either empty, or it contains 2^i elements, in sorted order. For a set of n integers, the algorithm maintains at most $\lceil \log_2(n+1) \rceil$ arrays. A lookup operation on this data structure can be realized by a sequence of $\mathcal{O}(\log_2(n))$ binary search operations, one for each array A_i, with total time complexity

$$C_{\text{lookup}}(n) \;\leq\; \sum_{i=0}^{\lfloor \log_2(n) \rfloor} \log_2(2^i) \;=\; \sum_{i=0}^{\lfloor \log_2(n) \rfloor} i \;\in\; \mathcal{O}\left(\log^2(n)\right). \tag{6.2}$$

When a new element has to be inserted, it is appended to the array A_0. If that leads to a situation in which $|A_0| > 1$, then A_0 is sorted and merged with A_1. If this causes A_1 to contain more than 2 elements, then A_1 is merged with A_2 and so forth, until there are no more violations of the constraint $A_i \leq 2^i$.

Now consider the total complexity of all insertion operations. After inserting n elements into the data structure, each element has participated in at most $\lceil \log_2(n+1) \rceil$ merge operations (because there are at most $\lceil \log_2(n+1) \rceil$ non-empty arrays). Therefore, the combined complexity of all insertion operations is

$$C^*_{\text{insert}}(n) \;\leq\; n \cdot \lceil \log_2(n+1) \rceil \;\in\; \mathcal{O}(n \cdot \log(n)), \tag{6.3}$$

and the amortized complexity per insertion operation is:

$$C_{\text{insert}} \;\in\; \mathcal{O}(\log(n)). \tag{6.4}$$

The worst-case complexity of an insertion operation, however, is still $\Theta(n)$; this happens, for example, when adding a new element to a set containing $2^k - 1$ elements (for some positive integer k). Overmars and van Leeuwen [84] present a variation of Bentley's scheme that exhibits the same lookup performance as before, but that achieves a $\mathcal{O}(\log^2(n))$ worst-case (i.e., non-amortized) insertion complexity.

The general technique described above is referred to as the *logarithmic method*. It can be directly applied to the task of updating an inverted index for a growing text collection, result-

Algorithm 6 On-line index construction with Logarithmic Merge. Building a schema-independent index for a potentially infinite text collection.

```
1: let $I_0$ denote the search engine's in-memory index
2: let $I_g$ denote the on-disk index partition of generation $g$
3: $I_0 \leftarrow \emptyset$ // initialize in-memory index
4: $currentPosting \leftarrow 1$
5: while there are more tokens to index do
6:   $T \leftarrow$ next token
7:   $I_0.addPosting(T,\ currentPosting)$
8:   $currentPosting \leftarrow currentPosting + 1$
9:   if $I_0$ contains more than $M - 1$ postings then
10:     $\mathcal{I} \leftarrow \{I_0\}$
11:     $g \leftarrow 1$
12:     while $I_g$ is not empty do
13:       $\mathcal{I} \leftarrow \mathcal{I} \cup \{I_g\}$
14:       $g \leftarrow g + 1$
15:     end while
16:     $I_g \leftarrow mergeIndices(\mathcal{I})$
17:     delete every index $I \in \mathcal{I}$
18:     $I_0 \leftarrow \emptyset$ // re-initialize in-memory index
19:   end if
20: end while
```

ing in the Logarithmic Merge update strategy (Algorithm 6). Where Bentley's method maintains a set of arrays of exponentially increasing size (i.e., $2^0, 2^1, \ldots, 2^k$), Logarithmic Merge maintains a set of on-disk indices I_j, containing $M \cdot 2^j$ postings each, where M is the number of postings that may be buffered in memory. Therefore, in analogy to the above analysis, the total number of postings transferred from and to disk when building an index for a text collection containing N postings, is:

$$D(N, M) \in \Theta\left(N \cdot \log\left(\frac{N}{M}\right)\right). \tag{6.5}$$

This is substantially better than the quadratic disk complexity of Immediate Merge. The set of index partitions maintained by Logarithmic Merge, over a sequence of index update cycles, is shown in Figure 6.3.

Lester et al. [68] generalize this scheme by allowing more general constraints on the relative size of the individual on-disk inverted files. As an additional point of comparison, I will use the variant of their *geometric partitioning* method in which they allow at most two on-disk

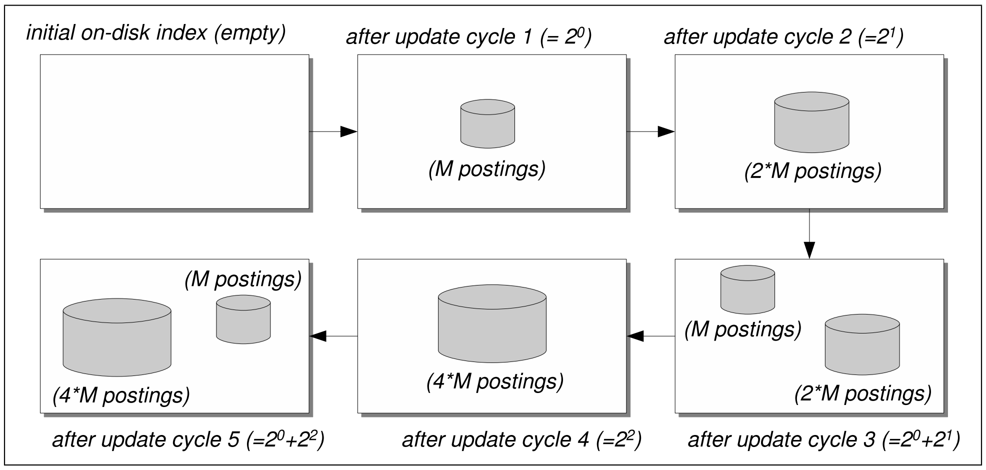

initial on-disk index (empty)
after update cycle 1 (= 2^0)
after update cycle 2 (=2^1)
(M postings)
(2*M postings)
(M postings)
(M postings)
(4*M postings)
(4*M postings)
(2*M postings)
after update cycle 5 (=2^0+2^2)
after update cycle 4 (=2^2)
after update cycle 3 (=2^0+2^1)

Figure 6.3: Index structures over time, with Logarithmic Merge as update strategy, assuming that the search engine can buffer M postings in RAM before it runs out of memory. Each cylinder represents a separate inverted file on disk.

inverted files to exist in parallel. This method has a total disk update complexity of

$$D(N, M) \in \Theta\left(N \cdot \sqrt{\frac{N}{M}}\right) \tag{6.6}$$

when building an index for a collection of size N (see [68] for an analysis). I shall refer to it as Sqrt Merge.

Both index partitioning strategies, Logarithmic Merge and Sqrt Merge, lead to a situation in which on-disk inverted lists are no longer stored in contiguous regions of the disk. Therefore, although they exhibit an improved update performance, compared to Immediate Merge, we can expect to see a reduced query processing performance, due to the additional disk seeks that need to be performed to fetch all segments of a query term's posting list from the individual index partitions. This slowdown, however, is likely to be relatively small, because the number of separate index partitions is tightly controlled in both cases.

Implementation Details

The actual implementation of Logarithmic Merge used in my experiments is slightly different from Algorithm 6. Instead of assigning a fixed generation label to each index partition, generation numbers are computed dynamically, based on the byte size of each partition. The adjustment is necessary because the search engine also provides support for document deletions (Section 6.5). In the presence of deletions, merging two inverted files containing x and y postings, respectively, no longer results in a new inverted file containing $x + y$ postings, as some of the postings involved in the merge operation might refer to files that have been deleted. Such postings will not show up in the target index, making it possible that a partition of generation $g + 1$ might contain fewer postings than one of generation g.

Deletions are taken into account by using the raw byte size of each index partition as the exclusive criterion to determine whether a merge operation needs to be performed or not. In particular, the following restriction is enforced:

$$\text{size}(I_{k+1}) > 1.5 \cdot \text{size}(I_k) \;\; \forall\, k. \tag{6.7}$$

The inter-partition growth factor $q = 1.5$ was chosen somewhat arbitrary. It does, however, need to be considerably smaller than 2, because merging two index partitions I_m and I_n results in a partition I_p with $\text{size}(I_p) < \text{size}(I_m) + \text{size}(I_n)$, due to term overlap between I_m and I_n.

Experiments

As before, I had the search engine build an index for the TREC collection, indexing documents and processing search queries in an interleaved fashion (cf. Figure 6.1), but this time allowing the engine to maintain more than just a single on-disk inverted file at a time. The results are depicted in Figure 6.4.

Plot 6.4(a) shows that both partitioning schemes, Logarithmic Merge and Sqrt Merge, exhibit a substantially better index maintenance performance than Immediate Merge. The total indexing time for Logarithmic Merge is 18.9 minutes (Sqrt Merge: 23.0 minutes), compared to an indexing performance baseline of 15.9 minutes achieved by static indexing (cf. Table 4.5). That is, the total overhead is only 3 minutes (+19%). At the same time, however, query performance is dramatically improved compared to No Merge, as shown in plot 6.4(b). When 100% of the collection have been indexed, Logarithmic Merge processes queries more than 3 times as fast as No Merge.

What is surprising about the experimental results is that there is virtually no difference between the query performance of Immediate Merge, Sqrt Merge, and Logarithmic Merge. This is unexpected, as the search engine needs to perform more disk seeks when processing a query from a partitioned index than from a single, non-partitioned index. At first blush, it is not entirely clear what exactly causes this abnormal behavior, but disk caching effects (because the index is only twice as large as the amount of RAM available to the operating system, and because the number of distinct search queries is rather small) are the most likely explanation.

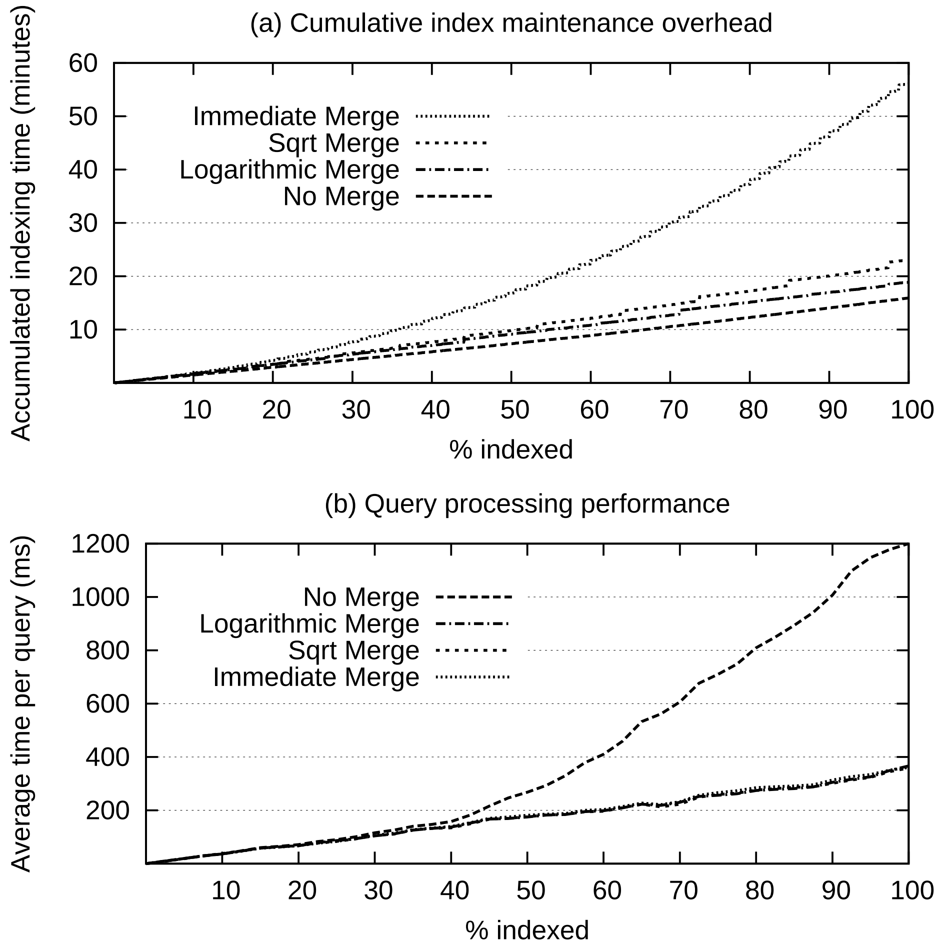


Figure 6.4: Indexing the TREC collection with interleaved query operations (command sequence from Figure 6.1). Index maintenance performance and query performance of different merge update strategies. In-memory update buffers: 32 MB.

I repeated the experiment with the larger GOV2 collection, so as to reduce the impact that the operating system's disk cache has on the search engine's query performance. The final index for GOV2 consumes 62 GB of disk space, 30 times more than the main memory installed in the 2-GB workstation hosting the collection. A fragment from the command sequence used for the revised experiment, comprising 27,204 update and query operations in total, is given by Figure 6.5.

The results obtained for GOV2 are shown in Figure 6.6. As before, the index partitioning schemes exhibit a total indexing overhead close to that of the No Merge strategy:

- No Merge: 309 minutes;
- Logarithmic Merge: 517 minutes (+67%);

```
@addfile gov2/GX148/01.txt
@rank[qap][count=20][id=85987] “⟨doc⟩”··· “⟨/doc⟩” by “cherry”, “juice”
@addfile gov2/GX066/63.txt
@rank[qap][count=20][id=11851] “⟨doc⟩”··· “⟨/doc⟩” by “radios”, “work”
@addfile gov2/GX008/63.txt
@rank[qap][count=20][id=16231] “⟨doc⟩”··· “⟨/doc⟩” by “population”, “florida”, “cities”
.....
```

Figure 6.5: Beginning of the combined insertion/query sequence for the GOV2 collection used in my experiments. This sequence is similar to the one shown in Figure 6.1, but for GOV2 instead of TREC. It consists of 27,204 @addfile and 27,204 @rank commands.

- Sqrt Merge: 1262 minutes (+308%).

Immediate Merge, on the other hand, is not competitive at all, requiring 9491 minutes (158 hours) to build an index for the text collection.

The difference between Logarithmic Merge and Sqrt Merge is greater than before, because the relative size of the in-memory buffers (256 MB), compared to the total size of the index (62 GB), is smaller than in the case of TREC (32 MB vs. 1.4 GB).

The query processing performance achieved by the different update policies is shown in plot 6.6(b). As before, No Merge has substantially lower query performance than the other strategies, requiring up to 300% more time per query than Immediate Merge. Regarding the relative performance of Immediate Merge and index partitioning, it can be seen that the difference is relatively small, as the three curves are virtually indistinguishable.

Figure 6.7 provides a different view of the relative query performance of the three update strategies, making it easier to see the exact differences. Plot 6.7(b) shows the absolute time difference per query, while plot 6.7(c) shows the relative query slowdown compared to Immediate Merge, both computed over a sliding window covering 150 search queries. By comparing the two plots to 6.7(a), it can be seen that the performance gap between partitioned and non-partitioned index is directly correlated to the number of partitions in the on-disk index. The sudden drop for Logarithmic Merge shortly before the 40% mark, for example, is caused by a merge operation reducing the number of active index partitions from 7 to 1. Before the re-merge operation, Logarithmic Merge can be expected to perform 6 additional disk seeks per query term (compared to Immediate Merge). The GOV2 query sequence contains 3.5 terms per query on average. Assuming an average disk seek latency of 8 ms, the expected slowdown is therefore: $6 \times 3.5 \times 8\text{ms} = 168\text{ms}$. The actual slowdown seen in the experiments is 154 ms.

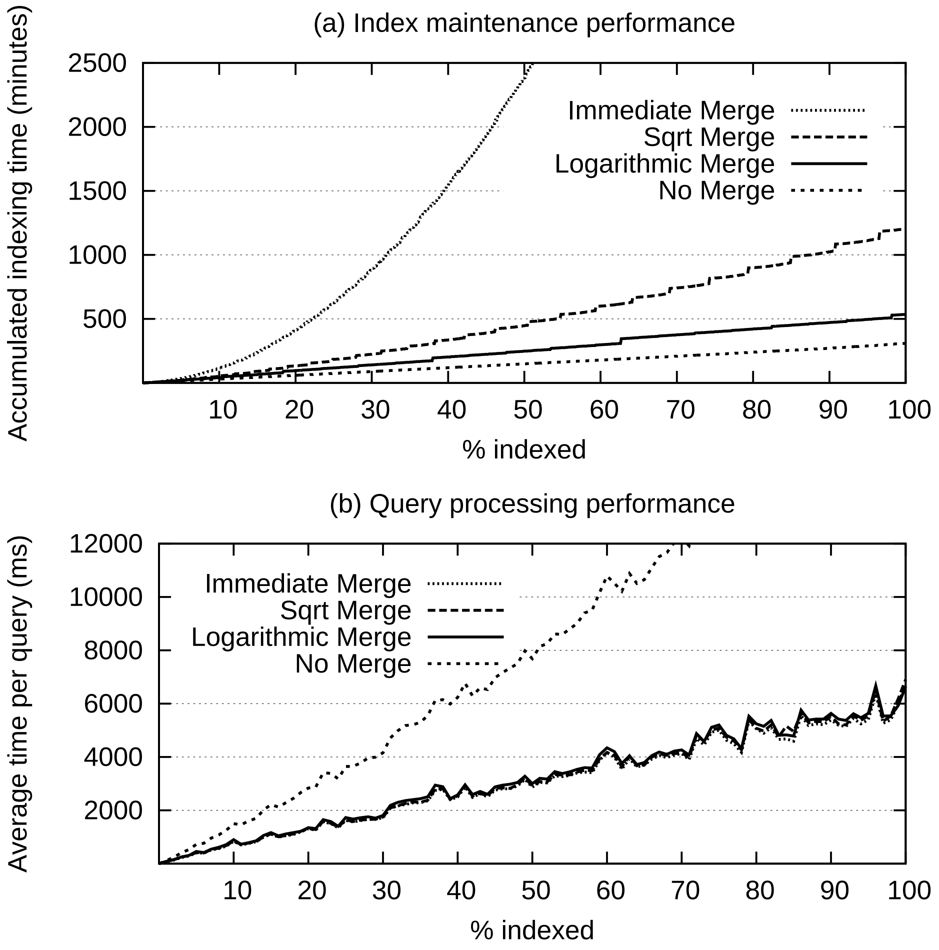


Figure 6.6: Index maintenance and query processing performance for GOV2, measured as the collection grows (100% $\widehat{=}$ 25.2 million documents, 43.6 billion tokens). Space for in-memory buffers: 256 MB.

The average per-query slowdown caused by the two index partitions maintained by Sqrt Merge hovers around 5% for the most time. The Logarithmic Merge strategy, usually maintaining 2–7 index partitions, leads to a slowdown between 10% and 20% for most queries. It is interesting that, while the absolute slowdown for Logarithmic Merge tends to increase (because of the growing number of index partitions), the relative slowdown shrinks. This is because the overall query cost increases approximately linearly with the size of the collection, but the slowdown introduced by Logarithmic Merge only increases logarithmically.

Table 6.2 shows the impact that the amount of main memory available to the search engine has on index maintenance performance. In contrast to the other two update policies, Logarithmic Merge's update performance is largely unaffected by available memory resources. Reducing the size of the in-memory buffers from 1024 MB to 32 MB only increases the main-

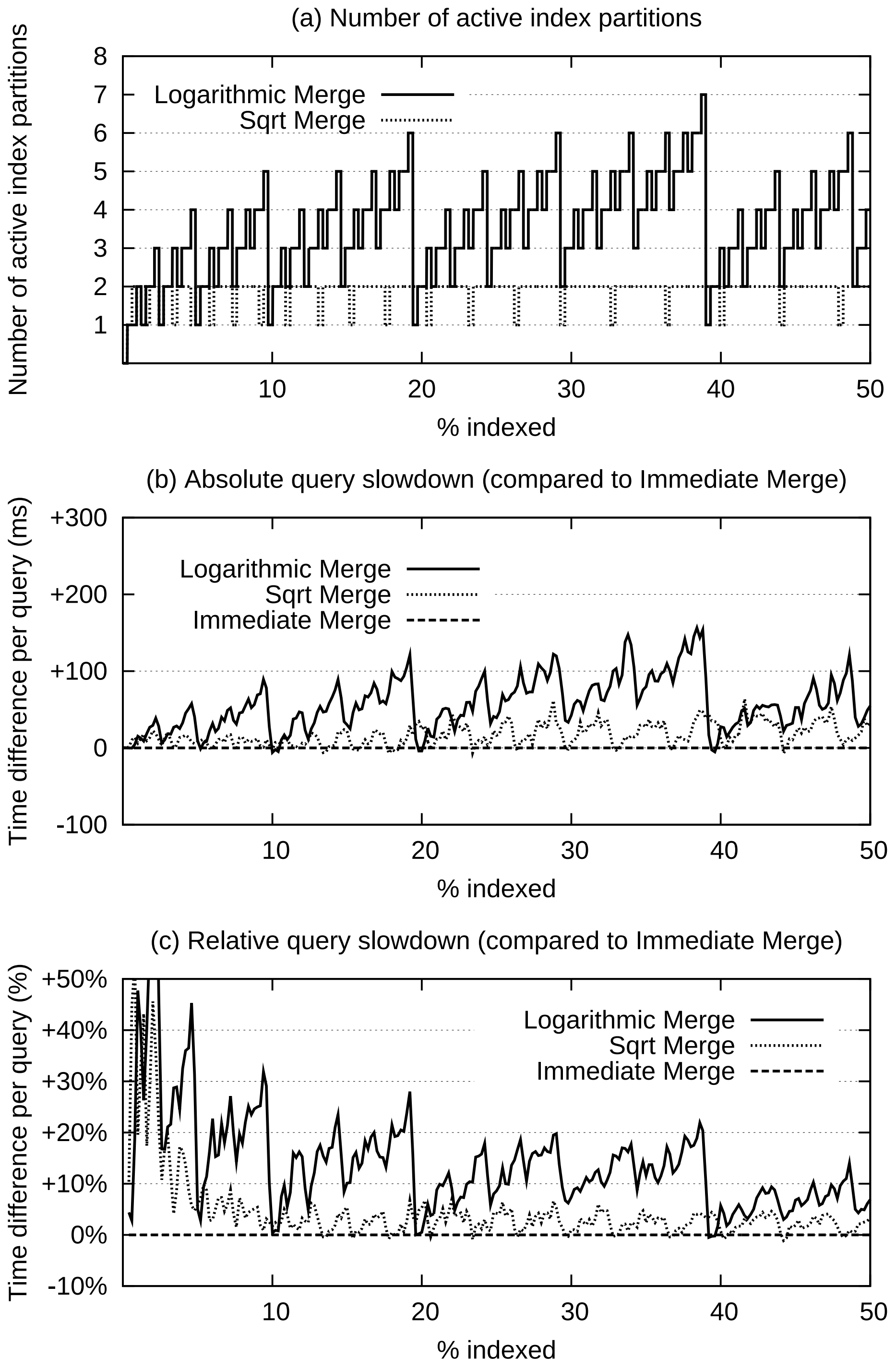


Figure 6.7: Query performance of IMMEDIATE MERGE, SQRT MERGE, and LOGARITHMIC MERGE for a growing text collection (GOV2). The relative performance (plots (b) and (c)) depends on the number of active index partitions maintained by the search engine (plot (a)).

	32 MB	64 MB	128 MB	256 MB	512 MB	1024 MB
Immediate Merge				158.2	77.9	38.6
Sqrt Merge				21.0	15.3	12.4
Logarithmic Merge	9.8	9.4	8.9	8.6	8.5	8.0
No Merge		5.3	5.3	5.1	5.1	5.1

Table 6.2: The impact of memory resources on update performance, for various update strategies. Total indexing time when building a schema-independent index for GOV2. All numbers in hours.

tenance overhead by 22%, from 8.0 hours to 9.8 hours. The method is therefore especially attractive in environments with scarce memory resources, such as file system search.

6.3 In-Place Update

The main shortcoming of the Immediate Merge update policy described and analyzed above is the necessity to copy the entire on-disk index during every physical index update, leading to the quadratic complexity seen in the experiments. The gravity of this problem can be reduced by maintaining multiple index partitions, as done by Sqrt Merge and Logarithmic Merge. But even with a partitioned index, the search engine copies posting lists back and forth even though the majority of the postings are not affected by the updates to the text collection. Moreover, dividing the on-disk index into multiple partitions decreases query performance, as shown in my experiments.

The typical in-place index update policy stores each on-disk posting list in a contiguous region of the disk [102] [69] [70] (although there are exceptions to this general rule [107]), pre-allocating some empty space at the end of each list. Whenever in-memory index data have to be combined with the on-disk index, the new postings can simply be appended at the end of the respective list, without the need to copy the entire list. Only when the free space at the end of a given list is insufficient to hold the new postings from the in-memory index, the entire list is relocated to a new position, pre-allocating again some free space at the end of its new location. The consensus regarding possible pre-allocation strategies is that best performance results are obtained when following a proportional overallocation strategy [103] [66] or by using historical information about the growth of each posting list to make a pre-allocation decision based on its predicted growth in the future [102].

The advantage of the basic in-place update strategy with proportional pre-allocation is that it leads to a linear number of postings transferred to/from disk (asymptotically optimal). Its main limitation is its inability to deal with large numbers of very short lists. Each list relocation requires a disk seek. Thus, if many lists need to be moved around during a physical index update, the in-place strategy, despite the small amount of data transferred from/to disk, will be very inefficient.

Lester et al. [70], therefore, recommend giving special treatment to very short posting lists. Instead of storing them in the postings file, Lester's implementation keeps short lists (up to 512 bytes) directly in the search engine's dictionary (realized by a B^+-tree). This method, although it increases update performance, has the disadvantage that it is likely to inflate the dictionary so much that it does not fit into main memory any more. On the other hand, the same may happen to the dictionary for any text collection of realistic size (cf. Section 4.4).

Surprisingly, even with all such optimizations integrated, in-place update is still outperformed by the comparatively simple IMMEDIATE MERGE strategy, except when the search engine's in-memory buffers are very small. Lester [66, Chapter 4] attributes this to the large number of random disk accesses required by in-place index update and the fact that the time it takes to perform a disk seek is approximately linear in the distance travelled by the disk's read/write head. Thus, as the index becomes larger and larger, the time spent on disk seeks (necessary to conduct list relocations), increases approximately linearly with the size of the index.

6.3.1 File-System-Based Update

Despite its limitations, in-place index maintenance – from a developer's point of view – represents an interesting alternative to merge update. For example, if it were possible to store every inverted list in a separate file in the file system, realizing list updates through simple file append operations, then this would dramatically simplify the design and implementation of the search engine.

Hans Reiser, the main developer of the ReiserFS[1] and Reiser4[2] file systems, and well known for his bold statements, motivates this approach in his own, distinctive words:

> *When filesystems aren't really designed for the needs of the storage layers above them, and none of them are, not Microsoft's, not anybody's, then layering results in enormous performance loss. The very existence of a storage layer above the filesystem means that the filesystem team at an OS vendor failed to listen to someone, and that someone was forced to go and implement something on their own.* [3]

[1]http://www.namesys.com/X0reiserfs.html (accessed 2007-03-04)
[2]http://www.namesys.com/v4/v4.html (accessed 2007-03-04)
[3]http://www.namesys.com/whitepaper.html (accessed 2007-03-04)

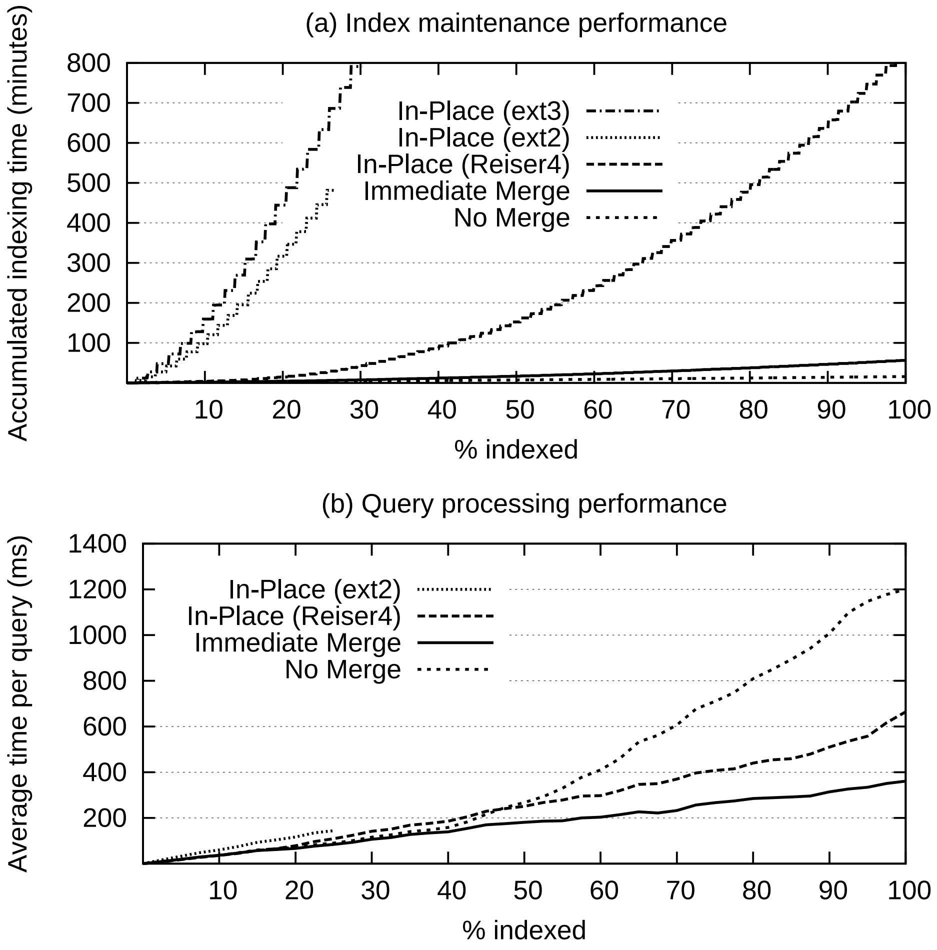


Figure 6.8: Index maintenance and query processing performance for TREC, measured as the collection grows (100% $\widehat{=}$ 1.5 million documents, 824 million tokens). File-system-based update is not competitive with merge-based update.

As crazy as it may sound, this suggestion is not entirely absurd. Pre-allocation strategies, for example, similar to those found in in-place index update, are standard components of file system implementations. Giving special treatment to infrequent terms with very short lists, as suggested by Lester et al. [70], is done in a similar way by Reiser4, which stores very small files directly in the directory tree instead of allocating a full disk block for their storage.

I repeated the first experiment from Section 6.2 (cf. Figure 6.1), this time using the file system implementation as the exclusive storage layer. All index data are stored in a single directory, containing one file for each posting list. The name of each file is equal to the term whose posting list it contains. Hence, the search engine does not need to maintain an explicit dictionary, but can rely on the lookup functionality provided by the file system. The experiment was repeated three times, using a different file system implementation each time: `ext2`, `ext3`, and Reiser4. Figure 6.8 shows the results obtained under the new circumstances.

At first sight, the outcome is not surprising. In terms of update performance, Reiser4 is about 15 times slower than Immediate Merge, `ext2` is about 15 times slower than Reiser4, and `ext3` is even slower than `ext2`, due to the additional journal activity. On the query side, the situation is no different. Reiser4 is clearly outperformed by Immediate Merge, and `ext2` is even slower than No Merge. `ext3`, not shown here, leads to approximately the same query performance as `ext2`. Obviously, the file system appears to be struggling with the large number of files that need to be maintained and periodically updated. It is unable to keep file fragmentation and lookup overhead at a low level.

However, it has to be pointed out that for small text collections, containing only a couple hundred million tokens, a file-system-based index implementation might actually be a feasible solution. After indexing the first 20% of the TREC collection (165 million tokens), for instance, the time difference between Reiser4 and Immediate Merge is only 10 minutes (14.6 minutes vs. 4.2 minutes; +246%). Thus, all in all, Reiser4 performs remarkably well, especially compared to other file system implementations, and might represent a fairly efficient index storage layer for small file systems. It just does not scale quite as well as custom index implementations.

6.3.2 Partial Flushing

One of the two main reasons for buffering postings in memory before applying them to the on-disk index, as a batch update operation, is that accumulating index data in memory decreases the ratio

$$\frac{\#\text{lists to be updated}}{\#\text{postings to be added}}$$

(cf. Section 6.1). Since the disk seeks performed during such a batch index update represent the main performance limitation of in-place update, it makes sense to think about techniques that can decrease this ratio without increasing the memory consumption of the search engine.

One such technique, *partial flushing*, is motivated by the Zipfian term distribution found in most natural-language text corpora. The main idea is that, while the majority of all posting lists are very short, the majority of all postings is found in long lists. For example, of all 2.1 million posting lists in an index for the TREC collection, 2.0 million (95.5%) contain less than 100 elements each. The total number of postings found in "long" lists (containing at least 100 postings), on the other hand, is 814 million (98.8%). Hence, if the main objective

Algorithm 7 In-place update with partial flushing. Building a schema-independent index for a potentially infinite text collection. Input parameters: $\vartheta \geq 1$, the long-list threshold; $\omega \in (0, 1)$, the efficiency criterion, used to determine when partial flushing is no longer beneficial.

```
 1: $I_M \leftarrow \emptyset$  // initialize in-memory index
 2: $I_D \leftarrow \emptyset$  // initialize on-disk index
 3: $currentPosting \leftarrow 1$
 4: $\omega' \leftarrow 1.0$  // assume the last partial flush reduced memory consumption by 100%
 5: while there are more tokens to index do
 6:   $T \leftarrow$ next token
 7:   $I_M.addPosting(T,\ currentPosting)$
 8:   $currentPosting \leftarrow currentPosting + 1$
 9:   if $I_M$ contains more than $M - 1$ postings then
10:     for each term $T$ in $I_M$ and its posting list $L_T$ do
11:       if $(L_T.length \geq \vartheta)$ or $(\omega' < \omega)$ then
12:         $I_D.addPostings(T,\ L_T)$  // add long list $L_T$ to on-disk index $I_D$
13:         remove $T$ and $L_T$ from $I_M$
14:       end if
15:     end for
16:     $\omega' \leftarrow 1 - \frac{\#\text{ of postings in } I_M}{M}$
17:   end if
18: end while
```

is to reduce the total number of list updates performed, then it makes sense to not transfer *all* in-memory posting lists to disk when the indexing process runs out of memory, but only *some* of them, namely all lists containing more than a certain number ϑ of postings (for some reasonable ϑ).

The effectiveness of this method depends on the characteristics of the text collection (Zipf parameter α — greater α means more strongly skewed distribution and greater savings), the "long list" threshold ϑ, and the total size of the search engine's memory buffers. When indexing the TREC collection ($\alpha \approx 1.22$) with 32 MB for in-memory posting buffers and a partial flushing threshold $\vartheta = 256$, the first application of partial flushing, taking place after indexing 10.6 million tokens, reduces the search engine's memory consumption by 59%. The second application reduces the memory consumption by 47%, the third by 38%, and so forth.

The savings achieved by each subsequent partial flush are smaller than those attained by the previous one, since the relative number of postings in short lists (containing fewer than ϑ postings each) increases. Moreover, the memory savings realized by flushing long lists is not quite as large as indicated by the Zipfian term distribution, mainly because of the memory occupied by the hash-based in-memory dictionary needed for building the in-memory index. Nonetheless, partial flushing can be used to reduce the total number of on-disk list updates.

Long list threshold	1	16	64	256	1024	4096	16384
Total indexing time (minutes)	817	1290	830	504	466	479	578
Total number of list updates ($\times 10^6$)	11.8	9.5	8.6	8.5	9.3	9.9	10.6
Postings per list update (average)	69.8	86.9	96.1	96.9	88.7	83.3	77.9

Table 6.3: Indexing TREC with in-place index maintenance (Reiser4) and 32 MB for in-memory buffers. Using a long-list threshold $\vartheta = 1024$, partial flushing decreases the number of on-disk list updates by 21% and the total index maintenance time by 43%.

Because the gains caused by partial flushing diminish at each subsequent application of the partial flushing technique, it makes sense to impose a limit on the number of partial flushings that may be performed in a row. In my implementation, this is realized by defining an efficiency criterion ω. If the relative memory savings achieved by the previous application of partial flushing are less than ω, then the following index update cycle will transfer the entire in-memory index to disk, not just the postings found in long lists. This procedure is formalized in Algorithm 7.

With $\vartheta = 256$, $\omega = 0.15$, and a memory limit of 32 MB, 5 partial flush operations on average are performed between two consecutive complete update operations.

Experiments

I repeated the experiment from before (Figure 6.1), this time employing the partial flushing strategy to reduce the total number of on-disk list updates in the in-place update policy realized through Reiser4. The partial flushing efficiency criterion was set to $\omega = 0.15$.

Table 6.3 shows that, with $\vartheta = 256$, the total number of list updates can be reduced by 28%, from 11.8 million to 8.5 million. As a result, the total indexing time is decreased by 38%, from 817 minutes to 504 minutes. Surprisingly, partial flushing does not always help. If the threshold ϑ is too small (≤ 64), indexing performance is worse than without partial flushing. This may be explained by the fact that Reiser4 stores short lists directly in the directory tree and not in separate disk blocks. Choosing ϑ too small essentially forces the file system to read most parts of the directory tree (in a sequential manner, similar to a re-merge operation), making it difficult to attain any savings by skipping parts of the directory structure in a physical index update cycle.

What is also interesting is that too small a threshold ϑ does not even reduce the number of list updates to be performed. For example, $\vartheta = 16$ leads to a greater number of list updates than $\vartheta = 256$. While partial flushing reduces the list update activity for short lists, it increases the number of list updates performed on long lists. If ϑ is chosen too small, this

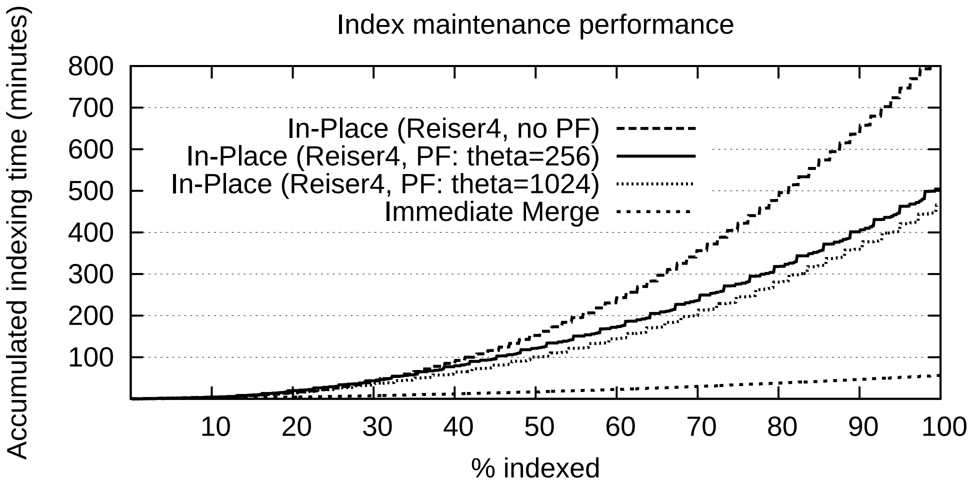


Figure 6.9: In-place update with partial flushing, employing the file system as the exclusive storage layer. When building an index for the TREC collection, a wide range of threshold values ϑ lead to improved index maintenance performance compared to the baseline (Reiser4, no partial flushing).

may lead to an overall increase of the number of list updates carried out (for $\vartheta = 16$, the number of list update operations is increased by 12% compared to $\vartheta = 256$).

Figure 6.9 shows the indexing performance of partial flushing for $\vartheta = 256$ and $\vartheta = 1024$ in comparison to pure in-place update and the Immediate Merge strategy. It emphasizes that, while partial flushing can lead to substantial savings over the original in-place strategy, its performance is still not even close to that achieved by Immediate Merge.

Although it is unclear what exact effect partial flushing would have on highly optimized implementations of in-place update (e.g., [102], [70]), it is highly unlikely that such implementations would not benefit from it at all. This claim is further supported by the results obtained for contiguous hybrid index maintenance in Section 6.4.

6.3.3 In-Place vs. Merge-Based Update

The question whether in-place or merge-based index update strategies lead to the best performance in dynamic text retrieval systems is vividly debated in the literature. Theoretically, in-place update with proportional pre-allocation exhibits an asymptotically linear disk complexity, because each posting – on average – is relocated at most a constant number of times. In practice, unfortunately, the large number of disk seeks triggered by list relocations transforms the linear complexity into a quadratic one, since each disk seek, on average, requires time linear in the size of the index [66].

Lester et al. [69] [70] show that, despite this limitation, in-place update may lead to better indexing performance than Immediate Merge — if the memory resources of the search engine are very limited and on-disk index updates have to be carried out frequently. However, this line of argumentation does not take into account that, under these circumstances, the search engine's dictionary will most likely be too large to fit completely into main memory.

With a merge-maintained index, the memory consumption of the search engine's dictionary is very low, thanks to the dictionary interleaving technique described in Section 4.4.2. If the system uses an in-place-updated index instead, the interleaving technique cannot be employed any more. Posting lists are relocated continually, and there is no predefined, global ordering on the terms in the index that can be utilized to find the index position of a term's posting list. The search engine, therefore, needs to maintain an explicit dictionary, containing the exact on-disk position of every posting list.

If memory resources are scarce, then this constitutes a significant disadvantage of in-place update. The search engine needs to store an explicit dictionary on disk and thus, at query time, requires an additional disk seek per query term. It is then no longer appropriate to compare in-place update to Immediate Merge. Instead, it must be compared to Sqrt Merge, the variant of *geometric partitioning* [68] in which the search engine maintains two independent index partitions and thus, at query time, performs two random disk accesses per query term.

Unfortunately, in-place update is not even close to being competitive with Sqrt Merge. I therefore conclude that, in most cases, in-place update is not an interesting option. It should only be used if the number of unique terms in the index is very small.

6.4 Hybrid Index Maintenance

The main shortcoming of in-place update is its inability to deal with large numbers of posting lists. The main shortcoming of merge-based index maintenance, on the other hand, is its need to copy large portions of existing posting lists even though they have not changed. It is possible to combine the advantages of both approaches into a family of update strategies referred to as *hybrid index maintenance.*

Hybrid index maintenance divides the on-disk index into two sections. The first section is updated according to a merge strategy and contains *short* posting lists (i.e., lists containing fewer than ϑ postings, for some threshold ϑ). The second section is updated using an in-place strategy; it contains *long* lists (i.e., lists containing ϑ postings or more).

The idea to distinguish between frequent terms and infrequent terms is not new:

- Cutting and Pedersen [35] conducted experiments with a merge update strategy based on B$^+$-trees in which they move posting lists into a secondary index once they become too long to be stored in a single block of the B$^+$-tree (a technique referred to as *pulsing*). However, their motivation was primarily ease of implementation, and not update performance.
- Shoens et al. [103] [107] investigated dividing the set of posting lists in the index into short lists and long lists. Short lists are stored in fixed-size buckets, with multiple lists sharing the same bucket, while long lists are updated according to an in-place strategy with pre-allocation. However, in their method, the marking of a list as *long* is triggered by a bucket overflow, which is only loosely related to the length of the given list.

To my knowledge, despite the existence of this previous work, the idea has never been fully expanded. Consequently, its performance has never been properly analyzed, in particular not on modern hardware (recall from Section 2.5.1 that hard drive characteristics have changed considerably over the last 15 years). Moreover, the method has never been used in conjunction with the novel index partitioning schemes developed in the past few years (cf. Section 6.2.1).

Based on whether they store each on-disk posting list in a contiguous or in a non-contiguous fashion, hybrid index maintenance strategies can be divided into two categories. These are examined in the following sections.

6.4.1 Hybrid Index Maintenance with Contiguous Posting Lists

Consider the Immediate Merge update strategy and in-place update with proportional pre-allocation. Both strategies, keeping on-disk posting lists in contiguous regions of the hard drive, lead to the same query performance (assuming that the dictionary is kept in memory or interleaved with posting lists). However, Immediate Merge, due to its sequential disk access pattern, exhibits better update performance.

Let us compare the update performance of the two competing strategies on a list-by-list basis. Suppose the hard drive provides a read/write bandwidth of b bytes/second and can carry out a random disk access in r seconds (worst-case). If the search engine, during an on-disk update operation, encounters a posting list that is bigger than $b \cdot r$ bytes, then it will achieve better performance by updating this list in-place than by updating it according to Immediate Merge.

Why is that? Suppose that the list in question has total size of s_i and s_{i+1} bytes after the completion of the i-th and $(i+1)$-th update cycle, respectively. If it is updated by merging the existing list with the incoming postings, then the corresponding index maintenance overhead caused by the search engine's disk activity during the $(i+1)$-th re-merge update is:

$$D_{i+1} = \frac{s_i + s_{i+1}}{b}, \tag{6.8}$$

where s_i corresponds to reading the old list from disk, and s_{i+1} corresponds to writing the new list into the inverted file that is the result of the re-merge operation.

If, on the other hand, the list is updated in-place, the disk overhead during the $(i+1)$-th update cycle is:

$$D_{i+1} = 2 \cdot r + \frac{s_{i+1} - s_i}{b}, \tag{6.9}$$

where the term $2 \cdot r$ stems from the necessity to seek to the on-disk position of the list and also to seek back to the original location. The second seek is not always necessary in pure in-place index maintenance, but is required when combining re-merge and in-place into a hybrid strategy.

The above analysis ignores some details of in-place update. For instance, it does not take into account the possibility that the list might need to be relocated during the update. Nonetheless, it does indicate that there is a point at which, for an individual list L, it becomes more efficient to update L in-place than to include it in a global re-merge operation. According to equations 6.8 and 6.9, this point is reached when $s_i > b \cdot r$. For a typical consumer hard drive ($r \approx 10$ ms, $b \approx 50$ MB/s), this happens for a list size of approximately 0.5 MB ($\approx$ 250,000 postings).

This line of argumentation immediately leads to the definition of *Hybrid Immediate Merge with contiguous posting lists* (HIM$_\text{C}$). The HIM$_\text{C}$ strategy starts off exactly like its non-hybrid counterpart Immediate Merge. However, whenever – during a re-merge update – it encounters a list L containing more than ϑ postings, for some predefined threshold ϑ, the list is marked as *long* and moved from the merge-maintained index into the in-place index. After that, the search engine always updates L in-place.

Every on-disk posting list is stored in a contiguous region of the hard disk: short lists reside entirely within the merge-updated index, while long lists exclusively reside in the in-place index — hence the name of the method. A more formal definition of HIM$_\text{C}$ is given by Algorithm 8. In the algorithm, the set of *long* lists is kept track of by the variable $\mathcal{L}$, updated whenever a given list exceeds the threshold ϑ (lines 14–17).

Partial Flushing

Since HIM$_\text{C}$ utilizes an in-place index to maintain long on-disk posting lists, the partial flushing technique from Section 6.3.2 can be applied here as well. In addition to the long list

Algorithm 8 On-line index construction according to Hybrid Immediate Merge with contiguous posting lists (HIM$_\mathrm{C}$). Input parameter: long list threshold ϑ.

```
1: let $I_{merge}$ denote the primary on-disk index (merge-maintained, initially empty)
2: let $I_{inplace}$ denote the secondary on-disk index (updated in-place, initially empty)
3: $I_{mem} \leftarrow \emptyset$  // initialize in-memory index
4: $currentPosting \leftarrow 1$
5: $\mathcal{L} \leftarrow \emptyset$  // the set of long lists is empty initially
6: while there are more tokens to index do
7:    $T \leftarrow$ next token
8:    $I_{mem}.addPosting(T,\ currentPosting)$
9:    $currentPosting \leftarrow currentPosting + 1$
10:   if $I_{mem}$ contains more than $M - 1$ postings then
11:      // merge $I_{mem}$ and $I_{merge}$ by traversing their lists in lexicographical order
12:      $I'_{merge} \leftarrow \emptyset$  // create new on-disk index
13:      for each term $T \in (I_{mem} \cup I_{merge})$ do
14:         if $(T \in \mathcal{L}) \vee (I_{mem}.getPostings(T).length + I_{merge}.getPostings(T).length > \vartheta)$
            then
15:            $I_{inplace}.addPostings(T,\ I_{merge}.getPostings(T))$
16:            $I_{inplace}.addPostings(T,\ I_{mem}.getPostings(T))$
17:            $\mathcal{L} \leftarrow \mathcal{L} \cup \{T\}$
18:         else
19:            $I'_{merge}.addPostings(T,\ I_{merge}.getPostings(T))$
20:            $I'_{merge}.addPostings(T,\ I_{mem}.getPostings(T))$
21:         end if
22:      end for
23:      $I_{merge} \leftarrow I'_{merge}$  // replace old on-disk index with new one
24:      $I_{mem} \leftarrow \emptyset$  // re-initialize the in-memory index
25:   end if
26: end while
```

threshold ϑ, a partial flushing threshold ϑ_{PF} needs to be chosen. When the indexing process runs out of memory, the lists for all terms with more than ϑ postings in total and more than ϑ_{PF} postings in the in-memory index are transferred to disk. All other lists remain in memory.

As before (cf. Section 6.3.2), the procedure is repeated until the relative memory savings achieved by a partial flush sink under a cut-off threshold ω, at which point the entire in-memory index, including all short lists, is written to disk.

Limitations of Contiguous Hybrid Index Maintenance

While the hybridization of Immediate Merge is straightforward, as demonstrated above, hybridizing the index partitioning schemes from Section 6.2.1 turns out to be more difficult. Equation 6.8 does not hold when the search engine maintains more than one index partition, because a merge operation then no longer necessarily involves all postings for a given term.

With Logarithmic Merge, for example, the total number of postings transferred to/from disk for a particular posting list L is:

$$D_L \in \Theta\left(\log\left(\frac{N}{M}\right) \cdot L.\mathit{length}\right) \tag{6.10}$$

(where N is the size of the collection, and M is the size of the in-memory update buffers). The expected number of postings read/written, for that list, during a single merge operation, then is:

$$E[D_L] \in \Theta\left(\frac{M}{N} \cdot \log\left(\frac{N}{M}\right) \cdot L.\mathit{length}\right), \tag{6.11}$$

since the total number of merge operations performed is $\lfloor N/M \rfloor$.

In contrast to Equation 6.8, the disk overhead caused by the list L suddenly depends on two additional parameters: the total size of the text collection, and the amount of main memory available to the search engine. Therefore, it is no longer possible to optimize update performance by choosing a fixed threshold ϑ. However, even with Logarithmic Merge, it is still the case that the search engine, when it encounters a list *fragment* containing more than ϑ postings, should move this fragment from the merge-maintained to the in-place-updated part of the on-disk index. But a firm distinction between short lists and long lists is not possible any more. This insight leads to the second variant of hybrid index maintenance.

6.4.2 Hybrid Index Maintenance with Non-Contiguous Posting Lists

Hybrid index maintenance with non-contiguous posting lists allows a given on-disk posting list to consist of multiple parts: one part resides in the in-place section of the index; one or more parts (depending on whether the base strategy is Immediate Merge or one of the index partitioning schemes) reside in the merge-maintained index section.

Consider again the non-hybrid update strategy Immediate Merge. Suppose that, after k re-merge operations, a posting list L reaches the point where it makes sense to transfer it to the in-place index ($L.\mathit{length} > \vartheta$), because seeking to the on-disk position of the list and appending new postings can be done more quickly than reading and writing all $L.\mathit{length}$ postings in every update cycle. Now, in the $(k + 1)$-th re-merge operation, we can expect approximately $\frac{\vartheta}{k}$ new postings for the given term. Contiguous hybrid index maintenance would transfer these new postings into the in-place index. However, k might be very large, and $\frac{\vartheta}{k}$ thus might be rather small. Consequently, the random disk access required to update the list in-place may easily take longer than just copying the $\frac{\vartheta}{k}$ new postings as part of a

Algorithm 9 On-line index construction according to HYBRID LOGARITHMIC MERGE with non-contiguous posting lists (HLM_{NC}). Input parameter: long list threshold ϑ.

```
1: let $I_g$ denote the on-disk index of generation $g$
2: let $I_{inplace}$ denote the search engine's in-place index (initially empty)
3: $I_0 \leftarrow \emptyset$ // initialize in-memory index
4: $currentPosting \leftarrow 1$
5: $\mathcal{L} \leftarrow \emptyset$ // the set of long lists is empty initially
6: while there are more tokens to index do
7:    $T \leftarrow$ next token
8:    $I_0.addPosting(T,\ currentPosting)$
9:    $currentPosting \leftarrow currentPosting + 1$
10:   if $I_0$ contains more than $M - 1$ postings then
11:      $\mathcal{I} \leftarrow \{I_0\}$ ; $g \leftarrow 1$
12:      while $I_g$ is not empty do
13:         $\mathcal{I} \leftarrow \mathcal{I} \cup \{I_g\}$ ; $g \leftarrow g + 1$
14:      end while
15:      $I_g \leftarrow \emptyset$ // create primary target index of merge operation
16:      for each term $T$ in $\mathcal{I}$ do
17:         $targetIndex \leftarrow I_g$
18:         if $\exists\ x : (T, x) \in \mathcal{L}$ then
19:            if $(\mathcal{I}.getPostings(T).length > \vartheta)\ \wedge\ (x \leq g)$ then
20:               $targetIndex \leftarrow I_{inplace}$
21:               $\mathcal{L}\ \leftarrow\ (\mathcal{L} \setminus \{(T, x)\})\ \cup\ \{(T, 0)\}$
22:            else
23:               $\mathcal{L}\ \leftarrow\ (\mathcal{L} \setminus \{(T, x)\})\ \cup\ \{(T, \max\{x, g\})\}$
24:            end if
25:         else
26:            if $(\mathcal{I}.getPostings(T).length > \vartheta)\ \wedge\ (\text{argmax}_k\{I_k \text{ is not empty}\} \leq g)$ then
27:               $targetIndex \leftarrow I_{inplace}$
28:               $\mathcal{L}\ \leftarrow\ \mathcal{L}\ \cup\ \{(T, 0)\}$ // mark $T$ as having postings in $I_{inplace}$
29:            end if
30:         end if
31:         $targetIndex.addPostings(T,\ \mathcal{I}.getPostings(T))$
32:      end for
33:      delete every index $I\ \in\ \mathcal{I}$
34:      $I_0 \leftarrow \emptyset$ // re-initialize in-memory index
35:   end if
36: end while
```

sequential re-merge operation.

Non-contiguous hybrid index maintenance starts like an ordinary non-hybrid update strategy. Like the contiguous variant, whenever it encounters a posting list that is longer than a predefined threshold ϑ, it transfers this list from the merge-maintained section of the index into the in-place section. However, it differs from the contiguous version in how it treats such a list after the transfer. In the non-contiguous version, there is no clear distinction between terms with long lists and terms with short lists. Instead, the decision whether a set of postings from a given posting list should be transferred to the in-place index depends entirely on the length of the list fragment participating in the current merge operation. If less than ϑ postings from a "long" list participate in a merge event, then these postings will not be transferred to the in-place index, but will remain in the merge-maintained index section instead.

The method can be combined with any merge-based update strategy, including various index partitioning schemes. Algorithm 9, for example, shows the hybridization of the Logarithmic Merge update strategy.

Implementation Details

The details of Algorithm 9 might not be immediately perspicuous. In contrast to Algorithm 8, the set $\mathcal{L}$, keeping track of all terms with postings in the in-place index, now contains elements of the form *(string, integer)*, and not simple strings any more.

For each term T with postings in the in-place index section, there is an element in $\mathcal{L}$, indicating the identity of the biggest inverted file in the merge-maintained index section that contains postings for T. If there is an index partition that is not involved in the current merge operation and that contains any of T's postings, then we *must not* move the postings to the in-place index (checked in line 19 of the algorithm). Similarly, for a term that does not have any postings in the in-place section, the algorithm only transfers its postings if *all* merge-maintained index partitions are involved in the merge operation (line 26).

This restriction is necessary because it is absolutely crucial that the postings in every posting list are stored in increasing order within each index partition, including the in-place section of the index. If the restriction $x \leq g$ were not enforced in line 19, then it could happen that, for some term T, a set of postings $\{p_1, \ldots, p_n\}$ is added to the in-place index and, in a later merge operation, another set of postings $\{p'_1, \ldots, p'_{n'}\}$ is added for the same term — with $p'_1 < p_n$. This would violate the ordering constraint for that term. In Section 6.6, which is concerned with file modifications, I discuss the impact that non-ordered posting lists can have on the search engine's update and query performance.

Variations

Non-contiguous hybrid index maintenance increases the amount of list fragmentation by splitting a posting list into a short part and a long part. It is possible to go a step further and to loosen the contiguity constraint on posting list fragments in the in-place section. Instead of storing the long part of a posting list in a contiguous fashion, the in-place index might be allowed to store this part itself in several, non-contiguous chunks. This procedure can be expected to improve update performance, but is likely to harm query efficiency.

In my experiments, I use two different variants of non-contiguous hybrid index maintenance. The first variant, denoted as X_{NC} (e.g., HLM_{NC}), enforces the contiguity of all list fragments in the in-place section. The second variant, denoted as $X_{\text{NC(A)}}$ (e.g., $\text{HLM}_{\text{NC(A)}}$), does not enforce their contiguity. Instead, whenever new postings are transferred to the in-place index section, they are simply appended at the end of the existing index, improving update performance, but increasing list fragmentation (the subscript (A) stands for *append*).

Expectations

Non-contiguous hybrid index maintenance can be expected to lead to better update performance than the contiguous variant. However, because it sometimes increases the degree of fragmentation found in the on-disk posting lists, it may lead to lower query performance. Compared to the basic Immediate Merge policy, for many terms the search engine maintains two on-disk list fragments (instead of one) if it uses HIM_{NC} to realize index updates.

However, this is not the case for all instantiations of this method. Suppose the search engine employs HLM_{NC} to maintain its on-disk index. Now consider a term T, and suppose T, on average, appears 300,000 times in each indexing buffer load, while the system's long list threshold is set to $\vartheta = 10^6$. Then T is highly unlikely to ever make it into an on-disk index partition of generation 3 (or higher), since, on average, such a partition would contain $4 \times 300{,}000 = 1.2$ million postings for T. List fragmentation for this term, therefore, will be lower with HLM_{NC} than with the non-hybrid Logarithmic Merge strategy.

6.4.3 Implementing the In-Place Index

Both variants of hybrid index maintenance, contiguous and non-contiguous, rely on the availability of an efficient in-place update implementation. This seems to indicate that hybrid index maintenance suffers from the same limitations as a plain in-place update strategy. However, this is not so. The main performance bottlenecks in in-place update all stem from the large number of index terms that a realistic search engine has to deal with — 30 million in the case of SBFS, and almost 50 million for GOV2. In hybrid index maintenance, this is no longer a problem. Even for the comparatively large GOV2 collection, a rather small long-list threshold of $\vartheta = 50{,}000$ only leads to 20,114 long lists (after the whole collection has been indexed). This greatly reduces the number of random disk accesses that need to be performed during an in-place update cycle. It also simplifies dictionary data structures and free space management.

In the implementation used in my experiments, the dictionary is realized through an uncompressed in-memory hash table. Disk space is allocated in large chunks (multiples of 1 MB, assigned according to proportional pre-allocation with $k = 2$), and free space is kept track of by means of a simple array, supporting search operations for free index regions through a straightforward linear scan.

6.4.4 Complexity Analysis

I now analyze the theoretical complexity of hybrid index maintenance. More specifically, I determine the number of disk write operations (measured by the number of postings written to disk) necessary to index a text collection of a given size. Since the number of read operations carried out during merge operations is bounded from above by the number of write operations (nothing is read twice), this allows me to calculate the total number of disk operations performed during index update activity (up to a constant factor).

I analyze the update complexity of HIM_{NC} and HLM_{NC}, showing that HLM_{NC} requires only $\Theta(N)$ disk operations to construct an index for a text collection of size N. Asymptotically, this is the optimal indexing performance and is only rivalled by the No Merge strategy used in off-line index construction (which provides uncompetitive query performance if used in an on-line environment). The analysis is based on the assumption that the term distribution can be expressed as a generalized Zipf distribution [117], i.e., that the number of times the i-th most frequent word appears in the text collection is inversely proportional to i^α (for some α). I will use the notation established in Section 4.2.3.

The results obtained in this section are not applicable to the contiguous version of hybrid index maintenance (e.g., HIM_{C}), because the total disk activity in that case is dominated by disk seeks, not by write operations. For non-contiguous hybrid index maintenance, however, because the search engine transfers approximately $\frac{\vartheta}{2}$ postings per disk seek, counting the number of postings written to disk can be expected to yield an accurate approximation of the true update performance.

An Analysis of HIM_{NC}

Consider the Hybrid Immediate Merge strategy with non-contiguous posting lists (HIM_{NC}). When building an index for a collection containing N tokens, HIM_{NC} performs a total of $\lfloor \frac{N}{M} \rfloor$ merge operations, where M is the number of postings that may be buffered in main memory. In every such merge operation, HIM_{NC} transfers all long posting lists (containing at least ϑ postings) it encounters to the in-place section of the on-disk index. For the merge-updated part of the on-disk index, which is the result of this merge operation, this means that it does not contain any lists that are longer than ϑ postings. Now, consider the merge-updated index section after the k-th merge operation, i.e., after $k \cdot M$ tokens have been indexed. This inverted file contains two types of lists:

1. genuine short lists, each having $\leq \vartheta$ postings;
2. short parts of long lists, each having $\leq \vartheta$ postings.

The latter happens if, for instance, the posting list for a term was moved from the merge-maintained to the in-place-maintained part of the index in the $(k-1)$-th re-merge operation, and fewer than ϑ postings have been accumulated for that term between the $(k-1)$-th and the k-th merge operation.

It is easy to give a lower bound for the number of postings $\hat{P}$ stored in the merge-maintained section of the index after the k-th index update cycle (using the notation from Section 4.2.3):

$$\begin{aligned} \hat{P}(k \cdot M, \vartheta) &\geq \hat{S}(k \cdot M, \vartheta) && (6.12) \\ &\in \Omega\left(\vartheta \cdot \left(\frac{k \cdot M}{\vartheta}\right)^{1/\alpha}\right) && (6.13) \end{aligned}$$

(counting only lists of type 1). Similarly, we have the following upper bound:

$$\begin{aligned} \hat{P}(k \cdot M, \vartheta) &\leq \hat{S}(k \cdot M, \vartheta) + \vartheta \cdot L(k \cdot M, \vartheta) && (6.14) \\ &\in \mathcal{O}\left(\vartheta^{1-1/\alpha} \cdot (k \cdot M)^{1/\alpha}\right) + \vartheta \cdot \mathcal{O}\left(\left(\frac{k \cdot M}{\vartheta}\right)^{1/\alpha}\right) && (6.15) \end{aligned}$$

(counting lists of both types, 1 and 2, and upper-bounding lists of type 2 by ϑ postings). It follows that

$$\hat{P}(k \cdot M, \vartheta) \in \Theta\left(\vartheta^{1-1/\alpha} \cdot (k \cdot M)^{1/\alpha}\right). \tag{6.16}$$

Under the chosen complexity model, this is also the number of disk write operations necessary to create the particular index. Since, for a text collection of size N, the number of postings written to the in-place section of the index is

$$\hat{L}(N, \vartheta) \in \Theta(N), \tag{6.17}$$

the total number of disk operations needed to build an inverted index for such a collection is:

$$\begin{aligned} & \Theta(N) \qquad \text{(in-place part)} \\ + \; & \sum_{k=1}^{\lfloor \frac{N}{M} \rfloor} \Theta\left(\vartheta^{(1-1/\alpha)} \cdot (k \cdot M)^{1/\alpha}\right) \qquad \text{(merge part)} \\ = \; & \Theta\left(\vartheta^{(1-1/\alpha)} \cdot \frac{N^{(1+1/\alpha)}}{M}\right). \end{aligned} \tag{6.18}$$

For $\alpha = 1.34$ (GOV2) and constant threshold ϑ, this results in a total index update disk complexity of $\Theta(\frac{N^{1.75}}{M})$, clearly better than the $\Theta(\frac{N^2}{M})$ complexity of non-hybrid Immediate Merge.

An Analysis of HLM$_{\text{NC}}$

Now consider the Hybrid Logarithmic Merge strategy with non-contiguous posting lists (HLM$_{\text{NC}}$). After M postings have been added to the index, two on-disk inverted files are created:

- a merge-maintained inverted file containing $\hat{S}(M, \vartheta)$ postings;
- an in-place-maintained inverted file containing $\hat{L}(M, \vartheta)$ postings.

I refer to the merge-maintained inverted file, created after M tokens have been added, as an index of generation 0.

After the first on-disk inverted file has been created, whenever $2^k \cdot M$ tokens have been added to the collection (for some integer k), HLM$_{\text{NC}}$ merges $k + 1$ inverted indices (k on-disk indices, plus the in-memory index), resulting in a new inverted file of generation k. However, because the search engine is following a hybrid strategy, all long lists ($\geq \vartheta$ postings) encountered during this merge operation are moved into the in-place section of the index instead of the primary target index of the merge operation.

Using the same argument as before (for HIM$_{\text{NC}}$), I can give a Θ-bound for the number of postings $\hat{P}$ in this new merge-maintained inverted file of generation k:

$$\hat{P}(2^k \cdot M, \vartheta) \in \Theta\left(\vartheta^{1-1/\alpha} \cdot (2^k \cdot M)^{1/\alpha}\right). \tag{6.19}$$

Therefore, the total number of disk write operations $D(k)$ performed during merge operations, before and including the creation of this new inverted file of generation k, is:

$$\begin{aligned} D(k) &= \hat{P}(2^k \cdot M, \vartheta) + 2 \cdot D(k-1) && (6.20) \\ &= \sum_{i=0}^{k} 2^{k-i} \cdot \hat{P}(2^i \cdot M, \vartheta) && (6.21) \\ &\in 2^k \cdot \sum_{i=0}^{k} 2^{-i} \cdot \Theta\left(\vartheta^{(1-1/\alpha)} \cdot (2^i \cdot M)^{1/\alpha}\right) && (6.22) \end{aligned}$$

(using equation 6.19). Hence, we have:

$$\begin{aligned} D(k) &\in \Theta\left(\vartheta^{(1-1/\alpha)} \cdot 2^k \cdot M^{1/\alpha} \cdot \sum_{i=0}^{k} (2^{\frac{1}{\alpha}-1})^i\right) && (6.23) \\ &= \Theta\left(\vartheta^{(1-1/\alpha)} \cdot 2^k \cdot M^{1/\alpha}\right) && (6.24) \\ &\subseteq O\left(\vartheta^{(1-1/\alpha)} \cdot 2^k \cdot M\right). && (6.25) \end{aligned}$$

For a text collection of size $N = 2^k \cdot M$, the number of disk operations spent on adding postings to the in-place part of the index is $\hat{L}(N, \vartheta) \in \Theta(N)$. Thus, the total number of disk operations needed to build an inverted index for a text collection of size N is:

$$\begin{aligned} & \Theta(N) && \text{(for the in-place part)} \\ + \; & O\left(\vartheta^{(1-1/\alpha)} \cdot N\right) && \text{(for the merge part).} \end{aligned}$$

Since ϑ is a constant, this means we need $\Theta(N)$ disk operations to build the index. HLM_{NC}'s disk complexity is asymptotically optimal. It is also practical (as opposed to the in-place update strategy, which, too, is asymptotically optimal, but impractical, due to the large number of random disk accesses), because HLM_{NC}'s update overhead is dominated by sequential data transfers, not by disk seeks.

6.4.5 Experiments

To evaluate the actual update and query performance of hybrid index maintenance, I repeated the experiment from before (command sequence shown in Figure 6.5), indexing all files in GOV2, and processing search queries in an interleaved fashion.

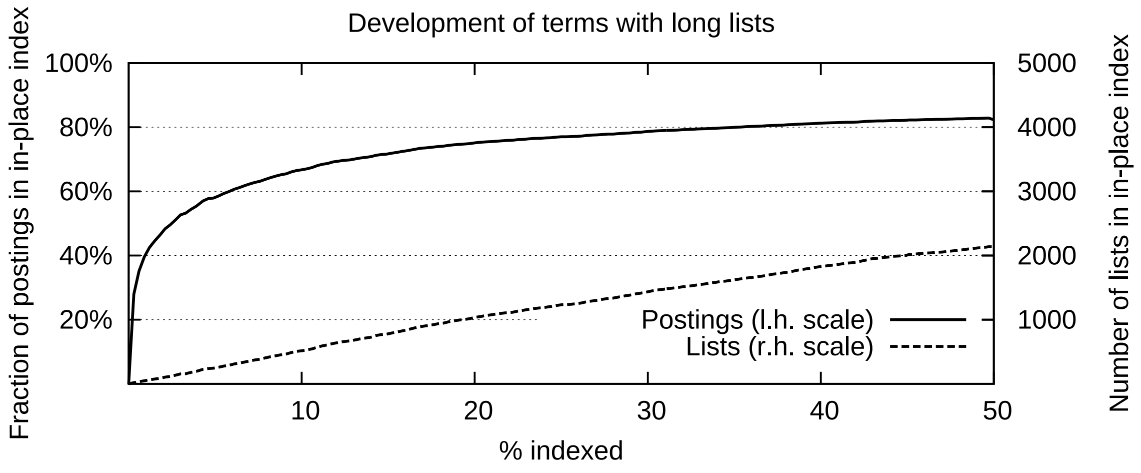


Figure 6.10: Indexing a random 50% of the GOV2 collection with HIM_C. Development over time: number of long lists and number of postings in long lists. Hybrid threshold: $\vartheta = 10^6$. Memory limit: 256 MB.

Contiguous Hybrid Index Maintenance

I evaluated the performance of HIM_C and compared it to its non-hybrid counterpart, Immediate Merge. Due to time constraints (each experiment would have taken more than 3 days to finish), only the first 50% of the total command sequence were processed.

Figure 6.10 shows how the number of terms and the number of postings in the in-place index section develop as the index grows. After 35% of GOV2 have been indexed, more than 80% of all postings are stored in the in-place section. At the same time, however, the total number of terms that are maintained in-place remains comparatively low, just above 2,000 after 50% of the collection have been processed.

The fact that the majority of all postings are maintained according to the Inplace strategy has an impact on the performance of the re-merge operations carried out in the merge-maintained index section. Figure 6.11(a) shows the cumulative index maintenance overhead for Immediate Merge and HIM_C (with and without partial flushing). With a threshold value $\vartheta = 10^6$, HIM_C reduces the maintenance overhead by 46% (compared to the non-hybrid baseline), from 39.7 hours to 21.3 hours. Applying partial flushing with a PF threshold $\vartheta_{\text{PF}} = 2^{16}$ results in a further reduction, down to 16.5 hours (-23%).

Plot 6.11(b) compares HIM_C's query performance with that of the non-hybrid update strategy Immediate Merge. Both strategies lead to approximately the same query performance, which could be expected, given that both maintain all on-disk lists in contiguous regions of the hard disk. The fact that HIM_C is able to process queries slightly faster than

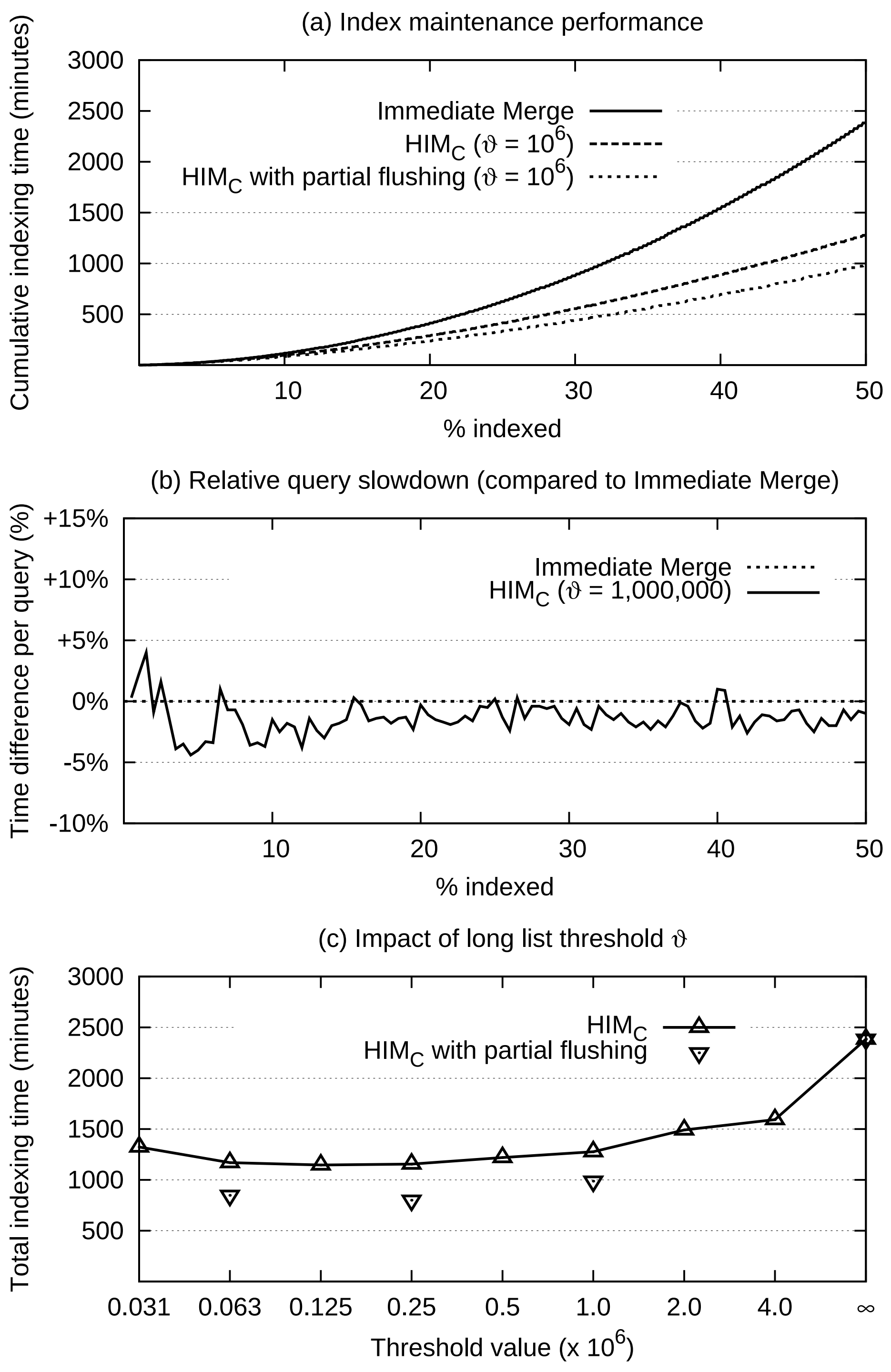


Figure 6.11: Indexing a random 50% of the GOV2 collection with hybrid and non-hybrid index maintenance (Immediate Merge vs. HIM$_C$; memory limit: 256 MB). The variant with partial flushing employs a PF threshold $\vartheta_{\text{PF}} = 2^{16}$, transferring all lists to disk that contain at least ϑ postings in total and for which there are at least ϑ_{PF} postings in the current in-memory index.

Immediate Merge (about 1–2% on average), stems from two sources:

- HIM_C's in-memory dictionary provides more efficient lookup operations than Immediate Merge's interleaved on-disk dictionary.
- Under some circumstances, the on-disk index implementation employed by Immediate Merge creates more than a single on-disk dictionary entry for a given term, resulting in one or more additional disk seek at query time. This only happens for a small number of terms, but has a visible impact on the results obtained here.

All in all, however, the performance difference is very small, and we may assume that both update strategies lead to equivalent query performance. This result is different from previous work [21], in which I characterized HIM_C as an *indexing performance vs. query performance trade-off*. The difference to these earlier results stems from the fact that, in the earlier work, I used a different implementation of the in-place index, completely relying on the file system implementation to enforce contiguity of posting lists. In the experiments presented here, contiguity is assured by a custom implementation of the in-place index.

Plot 6.11(c), finally, shows the update performance achieved by various long list thresholds ϑ. The optimal update performance is achieved for ϑ around 250,000. This is consistent with the rule of thumb that a list should be moved to the in-place index section when it becomes long enough so that reading it from disk requires more time than performing a single disk seek. For a typical consumer hard drive, this point is reached at 0.5 MB (approximately 125,000–250,000 vByte-compressed postings). With $\vartheta = 250{,}000$, HIM_C decreases the overall update overhead by 52% compared to Immediate Merge, and by 66% if partial flushing is enabled (a 44% performance improvement over HIM_C without partial flushing). It can be seen that HIM_C is largely insensitive to small variations of the threshold parameter ϑ. All values between $\vartheta = 62{,}500$ and $\vartheta = 500{,}000$ lead to approximately the same performance.

Since HIM_C exhibits the same query performance as Immediate Merge, but achieves an update performance that is more than twice as high as that of the non-hybrid strategy, we can say that HIM_C dominates Immediate Merge. There is no (performance-related) reason to choose Immediate Merge over HIM_C.

Non-Contiguous Hybrid Index Maintenance

The non-contiguous version of hybrid index maintenance was evaluated using the same command sequence as before. The results for HIM_NC are not shown here, because they are completely in line with what is suggested by the complexity analysis in the previous section. HIM_NC leads to better update performance than Immediate Merge, but is dominated by

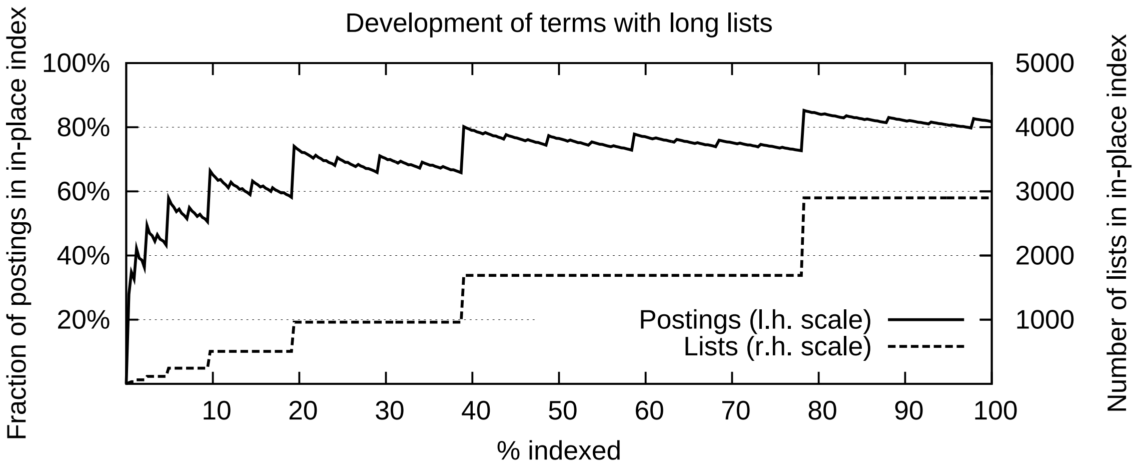


Figure 6.12: Indexing 100% of the GOV2 collection with HLM_{NC}. Development over time: number of long lists and number of postings in long lists. Hybrid threshold: $\vartheta = 10^6$. Memory limit: 256 MB.

Sqrt Merge. While both provide approximately the same query performance, Sqrt Merge exhibits a far more benign update complexity than HIM_{NC}.

The hybridization of Logarithmic Merge, HLM_{NC}, is more interesting. Figure 6.12 shows how the number of postings and the number of distinct terms in the in-place index section develop over time if HLM_{NC} is used to update the on-disk index structures. In contrast to the development shown in Figure 6.10, the two curves are not smooth any longer. Instead, re-merge operations (combining all on-disk index partitions into a single inverted file) are clearly visible at the 5%, 10%, 20%, 40%, and 80% marks. According to Algorithm 9, new terms may only be added to the in-place section if they do not appear in any index partition in the merge-maintained section. This usually is only the case after a complete re-merge. Despite this little edginess, the general development is similar to that shown in Figure 6.10, with 80% of all postings in the in-place index section after 40% of GOV2 have been indexed.

Figure 6.13 compares the two variants of non-contiguous Hybrid Logarithmic Merge, HLM_{NC} and $\text{HLM}_{\text{NC(A)}}$ (both defined in Section 6.4.2), to their non-hybrid counterpart. Plots 6.13(a) and 6.13(b) show their update performance, both in absolute terms and relative to the update performance of Logarithmic Merge. HLM_{NC} can build an index for GOV2 about 9% faster than Logarithmic Merge (531 minutes vs. 488 minutes). $\text{HLM}_{\text{NC(A)}}$, because it does not enforce the contiguity of posting lists in the in-place index section, performs even better, building the index in 450 minutes — 18% faster than Logarithmic Merge.

From 6.13(b), it can be seen that the relative performance advantage of HLM_{NC} and $\text{HLM}_{\text{NC(A)}}$ over Logarithmic Merge increases as the index grows. This is in line with the complexity analysis from Section 6.4.4, which showed that both methods can build the index in a time that is strictly better than $\Theta(N \cdot \log(N/M))$.

Figure 6.13(c) depicts the relative query performance of the three methods. It shows that $\text{HLM}_{\text{NC(A)}}$ leads to a query response time that is up to 5% higher than that of Logarithmic Merge. This is not entirely unexpected, as $\text{HLM}_{\text{NC(A)}}$ does not enforce the contiguity of inverted lists in the in-place index section. List fragmentation, therefore, is higher than with Logarithmic Merge, and this is reflected by the method's query processing performance.

What is surprising, however, is the fact that HLM_{NC}, although it enforces contiguity of all lists in the in-place section, also leads to weaker query performance than Logarithmic Merge. In Section 6.4.2, I argued that list fragmentation should be lower with HLM_{NC} than with Logarithmic Merge. This claim is supported by experimental results, depicted in Figure 6.14, which show that the average number of list fragments per query term is lower with HLM_{NC}. For example, after 75% of GOV2 have been indexed, the on-disk list for an average query term consists of 0.5 fragments less with HLM_{NC} than with Logarithmic Merge.

One could expect that this reduced degree of fragmentation would be reflected by the search engine's query processing performance. However, as can be seen from Figure 6.13(c), this is not so. HLM_{NC} is slower than Logarithmic Merge. One possible explanation for this contradiction could be that the operating system's file system implementation, unbeknownst to the search engine, introduces fragmentation into the engine's on-disk inverted files. Regrettably, this explanation is inconsistent with the fact that HIM_{C}, which employs the same in-place index structure as HLM_{NC}, achieves the same (or even slightly better) query performance as Immediate Merge.

I do not have an explanation for the anomaly shown in Figure 6.13(c). This is unfortunate, as the theoretical considerations from Sections 6.4.2 and 6.4.4 indicated that HLM_{NC} might dominate Logarithmic Merge, leading to both better update and better query performance. However, despite this result, HLM_{NC} and $\text{HLM}_{\text{NC(A)}}$ still have their right to exist, settling between Logarithmic Merge and No Merge in the update performance vs. query performance trade-off spectrum.

6.4.6 Hybrid vs. Non-Hybrid Index Maintenance

Hybrid index maintenance represents an interesting alternative to its non-hybrid, merge-based counterpart. However, especially for the non-contiguous version, it is not always clear which alternative is actually better. For example, when processing a search query, HIM_{NC} can be expected to lead to two random disk accesses per query term. This is the same as Sqrt Merge, but Sqrt Merge achieves a better update performance than HIM_{NC}.

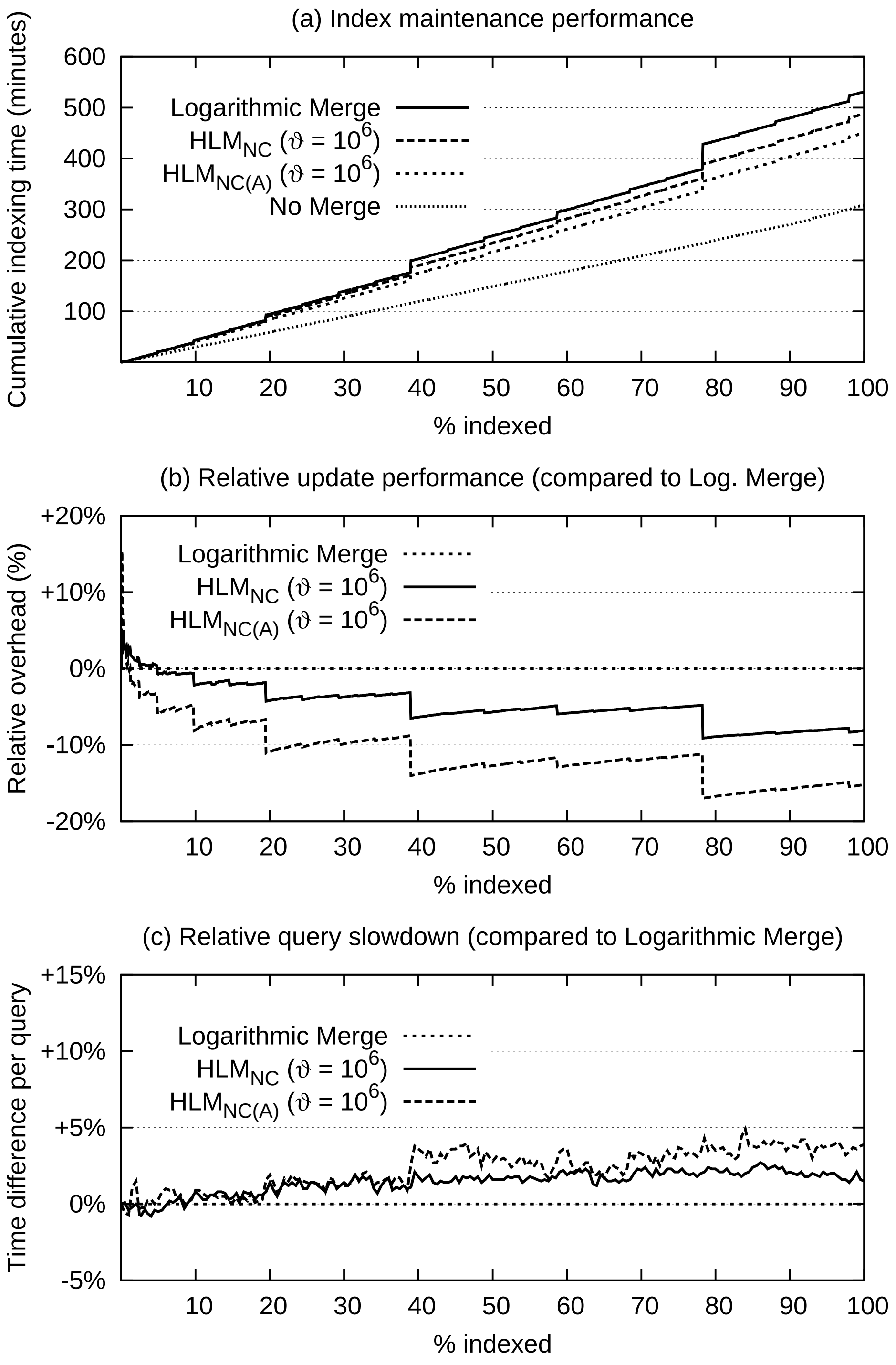


Figure 6.13: Indexing 100% of the GOV2 collection with hybrid and non-hybrid index maintenance (Logarithmic Merge vs. HLM_{NC} and $\text{HLM}_{\text{NC(A)}}$; memory limit: 256 MB). Relative query and update performance.

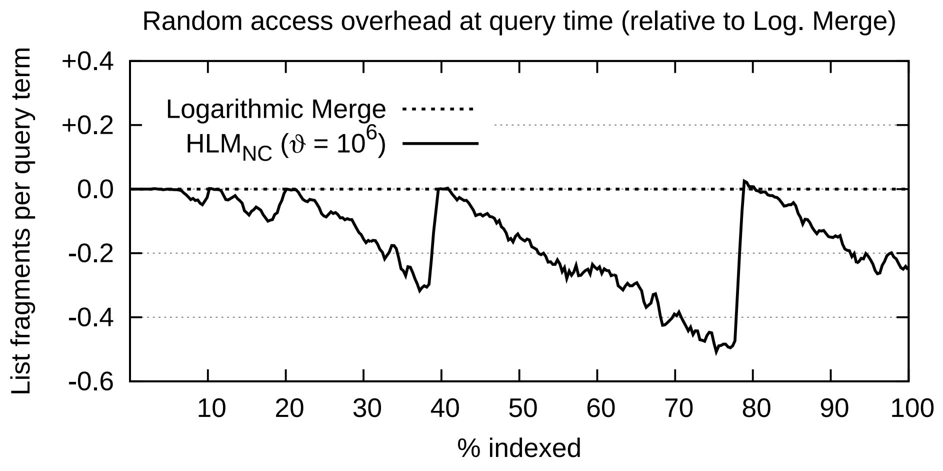


Figure 6.14: Number of list fragments per query term, for LOGARITHMIC MERGE and HLM$_{NC}$. Apart from a small anomaly around 80%, the number of list fragments for a random query term is consistently lower with HLM$_{NC}$ than with LOGARITHMIC MERGE, indicating that the former might lead to better query performance than the latter.

However, there are two situations in which hybrid index maintenance is in fact superior to a non-hybrid merge-based update strategy. If we classify a search system by the relative magnitude of its update and query load, then the applications for which hybrid index maintenance represents an interesting solution are found on both ends of the spectrum:

- If the system's workload predominantly stems from processing search queries, then each on-disk posting list should be stored in a contiguous region of the disk, implying that IMMEDIATE MERGE is an appropriate maintenance strategy. However, as has become apparent from the experiments, HIM$_C$ leads to the same query performance as IMMEDIATE MERGE (no list fragmentation), but offers greatly improved update performance. It should therefore always be preferred over IMMEDIATE MERGE.

- If the system's workload predominantly stems from index update operations, then LOGARITHMIC MERGE is an attractive update policy, because it keeps the index maintenance overhead very low. As shown in the experiments, HLM$_{NC}$ exhibits better update performance than LOGARITHMIC MERGE (up to 9% faster), while its query performance is only slightly reduced, within 3% of LOGARITHMIC MERGE. This makes the method attractive in environments with extremely high update load and relatively little query load.

6.5 File Deletions

In addition to file insertions, a file system usually also supports deletions (special-purpose backup file systems are among the rare exceptions). The update strategies described so far completely ignore this aspect of a dynamic search environment.

Published work on support for document deletions is scarce. Typically, deletions are dealt with in a delayed fashion. When a document is deleted, the corresponding postings are not immediately removed from the index, but only once a critical mass is exceeded or when the physical index update operation can be piggybacked on top of another index operation. The system described by Chiueh and Huang [25], for instance, maintains a *delete table*, realized by a binary search tree stored in main memory. Each deleted document receives an entry in this table. Whenever their search engine processes a query, the set of resulting documents is checked against the contents of the delete table, removing all search results referring to documents that have been deleted. In their implementation, a physical index update, integrating the information found in the delete table into the on-disk index structures, can be carried out

- during query processing: the posting list for each query term is adjusted to be consistent with the current contents of the delete table; the modified posting list is written back to disk;
- in a batch update: when the search engine runs out of memory (after a number of document insertions) and the in-memory index data have to be combined with the on-disk index, deletions are applied to all posting lists in the index.

As already pointed out in Section 5.5, the postprocessing approach employed by Chiueh and Huang has the disadvantage that document scores are computed for documents that no longer exist. Moreover, because the search engine does not know in advance how many of the top k documents identified are still alive, query optimization strategies like MaxScore [109] become less effective.

6.5.1 Utilizing Security Restrictions to Delay Index Updates

Fortunately, the security mechanisms set out in Chapter 5 can directly be applied to the problem of document deletions, having the same properties as before. Whenever a file F is deleted, it is – by definition – no longer searchable by any user. Within the search engine, this is guaranteed by removing F from the file table maintained by the search engine's security manager. As a result, postings referring to F are no longer visible to the query processing

```
@removefile trec/chunks/trec.00673
@rank[qap][count=20][id=215] “⟨doc⟩”··· “⟨/doc⟩” by “unexplained”, “highway”, “accidents”
@addfile trec/chunks/trec.00673
@rank[qap][count=20][id=8] “⟨doc⟩”··· “⟨/doc⟩” by “rail”, “strikes”
@removefile trec/chunks/trec.01257
@rank[qap][count=20][id=104] “⟨doc⟩”··· “⟨/doc⟩” by “oil”, “spills”
@addfile trec/chunks/trec.01257
.....
```

Figure 6.15: Command sequence used to measure the impact of garbage postings on query performance. Starting from a complete index for the TREC collection, documents are removed and re-inserted.

module. A retrieval score for F is never even computed, eliminating the problems caused by the postprocessing approach.

However, the search engine can only rely on the mechanisms provided by the security subsystem for so long. Ultimately, because the number of garbage postings in the index keeps growing as more and more files get deleted, the search engine needs to start a garbage collection process that removes all obsolete postings from the index. Otherwise, because garbage postings must still be read from disk when processing a query (even though they are not considered for ranking purposes), query performance is likely to deteriorate beyond any acceptable limit.

I examined the effect that uncollected garbage postings in the on-disk index have on query processing performance. After building an index for the TREC collection (823 million postings), a random file was deleted from the index and then immediately re-inserted. After each such index update event (file deletion, file insertion), a random query from the TREC query set was processed by the search engine, returning the top 20 search results. By re-inserting each file immediately after its deletion, the amount of non-garbage postings data in the index is kept constant, and the number of garbage postings in the index is the only variable factor. A fragment from the corresponding command sequence, comprising 50,000 search queries and update operations, is given by Figure 6.15.

Figure 6.16 shows the outcome of this experiment. Not unexpectedly, as the number of garbage postings increases, query processing becomes less efficient, due to the disk overhead associated with reading postings that are never used during query processing.

However, when increasing the total number of postings in the index by 150% (from 0.8 billion to 2.0 billion), the average time per search query grows by only 70% (from 340 ms to

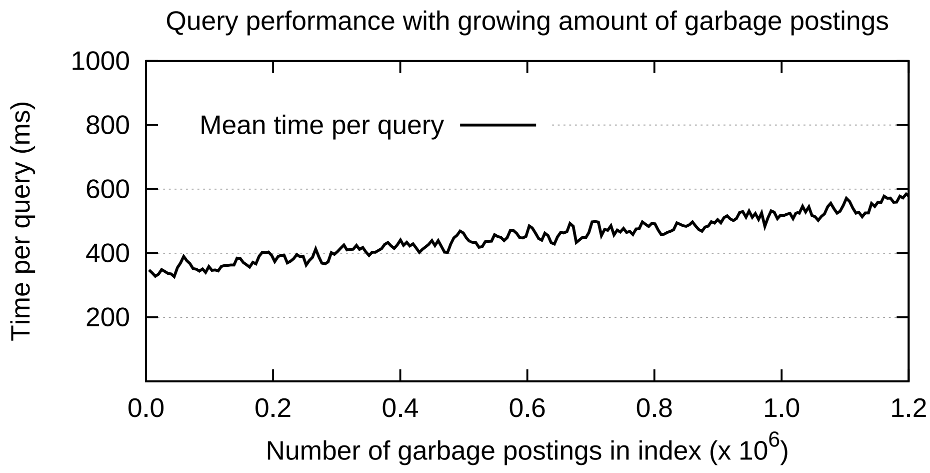


Figure 6.16: Average query processing performance (averaged over a sliding window covering 500 queries) for an increasing the number of garbage postings in the index. The amount of non-garbage postings in the index is kept constant. Starting from a complete index for the TREC collection (823 million postings), documents are removed and re-inserted (cf. Figure 6.15). New postings are folded into the existing index by Immediate Merge (memory buffers: 32 MB).

580 ms). This is because garbage postings do not contribute to document scores and thus carry a very light computational weight. Nonetheless, at some point all postings corresponding to deleted documents will need to be removed from the index, as they do not only harm query performance (violating Requirement 1 from Chapter 1), but also artificially increase the storage requirements of the search engine beyond its actual needs (violating Requirement 4). Such a *garbage collection* process is computationally expensive. Hence, it should be integrated into other index update operations, so as to reduce the overhead.

6.5.2 Collecting Garbage Postings

The process of removing garbage postings from an existing posting list is very similar to the process of computing the intersection of two sorted lists of integers (cf. Algorithm 1). Each indexed file F_i corresponds to an index address range $[s_i, e_i]$, subject to the constraint

$$s_i \leq e_i < s_{i+1} \leq e_{i+1} \;\; \forall\, 1 \leq i < |\mathcal{F}|, \tag{6.26}$$

where $|\mathcal{F}|$ is the number of active files in the index, and $e_i - s_i + 1$ is the number of tokens found in file F_i. When starting a garbage collection process, the search engine's index manager obtains the sequence

$$\mathcal{F} := [s_i, e_i]_{1 \leq i \leq |\mathcal{F}|} \tag{6.27}$$

Merge method	**SBFS**	**TREC**	**GOV2**
Without GC (compressed)	241.7 .. 243.7 sec	84.0 .. 88.7 sec	80.2 .. 83.8 min
Without GC (uncompressed)	279.5 .. 282.3 sec	104.4 .. 106.1 sec	96.1 .. 97.6 min
With GC (binary search)	395.8 .. 398.3 sec	170.4 .. 172.4 sec	129.1 .. 129.8 min
With GC (galloping search)	310.1 .. 312.4 sec	123.8 .. 128.9 sec	102.8 .. 106.7 min
Number of files in index	310,459	15,449	27,204

Table 6.4: Duration of the final merge operation in merge-based index construction, with and without built-in garbage collection. All numbers represent 95% confidence intervals, obtained by repeating each experiment 6 times.

from the security manager. It then processes each posting lists L in the index, one at a time, computing the intersection between L and and the file intervals in $\mathcal{F}$:

$$L' = \{p \in L \mid \exists\ [s, e] \in \mathcal{F} : s \leq p \leq e\}. \tag{6.28}$$

This intersection operation can be realized through iterated binary search, finding an interval $[s, e] \in \mathcal{F}$ for each posting p in the given posting list. Alternatively, a galloping search procedure very similar to the one shown in Algorithm 1 may be employed.

To quantify the exact overhead of the garbage collection process, I had the search engine build complete indices for SBFS, TREC, and GOV, using the merge-based index construction technique outlined in Chapter 4. However, in contrast to the experiments from Chapter 4, the final merge procedure was adjusted slightly so that the search engine applied its garbage collection mechanism to every posting list visited during the final merge. Because no files had been deleted prior to the final merge, the garbage collection process did not actually remove any postings from the index. It merely checked whether all postings in the index belonged to files that were still alive.

The results of this experiment are shown in Table 6.4. It can be seen that integrating the garbage collector into the original merge procedure leads to a substantial slowdown of the merge operation. When using galloping search to perform the set intersection operations, the time needed to perform the final merge for the TREC collection increases from 86 seconds to 126 seconds on average (+46%). For SBFS, the slowdown is a bit more moderate, from 243 seconds to 311 seconds on average (+28%). The slowdown is caused by two components of the garbage collection procedure: (1) decompressing compressed posting lists, and (2) running the set intersection algorithm. As can be seen from the table, the two components contribute about equally to the total garbage collection overhead.

The reason for the rather large differences between the overhead seen for the individual text collections is the great number of unique terms in SBFS (28.7 million) compared to TREC (2.1 million), making the performance baseline (merging without garbage collection and without decompression) perform worse for SBFS than for TREC (note that both collections contain approximately the same number of postings).

Table 6.4 also shows that the garbage collection implementation based on iterated binary search performs much worse than the one based on galloping search. This is not surprising. According to Zipf's law, the vast majority of all postings appears in long lists. For those lists, however, because their postings appear in almost every file, galloping search runs in time $\mathcal{O}(|L|)$, as opposed to $\mathcal{O}(|L| \cdot \log(|\mathcal{F}|)$ for binary search (here, $|L|$ denotes the number of postings in the posting list being processed by the garbage collector).

It is possible that the performance of the garbage collection procedure could be slightly improved by switching from galloping search to a linear scanning algorithm whenever postings from a list are found to appear in a large number of files. However, it is not clear how the optimal transition threshold should be chosen. Moreover, it would complicate the implementation and also is unlikely to lead to substantial savings compared to galloping search.

6.5.3 Performance Trade-offs and Garbage Collection Policies

Postings that belong to deleted files put the designer of a file system search engine into a dilemma. Leaving too many garbage postings in the index increases disk space consumption and query response times. Applying garbage collection too eagerly, on the other hand, increases the index maintenance overhead, even if the garbage collector is integrated into a routinely scheduled merge update operation, as shown above.

In previous work [19], I have characterized this as a trade-off between query processing performance and index maintenance performance. However, this characterization is not entirely accurate. Leaving garbage postings in the index does not only harm query performance, but also index maintenance performance, as all such postings are read from and written to disk during every physical update cycle. It may therefore make sense to remove garbage postings from the index even though their impact on query processing performance might still be tolerable. Consequently, any garbage collection policy should consist of two different components — one aiming at high index maintenance performance, the other ensuring that query performance stays at an acceptable level and is not excessively affected by garbage postings in the index.

A Garbage Collection Policy

The following two-tiered garbage collection policy has been implemented in Wumpus. It assumes that the search engine is based on a merge-maintained index with one or more index partitions.

1. **Garbage collection based on a global threshold**
   A pre-defined garbage threshold $\rho \in (0,1]$ is used to determine when the amount of garbage postings in the index has come to a point at which the resulting query slowdown is no longer tolerable. Whenever this threshold is reached, the search engine initiates a re-merge operation, combining all index partitions into a single inverted file and removing all garbage postings from the index.

2. **On-the-fly garbage collection**
   A pre-defined garbage threshold $\rho' \in (0,1]$ $(\rho' \leq \rho)$ is used to determine when it is sensible to integrate the garbage collection mechanism into an index update operation (merge operation) in order to improve index maintenance performance, not taking the garbage postings' impact on query performance into account.
   Whenever two or more index partitions are merged, and the relative number of garbage postings in these partitions is greater than ρ', the garbage collector is integrated into the merge process.

Rule 1 is aiming for high query performance. Rule 2 is aiming for high update performance, trying to combine the disk activity exhibited by ordinary merge operations and by the garbage collector, and to reduce the amount of garbage postings copied around during merge operations.

Whether, in the presence of rule 2, rule 1 is still necessary or not depends on the characteristics of the text collection and on the exact index update strategy employed by the search engine. Suppose Immediate Merge is used to fold incoming postings data into the on-disk index. If the search engine operates in a steady state, and document deletions occur at approximately the same rate as insertions, then rule 2 is likely to be able to replace rule 1 altogether, as the amount of garbage postings accumulated between two re-merge operations will be small. If, however, the collection exhibits periodic bursts of document deletions, in which a large portion of the postings in the index becomes obsolete, then rule 1 will still be necessary, as rule 2 alone would only remove the garbage data from the index when the next merge operation takes place (i.e., when the incoming postings data exceed the amount of available main memory). Query response time and index storage requirements, in that case, would be unreasonably high, for a potentially unlimited period of time (violating Requirement 1 and Requirement 4 from Chapter 1).

But even if the system operates in a steady state, it might be necessary to enforce rule 1 in addition to rule 2. Suppose the search engine employs Logarithmic Merge to update its index. Because of the exponentially increasing size of the individual index partitions (cf. Figure 6.3), the oldest partition, referred to as I_{old}, holds at least 50% of the postings in the index. Unfortunately, I_{old} is very rarely involved in any update operations (after all, this is the very idea behind index partitioning). Therefore, while rule 2 is applied over and over again, removing garbage data from the younger index partitions, garbage postings will keep accumulating in I_{old}, increasing query response time and index storage requirements.

Hence, both rules, 1 and 2, are in fact necessary and need to be enforced by the search engine. They complement each other well and, in combination, can be used to realize efficient support for document deletions.

Optimal Threshold Values

In the case of the global threshold ρ, it is difficult to pinpoint the optimal value, since its purpose is to limit the impact of garbage postings on query performance, and its optimal value depends on user preferences. For the local threshold ρ', however, targeting index maintenance performance, it is possible to identify an optimum that does not depend on user preferences. The parameter's optimal value is influenced by a number of external factors. It depends on the relative performance of the garbage collection mechanism, compared to an ordinary merge operation without garbage collection, and on the amount of garbage postings accumulated between two merge events.

Suppose the search engine operates in steady state, using Immediate Merge to realize on-disk index updates, and maintaining p live (i.e., non-garbage) postings in the index. Let g denote the total number of garbage postings in the index, and Δg the number of garbage postings accumulated between two merge events. Because the system is in steady state, we have $\Delta g \approx M$, where M is the number of postings that can be buffered in memory. Assume the application of the garbage collector increases the time required to perform a merge operation by a factor q (e.g., $q = 1.46$ for TREC, cf. Table 6.3). Finally, suppose the search engine uses an on-the-fly garbage threshold ρ', but no global threshold ρ.

The search engine integrates the garbage collector into a merge operation if and only if

$$\frac{g}{p+g} \geq \rho' \quad \Leftrightarrow \quad g \geq \frac{p \cdot \rho'}{1-\rho'}. \tag{6.29}$$

Hence, the number of merge operations performed between two applications of the garbage collector is

$$m = \left\lceil \frac{p \cdot \rho'}{\Delta g \cdot (1 - \rho')} \right\rceil - 1 \tag{6.30}$$

(not including the merge operation with activated garbage collector itself). The total index maintenance overhead of these m merge events, plus the last merge event, which includes the activation of the garbage collector, then is

$$O_G = c \cdot \left(q \cdot (p + (m+1) \cdot \Delta g) + \sum_{k=1}^{m} (p + k \cdot \Delta g) \right) \tag{6.31}$$

(where c is a constant translating the postings in the index into a time value). The average overhead per merge event is

$$O_G^{(\text{avg})} = c \cdot \frac{q \cdot (p + (m+1) \cdot \Delta g) + \sum_{k=1}^{m} (p + k \cdot \Delta g)}{m+1}. \tag{6.32}$$

By computing this overhead for various threshold values ρ', it is possible to identify $\rho'_{\text{opt}} \approx 0.1$ as the optimal parameter value for the TREC collection with 32 MB for the in-memory index buffers (corresponding to 10.6 million postings). However, a large range of threshold values ($0.04 \leq \rho' \leq 0.2$) leads to reasonable results, causing a total index maintenance overhead within 4% of the optimal value. This result is also confirmed by my experiments.

6.5.4 Experiments

I repeated the experiment from Figure 6.16, this time with the two-tiered garbage collection policy in place. After building a full index for the TREC collection, random documents were removed and re-inserted, as defined by the command sequence shown in Figure 6.15.

Figure 6.17 shows the results obtained when using Immediate Merge, for various garbage collection thresholds ρ' (as discussed before, the second threshold, ρ, has no effect when Immediate Merge is used for a collection in steady state). From plot 6.17(a) it can be seen that *eager garbage collection* ($\rho' = 0$) performs worst at first. After a certain point, though, this role is taken over by *no garbage collection* ($\rho = 1$). Because, in the latter case, the index keeps growing, the cumulative index update time with $\rho' = 1$ ascends quadratically (like in the case of incremental text collections), while the other threshold values lead to linear growth.

The two threshold values $\rho' = 0.05$ and $\rho' = 0.2$ result in almost the same update performance, with $\rho' = 0.05$ having a slight lead, implying that the garbage collection mechanism

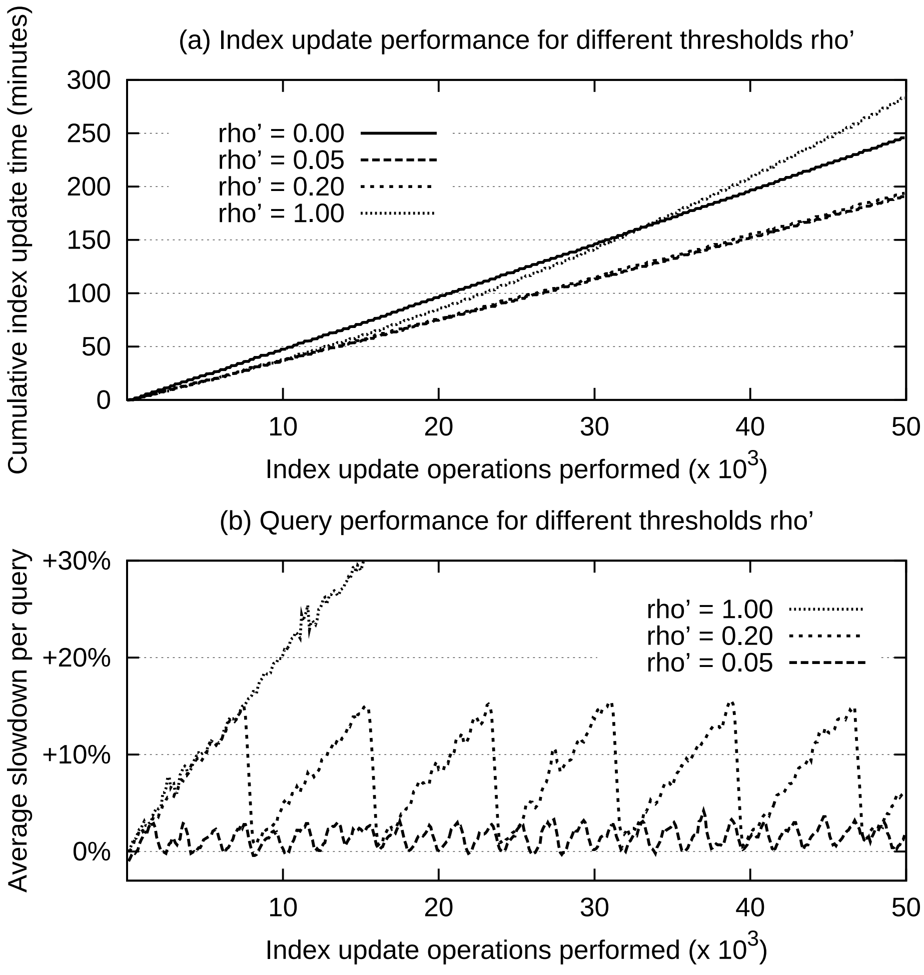


Figure 6.17: Index updates for a dynamic collection (TREC) in steady state. Index maintenance strategy: Immediate Merge. Global garbage collection threshold: $\rho = 1$ (i.e., never). On-the-fly garbage collection threshold: variable. In-memory update buffers: 32 MB.

is relatively robust against small variations in the threshold parameter ρ' — provided that extreme cases ($\rho' = 0$ or $\rho' = 1$) are avoided.

Plot 6.17(b) shows the query processing performance achieved by three different threshold values, relative to the query-optimal baseline value $\rho' = 0$. Garbage collection activity is clearly visible, with one peak in the line for $\rho' = 0.2$ after every fourth peak in the line for $\rho' = 0.05$. As expected, for a garbage threshold $\rho' = 0.2$, the maximum slowdown seen in the experiments remains just under 20%.

As a last experiment, I re-ran the command sequence from Figure 6.15 against a search engine configuration employing Logarithmic Merge instead of Immediate Merge. Because a merge operation carried out by Logarithmic Merge usually only affects a small

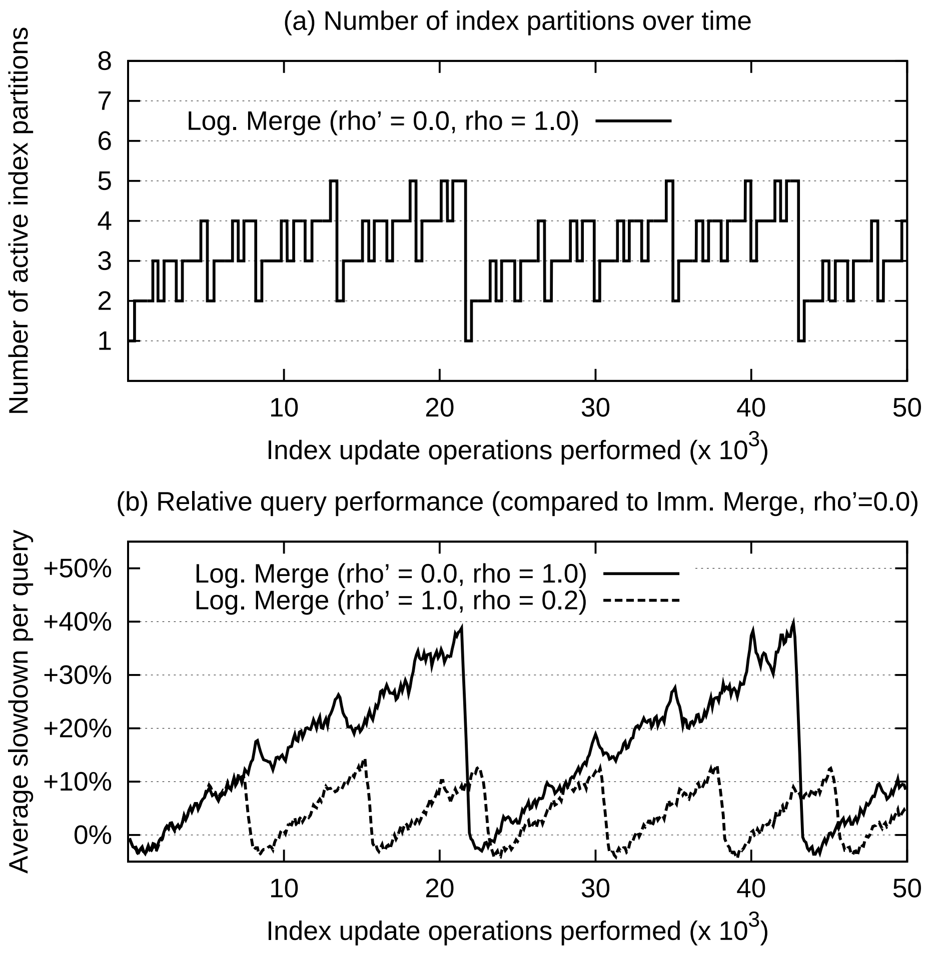


Figure 6.18: Index updates for a dynamic collection in steady state. Index maintenance strategy: Logarithmic Merge. Comparing two approaches: using ρ exclusively ($\rho' = 1$) vs. using ρ' exclusively ($\rho = 1$).

part of the entire index, the threshold parameter ρ can no longer be ignored. Figure 6.18 shows the results obtained by comparing two different garbage collection strategies:

- making exclusive use of the global threshold ρ;
- making exclusive use of the (local) on-the-fly garbage collection threshold ρ'.

It can be seen that, even if garbage postings are collected very eagerly (by setting $\rho' = 0$), query performance keeps decreasing if the local threshold ρ' is used as the only garbage collection criterion. Setting $\rho = 1$, as done in the experiment, allows garbage postings to pile up in the older index partitions, since these partitions are rarely involved in a merge operation. The relative query slowdown seen under these circumstances depends on the nature of the most recent merge operations. The greatest decrease in query time it witnessed immediately after a complete re-merge, reducing the number of active index partitions to 1.

Controlling the amount of garbage data in the index through the global threshold ρ, on the other hand, proves to be an effective way of keeping query performance close to the optimum (Immediate Merge with $\rho' = 0$). Both query performance and update performance can, however, be slightly improved by also using a local threshold ρ' in addition to the global one (not shown in the figure).

6.6 File Modifications

Thus far, I have only covered file insertions and file deletions. Research in the field of dynamic text collections usually focuses on these two types of operations and assumes that file modifications can be realized as a combination of delete and insert operations. This assumption may stem from the area of digital libraries, where document modifications are a rather rare phenomenon. In file system search, the situation is different. A large number of files that a user could be interested in searching are inherently dynamic and grow over time, through a series of append operations. Prominent examples are:

- system log files (e.g., `/var/log/messages`);
- e-mail folders, realized by files in the `mbox` format, each containing a sequence of e-mail messages.

While it might be argued that individual messages in an `mbox` file should be treated as independent documents, the same is clearly not the case for system log files. Treating every line in `/var/log/messages` as a separate document substantially limits the usability of the search engine for certain search operations. Hence, it is important to associate all postings for a given file with a contiguous region of the index address space.

Unfortunately, as shown in Figure 6.19, this might not always possible, since the address space region immediately following that assigned to the original contents of a file might already be occupied by another file. Re-indexing the whole file (i.e., treating the append operation as a delete/insert sequence) is one way to make sure that the file receives a contiguous region of the index address space. Alas, a system log file can easily contain several megabytes of text. The new data added at the end of the file, on the other hand, may be as little as a few bytes — a single line in the log file. Therefore, treating an append modification as a sequence of delete and re-insert is bound to lead to extremely low indexing performance — quadratic in the number of append operations. This is rather unfortunate, as the primary focus of the first

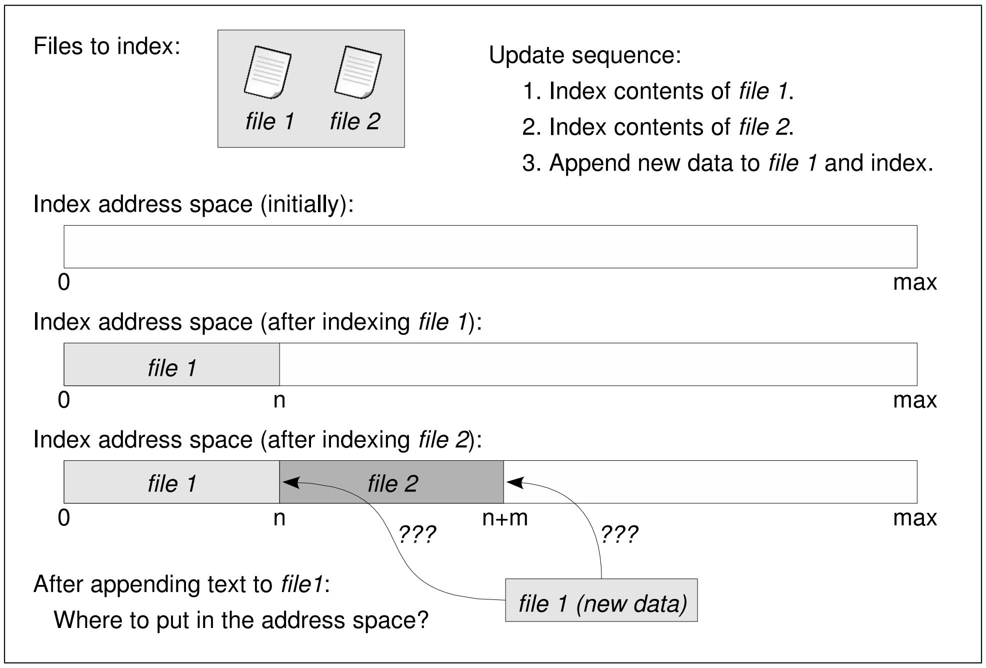

Files to index:
file 1
file 2
Update sequence:
1. Index contents of file 1.
2. Index contents of file 2.
3. Append new data to file 1 and index.
Index address space (initially):
0
max
Index address space (after indexing file 1):
file 1
0
n
max
Index address space (after indexing file 2):
file 1
file 2
0
n
n+m
max
???
???
After appending text to file1:
Where to put in the address space?
file 1 (new data)

Figure 6.19: A file append operation causing an address space conflict. Old data and new data in *file 1* should be assigned adjacent address space regions to be able to retrieve data from both parts of the file at the same time. Unfortunately, the address space after *file 1* is already occupied by *file 2*.

part of this chapter had been to move away from Immediate Merge's quadratic indexing complexity.

While file modifications in general can be more complex than simple append operations, append operations are by far the most common modification operations in a typical file system. Whenever a file is changed in a text editor, for example, the editor will usually write the entire file back to disk — even though only a small portion of the data in the file might have been changed. In such a case, it might be possible to slightly reduce the indexing overhead, by performing a `diff`[4] operation to find out which parts of a file need to be re-indexed, as done by Lim et al. [71]. However, as can be seen from Table 4.5 (Chapter 4), the reading and parsing overhead accounts for a large portion of the total workload of the indexing process, making it questionable whether substantial savings can be obtained by following this road.

6.6.1 Append Indexing Strategies

I decided to restrict myself to the case of true append operations, where it is clear from the operating system's file access pattern that no existing data were modified, and where only

[4]http://www.gnu.org/software/diffutils/ (accessed 2007-04-07)

the new data need to be read from disk, parsed, and indexed. I implemented and evaluated three different APPEND indexing strategies, as follows.

Re-Indexing

The *re-indexing* approach treats a file modification as a deletion that is followed by an insertion. This strategy can be expected to lead to high query performance and very low update performance. It serves as the baseline against which the two other implementations are compared.

Dynamic Address Space Transformations

When indexing the new text added to an existing file through an append operation, it might not be possible to assign the new postings an address space range that immediately follows that assigned to the existing postings for the file. However, there is plenty of room at the end of the address space, after the last file in the index. Therefore, instead of assigning the new data an address range that neighbors that of the old data, the old data can be relocated to a new place where there is sufficient address space to harbor both the old and the new text.

Of course, this relocation of the file's postings to a new index address is only virtual. All postings keep their original value. At query time, however, the posting list for each query term is adjusted in order to reflect the new position of each file in the address space. Such an *address space transformation* can be achieved by maintaining a set $\mathcal{T}$ of transformation rules

$$\{T_i\}_{1 \leq i \leq |\mathcal{T}|},\ T_i = (source_i, destination_i, size_i) \tag{6.33}$$

and by updating each posting p of a query term's posting list according to the transformation rules in $\mathcal{T}$:

$$p_{new} = \begin{cases} p - source_i + destination_i : & \text{if } \exists\ T_i \in \mathcal{T} :\ source_i \leq p < source_i + size_i; \\ p : & \text{otherwise.} \end{cases} \tag{6.34}$$

In the example from Figure 6.19, for instance, a single transformation rule T_1 is needed to move the index data for *file 1* to the free address space after *file 2*:

$$T_1 = (0, n + m, n). \tag{6.35}$$

The new data from *file 1* may then be indexed as usual, starting at index address $2n + m$. The transformation set $\mathcal{T}$ can be stored in a binary search tree to allow for efficient update operations. Applying $\mathcal{T}$ to a posting list L requires all of L's postings to be in memory and can be done by an operation similar to a galloping search used for list intersection (cf. Algorithm 1), followed by a sort operation to re-establish the original low-to-high ordering of the postings in the list. In the implementation used in my experiments, the sorting step is realized by radix sort, operating on 64-bit postings.

The search engine's indexing routines are completely oblivious to the append operation and treat it as an ordinary indexing operation, emitting a sequence of postings starting at the end of the used portion of the address space. The strategy can therefore be expected to lead to very high index maintenance performance. Query performance, on the other hand, can be expected to be comparatively poor. In addition, the requirement that each query term's posting list needs to be loaded into memory and decompressed prior to query processing may be a limitation if the index is very large compared to the amount of available main memory.

Also, note that this strategy requires a number of transformation rules T_i that is linear in the number of append operations carried out. If the file system exhibits a high append update frequency, then this can lead to problems for both query performance and memory requirements. The transformation set $\mathcal{T}$ should therefore periodically be integrated into the actual index data, either by adjusting all posting lists or by re-indexing all affected files.

Non-Monotonic Posting Lists

The third strategy is based on the idea that it is possible to keep old and new postings in a contiguous region of the address space by reserving more space than actually needed when first indexing a given file. This is similar to the proportional pre-allocation strategy used in in-place index update (cf. Section 2.5.1). When the search engine indexes (or re-indexes) a file F, containing $|F|$ tokens, it allocates an address space range with enough room for $k \cdot |F|$ postings (in my experiments, I use $k = 3$).

In the example from Figure 6.19, *file 1* would receive an address space range of size $3n$, and *file 2* would receive an address space range starting at $3n$ and ending at $3n + 3m$. When the new data appended to *file 1* need to be indexed, their postings can be inserted into the unused range starting at address n and ending at address $3n - 1$.

Whenever the range assigned to a given file is exhausted because too much text has been appended to an existing file, the entire file is re-indexed, with its postings starting at a new index address at the end of the used portion of the address space. This leads to a logarithmic number of re-index operations and a total indexing complexity that is linear in the size of the file, just like in the case of in-place index update with proportional pre-allocation.

As simple as this strategy might seem, it leads to a fundamental problem: the postings in a given posting list are no longer monotonically increasing. This is problematic because many components of the search engine rely on this very fact. Hence, in order to realize append operations by reserving some free address space at the end of every indexed file, the following major changes to the search engine are necessary:

- If the search engine employs a compressed in-memory index construction strategy, then the compression method used to encode incoming postings for a given term needs to be adjusted, allowing for the fact that the difference between two consecutive postings for the same term may be negative.
  In my experiments, the search engine used vByte [100] to compress incoming postings. A delta value of 0 was used to indicate a violation of the monotonicity constraint and to use index offset zero as the point of reference when decoding the following posting.
- When transferring postings from memory to disk (after the indexing process runs out of memory), the postings for each term first need to be decompressed and re-sorted (using radix sort in my current implementation), then compressed again.
- Posting lists now always need to be decompressed during index merge operations. Because postings are added to the index in a non-monotonic fashion, it is no longer possible to merge a set of partitions by concatenating their list segments. Instead, a true multi-way merge operation needs to be performed.
- When processing a search query, the postings for each query term, as found in the in-memory index (used to buffer incoming postings before transferring them to disk) need to be sorted first. Again, this necessitates prior decompression and thus requires the in-memory list for each query term to fit completely into main memory (note that this is not the same as in the case of dynamic address space transformation, where the *entire* list for each query term needed to fit into main memory).

In addition to these major changes to the indexing / query processing logic of the search engine, several minor changes are necessary, for example to the implementation of the search engine's index-to-text map, which also assumes monotonicity of incoming postings.

6.6.2 Experiments

I evaluated all three Append strategies by having the search engine build an index for the TREC collection in the following way. The collection was split into 100 sub-collections. One

```
@system cat trec/chunks/trec.05326 >> trec/part.91
@update APPEND trec/part.91
@rank[qap][count=20][id=435] "⟨doc⟩"··· "⟨/doc⟩" by "sun", "beds", "safe"
@system cat trec/chunks/trec.13366 >> trec/part.73
@update APPEND trec/part.73
@rank[qap][count=20][id=303] "⟨doc⟩"··· "⟨/doc⟩" by "osteoporosis"
.....
```

Figure 6.20: Fragment from the APPEND command sequence used in the experiments.

after the other, each of the 15,449 files in the collection was appended to a random sub-collection. The search engine then was notified of the append operation, and a search query was processed against the new contents of the index. A fragment from the actual command sequence used is given by Figure 6.20. The sequence is not meant to be representative of a typical file system search scenario. Instead, it is used to explore the performance of the different strategies under extreme conditions, where the file system is updated exclusively through append operations.

For all three configurations, the garbage collection thresholds were set to $\rho = 0.4$ (global threshold) and $\rho' = 0.1$ (on-the-fly threshold). LOGARITHMIC MERGE was used as the index update strategy. The results obtained are shown in Figure 6.21. Plot 6.21(a) depicts the cumulative time spent on index updates. Plot 6.21(b) depicts the average time per query, computed over a sliding window covering 150 search queries.

As expected, the re-indexing strategy leads to unacceptable index maintenance performance, even worse than IMMEDIATE MERGE. Its query performance stays relatively high, also as expected, only slightly worse than in the original experiments from Section 6.2, due to garbage postings in the index and interference of index update operations with search queries (both are competing for the operating system's disk cache).

Dynamic address space translation, conversely, leads to very high update performance, close to that of LOGARITHMIC MERGE in the original, non-APPEND experiments. Its query performance, however, is not competitive at all. On average, a query takes 4 times as long to be processed as with the re-indexing strategy — 1.6 seconds per query when 100% of the collection have been indexed.

The third strategy, pre-allocating index address space and maintaining non-monotonic posting lists, represents an interesting trade-off between the two extremes. The index maintenance overhead is approximately three times as large as in the case of the transformation strategy, for the reasons listed above. Perhaps surprisingly, however, query performance is

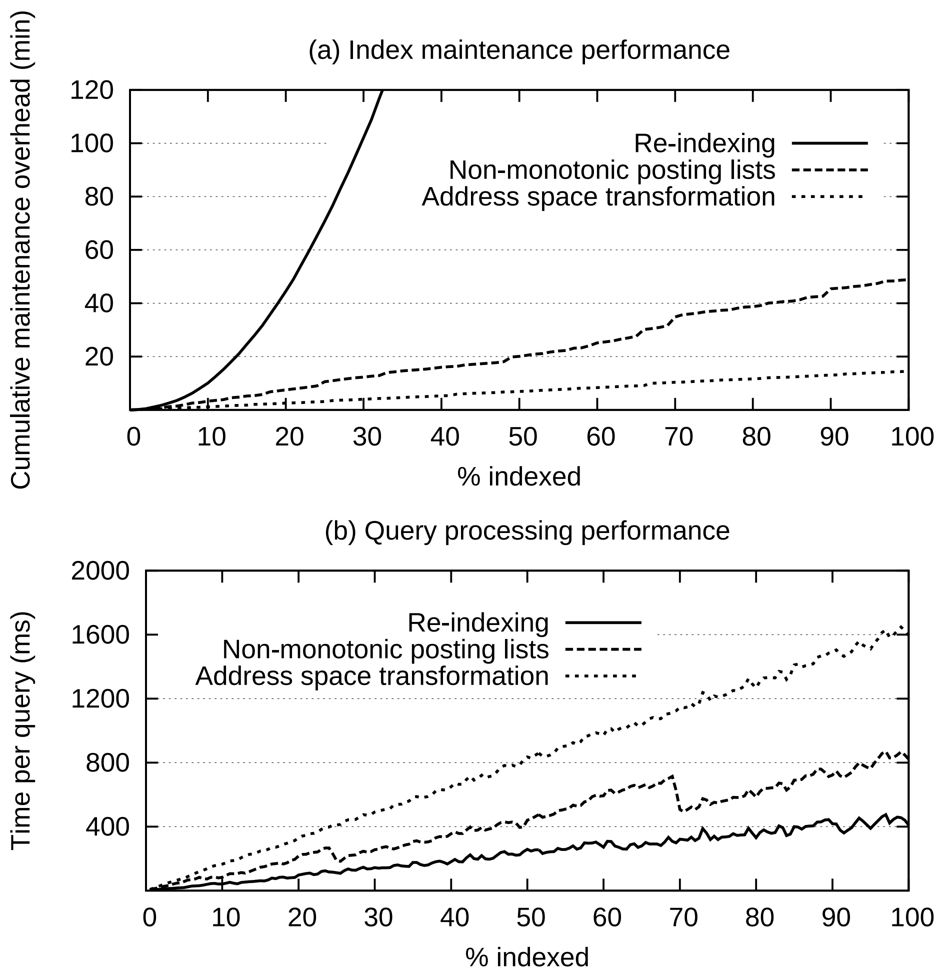


Figure 6.21: Update and query performance for three different APPEND update strategies.

quite a bit lower than in the case of re-indexing, with a relative slowdown between 50% (around 70% indexed) and 100% (at 100% indexed).

The query slowdown stems from two sources. First, the postings in the in-memory index need to be decompressed and sorted before the query can be processed. Second, the query processor requires random access to all posting lists: the MAXSCORE [109] heuristic can only be employed if parts of a query term's posting list may be skipped; the lists for "⟨doc⟩" and "⟨/doc⟩" need to support random access in order to efficiently find the document that a particular posting refers to. If the search engine maintains a set of n independent index partitions, as is the case if LOGARITHMIC MERGE is employed to realize on-disk index updates, then a random list access to a term's posting list requires n binary search operations, one for each index partition, because the individual index partitions no longer form a partitioning of the used portion of the address space (they may overlap).

The relationship between the number of independent index partitions and the query slowdown can be seen from Figure 6.21, where the re-merge operation at around *70% indexed* (corresponding to a small jump in the cumulative indexing time) results in a reduced query response time, from 710 ms down to 490 ms per query.

All in all, the address space pre-allocation strategy seems to be the best choice. Re-indexing is definitely not an option, because of its unbearably low update performance. Address space transformation, on the other hand, leads to very low query performance and exhibits substantial memory requirements (due to the necessity to load the query terms' posting lists completely into main memory — in uncompressed form). The relatively low update performance of the pre-allocation strategy, although undesirable, is unlikely to cause severe problems in the context of file system search. As we have seen in Section 4.3.3 (Table 4.5), converting the input files to plain text tends to be the main bottleneck when indexing real file system data. Hence, the additional overhead introduced here is not much greater than the one already present due to external constraints.

6.7 Meeting Real-Time Requirements

In the previous sections of this chapter, I have demonstrated that it is possible to have the search engine provide efficient support for index update operations – insertions, deletions, modifications – exacted by a fully dynamic text collection. These update operations, however, only feature a benign *amortized* complexity; the worst-case complexity of each update operation can still be very high. For example, if Logarithmic Merge is used as the sarch engine's index update policy, then it is guaranteed that the system requires $\mathcal{O}(N \cdot \log(N))$ disk operations to build an index for a dynamic collection containing N tokens. But it is not guaranteed that each token can be indexed in time $\mathcal{O}(\log(N))$. Instead, indexing a single token may require a very long time, if the token is unlucky enough to cause the search engine to run out of memory.

Figure 6.2(a), for example, shows that the vast majority of all file insertions for the TREC collection can be carried out in under a second. Every now and then, however, the search engine needs to perform a physical index update, transferring postings from memory to disk. The duration of this operation depends on the update strategy chosen. For No Merge, it takes between 1 and 2 seconds. For Immediate Merge (and all other merge-based update strategies), it may take up to several minutes.

Asynchronous Index Maintenance

In order to achieve true real-time index updates, the search engine needs to be modified so that garbage collection and index merging procedures are no longer integrated into index update operations (insert/delete), but are carried out in an asynchronous fashion.

Asynchronous index maintenance is realized by a low-priority background process that takes care of all disk-intensive index maintenance tasks. As this background process is not time-critical, it will get interrupted whenever the search engine has to respond to a document insertion/deletion or to an incoming search query, ensuring low query response time and low update latency.

Equipping the search engine with the ability to perform asynchronous index maintenance requires the following changes to its update procedures:

1. During an insert operation: When the indexing process runs out of memory, it creates a new on-disk index partition, but does not perform a merge operation. Instead, it starts a background process that takes care of merging index partitions (if necessary).

2. During a delete operation: If the critical amount of garbage postings ρ is exceeded, the search engine starts a background process that removes all garbage postings from the index.

3. The background process periodically checks for other system activity and immediately puts itself to sleep if such activity is detected.

4. The index maintenance procedures write data to the file system in very small chunks (a few hundred kilobytes). They do so without making use of the operating system's disk caching facilities (by opening files in mode `O_DIRECT` or `O_SYNC`).

Points 1 and 2 are immediately obvious from the meaning of the word "asynchronous".

Point 3 is necessary because most operating systems, by default, do not perform real-time CPU scheduling. Hence, assigning the background process a low priority is not sufficient to guarantee that other processes will always be given preference over it. This is particularly so because the background process exhibits a very high disk I/O level, and many operating systems will dynamically increase the priority level of such a process, employing techniques like multi-level feedback queues.

Point 4, finally, is needed because the operating system usually serves write requests in an asynchronous manner, buffering data in memory and writing them to disk at a later point. Unfortunately, this means that the background process can interfere with other system activity even after it has been suspended. By writing data to disk in small chunks, it is guaranteed that they will not delay disk operations required by index updates or search queries for too long. By performing only synchronized write operations, the background process never performs a disk access after it has detected other system activity — minimizing its impact on other processes.

Experiments

I tested the search engine's ability to perform real-time updates by splitting the command sequence from Figure 6.15 into an update sequence (containing `@addfile` and `@removefile` commands) and a query sequence (containing `@rank` commands). Commands from the update sequence were assumed to arrive in the system according to an exponential distribution with mean $\mu_{\mathrm{U}} = 100$ ms (5 insertions and 5 deletions per second). Commands from the query sequence were assumed to arrive according to an exponential distribution with mean $\mu_{\mathrm{Q}} = 5000$ ms (one query every 5 seconds).

After building a complete index for the TREC collection, three processes were started: a search engine process, an update process, and a query process. The update and query process's sole purpose was to send the commands from each sequence to the search engine – at pre-defined points in time – and to measure the time it took the search engine to respond to each command. The engine was configured to maintain its on-disk index structures according to *Asynchronous Immediate Merge*, and to allow 1 update operation and 4 query operations to be carried out concurrently (multi-threaded).

The results of this experiment are depicted in Figure 6.7. Plot 6.7(a) shows that, as with synchronous index maintenance, the majority of all update operations (81.9%) takes less than 100 ms to be carried out. On average, an update operation (insert/delete) is integrated into the index structures after 119 ms (standard deviation: 287 ms).

Compared to the synchronous case (cf. Figure 6.2), in which the search engine managed to carry out 94.5% of all update operations in less than 100 ms, the percentage achieved here (81.9%) may seem low. However, the experimental conditions here are different from the ones for Figure 6.2. Synchronous index maintenance assumes that an update event always takes place *after* the previous event has been processed. When modeling the arrival of update events according to an exponential distribution, this is no longer the case. In fact, with $\mu_{\mathrm{U}} = 100$ ms, the time difference between two consecutive update events U_i and U_{i+1} is less than 40 ms in 33% of all cases:

$$\int_0^{0.04} \lambda \cdot e^{-\lambda \cdot x} dx \;\approx\; 0.33. \tag{6.36}$$

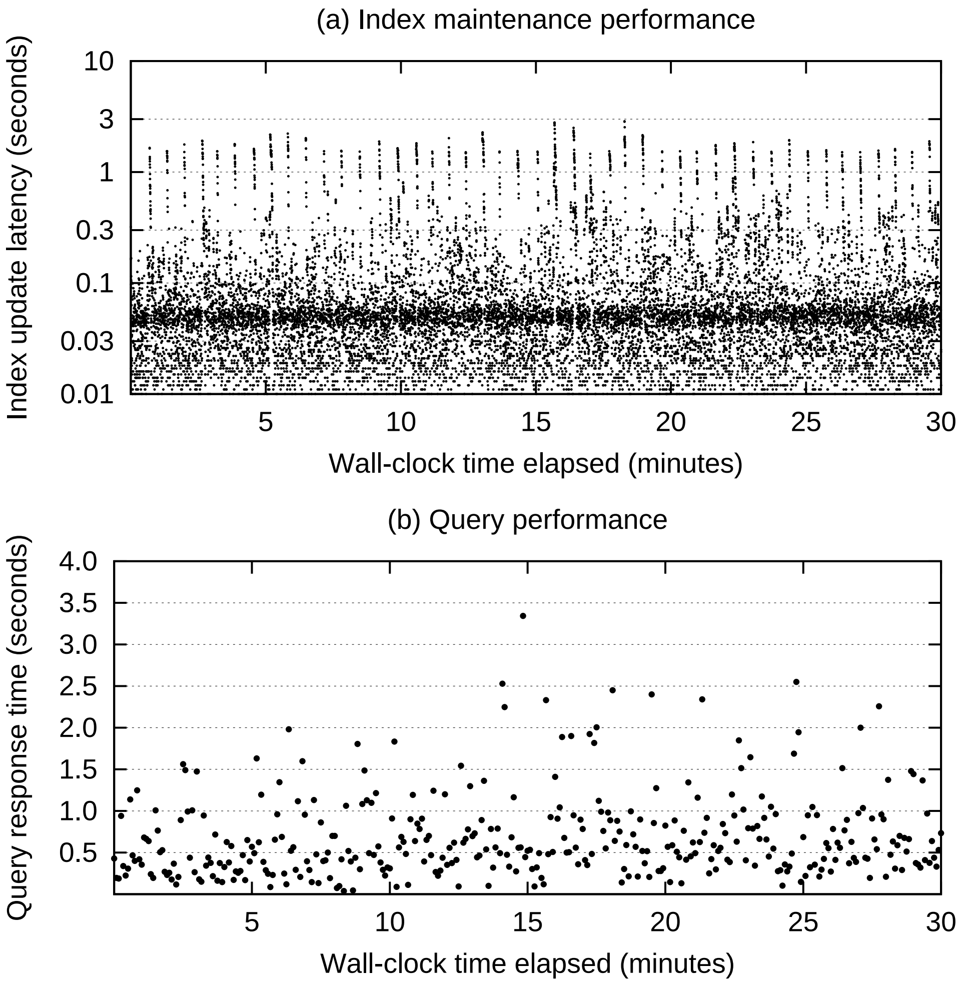


Figure 6.22: Real-time index updates. Performing 10 index updates and 0.2 query operations per second on a full index for the TREC collection. Index maintenance activity (merging partitions, collecting garbage postings) takes place in a low-priority background process.

Whenever this happens, achieving a response time below 100 ms becomes very difficult. Assuming a raw service time of 70 ms per update operation, event U_{i+1} would need to wait 30 ms before it could even be forwarded to the search engine (it cannot be processed until U_i has been finished), leading to a total expected response time of at least 100 ms.

If asynchronous index maintenance is used, then the maximum time required to perform an update operation no longer depends on the time it takes to carry out a physical index update. Out of the 18,000 update events that took place during the 30-minute experiment, not a single one required more than 3 seconds until it was completely processed by the search engine and all changes had been propagated to the index structures.

Plot 6.7(b) shows that search queries – submitted to the engine concurrently with update commands, and competing with update and index maintenance operations for disk and CPU – can still be processed very quickly. 81.7% of all queries are processed in under 1 second. Only 3.1% require more than 2 seconds. The average query response time is 686 ms (standard deviation: 508 ms).

Adaptivity

As a last comment, it is worth pointing out that with the rather high update arrival rate ($\mu_{\mathrm{U}} = 100$ corresponds to 5 insertions per second and thus a total of 134 GB of new text per day), the search engine's index maintenance strategy (Immediate Merge) is no longer able to keep up with the rate at which new index data arrive. On average, the search engine's update buffers are exhausted after approximately 200 files insertion, or 40 seconds. A merge operation takes at least 80 seconds — even longer if executed with integrated garbage collection (cf. Table 6.4). Therefore, we encounter the interesting situation in which, after running out of memory the first time, the indexing process runs out of memory a second time before the background process has finished its merge operation.

Fortunately, this does not cause any problems. The foreground index update task simply creates a new index partition from the in-memory data, without invoking a new merge operation. Thus, effectively, the search engine temporarily switches from its original Immediate Merge strategy to a No Merge policy, by increasing the number of index partitions until the background process has finished its merge update cycle. Therefore, if the update load becomes too high, the background process will never run, and the search engine will automatically switch from its original strategy, targeting high query performance, to the more update-efficient No Merge.

This shows that the index maintenance strategies presented in this chapter can in fact be used to provide support for real-time index updates, even when the system's update load is very high. By realizing all disk-intensive update tasks in a background process, their original *amortized* complexity can effectively be transformed into actual *worst-case* performance guarantees (assuming that the system is not permanently under full load, but its idle time is sufficient for the indexing process to transfer the current in-memory index to disk whenever it runs out of memory).

It also shows that index partitioning schemes like Logarithmic Merge, although designed with high update performance in mind, may in fact lead to higher query performance than Immediate Merge when used in an environment with real-time update constraints. This is because the situation in which the search engine runs out of memory a second time, before the current background merge operation could be finished, is more likely to occur with Immediate Merge than with Logarithmic Merge. Therefore, potentially, Immediate Merge could lead to a larger number of index partitions than Logarithmic Merge — at least temporarily.

6.8 Discussion

This chapter provides an overview of techniques that can be used to realize high-performance index updates with low amortized complexity. It covers the three main types of update events seen in dynamic collections: insertions, deletions, and append operations.

I evaluated various index maintenance strategies, in-place and merge-based. The evaluation included index partitioning schemes, such as Logarithmic Merge, that represent interesting alternatives to the traditional in-place and re-merge update policies with their strictly contiguous on-disk inverted lists.

With respect to the quest for the optimal index maintenance strategy for incremental text collections, I have shown that neither candidate (in-place or merge-based) alone deserves the title "best strategy". Instead, a combination of the two approaches, *hybrid index maintenance*, leads to the best results. Under high query load, when on-disk posting lists need to be stored in a contiguous fashion, Hybrid Immediate Merge with contiguous posting lists (HIM_C) is the method of choice, as it dominates Immediate Merge (same query performance, but substantially better update performance). Under high update load, on the other hand, when index maintenance performance is more important than query performance, Hybrid Logarithmic Merge with non-contiguous posting lists (HLM_NC or $\text{HLM}_{\text{NC(A)}}$) is a very attractive strategy, as it leads to better update performance than Logarithmic Merge and better query performance than No Merge. Unfortunately, I was not able to experimentally verify HLM_NC's theoretically suggested dominance over Logarithmic Merge.

Support for document deletions blends nicely into the incremental index update strategies, thanks to the secure query processing framework from Chapter 5. Postings referring to deleted documents can be removed from the index in a delayed fashion by executing a garbage collection procedure. This procedure can either be integrated into regular update tasks (on-the-fly garbage collection) or carried out separately, whenever the amount of garbage postings exceeds a predefined threshold. For optimal performance, a combination of both approaches is necessary.

All techniques presented may relatively painlessly be modified to operate at a low priority level in the background, resulting in a search engine that supports actual real-time index updates. When the search engine runs out of memory, the in-memory index is transferred to disk in a synchronous fashion. However, possible merge operations resulting from the creation of the new on-disk inverted file, are carried out asynchronously, by a separate process.

It should be possible to reduce the worst-case update latency even further, beyond what is shown in Section 6.7, by also delegating the construction of the new on-disk inverted file to a background process. In order for this to be possible, the search engine would need to reserve a certain amount of main memory that can be used to start building a new in-memory index (accumulating postings for incoming documents) while the previous in-memory index is being transferred to disk. Query processing complexity would increase slightly, because posting lists would potentially need to be fetched from two different in-memory indices instead of only one. On the other hand, by employing this strategy, it should be possible to guarantee sub-second update response times (assuming a reasonable update arrive rate).

In addition to all their other benefits, the need for real-time updates is another strong argument in favor of index partitioning schemes, such as Logarithmic Merge. Most structural components required by index partitioning (processing queries from two or more inverted files; merging a set of inverted files) have to be present in the search engine anyway, in order to support real-time index updates. Hence, the implementational overhead caused by supporting those schemes is negligible, and there really is no excuse any more for not supporting them.

A remaining facet that has not been covered yet is the degradation of compression effectiveness. By removing postings from the index, and not re-claiming the index address space that these postings used to occupy, the search engine creates gaps in the address space. These gaps need to be "jumped across" by the Δ-based postings compression scheme. For example, if a random 50% of all postings are removed from the index, then this may increase the compression inefficiency of the remaining postings by up to 1 bit per posting.

Fortunately, the actual degradation seen in practice is less than 1 bit per posting, because many terms exhibit a strong tendency to form clusters. That is, a random occurrence of a term T is likely to be followed by another occurrence of T, within the same document. Such clusters are unaffected by the gaps introduced outside the document. Nonetheless, gaps will keep piling up as more and more data are deleted from the index. One possible way to deal with this problem is to integrate an *address space compaction mechanism* into the garbage collection procedure. In addition to removing postings referring to deleted documents, the garbage collector would then also adjust the value of the remaining postings, so as to re-claim unused portions of the index address space.

Chapter 7

Interaction with the Operating System

A last remaining aspect of file system search that has not yet been addressed is how the search engine should interact with other components of the computer system that it runs on. For example, in order to update its index in a timely manner, the search engine somehow needs to be notified about changes to the file system whenever they take place. Also, the resulting I/O-intensive index update operations need to be scheduled in a reasonable way, so as to keep their impact on the performance of other processes running on the same computer as low as possible. These and other aspects of file system search are the topic of this chapter.

In Section 7.1, I examine several file system notification services for GNU/Linux. I discuss the existing notification infrastructure found in the Linux kernel (`dnotify` and `inotify`) and argue that there is a set of file system events that are crucial for efficient file system search, but that are not currently supported by any major operating system.

Section 7.2 discusses the problems arising from the presence of many different file types in the file system. The greater the set of file types supported by the search engine, the greater is its utility to the user. At the same time, unfortunately, supporting a large variety of different file formats also increases the implementation efforts of the search engine developers.

Section 7.3 focuses on some temporal aspects of file systems. I show that the indexable contents of a file system are typically heavily skewed towards very old files that rarely, if ever, get changed. I argue that this aspect should be taken into account when building the index.

I finish this chapter with a brief discussion of index update scheduling policies and suspend-resume index maintenance strategies necessary to react to a sudden system shutdown that might interrupt an ongoing index maintenance operation (Section 7.4).

7.1 File System Notification

In order to be able to update its index structures in real time, the search engine needs a hook into the operating system that allows it to be notified of all changes that are happening in the file system. Such event notification services are provided by most operating systems. In my discussion of existing notification systems, I limit myself to the case of Linux.

7.1.1 Essential File System Events

Obviously, the search engine must be able to be notified of basic file system events, such as file creations, file deletions, name changes, and modifications to a file's set of access permissions. However, a few more, not so obvious, requirements exist:

1. The search engine has to be notified of all changes in the file system without specifically registering for a particular directory. A notification interface that requires explicit per-directory registration necessarily leads to a long start-up delay (scanning the entire file system when the computer is turned on) and also to a number of race conditions that arise when the contents of a newly created directory change before the search engine has a chance to register for changes in that directory.

2. When the contents of a file are modified, it is not sufficient to merely inform the search engine that a change has happened. In order to appropriately react to partial documents modifications (e.g., append operations), the search engine also needs to know what exact parts of the file are affected by that change.

3. There needs to be a way to notify the search engine that a user is about to remove a storage device from the file system. Typically, the search system will keep a small number of files open for every mount point (so that it does not need to open/close its index files whenever it has to process a search query). If it is not notified of such a user intent, it will not close its files, and the device may not be removed from the system without the risk of data loss.

None of these requirements are met by the current notification interface(s) provided by the Linux kernel. In Windows, requirement 1 is met by the `FindFirstChangeNotification` system call (and related functions), but requirements 2 and 3 are not fulfilled in Windows, either.

7.1.2 Event Notification in Linux: `dnotify` and `inotify`

File system event notification in Linux is traditionally realized through the interface provided by `dnotify`. With `dnotify`, a user process can register to be notified about all changes taking place in a particular directory by obtaining a handle to the directory and issuing a special `fcntl` command. Thereafter, whenever the contents of the directory are changed, the operating system kernel sends a signal to the process, notifying it of the event.

The main limitation of `dnotify` is that a process needs to maintain an open file handle for every directory that it is watching. This design decision was motivated by the originally intended use of `dnotify`, aiming at graphical user interfaces, such as file managers, that need to monitor a small number of directories in order to update the information displayed to the user. In the context of file system search, however, the indexing process needs to watch an entire file system, containing thousands of different directories. This is likely to be impossible, due to a limited number of open file handles per process supported by the operating system.

Apart from the directory in which a file system event took place, `dnotify` provides virtually no detailed information about the nature of an event. For instance, if a file is moved from directory A to directory B, then this results in two different events. With Linux's preemptive kernel, however, the two events are not necessarily received by the search engine right after another, but may instead be separated by some other events. Hence, moving a file from A to B will cause the search engine to re-index the file instead of just updating some meta-data.

Finally, `dnotify` sends event notifications to registered processes in the form of system signals, a rather ugly and inconvenient way of inter-process communication.

Many of the shortcomings of `dnotify` were eliminated with the advent of `inotify`[1], the now-standard notification system in Linux. `inotify` has been in development since 2004 and became part of the official Linux kernel with the release of Linux 2.6.13 in August 2005. With `inotify`, a process no longer needs to keep an open file handle for every directory that it is watching. After opening a given directory and sending a registration request to the kernel, the directory may be closed. Hence, the number of directories that can be watched by a process is no longer limited by the number of open file handles.

`inotify` also provides more information to user-space processes than `dnotify`. For example, events are time-stamped so that, for a file relocation from directory A to directory B, the two corresponding events, in A and B, can be re-connected by the search engine.

Regrettably, some of the limitations of `dnotify` are shared by `inotify`. For example, `inotify` does not provide any information about which part of a file is changed by a write

[1]http://www.kernel.org/pub/linux/kernel/people/rml/inotify/README (accessed 2007-04-11)

operation. In the case of system log files, this means that the entire log file needs to be re-parsed (and potentially re-indexed) whenever a single line has been added to the file.

More importantly, however, `inotify` still requires the watching process to register for each watched directory separately — exacting an exhaustive scan of the entire file system during system start-up. This design decision is mainly motivated by security considerations: a user process should only be notified about events taking place in directories for which it has read permission. By forcing the process to first open a directory before it can register for event notification, the operating system can easily make sure that the process has the necessary permissions. Unfortunately, the concept is only a half-hearted attempt to provide a secure notification interface. For example, when the access permissions of a directory (or one of its ancestors in the file system tree) are changed after a process has registered for changes, the process will still be notified of all events in that directory, even though it might no longer possess the read privilege for it.

7.1.3 An Alternative to `inotify`: `fschange`

Because the need to explicitly and individually register for each watched directory is such a fundamental limitation of the existing event notification infrastructure, it makes sense to think about alternatives. `fschange`[2] is such an alternative. It was designed with the specific needs of file system search and file-system-wide backup systems in mind. In these applications, it is not necessary to verify the calling process's access privileges when it registers for notification. The respective user-space process always runs with super-user privileges and has permission to read every directory in the file system.

Instead of enforcing security restrictions on a per-directory basis, like `inotify`, `fschange` provides complete event information about the entire file system through a file in the `/proc` file system. When a process reads from this file, it receives event descriptions for all events that changed the status of the file system. Prior registration for specific directories is not necessary. The file, however, can only be accessed by a process with super-user privileges, ensuring that the file system's access permissions are not violated by the notification service.

`fschange` provides most of the event types supported by `dnotify` and `inotify`, plus some additional ones. Figure 7.1 shows a sequence of user actions affecting the state of the file system (left-hand side) and the corresponding event lines generated by `fschange` (right-hand side). Some details worth pointing out:

[2] http://stefan.buettcher.org/cs/fschange/ (accessed 2007-04-11)

```
[root@stu x]# pwd
/mnt/x
[root@stu x]# cat > testfile
1234567890
[root@stu x]# cat >> testfile
1234567890
[root@stu x]# mv testfile filetest
[root@stu x]# rm filetest
[root@stu x]# umount /mnt/x
umount: /mnt/x: device is busy
umount: /mnt/x: device is busy
[root@stu x]# cd ..
[root@stu mnt]# umount /mnt/x
```

```
[root@stu ~]# cat /proc/fschange
CREATE /mnt/x/testfile
WRITE /mnt/x/testfile 0 11
WRITE /mnt/x/testfile 11 22
RENAME /mnt/x/testfile /mnt/x/filetest
UNLINK /mnt/x/filetest
UMOUNT_REQ /mnt/x
UMOUNT_REQ /mnt/x
UMOUNT_REQ /mnt/x
UMOUNT /mnt/x
CREATE /etc/mtab.tmp
TRUNCATE /etc/mtab.tmp 0
CHMOD /etc/mtab.tmp 420
WRITE /etc/mtab.tmp 0 437
CHOWN /etc/mtab.tmp 0 0
RENAME /etc/mtab.tmp /etc/mtab
```

Figure 7.1: File system event notification with `fschange`. Left-hand side: User actions. Right-hand side: `fschange` events.

- A `WRITE` event includes information about the first and the last byte inside the file that were affected by the operation. This information can be used by the search engine to react to append operations in an appropriate way.
- Changing the name of a file (or its location in the file system) is treated as a single event, and there is no need for a time-stamping facility that could be used to re-connect two separate events that originated from the same user action.
- An unsuccessful `umount` command ("device is busy") results in an `fschange` event of the type `UMOUNT_REQ`. The search engine, when it sees this event, can close all its open files on the given device.

The third point addresses the unfortunate side-effect of the search engine's maintaining open files on all active storage devices. By modifying the `umount` command so that, after an unsuccessful attempt to unmount a device, it waits a few seconds and then tries again, it is possible to allow the search engine to flush all its pending data to disk and close its open files on the respective device. An unsuccessful `unmount` attempt can be easily detected by the search engine from the file access pattern observed by processing the `fschange` event stream. In such a case, the engine can re-open its data files for the given mount point and resume its normal operation.

In the context of file system search, `fschange` solves all of the shortcomings of Linux's existing notification interface. As it does not take user permissions into account, however, it only complements `dnotify` and `inotify` instead of replacing them.

7.2 Dealing with Different File Types

A major implementational hurdle in the development of a file system search engine is the necessity to support a large number of different file types. Not only does the indexing process need to be able to extract indexable text data from a variety of input file formats; the search engine also needs to be able to display the contents of each file in a meaningful fashion when presenting search results to a user.

This problem is bigger than it seems at first, as there is no standardized set of libraries that can be used to extract the indexable data from a given file. For this reason, the first release of Google's desktop search[3], for example, did not include support for Adobe's PDF file format, severely limiting the search engine's utility to many users.

The ability to obtain a textual representation of every file in the file system is such a fundamental exigency, not only for file system search, but also for other applications, that it seems reasonable to integrate it into the core file system.

A possible approach could be the one envisioned by Hans Reiser[4,5] or by the proponents of the semantic file system [50]. Within these frameworks, the distinction between files and directories no longer exists. A file can be accessed like a directory, allowing a process to effortlessly access various attributes or components of the file. For example, the artist of an MP3 audio file `britney.mp3` could be obtained by reading from `britney.mp3/artist`. Similarly, the raw textual representation of a PDF document could be obtained through `thesis.pdf/text`. An application that knows how to interpret a certain file type (e.g., Adobe Acrobat) can register with the operating system to provide a file system plug-in that may be used to extract the desired information from a given file. This way, it is no longer necessary to develop a large number of independent text extraction procedures, one for each file type, or to rely on non-standardized libraries when providing support for various (proprietary) file types.

7.3 Temporal Locality in File Systems

In Section 4.1, I considered the spatial aspect of a file system — the fact that a typical UNIX file system consists of more than just a single storage device. I argued that this fact should be taken into account by dividing the file system's content index into several sub-indices, one for each storage device.

[3]http://desktop.google.com/ (accessed 2007-04-10)
[4]http://www.namesys.com/v4/v4.html (accessed 2007-04-27)
[5]http://www.namesys.com/whitepaper.html (accessed 2007-04-27)

File type	**Last modification**			
	less than 1 day	1–7 days	7–30 days	older
Plain text & source code	200	36	525	140,836
HTML	39	103	744	109,758
XML	6	0	0	26,263
PDF	8	3	25	1,549
Postscript	5	1	1	1,649
`man` pages	0	0	0	5,771
E-mail files	7	6	5	22,289
Total	0.1%	0.0%	0.4%	99.4%

Table 7.1: File age statistics for SBFS. More than 99% of all indexable files in the file system are older than 30 days. The category *e-mail files* contains individual e-mail messages, but also e-mail folders in the `mbox` format.

In addition to the spatial aspect, a file system also has a number of temporal aspects. Files can be young or old, and old files are less likely to be changed than younger ones. Ramakrishnan et al. [88] report that, in a typical office work environment with shared files, fewer than 10% of all files in the file system are accessed during a given day. About 50% of the file operations logged in Ramakrishnan's experiments were file modifications or deletions. Since there is usually a significant overlap between the files accessed during two subsequent days, it is unlikely – even over a longer period, like one week – that more than 20% of the files in the file system are modified.

Ousterhout et al. [83] report that about 50% of all newly-created information is deleted or overwritten within 5 minutes. This can be explained by a large number of temporary files (e.g., intermediate files produced by a compiler) and also by typical user behavior: a user tends to periodically save the file that she is working on, even when she has not finished editing the file yet. Therefore, many of the write operations seen in Ramakrishnan's experiments are likely to refer to the same files, implying that the total number of files modified during a typical workday is far less than 5% of all files in the file system.

This claim is further supported by the file age statistics for the SBFS collection that are shown in Table 7.1. It can be seen that more than 99% of all indexable files in the file system are older than 30 days. Many of those files are part of the operating system environment, such as `man` pages (nearly 6,000 files in `/usr/share/man`) or other system documentation files (e.g., 25,000 files in various formats in `/usr/share/doc`). However, the majority of these files are in fact user-generated (or downloaded) content that has accumulated over the years.

All these files need to be indexed, because they might be targeted by a legitimate search query. However, only a small fraction of all files are expected to change. This can be taken into account by maintaining two different indices per storage device. One index, I_Y, is used to store information about all young files in the file system, perhaps younger than a week. The other index, I_O contains information about old files. Most file system changes are confined to young files. Index maintenance operations, therefore, are predominantly limited to the relatively small index I_Y and can be carried out much more efficiently than before. I_O can be updated periodically, maybe once per week, without causing any substantial performance problems.

Dividing the index into two parts also allows the search engine to apply different index compression techniques to the individual indices. For instance, I_O might be compressed using a slower, but more space-efficient algorithm than the one used for I_Y. I_O's inverted lists are decompressed and re-compressed rather infrequently, and choosing a compression scheme with a more complicated encoding routine than vByte no longer represents a performance bottleneck.

7.4 Scheduling Index Updates

Requirement 4 from Chapter 1 states that the search engine's indexing activity should not interfere with the normal operation of other processes in the system. Typical desktop search engines meet this requirement by pausing their indexing process as soon as they detect any system activity. After a short period of system inactivity, usually between 10 and 30 seconds, the indexing task is resumed.

In Section 6.7, I have shown that it is possible to delegate index maintenance operations to a low-priority background process so that index updates and search queries can still be processed at a very high rate, unaffected by the search engine's shoveling around large amounts of index data. The techniques employed to achieve real-time index updates can easily be adjusted so that the search engine's indexing activity does not interfere with other processes. Thus, pausing the indexing process for 15–30 seconds in many cases is not necessary. This aspect is especially important, since typical user-interactive applications, such as word processors, are CPU-bound with very short periods of CPU bursts, while the search engine's indexing task is largely I/O-bound (cf. Section 4.3.2). It is therefore unlikely to interfere with such applications.

What is more interesting than how to interact with other active processes in the system is how to react to a sudden system shutdown. Suppose the search engine, at some point, needs to initiate a garbage collection process because the amount of garbage postings in the index has reached a predefined threshold. Suppose the time necessary to complete this process is about an hour, but the computer is only turned on for short periods of time, perhaps 30 minutes.

If the search engine employs a merge-based maintenance strategy, then it will have a hard time ever finishing the garbage collection procedure. If, instead, it maintains its index structures following an in-place approach or hybrid index maintenance (cf. Section 6.4), then it is possible to apply the garbage collection mechanism to each inverted list individually. This is another argument in favor of hybrid index maintenance, because it enables the search engine to interrupt the garbage collector at any given point in time, without compromising the consistency of the on-disk index structures. With a purely merge-maintained index, this is also possible. It would, however, require rather complicated check-pointing mechanisms to guarantee index consistency and to make the garbage collector properly resume its work after a system shutdown.

Chapter 8

Conclusion

This thesis has covered a large variety of issues that arise in the context of multi-user file system search. Taken together, the techniques that I proposed in the previous chapters can be used to implement a search engine that

- exhibits low memory requirements;
- respects user permissions and always returns search results that are consistent with the file access privileges defined by the file system;
- updates its index structures in real time, usually within a few seconds after a file system change is reported to the search engine.

All methods have been implemented and are part of the publicly available Wumpus[1] search engine. The methods have been thoroughly evaluated, to prove their effectiveness in the context of dynamic multi-user information retrieval systems.

Contributions

The main contributions of this thesis are:

- I developed a way to model some statistical properties of inverted files, including the number of posting lists below a certain length threshold, and the number of postings contained in such lists. The model, based on the assumption of a Zipfian term distribution, was shown to yield an accurate description of the lists in an inverted index. It was used several times in this thesis, to theoretically demonstrate the usefulness of various techniques for creating, updating, and organizing an inverted index.

[1]http://www.wumpus-search.org/ (accessed 2007-04-30)

- The interleaved dictionary organization from Section 4.4.2 brings the memory consumption of the index's lookup data structure to a very low level, sub-linear in the size of the text collection. This is important in the context of file system search, where memory resources are scarce. It is also fundamental for the efficiency of the index update strategies in Chapter 6. With the chosen data structure, dictionary maintenance, typically a non-trivial problem in the design of any update strategy (cf. Lester [66, Chapter 7]), essentially comes for free, since the dictionary now is part of each inverted file.

- In Chapter 5, I quantified the amount of information that may leak to an unauthorized user if the search engine follows a postprocessing approach to impose security restrictions upon its search results. I showed that an arbitrary user, by carefully designing search queries and analyzing the order in which search results are returned by the search engine, can obtain a very accurate approximation of the global term statistics of the file system, including information about files that cannot be read by the user.

- To cure this deficiency, I developed a secure query processing framework, based on the GCL retrieval model. I discussed possible performance optimizations within this new framework and showed that, in addition to its security advantages, it also leads to improved query processing performance compared to the postprocessing approach.

- The hybrid index maintenance strategies from Section 6.4 represent an interesting family of index update policies, applicable on both ends of the *updates vs. queries* workload spectrum. Hybrid Immediate Merge with contiguous posting lists (HIM_C) exhibits the same query performance as Immediate Merge, but achieves considerably lower update complexity. Hybrid Logarithmic Merge with non-contiguous posting lists (HLM_NC), based on a theoretical analysis, was shown to lead to lower update complexity than Logarithmic Merge. This result was confirmed through experiments. HLM_NC's query performance is only slightly weaker than that of Logarithmic Merge, implying that the method represents a useful trade-off point between No Merge and Logarithmic Merge.

  HIM_C is one of the rare cases where one index maintenance strategy actually dominates another strategy and should always be preferred over it. Regardless of the relative query and update load of the system, HIM_C is always better than Immediate Merge.

- I introduced the concept of *partial flushing* (Section 6.3.2), a method that transfers only *some* posting lists (instead of all of them) to disk when the search engine runs out of main memory. Partial flushing is motivated by the Zipfian distribution of terms in a text collection, where the majority of all postings is found in a rather small number of posting lists. It can be used in conjunction with any in-place index update policy (including hybrid strategies). The method was shown to exhibit a greatly reduced index update overhead compared to an already highly optimized performance baseline. In my experiments, its update performance was up to 44% higher than that of HIM_C without partial flushing.

- I proposed a set of techniques that can be used to deal with document deletions and modifications, two aspects that are usually ignored when dynamic text collections are discussed in the literature.

  With respect to file deletions (Section 6.5), I showed that a two-tiered garbage collection policy, in conjunction with the secure query processing framework from Chapter 5, is an appropriate instrument to control the amount of garbage postings in the index and keep both query and update performance at a high level.

  In the context of file modifications (Section 6.6), I pointed out the main difficulties stemming from the necessity to support such operations. Limiting myself to the case of append operations, I proposed a proportional address space pre-allocation strategy, similar in spirit to proportional pre-allocation in in-place index update, that results in low update overhead and that does not increase the search engine's main memory consumption.

- My discussion of real-time index updates in Section 6.7 has shown that a small number of fairly simple modifications to the search engine are sufficient to transform the amortized complexity of the index maintenance techniques examined in Chapter 6 into real-time index updates. I am not aware of any previous discussion of actual real-time updates in the context of text retrieval systems.

- Finally, in Chapter 7, I presented a discussion of index update problems that are specific to file system search. In particular, I argued that there is a set of file system event notifications that are crucial to real-time index updates and that are currently not supported by any event notification system. As an alternative to the existing Linux notification infrastructure, I proposed `fschange`, a Linux kernel module that provides a process with fine-grained information about all content-changing file system events, without the need to register for specific directories prior to notification.

Applicability in Other Contexts

Many of the techniques presented in this thesis are not only applicable to file system search, but also to problems that arise in other contexts, both inside and outside the field of information retrieval.

For example, real-time index updates are an attractive feature for every search engine, particularly in such genuine real-time environments as newswire services and on-line auctioning environments. Likewise, applying tight security restrictions to the search results produced by the search engine is important in many scenarios outside of file system search; for instance, in enterprise search and in any other collaborative work environment.

Variations of the dictionary interleaving scheme proposed in Section 4.4.2 can be used for every disk-based data repository storing variable-size objects. In fact, a similar approach is taken by many file system implementations (such as `ext2`[2]) that try to keep each file's meta-data (i-node information) close to the actual file contents so that disk seeks between file data and meta-data are less costly. Similarly, the incremental update mechanisms from Sections 6.2 and the hybrid update strategies from 6.4 are of possible interest to any application that needs to maintain a large number of dynamic, variable-size records.

Limitations

In all my experiments, I have assumed that main memory is a scarce resource and thus that the majority of all index data need to be held in secondary storage. In my experiments, secondary storage is realized by hard disk drives, and the operational characteristics of hard drives, in particular the very high cost of random access operations, influenced the design of many of the data structures presented in this thesis. For example, if disk seeks came for free, then there would be no need for the interleaved dictionary structure from Section 4.4.2. Moreover, if fragmented posting lists could be accessed without a severe performance penalty, then even the most simple index maintenance strategy, NO MERGE, might lead to good performance results.

Therefore, if novel storage devices with negligible random access overhead, like flash memory and solid-state disks, ever become mainstream enough to replace hard disk drives as the standard means of data storage in computer systems, then many of the techniques proposed in this thesis will need to be re-evaluated — just like the changing behavior of hard drives over the last 15 years required a re-evaluation of the index update methods proposed in the early 1990s. However, as I already pointed out in Chapter 3, this is unlikely to happen any time

[2]http://e2fsprogs.sourceforge.net/ext2.html (accessed 2007-04-29)

soon, as solid-state storage devices are still astronomically expensive, compared to traditional magnetic storage.

Future Work

There are several open questions that have not been answered in this thesis. Some are related to the core algorithms of the search engine; others are concerned with its integration into the operating system.

First and foremost, optimization strategies for complex GCL queries have not been studied very well yet. The retrieval framework presented in this thesis provides full support for all GCL retrieval operators and thus allows the user (or a third-party application) to express a wide variety of search constraints. However, a comprehensive theory that could be used to perform query optimizations within this framework does not exist. This became apparent in Section 5.6, where I employed the (rather simplistic) optimization heuristic to move user-specific security restriction operators as far up in the GCL operator tree as possible. For the disjunctive Boolean queries in keyword search (and in my experiments), this seems to be a reasonable strategy. However, there are cases where different arrangements lead to better performance results. Consider, for instance, the GCL query

$$(\text{“peanuts”} \wedge \text{“butterflies”}) \ntriangleright [4]$$

(i.e., “peanuts” and “butterflies” within at most 3 words) and its secure version

$$(\ (\text{“peanuts”} \wedge \text{“butterflies”}) \ntriangleright [4]\ ) \triangleleft F_U.$$

It is quite possible that the user U is an avid admirer of butterflies, but does not have access to the secret peanut knowledge stored in the file system. In that case, the search engine might spend a long time identifying all matches for

$$(\text{“peanuts”} \wedge \text{“butterflies”}) \ntriangleright [4],$$

only to find out that none of them lies within F_U. By re-writing the GCL expression to

$$(\ ((\text{“peanuts”} \triangleleft F_U) \wedge \text{“butterflies”}) \ntriangleright [4]\ ) \triangleleft F_U,$$

the query could have been processed far more efficiently. The optimal GCL operator tree seems to be closely related to the selectivity of the security restriction predicate F_U at each node in the tree, but it would be nice to have a proper theory at one's command, to guide the search engine through the query re-writing procedure.

Another open question is concerned with index consistency issues and the question of how the search engine should deal with spontaneous system failures. From the rather simplistic index structure employed by the index partitioning schemes from Section 6.2.1, it seems that recovering from a system crash might not be infeasible. In particular, the search engine's ability to carry out index maintenance operations in the background implies that there will always be a coherent set of on-disk index partitions, even after a crash.

Nonetheless, it would be nice to have actual robustness guarantees, also involving the auxiliary components of the search engine, such as its security-related meta-data. One approach that seems especially promising is to integrate the search engine into the file system's journalling activity. A journal entry then would only be cleared after the change has made it into the persistent part of the search engine's index structures. When recovering from a crash, the journal would not only be used to restore the file system's consistency, but also to allow the search engine to re-obtain the index information it lost due to the system failure.

This aspect leads to a final question regarding the relationship of file system and search engine. In all my considerations, I have always assumed that the search engine is running in a user-space process, outside the core operating system. What if the retrieval system itself became part of the operating system, and its index structures part of the file system? This would facilitate the realization of some interesting features, such as the ability to let the index data for a particular file count towards the file owner's disk quota, which would be rather difficult to implement outside the operating system kernel.

Bibliography

[1] Vo Ngoc Anh and Alistair Moffat. Index Compression Using Fixed Binary Codewords. In *Proceedings of the Fifteenth Conference on Australasian Database*, pages 61–67, Dunedin, New Zealand, January 2004.

[2] Vo Ngoc Anh and Alistair Moffat. Inverted Index Compression using Word-Aligned Binary Codes. *Information Retrieval*, 8(1):151–166, January 2005.

[3] Vo Ngoc Anh and Alistair Moffat. Pruned Query Evaluation Using Precomputed Impacts. In *Proceedings of the 29th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 372–379, Seattle, USA, August 2006.

[4] Vo Ngoc Anh and Alistair Moffat. Pruning Strategies for Mixed-Mode Querying. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, pages 190–197, Arlington, USA, November 2006.

[5] Peter Bailey, David Hawking, and Brett Matson. Secure Search in Enterprise Webs: Tradeoffs in Efficient Implementation for Document Level Security. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, pages 493–502, Arlington, USA, 2006.

[6] Holger Bast, Christian Worm Mortensen, and Ingmar Weber. Output-Sensitive Autocompletion Search. In *Proceedings of the 13th International Conference on String Processing and Information Retrieval*, pages 150–162, Glasgow, Scotland, UK, October 2006.

[7] Holger Bast and Ingmar Weber. Type Less, Find More: Fast Autocompletion Search With a Succinct Index. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 364–371, Seattle, USA, August 2006. ACM Press.

[8] Rudolf Bayer. Binary B-Trees for Virtual Memory. In *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control*, pages 219–235, San Diego, USA, November 1971.

[9] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1:173–189, 1972.

[10] Jon L. Bentley and James B. Saxe. Decomposable Searching Problems I: Static-to-Dynamic Transformations. *Journal of Algorithms*, 1(4):301–358, 1980.

[11] Jon Louis Bentley and Andrew Chi-Chih Yao. An Almost Optimal Algorithm for Unbounded Searching. *Information Processing Letters*, 5(3):82–87, 1976.

[12] Deepavali Bhagwat and Neoklis Polyzotis. Searching a File System Using Inferred Semantic Links. In *Proceedings of the Sixteenth ACM Conference on Hypertext and Hypermedia*, pages 85–87, Salzburg, Austria, September 2005.

[13] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[14] George Boole. The Calculus of Logic. *The Cambridge and Dublin Mathematical Journal*, 3:183–198, 1848.

[15] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast Incremental Indexing for Full-Text Information Retrieval. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB 1994)*, pages 192–202, Santiago de Chile, Chile, September 1994.

[16] Nader H. Bshouty and Geoffrey T. Falk. Compression of Dictionaries via Extensions to Front Coding. In *Proceedings of the Fourth International Conference on Computing and Information (ICCI 1992)*, pages 361–364, Washington, DC, USA, 1992. IEEE Computer Society.

[17] Stefan Büttcher and Charles L. A. Clarke. A Security Model for Full-Text File System Search in Multi-User Environments. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST 2005)*, pages 169–182, San Francisco, USA, December 2005.

[18] Stefan Büttcher and Charles L. A. Clarke. Indexing Time vs. Query Time Trade-offs in Dynamic Information Retrieval Systems. In *Proceedings of the 14th ACM Conference on*

Information and Knowledge Management, pages 317–318, Bremen, Germany, November 2005.

[19] Stefan Büttcher and Charles L. A. Clarke. Indexing Time vs. Query Time Trade-offs in Dynamic Information Retrieval Systems. University of Waterloo Technical Report CS-2005-31, October 2005.

[20] Stefan Büttcher and Charles L. A. Clarke. Memory Management Strategies for Single-Pass Index Construction in Text Retrieval Systems. University of Waterloo Technical Report CS-2005-32, October 2005.

[21] Stefan Büttcher and Charles L. A. Clarke. A Hybrid Approach to Index Maintenance in Dynamic Text Retrieval Systems. In *Proceedings of the 28th European Conference on Information Retrieval*, pages 229–240, London, UK, April 2006.

[22] Stefan Büttcher, Charles L. A. Clarke, and Brad Lushman. Term Proximity Scoring for Ad-Hoc Retrieval on Very Large Text Collections. In *Proceedings of the 29th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 621–622, Seattle, USA, August 2006.

[23] Stefan Büttcher, Charles L. A. Clarke, and Ian Soboroff. The TREC 2006 Terabyte Track. In *Proceedings of the 15th Text REtrieval Conference (TREC 2006)*, Gaithersburg, USA, November 2006.

[24] Ben Carterette and Fazli Can. Comparing Inverted Files and Signature Files for Searching a Large Lexicon. *Information Processing and Management*, 41(3):613–633, 2005.

[25] Tzi-cker Chiueh and Lan Huang. Efficient Real-Time Index Updates in Text Retrieval Systems. Technical report, SUNY at Stony Brook, NY, USA, August 1998.

[26] Kenneth Church, William Gale, Patrick Hanks, and Donald Hindle. Using Statistics in Lexical Analysis. In Uri Zernik, editor, *Lexical Acquisition: Exploiting On-Line Resources to Build a Lexicon*, pages 115–164, 1991.

[27] David R. Clark and J. Ian Munro. Efficient Suffix Trees on Secondary Storage. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1996)*, pages 383–391, Atlanta, USA, January 1996.

[28] Charles L. A. Clarke and Gordon V. Cormack. Shortest-Substring Retrieval and Ranking. *ACM Transactions on Information Systems*, 18(1):44–78, January 2000.

[29] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. Fast Inverted Indexes with On-Line Update. Technical report, University of Waterloo, Waterloo, Canada, November 1994.

[30] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. Schema-Independent Retrieval from Heterogeneous Structured Text. Technical report, University of Waterloo, November 1994.

[31] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An Algebra for Structured Text Search and a Framework for Its Implementation. *The Computer Journal*, 38(1):43–56, 1995.

[32] Charles L. A. Clarke, Gordon V. Cormack, Derek I. E. Kisman, and Thomas R. Lynam. Question Answering by Passage Selection. In *Proceedings of the 9th Text REtrieval Conference*, Gaithersburg, USA, November 2000.

[33] Charles L. A. Clarke, Gordon V. Cormack, and Thomas R. Lynam. Exploiting Redundancy in Question Answering. In *Proceedings of the 24th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 358–276, New Orleans, USA, November 2001.

[34] Charles L. A. Clarke, Nick Craswell, and Ian Soboroff. Overview of the TREC 2004 Terabyte Track. In *Proceedings of the 13th Text REtrieval Conference*, Gaithersburg, USA, November 2004.

[35] Douglass R. Cutting and Jan O. Pedersen. Optimization for Dynamic Inverted Index Maintenance. In *Proceedings of the 13th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 405–411, New York, USA, September 1990.

[36] Owen de Kretser and Alistair Moffat. Effective Document Presentation with a Locality-Based Similarity Heuristic. In *Proceedings of the 22nd ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 113–120, Berkeley, USA, 1999.

[37] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive Set Intersections, Unions, and Differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 743–752, San Francisco, USA, January 2000.

[38] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Experiments on Adaptive Set Intersections for Text Retrieval Systems. In *Revised Papers from the Third International Workshop on Algorithm Engineering and Experimentation (ALENEX 2001)*, pages 91–104, Washington, DC, USA, January 2001.

[39] Susan Dumais, Edward Cutrell, JJ Cadiz, Gavin Jancke, Raman Sarin, and Daniel Robbins. Stuff I've Seen: A System for Personal Information Retrieval and Re-use. In *Proceedings of the 26th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 72–79, Toronto, Canada, 2003.

[40] P. Elias. Universal Codeword Sets and Representations of the Integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, 1975.

[41] Christos Faloutsos and Raphael Chan. Fast Text Access Methods for Optical and Large Magnetic Disks: Designs and Performance Comparison. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases (VLDB 1988)*, pages 280–293, Los Angeles, USA, September 1988.

[42] Christos Faloutsos and Stavros Christodoulakis. Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation. *ACM Transactions on Information Systems*, 2(4):267–288, 1984.

[43] G. Fernandez and L. Allen. Extending the UNIX Protection Model with Access Control Lists. In *Proceedings of USENIX*, pages 119–132, San Francisco, USA, 1988.

[44] Edward A. Fox and Whay C. Lee. FAST-INV: A Fast Algorithm for Building Large Inverted Files. TR-91-10. Technical report, Virginia Polytechnic Institute and State University, Blacksburg, USA, March 1991.

[45] Aviezri S. Fraenkel and Shmuel T. Klein. Novel Compression of Sparse Bit-Strings — Preliminary Report. *Combinatorial Algorithms on Words, NATO ASI Series*, 12:169–183, 1985.

[46] Valery I. Frants, Jacob Shapiro, Isak Taksa, and Vladimir G. Voiskunskii. Boolean Search: Current State and Perspectives. *Journal of the American Society for Information Science*, 50(1):86–95, 1999.

[47] William Gale and Geoffrey Sampson. Good-Turing Frequency Estimation Without Tears. *Journal of Quantitative Linguistics*, 2:217–237, 1995.

[48] R. Gallager and D. van Voorhis. Optimal Source Codes for Geometrically Distributed Integer Alphabets. *IEEE Transactions on Information Theory*, 21(2):228–230, March 1975.

[49] Alexander Gelbukh and Grigori Sidorov. Zipf and Heaps Laws' Coefficients Depend on Language. In *Proceedings of the Second Conference on Intelligent Text Processing and Computational Linguistics (CICLing-2001)*, pages 332–335, 2001.

[50] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James O'Toole. Semantic file systems. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles (SOSP 1991)*, pages 16–25, October 1991.

[51] Solomon W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, IT-12:399–401, July 1966.

[52] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. *New Indices for Text: PAT Trees and PAT Arrays*, pages 66–82. Information Retrieval: Data Structures and Algorithms. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[53] Donna Harman, R. Baeza-Yates, Edward Fox, and W. Lee. Inverted Files. In *Information Retrieval: Data Structures and Algorithms*, pages 28–43, Upper Saddle River, USA, 1992.

[54] Herbert Stanley Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, New York, USA, 1978.

[55] Steffen Heinz and Justin Zobel. Efficient Single-Pass Index Construction for Text Databases. *Journal of the American Society for Information Science and Technology*, 54(8):713–729, June 2003.

[56] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst Tries: A Fast, Efficient Data Structure for String Keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.

[57] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. In *Proceedings of the Institute of Radio Engineers*, volume 40, pages 1098–1101, September 1952.

[58] F. Jelinek and R. Mercer. Interpolated Estimation of Markov Source Parameters from Sparse Data. In E. S. Gelsema and L. N. Kanal, editors, *Pattern Recognition in Practice*, pages 381–402, Amsterdam, The Netherlands, 1980.

[59] Karen Spärck Jones, Steve Walker, and Stephen E. Robertson. A Probabilistic Model of Information Retrieval: Development and Comparative Experiments (parts 1 and 2). *Information Processing and Management*, 36(6):779–840, 2000.

[60] Jyrki Katajainen and Erkki Mäkinen. Tree Compression and Optimization with Applications. *International Journal of Foundations of Computer Science*, 1(4):425–448, December 1990.

[61] Jon M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998)*, pages 668–677, San Francisco, California, United States, 1998.

[62] Wessel Kraaij, Thijs Westerveld, and Djoerd Hiemstra. The Importance of Prior Probabilities for Entry Page Search. In *Proceedings of the 25th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 27–34, Tampere, Finland, August 2002.

[63] John D. Lafferty and ChengXiang Zhai. Document Language Models, Query Models, and Risk Minimization for Information Retrieval. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 111–119, New Orleans, USA, September 2001.

[64] D. Y. Lee, J. Hwang, and G. N. Bae. Effect of Disk Rotational Speed on Contamination Particles Generated in a Hard Disk Drive. *Microsystem Technologies*, 10(2):103–108, 2004.

[65] Joon Ho Lee. Properties of Extended Boolean Models in Information Retrieval. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 182–190, Dublin, Ireland, 1994.

[66] Nicholas Lester. *Efficient Index Maintenance for Text Databases.* PhD thesis, RMIT University, Melbourne, Australia, 2006.

[67] Nicholas Lester, Alistair Moffat, William Webber, and Justin Zobel. Space-Limited Ranked Query Evaluation Using Adaptive Pruning. In *Proceedings of the 6th International Conference on Web Systems Information Systems Engineering*, pages 470–477, New York, USA, 2005.

[68] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast On-Line Index Construction by Geometric Partitioning. In *Proceedings of the 14th ACM Conference on Information and Knowledge Management*, pages 776–783, Bremen, Germany, November 2005.

[69] Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems. In *Proceedings of the 27th Conference on Australasian Computer Science*, pages 15–23, Dunedin, New Zealand, 2004.

[70] Nicholas Lester, Justin Zobel, and Hugh E. Williams. Efficient Online Index Maintenance for Text Retrieval Systems. *Information Processing & Management*, 42:916–933, July 2006.

[71] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ramesh Agarwal. Dynamic Maintenance of Web Indexes Using Landmarks. In *Proceedings of the 12th International World Wide Web Conference (WWW 2003)*, pages 102–111, Budapest, Hungary, May 2003.

[72] Erkki Mäkinen. A Survey on Binary Tree Codings. *The Computer Journal*, 34(5):438–443, 1991.

[73] M. Mallary, A. Torabi, and M. Benakli. One Terabit Per Square Inch Perpendicular Recording Conceptual Design. *IEEE Transactions on Magnetics*, pages 1719–1724, July 2002.

[74] Udi Manber. Finding Similar Files in a Large File System. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Francisco, USA, January 1994.

[75] Udi Manber and Eugene W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[76] Udi Manber and Sun Wu. GLIMPSE: A Tool to Search Through Entire File Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 23–32, January 1994.

[77] Donald Metzler, Victor Lavrenko, and W. Bruce Croft. Formal Multiple-Bernoulli Models for Language Modeling. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 540–541, Sheffield, United Kingdom, 2004.

[78] Patrick C. Mitchell. A Note about the Proximity Operators in Information Retrieval. In *Proceedings of the 1973 Meeting on Programming Languages and Information Retrieval (SIGPLAN 1973)*, pages 177–180, Gaithersburg, USA, 1973.

[79] Alistair Moffat and Timothy C. Bell. In-Situ Generation of Compressed Inverted Files. *Journal of the American Society of Information Science*, 46(7):537–550, 1995.

[80] Alistair Moffat and Justin Zobel. Compression and Fast Indexing for Multi-Gigabyte Text Databases. *Australian Computer Journal*, 26(1):1–9, 1994.

[81] Alistair Moffat and Justin Zobel. Self-Indexing Inverted Files for Fast Text Retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.

[82] Spencer W. Ng. Advances in Disk Technology: Performance Issues. *The Computer Journal*, 31(5):75–81, May 1998.

[83] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles (SOSP 1985)*, pages 15–24, New York, USA, 1985.

[84] Mark H. Overmars and Jan van Leeuwen. Dynamization of Decomposable Searching Problems Yielding Good Worsts-Case Bounds. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 224–233, London, UK, March 1981. Springer-Verlag.

[85] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford Digital Library Technologies Project, 1998.

[86] Michael Persin, Justin Zobel, and Ron Sacks-Davis. Filtered Document Retrieval with Frequency-Sorted Indexes. *Journal of the American Society for Information Science*, 47(10):749–764, October 1996.

[87] Jay M. Ponte and W. Bruce Croft. A Language Modeling Approach to Information Retrieval. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 275–281, Melbourne, Australia, 1998.

[88] K. K. Ramakrishnan, Prabuddha Biswas, and Ramakrishna Karedla. Analysis of File I/O Traces in Commercial Computing Environments. In *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 78–90, New York, USA, 1992.

[89] Yves Rasolofo and Jacques Savoy. Term Proximity Scoring for Keyword-Based Retrieval Systems. In *Proceedings of the 25th European Conference on IR Research*, pages 207–218, Pisa, Italy, April 2003.

[90] Dennis M. Ritchie. The UNIX Time-Sharing System: A Retrospective. *Bell Systems Technical Journal*, 57(6):1947–1969, 1978.

[91] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, 1974.

[92] Stephen E. Robertson. The Probabilistic Character of Relevance. *Information Processing and Management*, 13(4):247–251, 1977.

[93] Stephen E. Robertson, C. J. van Rijsbergen, and Martin F. Porter. Probabilistic Models of Indexing and Searching. In *Proceedings of the Joint ACM/BCS Symposium on Information Storage and Retrieval (SIGIR 1980)*, pages 35–56, Cambridge, UK, June 1980.

[94] Stephen E. Robertson, Steve Walker, and Micheline Hancock-Beaulieu. Okapi at TREC-7. In *Proceedings of the Seventh Text REtrieval Conference*, Gaithersburg, USA, November 1998.

[95] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at TREC-3. In *Proceedings of the Third Text REtrieval Conference*, Gaithersburg, USA, November 1994.

[96] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *The Computer Journal*, 27(3):17–28, 1994.

[97] Airi Salminen and Frank William Tompa. Pat Expressions: An Algebra for Text Search. *Acta Linguista Hungarica*, 41:277–306, 1994.

[98] Gerard Salton and Michael E. Lesk. Computer Evaluation of Indexing and Text Processing. *Journal of the ACM*, 15(1):8–36, January 1968.

[99] Gerard Salton, A. Wong, and C. S. Yang. A Vector Space Model for Automatic Indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[100] Falk Scholer, Hugh E. Williams, J. Yiannis, and Justin Zobel. Compression of Inverted Indexes for Fast Query Evaluation. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 222–229, Tampere, Finland, August 2002.

[101] Ernst J. Schuegraf. Compression of Large Inverted Files with Hyperbolic Term Distribution. *Information Processing and Management*, 12(6):377–384, 1976.

[102] Wann-Yun Shieh and Chung-Ping Chung. A Statistics-Based Approach to Incrementally Update Inverted Files. *Information Processing and Management*, 41(2):275–288, 2005.

[103] Kurt A. Shoens, Anthony Tomasic, and Héctor García-Molina. Synthetic Workload Performance Analysis of Incremental Updates. In *Proceedings of the 17th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 329–338, Dublin, Ireland, January 1994.

[104] Z. K. Silagadze. Citations and the Zipf-Mandelbrot's Law. *Complex Systems*, 11:487–499, 1997.

[105] Fei Song and W. Bruce Croft. A General Language Model for Information Retrieval. In *Proceedings of the 8th International Conference on Information and Knowledge Management*, pages 316–321, Kansas City, USA, 1999.

[106] Ken Thompson. Programming Techniques: Regular Expression Search Algorithm. *Commununications of the ACM*, 11(6):419–422, June 1968.

[107] Anthony Tomasic, Héctor García-Molina, and Kurt Shoens. Incremental Updates of Inverted Lists for Text Document Retrieval. In *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, pages 289–300, New York, USA, 1994.

[108] Stephen Tomlinson. Experiments with the Negotiated Boolean Queries of the TREC 2006 Legal Discovery Track. In *Proceedings of the 15th Text REtrieval Conference (TREC 2006)*, Gaithersburg, USA, November 2006.

[109] Howard Turtle and James Flood. Query Evaluation: Strategies and Optimization. *Information Processing & Management*, 31(1):831–850, November 1995.

[110] Peter Weiner. Linear Pattern Matching Algorithm. In *14th Annual Symposium on Foundations of Computer Science*, pages 1–11, Iowa City, USA, October 1973.

[111] Eric W. Weisstein. Euler-Mascheroni Constant. From MathWorld – http://mathworld.wolfram.com/Euler-MascheroniConstant.html.

[112] Hugh E. Williams and Justin Zobel. Compressing Integers for Fast File Access. *The Computer Journal*, 42(3):193–201, 1999.

[113] Hugh E. Williams and Justin Zobel. Searchable Words on the Web. *International Journal on Digital Libraries*, 5(2):99–105, April 2005.

[114] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes (2nd ed.): Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers Inc., San Francisco, USA, 1999.

[115] Peter C. K. Yeung, Stefan Büttcher, Charles L. A. Clarke, and Maheedhar Kolla. A Bayesian Approach for Learning Document Type Relevance. In *Proceedings of the 29th European Conference on Information Retrieval (ECIR 2007)*, pages 753–756, Rome, Italy, April 2007.

[116] Chengxiang Zhai and John Lafferty. A Study of Smoothing Methods for Language Models Applied to Ad Hoc Information Retrieval. In *Proceedings of the 24th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 334–342, New Orleans, Louisiana, USA, 2001. ACM Press.

[117] George K. Zipf. *Human Behavior and the Principle of Least-Effort.* Addison-Wesley, Cambridge, USA, 1949.

[118] Justin Zobel, Steffen Heinz, and Hugh E. Williams. In-Memory Hash Tables for Accumulating Text Vocabularies. *Information Processing Letters*, 80(6):271–277, 2001.

[119] Justin Zobel and Alistair Moffat. Inverted Files for Text Search Engines. *ACM Computing Surveys*, 38(2):6, 2006.

[120] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted Files versus Signature Files for Text Indexing. *ACM Transactions on Database Systems*, 23(4):453–490, December 1998.