Execution Performance Issues in Full-Text Information Retrieval

Eric W. Brown

Technical Report 95-81
October 1995

Computer Science Department
University of Massachusetts at Amherst

EXECUTION PERFORMANCE ISSUES IN FULL-TEXT INFORMATION RETRIEVAL

A Dissertation Presented

by

ERIC WILLIAM BROWN

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

DOCTOR OF PHILOSOPHY

February 1996

Department of Computer Science

To Jennifer

ACKNOWLEDGEMENTS

The long road to a Ph.D. is never walked alone. Along the way many people have offered me encouragement, support, and guidance. To all of them I give my deepest thanks, especially

My advisor, Bruce Croft, for his insight, direction, perspective, and vision.

Eliot Moss and Tony Hosking, for teaching sound research principles and instilling a passion for experimental performance evaluation.

Jamie Callan, for tutoring me in information retrieval, fielding countless questions about INQUERY, and working as my closest colleague throughout this effort.

The rest of my dissertation committee, Howard Turtle and Graham Gal, for their comments and suggestions, which have greatly improved this dissertation.

The students and staff of the Object Systems Lab and Center for Intelligent Information Retrieval who have contributed to the systems used in this research.

My close friends and comrades, for provocative conversation, intellectual stimulation, welcome distraction, and occasional commiseration.

My family, for their understanding and patience.

My wife, Jennifer, who, more than anyone else, made this all possible with her encouragement, support, and love.

This research has been supported by the National Science Foundation Center for Intelligent Information Retrieval at the University of Massachusetts, Amherst.

ABSTRACT

EXECUTION PERFORMANCE ISSUES IN FULL-TEXT INFORMATION RETRIEVAL

FEBRUARY 1996

ERIC WILLIAM BROWN

B.Sc., UNIVERSITY OF VERMONT

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor W. Bruce Croft

The task of an information retrieval system is to identify documents that will satisfy a user's information need. Effective fulfillment of this task has long been an active area of research, leading to sophisticated retrieval models for representing information content in documents and queries and measuring similarity between the two. The maturity and proven effectiveness of these systems has resulted in demand for increased capacity, performance, scalability, and functionality, especially as information retrieval is integrated into more traditional database management environments.

In this dissertation we explore a number of functionality and performance issues in information retrieval. First, we consider creation and modification of the document collection, concentrating on management of the inverted file index. An inverted file architecture based on a persistent object store is described and experimental results are presented for inverted file creation and modification. Our architecture provides performance that scales well with document collection size and the database features supported by the persistent object store provide many solutions to issues that arise during integration of information retrieval into

more general database environments. We then turn to query evaluation speed and introduce a new optimization technique for statistical ranking retrieval systems that support structured queries. Experimental results from a variety of query sets show that execution time can be reduced by more than 50% with no noticeable impact on retrieval effectiveness, making these more complex retrieval models attractive alternatives for environments that demand high performance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Documents play a central role in our daily acquisition and distribution of information. They serve as both a medium and a repository for information, coming in a variety of shapes and sizes. Newspaper and magazine articles supply us with our daily news. Manuals instruct us in all sorts of activities. Letters enable us to correspond professionally and socially. Reports keep us current in the work and business of others. Over the ages people have produced an enormous wealth of documents. Today we continue to add to this wealth by perpetually generating new documents. With such an abundance of documents available, finding a particular document of interest can amount to a Herculean task.

To make this task feasible, information retrieval (IR) systems were developed. The function of an information retrieval system is to satisfy a user's information need by identifying the documents in a collection of documents that contain the desired information. Since the inception of IR systems over thirty years ago, a great deal of effort has been spent on improving the ability of IR systems to correctly identify interesting and relevant documents. In the work presented in this dissertation, we now concentrate on system implementation issues and, in particular, how to improve the execution performance of these systems so that their operations can be carried out quickly and efficiently.

In the remainder of this chapter, we provide an overview of the problems considered in this dissertation and the approaches taken to solving them, summarize the research conducted and the results achieved, describe the contributions of this work, and outline the rest of this dissertation.

1.1 Overview

A document is any written work that conveys information. Examples include books, reports, articles, and letters. The fundamental element of any document is text, the written form of human language. Text is a powerful mechanism for storing information, allowing us to record anything that can be expressed verbally. This power comes from the endless variety and flexibility of human language. When creating text, we have a huge vocabulary of terms at our disposal and infinitely many ways of combining those terms to express what we wish to communicate.

While this flexibility makes for rich and interesting documents, it has the potential to impede human understanding of the information stored in a document. Documents can be long and detailed, requiring careful study before their true information content is discovered. This situation is acceptable when the set of documents that we must examine is restricted to those that contain the information we seek. But what if we have a large number of documents and do not know which ones contain the desired information? Individual inspection of each document is impractical. Without a method for identifying relevant documents, a large collection of information-rich documents is useless.

One of the first solutions to this problem appeared nearly four thousand years ago when catalogues of documents in libraries were created to aid in keeping track of those documents [44]. A catalogue provides a compact listing of the documents available in the library. Each document entry in the catalogue includes some number of attributes for the respective document, such as author, title, or subject. The attributes can be used to identify potentially interesting documents without actually having to examine the documents themselves.

More recently, in the 16th century, primitive *indexes* for documents were created. An index is a list of certain keywords or topics. Each entry in the list contains pointers into the documents where descriptions and discussions of the respective keyword or topic may be found. Unfortunately, deciding what keywords and topics should go into an index and

which discussions are worthy of an index pointer is a tedious and subjective human task prone to omissions. Ultimately, both indexes and catalogues suffer from the restriction that an information search must be based on a set of limited, predetermined document characteristics, i.e., the keywords of an index or the attributes of a catalogue.

A different kind of index that avoids this shortcoming is the *concordance*. A concordance is an alphabetical list of all of the terms that appear in a collection. For each term, the list gives a pointer to every occurrence of the term in the collection, along with a portion of the text surrounding the term to suggest the context of the occurrence. The full-text index provided by a concordance is free from the restrictions of predetermined keywords and can be used to locate all of the passages that contain the terms of interest. A concordance for a large document such as the Bible, however, might require a good portion of a lifetime to construct by hand, and such an effort can take a significant toll on the concordance compiler. In the case of Alexander Cruden, author of one of the better known Bible concordances [24] (first published in 1737), the effort involved in compiling the concordance is believed to have led to his insanity [48].

With the advent of the computer age in the latter half of the 20th century, concordance construction could be automated, greatly simplifying the task. What used to take years could now be accomplished in minutes. In spite of being relatively complete and simple to construct, a concordance still provided a rather unsophisticated solution to our original problem. Trying to locate information in a large collection of documents using a concordance can be an exercise in frustration, leading to the retrieval of many unrelated documents that just happen to contain terms that we believe are indicative of the information we seek. A more intelligent solution to the problem at hand was still needed.

Over thirty years ago, work towards this intelligent solution began with the birth of *information retrieval* systems. Information retrieval is the process of identifying and retrieving relevant documents based on some expressed interest in documents of a particular nature. The distinguishing characteristic of information retrieval is that the search for

interesting documents is based on the information *content* of the documents, rather than just the terms, keywords, or attributes associated with the document. To support document searching based on information content, an information retrieval system consists of three basic elements: a document representation, a query representation, and a measure of similarity between queries and documents. The document representation provides a formal description of the information contained in the documents, the query representation provides a formal description of the information need, and the similarity measure defines the rules and procedures for matching the information need with the documents that satisfy that need.

These three elements collectively define a retrieval model. Research in information retrieval has produced a number of retrieval models, of which the three most prominent are the Boolean, vector-space, and probabilistic retrieval models. In all of these models, a document is represented by a set of indexing *features* that have been assigned to the document. Indexing features are commonly the terms that occur in the document collection, although they may also be more semantically meaningful concepts extracted from the text by sophisticated indexing methods (e.g., citations, phrases). Unless further distinction is necessary, we will use the word "term" to mean any indexing feature.

In Boolean retrieval, a document is represented as a set of terms $d_j = \{t_1, \ldots, t_k\}$, where each t_i is a term that appears in document d_j. A query is represented as a Boolean expression of terms using the standard Boolean operators *and*, *or*, and *not*. A document matches the query if the set of terms associated with the document satisfies the Boolean expression that represents the query. The result of the query is the set of matching documents.

The vector-space model [73] enhances the document representation of the Boolean model by assigning a weight to each term that appears in a document. A document can then be represented as a vector of term weights. The number of dimensions in the vector-space is equal to the number terms used in the overall document collection, or $\mid T \mid$, where T is the set of terms used in the collection, commonly referred to as the *vocabulary* or *lexicon*.

The weight of a term in a document is calculated using a function of the form $tf \cdot idf$, where *tf* (term frequency weight) is a function of the number of occurrences of the term within the document, and *idf* (inverse document frequency weight) is an inverse function of the total number of documents that contain the term. The first component incorporates the notion that the ability of a term to describe a document's content is directly related to the number of times the term occurs within that document. The second component incorporates the notion that a term's discriminatory power weakens as the term appears in more and more documents.

A query in the vector-space model is treated as if it were just another document, allowing the same vector representation to be used for queries as for documents. This naturally leads to the use of the vector inner product as the measure of similarity between the query and a document. This measure is typically normalized for vector length, such that the actual similarity measure is the cosine of the angle between the two vectors. After all of the documents in the collection have been compared to the query, the documents are sorted by decreasing similarity measure and a ranked listing of documents is returned as the result of the query.

The probabilistic retrieval model is based on the Probability Ranking Principle, which states that an information retrieval system is most effective when it responds to an expressed information need with a list of documents ranked in decreasing order of probability of relevance, and the probabilities are estimated as accurately as possible given all of the available information [70]. In this model, the answer to a query is generated by estimating $P(relevant \mid d)$ (the probability of the information need being satisfied given document d) for every document, and ranking the documents according to these estimates. Using Bayes' theorem, $P(relevant \mid d)$ can be expressed as a function of the probabilities of the terms in d appearing in relevant and non-relevant documents. The query gives an estimate for the probability of a given term appearing in relevant documents, and the document collection gives an estimate for the probability of a given term appearing in non-relevant documents.

This results in a *tf* · *idf* style term weighting function, similar to that used in the vector-space model. The probabilistic version, however, is more formally motivated.

Although these models differ in many of their details, they each incorporate the belief that a query and its relevant documents will have terms in common. An important query evaluation step for all of these models is matching query terms with the documents that contain those terms. Scanning the document collection for occurrences of the query terms is an unsatisfactory implementation of this step, especially when the document collection is quite large. Instead, an *inverted file* index [73, 29, 42] is used to support this process. An inverted file contains an inverted list for every term that appears in the document collection. A term's inverted list identifies all of the documents that contain the corresponding term. Each document entry in an inverted list may additionally contain a term weight for the document (often just the number of occurrences of the term within the document) and the locations of each occurrence of the term within the document.

Using an inverted file, we match query terms to documents by obtaining the inverted lists for the query terms and processing the document entries in those lists. The particular retrieval model will dictate exactly what information is stored in the inverted lists and how that information is used in the query evaluation process. Regardless of how the inverted list contents are used, the fundamental advantage of an inverted file is that the set of documents that must be considered during the query evaluation process is constrained to those that contain at least one of the query terms. Moreover, the documents in this constrained set do not even need to be accessed during query evaluation. All of the information required to evaluate a query can be stored in the inverted lists, such that a document need only be accessed when the user selects it from the query result list for viewing.

The issues of what information to store in an inverted list and how to use that information to generate a query result are at the heart of the question that most of the IR research to date has focused on: how to define the elements of a retrieval model for best retrieval effectiveness. Retrieval effectiveness is a measure of an IR system's ability to correctly

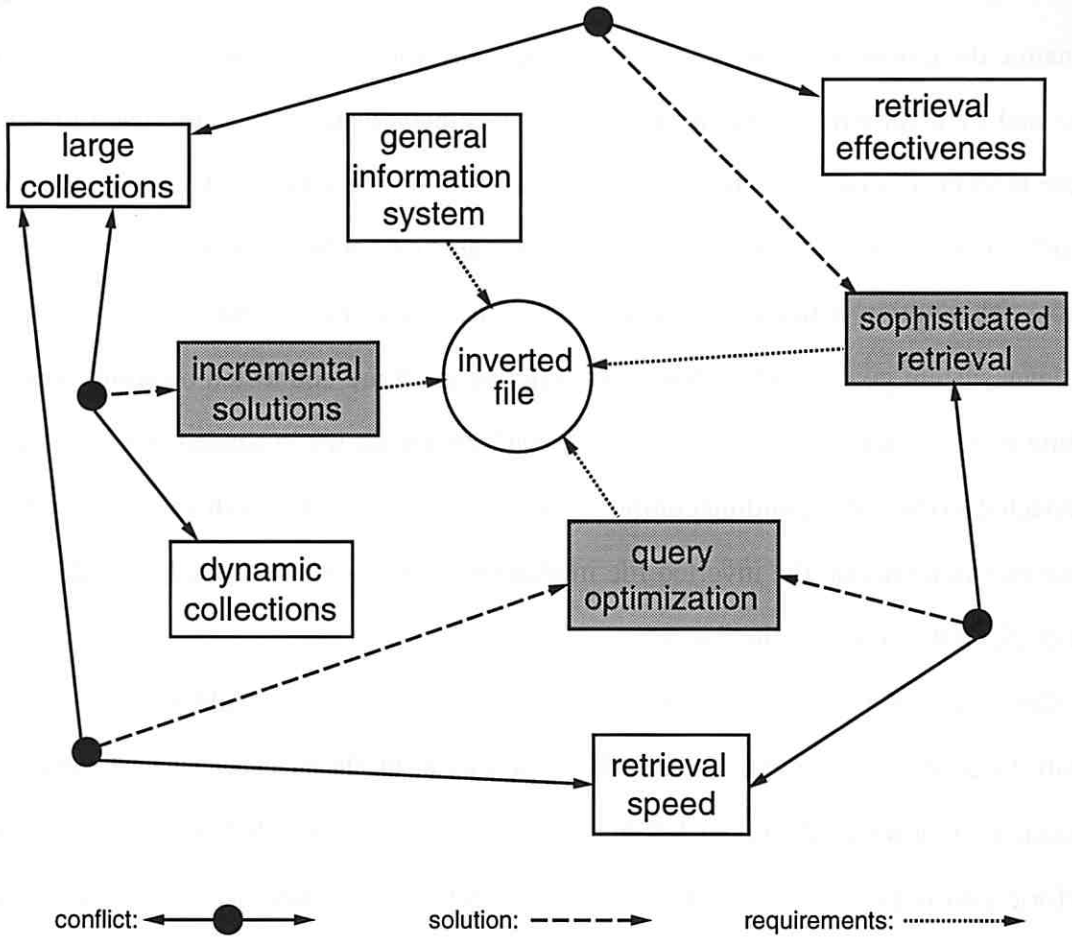


Figure 1.1 Inverted file issues

identify the documents that are relevant to a given query. While improving retrieval effectiveness remains an important area of research, a number of new challenges have appeared that are rapidly becoming much more pressing. First, IR systems are being asked to manage larger and larger document collections. Second, the traditional view of document collections as static and archival is being replaced by the desire for dynamic collections that can be updated efficiently or built incrementally. Third, information retrieval is being integrated into more comprehensive information management systems. For this to happen, IR systems must provide reliable, efficient, multi-user access—features common to more traditional data management systems.

The goal of this dissertation is to provide solutions to the challenges created by large, dynamic document collections, and to lay the foundation for a solution to the challenges imposed by a comprehensive information management system. Our approach to solving these problems is based on the following observation: the speed and functionality of an information retrieval system are determined to a large extent by the inverted file implementation. This notion is depicted in Figure 1.1. The figure shows a number of information retrieval system goals in white boxes. Conflicting goals are connected by solid lines emanating from a black dot. Solutions to these conflicts are shown in shaded boxes, which are connected to the corresponding conflict by a dashed line. Finally, both solutions and goals place requirements on the inverted file implementation, shown as dotted lines directed at the circle in the center of the figure.

Consider first the goal of retrieval effectiveness in the upper right hand corner. For small document collections, a simple Boolean model might suffice. On large document collections (shown in the upper left hand corner), however, simple boolean retrieval will perform poorly [72, 1, 85]. To resolve the conflict between these two goals and provide better retrieval effectiveness on large document collections, we turn to more sophisticated retrieval models. Sophisticated retrieval models place additional requirements on the inverted file implementation, such as storage of term weights and occurrence locations.

Both large document collections and sophisticated retrieval conflict with the goal of fast document retrieval, shown at the bottom of Figure 1.1. These two conflicts lead to the use of query optimization techniques to improve retrieval speed. Query optimization techniques can require alternative inverted file access methods and storage of additional information in the inverted lists, placing further requirements on the inverted file implementation.

The goal of supporting a dynamic document collection (shown in the lower left hand corner) conflicts with the goal of supporting a large document collection. If a document collection is small enough, modifications to the collection can be incorporated into the inverted file simply by re-indexing the entire document collection from scratch. With

larger document collections, this solution is impractical. Instead, incremental solutions are required that allow in-place modifications of the existing inverted file. The functionality requirements imposed by a dynamic inverted file introduce a whole new set of issues that must be considered in the inverted file implementation.

The last goal depicted in Figure 1.1 is the incorporation of information retrieval into a general information management system, shown in the top middle of the figure. For example, a traditional database management system (DBMS) provides excellent support for structured, record based data. However, a DBMS provides only limited support for text data types and generally lacks the sophisticated full-text search capabilities provided by an IR system. Combining these two technologies into a single, comprehensive system will result in a more powerful and useful information management system.

Before this integration can take place, an IR system must meet the data management standards set by the DBMS. A large part of the functionality provided by a DBMS is support for consistent, reliable multi-user access and update of the database. This is accomplished through the use of transactions, concurrency control, and recovery—features typically absent from an IR system. The incorporation of these mechanisms into an IR system will have a significant impact on the inverted file implementation, imposing a variety of new functionality requirements for controlled access and manipulation of the inverted lists.

The above observations lead to a problem solving approach centered on the inverted file implementation. This dissertation presents a comprehensive solution to managing an inverted file that either directly satisfies the requirements stated above, or enables other new strategies to be applied in the problem solving effort. Since we are concerned with execution performance issues, the solution is fully implemented and evaluated empirically. The experimental test-bed is provided by INQUERY [12], a full-text probabilistic information retrieval system based on a Bayesian inference network model [88]. INQUERY was chosen for the following reasons:

- INQUERY uses a general inverted file that includes term occurrence locations, allowing exploration of more complex inverted list data structures. This exploration would not be possible in a system that stores term weights only in its inverted file.

- The inference network-based retrieval model exemplifies the sophisticated retrieval solution of Figure 1.1.

- The inference network-based retrieval model provides a general framework in which a variety of retrieval models can be represented, suggesting that results obtained in this environment have a better likelihood of generalizing to other retrieval models.

- INQUERY has been shown to provide a high level of retrieval effectiveness [39, 40], increasing the impact of the results presented in this dissertation. A fast system is useless if it provides poor retrieval effectiveness.

- INQUERY is a commercial quality system and is currently used in a number of installations [21], again increasing the impact of the results presented in this dissertation.

1.2 Research Summary

The research conducted for this dissertation covers two main areas: indexing and query evaluation. Indexing includes the initial creation, modification, and overall management of the inverted file. The specific indexing problems addressed are:

1. Efficient inverted file creation for large document collections.

2. Efficient additions of new documents to an existing document collection.

3. Design of an overall architecture that enables solutions to the first two problems and provides a foundation for future work on the comprehensive information management system problem.

These problems have a strong systems orientation, focusing on the management of large amounts of data that must be moved back and forth between disk and main memory. As such, any approach to solving these problems must be sensitive to basic computer architecture issues and tradeoffs. In particular, the size and access characteristics of the data to be managed must be taken into account when deciding how to make use of various computer resources (e.g., CPU, disk, main memory). With these considerations in mind, the following hypotheses are put forth:

1. Fast, scalable document indexing can be achieved by localizing sort and insertion operations, building intermediate results in main memory, minimizing I/O, and favoring sequential I/O over random I/O.

2. Document additions can be efficiently supported by an inverted list data structure that minimizes access to the existing inverted file during the update.

3. A general, "off-the-shelf" data management system can be used to manage an inverted file if the data management system provides the appropriate data model and extensibility mechanisms.

A general document indexing scheme based on the previous work of Witten et al. [90] was implemented. The extension to their work is a double buffering scheme for parsing documents and building inverted lists in main memory without the use of a term dictionary. The overall indexing scheme is able to index documents at a rate of over 500 MB an hour on a current, midrange workstation, and results show that the technique scales well with document collection size. The issues identified in the first hypothesis were considered throughout the implementation, and the results obtained lead to the acceptance of that hypothesis.

An exploration of possible solutions to the problem of managing an inverted file was conducted, leading to the conclusion that a persistent object store provides the appropriate level of performance and functionality for this task. In particular, the Mneme persistent

object store [62] was used as the "off-the-shelf" data management system in the inverted file architecture. The data model provided by Mneme allowed the design of an inverted list data structure that meets the requirements stated in the second hypothesis. Experimental results show that the new inverted file architecture supports document additions with costs significantly less dependent on the size of the existing document collection than traditional techniques, which require redundant indexing of the document collection or scanning of the entire existing inverted file. Moreover, additions in the new implementation are performed in-place, substantially reducing temporary disk space costs. These results confirm the second hypothesis, although there is still room for improvement.

Other inverted file management tasks were explored within the context of the Mneme based architecture. While many of these additional features have been implemented, including document deletions, concurrency control, recovery, and transactions for multi-user access, a full evaluation of these features is beyond the scope of this dissertation. The implementation of these features, however, does lead to the acceptance of the third hypothesis above.

A single problem was addressed within the context of query evaluation, namely, how to provide fast evaluation of *structured queries* in statistical ranking retrieval systems. Retrieval systems of this kind are characterized by a statistical or probabilistic term weighting function and a query language that provides a variety of query operators for combining term weights, proximity information, and the results of nested query operators. A structured query can be represented as a tree with operators at the internal nodes and terms at the leaves. During query evaluation, a document's score is calculated by propagating term weights for the document from the leaves toward the root, combining the term weights according to the semantics of the query operators at the internal nodes to produce a final score for the document at the root of the query tree.

A technique for reducing query evaluation costs can be categorized as either *safe* or *unsafe*. A safe technique has no impact on retrieval effectiveness, while an unsafe technique

may trade retrieval effectiveness for execution efficiency. A number of safe optimization techniques were explored, including their implications for the inverted file implementation and expected impact on query evaluation time. The main focus here, however, was on unsafe optimization techniques. Our research was guided by the following observation: relevance scores are generated for a significant percentage of the documents in a document collection when evaluating a query. This observation has been made by others. Moffat and Zobel [58] found that for queries containing around 40 terms, nearly 75% of the documents in the collection are scored. Even relatively short queries suffer from this problem. We have observed that for queries containing around 8 terms, 35% of the documents in the collection are scored. If the document collection contains 1 million documents, hundreds of thousands of documents will be scored, far exceeding the number of documents an end user is likely to be interested in. In light of this, the following hypothesis is put forth:

- The set of documents to score, called the *candidate document set*, can be significantly constrained with minimal effort, which in turn will produce a significant savings in query evaluation execution time.

A new optimization technique was developed based on this hypothesis. The technique populates the candidate document set in a light-weight preprocessing step using heuristics to select the documents most likely to be relevant to the query. These documents are then fully scored to generate the answer to the query. An evaluation of the new optimization technique on large document collections using a variety of query sets showed that the candidate document set can be reduced by over 90%. This in turn translates into a savings in wall-clock execution time of over 50%, proving the above hypothesis. Furthermore, retrieval effectiveness is maintained in the portion of the query result most likely to be viewed by the end user.

The new optimization technique was also compared to and combined with a previously proposed optimization technique, *term-elimination*. While the individual techniques perform comparably on certain query sets, our new technique was shown to be more robust in

all situations. Moreover, the two techniques are complementary, such that combining them yields an additional improvement in performance.

Finally, the efficacy of applying the new optimization technique and term-elimination on short, unstructured queries was evaluated, and the usefulness of high frequency (i.e., low *idf*) query terms was explored. It was found that high frequency query terms can often be eliminated to yield substantial improvements in both execution speed *and* retrieval effectiveness. While this is gratifying, it is actually indicative of a problem in the retrieval model, suggesting that high frequency terms are not being handled properly. Appropriate query modifications were explored to better incorporate high frequency query term information into final document scores. These efforts led to a better understanding of both the impact of high frequency query terms, and which techniques provide the best combination of retrieval effectiveness and execution speed.

1.3 Research Contributions

The contributions of this thesis work are primarily practical in nature, with implications for information retrieval system implementation. The contributions include:

- Implementation and evaluation of a fast, scalable indexing system.
- Design and implementation of an inverted file management architecture using "off-the-shelf" data management technology, providing opportunities for all aspects of an information retrieval system to benefit from traditional database management features, such as buffer management and efficient low-level storage management.
- Development and evaluation of an incremental indexing strategy enabled by the above architecture.
- Ground work for a comprehensive information management system where information retrieval is a full-featured component.

- Development and evaluation of a structured query optimization that reduces execution time by over 50% with no noticeable impact on retrieval effectiveness.

- An investigation of the impact of high frequency query terms in short, unstructured queries and how to handle them for best retrieval effectiveness and execution performance.

1.4 Outline of the Dissertation

In the remainder of this dissertation, we begin with a survey of related work (Chapter 2). We then consider the problems of indexing a document collection and managing an inverted file, describe our solutions, and present results (Chapter 3). Next, we address the problem of providing fast evaluation of structured queries, describe our solution, and present results (Chapter 4). Finally, we summarize the conclusions drawn from this research and discuss future work (Chapter 5).

CHAPTER 2

RELATED WORK

In this chapter we survey related work that is not specifically addressed in other parts of the dissertation. We begin with a discussion of inverted file implementation issues and alternatives, and then survey work on query optimization techniques for information retrieval.

2.1 Inverted File Management

Inverted file management has been pursued from a number of perspectives. We begin with a discussion of efforts to support information retrieval with a traditional database management system, which range from treating IR as just a relational database application, to loose integration of separate IR and database management systems. We then consider custom inverted file management solutions, and briefly review alternative indexing schemes for information retrieval.

2.1.1 Traditional Database Support for IR

The first body of work related to the research presented in this dissertation is the general technique of providing information retrieval services using a standard database management system (DBMS). Documents are stored by the DBMS and represented in such a way that the query language of the DBMS can be used to construct information retrieval style queries. Some of the earliest work was done by Crawford and MacLeod [18, 54, 17, 55], who describe how to use a relational database management system (RDBMS) to store document data and construct information retrieval queries. Similar work was presented more recently

by Blair [5] and Grossman and Driscoll [38]. Others have chosen to extend the relational model to allow better support for IR. Lynch and Stonebraker [53] show how a relational model extended with abstract data types can be used to better support the queries that are typical of an IR system.

In spite of evidence demonstrating the feasibility of using a standard or extended RDBMS to support information retrieval, the poor execution performance of such systems has led IR system builders to construct production systems from scratch. Additionally, most of the work described above deals only with document titles, author lists, and abstracts. Techniques used to support this relatively constrained data collection may not scale to true full-text retrieval systems. Moreover, sophisticated retrieval models such as the inference network-based retrieval model are difficult to represent using an RDBMS. A custom retrieval engine will inevitably provide superior performance and is certain to better represent the semantics of the retrieval model.

Other work in this area has attempted to integrate information retrieval with database management [27, 74], and is representative of our comprehensive information management system goal. The services provided by a database management system and an IR system are distinct but complementary, making an integrated system very attractive. In this case, a separate, self-contained information retrieval system is loosely coupled with a more traditional database management system. There is a single user interface to both systems, and a preprocessor is used to delegate user queries to the appropriate subsystem. Additionally, the DBMS is used to support the low level file management requirements of the whole system.

Whether an RDBMS is used to implement an IR system or provide low-level storage support for a loosely coupled IR system, the inverted file index required by the IR system must be managed efficiently. We will see in Chapter 3 that the data management requirements of an inverted file are not easily satisfied by an RDBMS. Rather than use an RDBMS, we propose the use of a persistent object store, favoring a data management system that

more naturally satisfies the unusual storage requirements of an inverted file. In particular, the inverted lists in an inverted file will come in a broad range of sizes, with some of the lists being very large. We will see that the persistent object store offers a straight forward solution to the problem of managing these large objects.

Generic support for storage of large objects has been pursued elsewhere in the database community. The EXODUS storage manager [13] supports large objects by storing them in one or more fixed size pages indexed by a B+tree on byte address. For example, to access the 12 bytes starting at byte offset 10324 from the beginning of a large object, the object's B+tree would be used to look up 10324 and locate the data page(s) containing the desired bytes.

The Starburst long field manager [50] supports large objects using a sequence of variable length segments indexed by a descriptor. As an object grows, a newly allocated segment will be twice as large as the previously allocated segment. This growth pattern continues up to some maximum segment size, after which only maximum size segments are allocated. The last segment in the object is trimmed to a page boundary to limit wasted space. This known pattern of growth allows a segment's size to be implicitly determined, eliminating the need to store sizes in the descriptor. A key component of this scheme is the use of a buddy system to manage extents of disk pages from which segments are allocated. This scheme is intended to provide efficient sequential access to large objects, assuming they are typically read or written in their entirety.

Biliris [3] describes an object store that supports large objects using a combination of techniques from EXODUS and Starburst. A B+tree is used to index variable length segments allocated from disk pages managed by a buddy system. This scheme provides the update characteristics of EXODUS with the sequential access characteristics of Starburst. A comparative performance evaluation of the three schemes can be found in [4].

2.1.2 Custom Inverted List Management

Efficient management of full-text database indexes has received a fair amount of attention. Faloutsos [29] gives an early survey of the common indexing techniques. Zobel et al. [97] investigate the efficient implementation of an inverted file index for a full-text database system. Their focus is on compression techniques to limit the size of the inverted file index. They also address updates to the inverted file using large fixed length disk blocks, where each block has a heap of inverted lists at the end of the block and a directory into the heap at the beginning of the block. As inverted lists grow they are rearranged in the heap or copied to other blocks with more space. Techniques for handling inverted lists larger than a disk block are not discussed, nor is the disk block technique fully evaluated.

A more sophisticated inverted list implementation was proposed by Faloutsos and Jagadish [31]. In their scheme, small lists are stored as inverted lists, while large lists are stored as signature files. They have a similar goal of reducing the processing costs for long inverted lists, but their solution is inappropriate for the inference network model. In [32], Faloutsos and Jagadish examine storage and update costs for a family of long inverted list implementations, where the general case is their "HYBRID" scheme. The HYBRID scheme essentially chains together chunks of the inverted list and provides a number of parameters to control the size of the chunks and the length of the chains. At one extreme, limiting the length of a chain to one and allowing chunks to grow results in contiguous inverted lists, where relocation of the inverted list into a larger chunk is required when the current chunk is filled. At the other extreme, fixed size chunks and unlimited chain lengths give a standard linked list.

Harman and Candela [41] use linked lists for a temporary inverted file created during indexing. Their linked list nodes are quite small, consisting only of a single document posting. Accessing the inverted file in this format during query processing is much too inefficient, so the nodes in a linked list are ultimately conglomerated into a single inverted list before the file is used for retrieval.

Tomasic et al. [84] propose a new inverted file data structure to support incremental indexing, and present a detailed simulation study over a variety of disk allocation schemes. The study is extended with a larger synthetic document collection in [76], and a comparison is made with traditional indexing techniques. Their data structure manages small inverted lists in buckets (similar to the disk blocks in [97]) and dynamically selects large inverted lists to be managed separately. It is notable that they expect the scheme with the best incremental update performance to have the worst query processing performance due to fragmentation of the long inverted lists.

Moffat and Zobel [60] describe an inverted list implementation that supports jumping forward in the list using *skip pointers*. This is useful for document based access into the list during conjunctive style processing. The purpose of these skip pointers is to provide synchronization points for decompression, allowing just the desired portions of the inverted list to be decompressed.

Properly modeling the size distribution of inverted file index records and the frequency of use of terms in queries is addressed by Wolfram in [91, 92]. He suggests that the informetric characteristics of document databases should be taken into consideration when designing the files used by an IR system. This is an underlying theme of the work described here, where term frequency and access characteristics are carefully considered throughout.

2.1.3 Inverted File Alternatives

The most popular alternative to an inverted file is the signature file [30]. A signature file contains document signatures, one for each document in the collection. A document's signature is a bit-string created by applying a hash function to each of the terms in the document (documents may be sub-divided into blocks, with a separate signature for each block). The hash function identifies one or more bits in the signature that should be set to "1." The width of the bit-string and the number of bits set by the hashing function are parameters that control the likelihood of different terms setting overlapping bits.

During query evaluation, a signature is created from the terms in the query in the same way. The query signature is then compared with all of the document signatures in the signature file. A document will potentially match the query if the intersection of its signature and the query signature is equal to the query signature. The match is "potential" because terms different from those in the query may set the same signature bits as the query terms, resulting in a *false drop*. In this case, a document is flagged as matching the query, when in fact it does not. Note that the opposite cannot occur. If a document does contain all of the query terms, this strategy will never fail to flag the document as matching. The possibility of false drops means that documents with matching signatures must be processed further to determine whether or not they truly match the query.

Signatures are commonly stored and manipulated in *bit-slices*. The n^{th} bit-slice contains the n^{th} bit from all of the signatures, stored as a sequential string. With this organization, we need to process only the bit-slices identified by the query signature, greatly reducing the amount of data that must be read from the signature file. The cost of using a bit-sliced organization is more expensive updates. This organization, however, is particularly amenable to parallel processing, and a number of parallel implementations have been described in the literature [63, 82].

It has long been argued that signature files provide performance superior to that obtained with inverted files. Any performance advantage, however, comes at the cost of a more restricted retrieval model—signature files typically support Boolean queries only. Croft and Savino [23] show how signature files can be extended to support document ranking, but ultimately find that equivalent performance can be obtained by using an inverted file. More recently, Zobel et al. [96] give both analytical and empirical results that show inverted files to be superior to signature files in all respects, regardless of the retrieval model. Given their greater flexibility in terms of retrieval model and the recent results demonstrating their superior performance, inverted files appear to be the index of choice for a full-text information retrieval system.

2.2 Query Optimization

The database community has a rich history of query optimization techniques. In [37], Graefe gives a comprehensive survey of query execution and optimization techniques, concentrating mainly on the relational model. These techniques are generally based on an algebra or calculus where query manipulations can be performed to reduce execution time without modifying the semantics of the query. While some of these execution techniques are applicable to information retrieval (e.g., set intersection techniques), the vague nature of ranked retrieval makes it drastically different from the traditional database query paradigm, where there is a single correct answer to any given query. In ranked information retrieval, we can trade answer precision for speed using unsafe optimization techniques.

The unsafe query optimization techniques have their roots in the upper bound optimizations used to solve the nearest neighbor problem in information retrieval. In this model, a query and the documents in the collection are represented as vectors in an n-dimensional space, where n is the number of terms in the vocabulary. The problem is to find the document closest to the query in this vector space. Distance in the vector space is defined by the similarity measure used between a document and the query. This is typically some form of dot product between the vectors. The dot product is limited to the terms that appear in the query, so only documents that contain at least one of the query terms need be considered in the nearest neighbor search. Inverted lists are used to identify documents that are the potential nearest neighbor to the query. When a previously unseen document is encountered in an inverted list, the document's representation vector is retrieved to calculate its exact similarity to the query. If this document is closer to the query than the current nearest neighbor, it becomes the new nearest neighbor. When the inverted lists for all of the terms in the query have been processed, the current nearest neighbor is returned as the answer to the query.

Smeaton and van Rijsbergen [78] describe how an upper bound on the similarity of any unseen document can be calculated based on the unprocessed query terms. If this upper

bound is less than the similarity of the current nearest neighbor, processing may stop. By processing terms in order of increasing inverted list length, they achieve a 40% reduction in the number of similarity calculations required to find the nearest neighbor.

An alternative technique for locating the nearest neighbor uses counters to gradually accumulate a document's similarity to the query. The accumulated similarity is based solely on the information stored in the inverted lists, thus eliminating the need to retrieve the document representation vectors. After all inverted lists have been processed, the nearest neighbor is identified by selecting the maximum similarity from the counters. Perry and Willett [64] show how the upper bound technique can be applied to this processing strategy to reduce main memory requirements. The upper bound on the similarity of a previously unseen document is calculated in the same way as before. If this upper bound is less than the current best similarity for any previously seen document, the new document is not allocated a counter since it cannot be the nearest neighbor. The overall number of counters is reduced, resulting in main memory savings.

This processing strategy can be extended to support full ranking by computing the complete similarity for every document encountered and sorting the set of counters to produce the final ranking. This strategy is at the core of most modern ranking retrieval systems, and can be restated as follows. A query consists of a set of terms, where each term contributes a term weight for every document in which it appears. To evaluate the query, the term weights for a given document are combined according to the semantics of the particular similarity measure to produce a final score for the document. The documents are then ranked by their final scores to produce the answer to the query. In essence, this procedure involves allocating an array large enough to hold an identifier and final score for each document, updating this array as each term weight from the terms is processed, and sorting the final array by score.

In this processing strategy the goal of a query optimization is to avoid processing the term weights that do not contribute significantly to the final document ranking. This can

be accomplished by identifying some subset of the term weights that will result in a final ranking close to the "exact" ranking achieved when all term weights are processed. As this subset becomes smaller and smaller, we expect the final ranking to differ more and more from the exact ranking. The question now is how to select this subset. There are a variety of methods to make this selection, and they all can be classified based on how they decide the following:

- which term weight to process next
- when to stop

Both of these seemingly simple questions have interesting and subtle implications for performance and implementation. The order in which term weights are processed will affect the rate at which the array of scores is populated with discriminating information, and has implications for the inverted list organization. The stopping condition is intimately related to the term weight processing order and will determine how much work will be done to answer the query and what claims can be made about the quality of the answer returned. We consider possible answers to these questions below.

2.2.1 Term Weight Magnitude Ordering

The first term weight processing order is to greedily process term weights in order of decreasing contribution to the final ranking. For a similarity measure that treats all term weights equally, this is equivalent to processing term weights in order of decreasing magnitude. This ordering is very appealing in that the document ranking scores will initially grow very quickly and the relative order of the documents should be established early in the processing. Term weights processed later in the order will be smaller, having less chance to change the relative ranking of the documents.

To support this processing order, the term weights must be extracted from the inverted lists in decreasing sorted order. Practically speaking, this would be accomplished by storing

the document entries in the inverted lists in decreasing term weight order. The next term weight to process would be chosen by examining the next term weight in each inverted list and selecting the largest of these values.

The stopping condition for this processing order can be defined in a number of ways. First, we might simply stop after processing some arbitrary percentage of the term weights, assuming that retrieval effectiveness is a logarithmic function of the number of term weights processed and execution time is a linear function of the number of term weights processed. Determining what these functions actually look like might be done experimentally or analytically. The problem with this scheme is that, short of processing all of the term weights, it gives us no guarantees on the correctness of the final ranking obtained. This scheme was proposed by Wong and Lee [93], who describe two estimation techniques for determining how many term weights must be processed to achieve a given level of retrieval effectiveness.

An alternative to this ad-hoc stopping condition would be a stopping condition that takes advantage of the organization of the term weights. Each term will contribute at most one term weight to each document being considered. If we keep track of which terms have contributed a term weight to a given document so far, we can calculate an upper bound on the final score for that document using the current term weights from each of the terms which have not contributed a term weight for that document (since a term's term weights are processed in decreasing sorted order). Moreover, we can use the current partially computed score for a document as a lower bound for that document's final score. At any given time, if a document's lower bound exceeds all other document's upper bounds, then further consideration of that document can stop and the document can be returned as the current best document. With this stopping condition, we can guarantee that the top n documents will be returned in the correct order, making the scheme safe for the top n documents. The disadvantage of this scheme is the computational costs of the required bookkeeping, which

may exceed any savings in term weight processing. This scheme is described by Pfeifer and Fuhr [66].

If we are more concerned with obtaining the top n documents and less concerned with their relative ranking, we can define another stopping condition. At any given time, an upper bound on the remaining increase in any document's score is given by the sum of the current term weights from each of the terms. Assume the documents are ranked by their current partially computed scores. When the $n+1^{st}$ document's current score plus the upper bound on the remaining document score increase is less than the n^{th} document's score, we know that the top n documents will not change and processing can stop. We can return the top n documents, but we cannot guarantee their relative ranking.

Rather than place a hard limit on the size of the set of documents returned, thresholds can be established that determine how a term weight is processed. Such a scheme is described by Persin [65]. If a document is not in the set of documents currently being considered and has no current score (i.e., no term weights have been processed for that document), an *insertion* threshold is used to determine if a term weight for that document is significant enough to place the document into the consideration set. If the document is already in the consideration set, an *addition* threshold is used to determine if a term weight is significant enough to modify a document's current score. The addition threshold allows us to stop processing an inverted list as soon as its term weights fall below the addition threshold. The insertion threshold ensures that we consider only documents which have a significant term weight contribution from the terms. With this scheme, we can make no claims about the quality of the final ranking.

2.2.2 Document Based Ordering

None of the previous schemes can guarantee that a complete score for a given document has been computed. All that might be guaranteed is that the top n documents have been returned, and in one case, that they are correctly ranked. If we require that complete final

scores be calculated for all documents ranked, then the term weight processing order may be document driven using a document-at-a-time query processing strategy. In this scenario, once the current document to process has been identified, the term weights for all of the query terms that appear in that document must be processed. This requires document based access into the inverted lists and is most easily supported by storing the document entries in the inverted lists in document identifier order. Now we must decide the order in which to process the term weights for the current document. The order of decreasing contribution to the document's final score is most useful. Assuming a $tf \cdot idf$ style term weighting function, this can be accomplished by processing the term weights in decreasing order of idf.

This per document term weight processing order allows us to use the following stopping condition. Assume we wish to return the top n documents. We begin by initializing the set of top n documents with complete scores for the first n documents. We then identify the minimum score S from these top n documents. For each of the remaining documents, an upper bound on the current document's final score can be calculated from its currently accumulated score and the idf of the terms not yet processed for the document. If this upper bound becomes less than S, processing of the current document can stop because it cannot appear in the top n documents. If a complete score for the document is computed which is greater than S, the document is placed in the set of top n documents and S is recalculated. This scheme guarantees that the top n documents are returned, correctly ranked and with complete final scores. Processing savings will accrue whenever a document's upper bound descends below S and the document is eliminated from consideration before its complete score is calculated. I/O savings may accrue if we have the ability to skip portions of inverted lists. Frequent terms will occur late in the processing order and will have long inverted lists. Many documents will be eliminated from consideration before these frequent terms are processed, such that much of the inverted list information for these terms can be skipped. This scheme is called max-score by Turtle and Flood [89].

The document processing order used above will attempt to calculate a score for every document that appears in the inverted lists of the query terms. In fact, we can identify another stopping condition at which point all document processing can stop. As processing proceeds, all of the term weights from short inverted lists will eventually be processed, such that those terms no longer need to be considered. If the upper bound contribution of the remaining terms which still have term weights to process descends below S, then all processing can stop. We may be able to achieve this condition more quickly by altering the document processing order to process first those documents which appear in the shortest inverted lists, encouraging the early exhaustion of these lists.

2.2.3 Term Based Ordering

The last term weight processing order is term based, where all of the term weights for a given term are processed at once. This corresponds to term-at-a-time query processing (see [89] for a comparison of term-at-a-time and document-at-a-time processing). As with the per document term weight processing order above, terms are processed in decreasing order of document score contribution, approximated by the term's *idf* score. This strategy will cause the terms to be processed in order of inverted list length, from shortest to longest.

The first stopping condition we will consider was originally described by Buckley and Lewit [10] and later discussed by Lucarella [52]. It is intended to eliminate processing of entire inverted lists, and is similar to the third stopping condition described in Section 2.2.1. Assume that we are to return the top n documents to the user. After processing a given term, the documents can be ranked by their currently accumulated scores, establishing the current set of top n documents. An upper bound on the increase of any document's score can be calculated from the unprocessed terms in the query, assuming the maximum possible term weight contribution from each of those terms. If the $n+1^{st}$ document's score plus the upper bound increase is less than the n^{th} document's score, then we know that the set of top n documents has been found. At this point we can stop processing and guarantee that

the top n documents will be returned. We cannot, however, guarantee either the relative ranking of the documents within the set or that complete scores have been calculated for those documents.

This scheme elegantly addresses the irony where the most expensive terms to process contribute the least to the final score. Since the terms are processed in order of decreasing score contribution, the upper bound score increase will diminish as quickly as possible, and the most expensive terms to process will be eliminated by the stopping condition. Note also that since the processing order and stopping condition are completely term based, there are no constraints on the organization of the document term weights within an inverted list.

There are three variations on this stopping condition, all of which are similar to the last stopping condition described in Section 2.2.1. The first variation was proposed by Harman and Candela [41], called *pruning*. Rather than place a limit on the number of documents returned to the user, we can establish an insertion threshold for placing new documents in the candidate set. In this case, the insertion threshold is term based, such that a term's potential score contribution must exceed some threshold in order for the term to contribute new documents to the candidate set. Processing will then have two distinct phases. First, during a disjunctive phase, documents will be added to the candidate set and partial scores updated as usual. Then, after the insertion threshold is reached, a conjunctive phase will occur where terms are not allowed to add new documents, only update the scores of existing documents. This scheme can make no guarantees about the membership of the set. It does, however, calculate complete scores for the documents in the candidate set, guaranteeing a correct relative ranking.

The second variation was proposed by Moffat and Zobel [60, 58, 59]. Rather than use an insertion threshold related to a term's potential score contribution, a hard limit is placed on the size of the candidate document set. The disjunctive phase proceeds until the candidate set is full. Then, the conjunctive phase proceeds until all of the query terms have been processed. This variation makes the same guarantees as the previous one.

The third variation is a term-at-a-time version of max-score described by Turtle and Flood [89]. New documents are added to the candidate set until the upper bound score of an unseen document (determined from the maximum possible term weight contributions of the unprocessed terms) falls below the current partial score of the n^{th} document. At this point, we know that no unseen document can appear in the top n documents. Processing then continues in a conjunctive fashion, updating the scores for just those documents currently in the candidate set. When a given document's score is updated, its maximum possible score is computed assuming it contains all of the unprocessed terms. If this maximum score is less than the n^{th} score, this document is eliminated from the candidate set. This variation will guarantee that the top n documents are returned in the correct order.

During the conjunctive processing phase of the last three variations, access into the inverted lists will be document based. This suggests that, for the most efficient processing, document entries within the inverted lists should be sorted by document identifier. Moreover, as in Section 2.2.2, the ability to skip portions of inverted lists should provide significant I/O savings during this processing phase.

There are two other optimization techniques that do not easily fit into the the taxonomy used above. First is the two stage query evaluation strategy of the SPIDER information retrieval system [75, 47]. In SPIDER, a signature file is used to identify documents that potentially match the query, and an upper bound is calculated for each document's similarity to the query. Non-inverted document descriptions are then retrieved for these documents in order of best upper bound similarity and used to compute an exact similarity measure. As soon as a document's exact similarity measure exceeds all other documents' upper bound (or exact) similarity measures, this document can be returned as the best matching document. Correct document scores and rankings are guaranteed.

The second optimization technique, *list pruning*, was proposed by Smith [79] for the p-norm retrieval model (an extended Boolean retrieval model). During term-at-a-time evaluation, intermediate result lists are pruned by removing all document entries whose

current score is less than some score threshold. This threshold may be constant, or it may be determined dynamically based on the contents of the intermediate result. Pruned intermediate result lists require less computation as query evaluation proceeds, resulting in potential execution time savings. A document eliminated from one part of the query may be re-introduced in another part, however, allowing documents to have inaccurate final scores. The accuracy of the final document ranking, therefore, cannot be guaranteed.

CHAPTER 3

INDEXING

In this chapter we consider two problems: efficiently building an inverted file index for a document collection, and updating that index to reflect modifications to the document collection. Indexing is an important procedure in any information retrieval system—a document collection cannot be searched efficiently (if at all) unless it has been indexed. A variety of indexing procedures have been proposed in the literature [41, 35, 42], although only recently have procedures been described that claim to index large document collections efficiently [57, 90]. While we are certainly concerned with finding an efficient indexing technique for large document collections, we are equally concerned with supporting *dynamic* document collections. A document collection is dynamic if new documents can be added to an existing collection, old documents can be deleted from an existing collection, or existing documents can be modified. We will, therefore, pursue a more comprehensive solution to the problem of building and managing a document collection index.

The ability to modify an existing document collection is a natural requirement for any information retrieval system. New documents will forever be created, discovered, delivered, or requested. If the information contained in these new documents is to be integrated into and accessible from the current information base, then the new documents must be added to the existing document collection. Some applications have very explicit requirements for supporting document collection modification. For example, an on-line news wire service with a current events document collection must grow the collection frequently and efficiently. There will be a continuous stream of new articles coming in on the news wire. In order to answer queries about recent newsworthy events, the new articles

must be added to the current events document collection shortly after they are received. Additionally, old news articles will eventually expire and must be deleted from the current events document collection. Articles may expire either because their content is relevant only for a certain period of time, or because the size of the current events collection must be held below some threshold due to performance requirements or capacity limitations. Expired articles will either be discarded or archived in a larger secondary document collection, leading to further document addition operations.

Even if all of the documents that are to be added to the document collection are available simultaneously, the ability to add new documents to an existing document collection can be useful. As we will see in Section 3.1 when we consider the mechanics of document indexing in more detail, if the inverted file does not support growth, the indexing process can require substantial temporary disk space resources, especially if the document collection to be indexed is large. If instead the inverted file does support growth, then temporary disk space requirements can be significantly reduced using *incremental indexing*. An incremental indexing strategy indexes the documents in batches, where each batch indexing step requires little or no temporary disk space and yields a complete index for the documents processed so far. The key to this strategy is the ability to build on the output of previous batch indexing steps by growing the inverted file that was built during those steps. Underlying all of this is the ability to add new documents to an existing collection.

Modifications to documents in an information retrieval system may come about for a number of reasons. Consider, for example, a collaborative authoring system. In this application, multiple authors will be simultaneously modifying documents in the collection. The information retrieval system must be able to incorporate these modifications in order to faithfully track the information content of the document collection. Of course, document modifications are not restricted to applications specifically intended to support document creation. An information retrieval system that stores manuals or documentation will inevitably be asked to modify those documents as they are revised and updated. Although

document modifications arise in a variety of situations, most of these situations can be accommodated using a versioning scheme. A modified document is simply a new version of the original document, and is added to the document collection as a new document, distinct from the original. The original document can then be deleted, or a higher level mechanism can be used to track multiple versions of the same document in the document collection. Either way, as long as document additions and deletions are supported by the information retrieval system, no extra functionality is required to support document modifications.

The level of functionality provided by the inverted file implementation will determine how well the overall system can satisfy the requirements of a dynamic document collection. During query evaluation, rather than operate on the documents themselves, the retrieval engine processes the contents of the inverted file. As far as the retrieval engine is concerned, the membership of the document collection is defined by the inverted file. A document has not been truly added to the document collection until the inverted file has been updated to reflect that addition. The same holds true for document deletions. The question of how to support a dynamic document collection is in large part a question of how to support a dynamic inverted file.

In the rest of this chapter we will pursue this question in detail. We begin with a discussion of the general indexing process—how to build the inverted file in the first place. For large document collections, building an inverted file efficiently is a difficult problem. We have extended a previously described indexing technique to produce a fast, scalable indexing system. The output of this system is complete inverted lists for the input document collection. These lists are handed to the *Inverted File Manager*, which is responsible for the low-level storage and retrieval of the inverted file. The Inverted File Manager is the core system component that determines the overall functionality available for inverted file manipulation. We will describe the issues pertinent to building an Inverted File Manager, the particular solution we have chosen, and our implementation of that solution. This discussion is followed by an experimental evaluation of our solution. The measurements

will focus on indexing whole document collections from scratch and adding new documents to an existing document collection. These two activities represent the most common and crucial indexing activities that must be performed by an information retrieval system. This emphasis stems from the traditional role of IR systems in managing archival document collections, which are either static or growing. The experimental results are followed by conclusions.

3.1 Document Inversion

The process of indexing a document collection and building its inverted file is called *inversion*. Initially, we can easily identify the terms that appear in a given document simply by inspecting the document—the terms are what make up the document. Ultimately, what we want is the inverse of this, such that given a term, we can identify the documents that contain that term. Suppose we create a tuple $\langle d, t, l \rangle$ to represent each document/term occurrence pair, where d is a document identifier, t is a term identifier, and l is the location of the occurrence of term t in document d. An example is given in Figure 3.1. There is a tuple for every term occurrence in the document collection. When we scan a document collection from start to finish, the tuples for the collection will come out in an order sorted first on d and second on l. For an inverted document collection, we want these tuples sorted first on t, second on d, and third on l. As such, the inversion process can be viewed as a large tuple sorting problem, going from the collection sort order to the inverted sort order.

A closer look at the problem, however, shows that a full sort of the collection tuples is not actually necessary. A comparison of the collection sort order and the desired inverted sort order reveals that the collection sort order is partially in the desired inverted sort order. In the collection sort order, the tuples are fully sorted on d. In the inverted sort order, all of the tuples for a given t are sorted by d. Furthermore, in both the collection sort order and the inverted sort order, all of the tuples for a given $\langle d, t \rangle$ pair are sorted by l. This suggests the following inversion strategy. First, maintain a separate list of tuples for each

Documents	*Terms*	*Tuples* *Collection Order*	*Tuples* *Inverted Order*
1. The cat ate the snake	1. the	<1, 1, 1>	<1, 1, 1>
2. The dog chased the cat	2. cat	<1, 2, 2>	<1, 1, 4>
3. The snake chased the dog	3. ate	<1, 3, 3>	<2, 1, 1>
	4. snake	<1, 1, 4>	<2, 1, 4>
	5. dog	<1, 4, 5>	<3, 1, 1>
	6. chased	<2, 1, 1>	<3, 1, 4>
		<2, 5, 2>	<1, 2, 2>
		<2, 6, 3>	<2, 2, 5>
		<2, 1, 4>	<1, 3, 3>
		<2, 2, 5>	<1, 4, 5>
		<3, 1, 1>	<3, 4, 2>
		<3, 4, 2>	<2, 5, 2>
		<3, 6, 3>	<3, 5, 5>
		<3, 1, 4>	<2, 6, 3>
		<3, 5, 5>	<3, 6, 3>

Figure 3.1 Document collection tuples

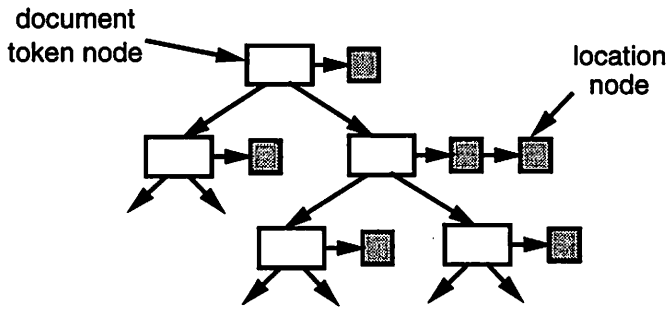
term in the collection. Then, scan the document collection and process the tuples in their collection order. As each tuple is processed, append it to the tuple list for the term that appears in the tuple. The document id order and term occurrence location order will be preserved automatically in the new term based tuple lists, and the inverted tuple order will be obtained.

We must consider a number of issues before implementing this inversion strategy. First, large document collections contain a large number of distinct terms. The 1 GB TIPSTER [39] document collection used in the experiments below (**Tip1**) contains 639,914 terms. During the inversion process we need appropriate data structures to keep track of 639,914 distinct term lists. Second, large document collections contain a large number of term occurrences. The 1 GB TIPSTER document collection contains 112,812,693 term occurrences, translating into 112,812,693 tuples. If a four byte integer is used for each element of a tuple, each tuple will occupy 12 bytes and the total memory requirement for all of the tuples will be 1.3 GB. If the inversion process is run on a workstation equipped

with 64 MB of main memory (a likely scenario these days), all of the tuples clearly will not fit in main memory. It is therefore inevitable that inverting a large document collection requires some amount of disk I/O. Careful management of this disk I/O is essential for efficient inversion of large document collections.

There are two basic guidelines regarding disk I/O that will govern our implementation. First, perform as little I/O as possible. Second, when I/O must be performed, favor sequential I/O over random I/O in an effort to avoid disk head positioning. The first guideline is somewhat obvious. The second guideline is based on the costs associated with the different components of a disk access [14]. The time to perform a disk access is made up of head positioning time, which includes seeking and rotational latency, and data transfer time. Average head positioning times are currently around 15 milliseconds, and data transfer rates are around 5 MB per second. Given the relatively fast data transfer rates and slow head positioning times, it is advantageous to amortize the head positioning cost over larger data transfers. Sequential I/O provides this desirable behavior, while random I/O does not.

With these guidelines in mind, the following document indexing procedure was implemented for INQUERY. The overall process is a unique combination of the main memory linked list and multiway merge schemes with compressed temporary files described by Witten et al. [90], and consists of two main operations: parsing and merging. The subsystem responsible for parsing is called the *Parser*. It creates *partial* inverted lists by scanning, lexically analyzing, and inverting documents. A partial inverted list contains document entries for a subset of the documents in the collection. It must be combined with other partial inverted lists for the same term to create a *final* inverted list for the document collection. The Parser buffers partial inverted lists in main memory and flushes them to temporary files when the buffer is full. The subsystem responsible for merging is called the *Merger*. After all of the documents have been parsed, the Merger combines the temporary files to produce the final inverted lists for the collection.



document token node

```
struct doc_token
  {
    struct doc_token *left;
    struct doc_token *right;
    struct location *loc_list;
    int tf;
    char term[];
  }
```

location node

```
struct location
  {
    int position;
    struct location *next;
  }
```

Figure 3.2 Document buffer binary tree

3.1.1 Parsing

Document indexing begins with parsing. The Parser scans and lexically analyzes each document, producing a stream of tokens from the documents. The Parser checks each scanned token against a *stop words list* (a list of terms too frequent to be worth indexing) [33, 34] and discards any tokens that it finds in the list. Tokens that survive the stop words list are run through a *stemmer* [36]. Stemming reduces a term to its root form, mapping different morphological variants to a common stem. This process *conflates* different representations of the same concept into a single representation, improving retrieval effectiveness by eliminating mismatches between morphological variants of the same term. It also compresses the index by reducing the total number of terms that are indexed. Our indexing implementation uses document scanning, stopping, and stemming utilities developed by others at the University of Massachusetts [12]. Our contribution to the implementation is the portion of the system the handles the tokens from this point on.

The next step in the parsing process is assembly of the stemmed tokens and their locations into partial inverted lists for the current document. This occurs gradually as the Parser stores the location of each token occurrence in the *document buffer*. The document buffer is organized as a binary search tree of token nodes sorted on term strings, depicted in Figure 3.2. Each token node in the tree contains a count of the number of times the associated term occurs in the current document and a pointer to a linked list of location nodes containing the locations of each occurrence of the term. The Parser searches the binary tree for each scanned token and either finds a token node for the current token in the tree, or creates and inserts a new token node for the current token. The Parser then creates a location node for the current token and adds the location node to the head of the linked list of locations for the token.

The primary motivation for building partial inverted lists on a per document basis is to reduce the time spent searching for each token's partial inverted list as the tokens are parsed out of the document. Since the document buffer contains inverted list entries just for the current document, the number of token nodes in the binary tree will grow only to the size of the vocabulary used within the current document. Documents in the 3.2 GB TIPSTER collection [39] contain an average of 132 unique terms, while the entire collection contains 1,062,677 unique terms. Searching for each parsed token in the binary search tree requires $O(\lg(n))$ time, where n is the number of nodes in the binary search tree. For the average document, we will traverse $O(\lg(132) = 7)$ binary tree nodes for each parsed token using a per-document binary search tree. In comparison, if the binary search tree contained a node for every term in the collection, we would traverse $O(\lg(1062677) = 20)$ binary tree nodes for each parsed token.

When all of the tokens have been parsed out of the current document, the document buffer is flushed to the *batch buffer*. The batch buffer holds partial inverted lists for a batch of documents, where a batch consists of as many documents as can be parsed before the batch buffer is full. The batch buffer is organized as a hash table of token nodes keyed on

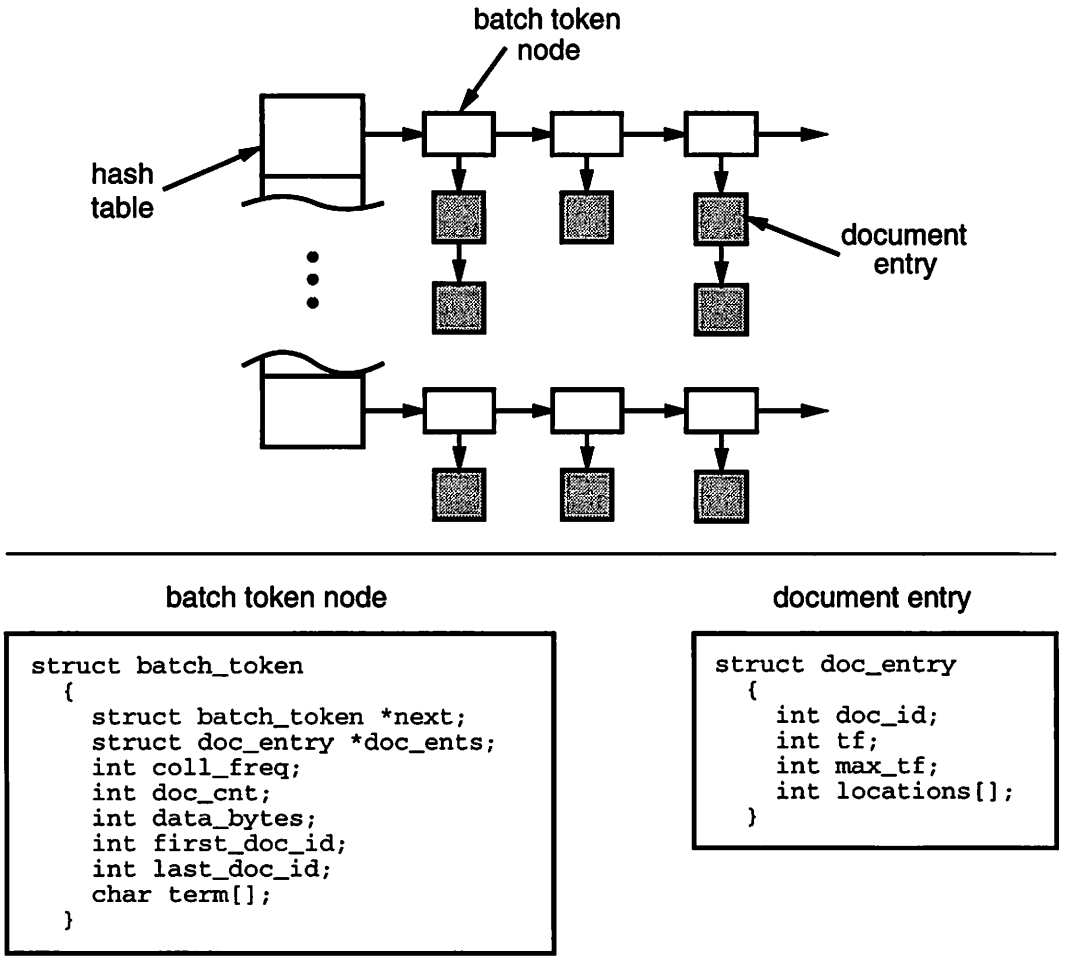


Figure 3.3 Batch buffer hash table

term strings, depicted in Figure 3.3. The batch buffer could actually be organized using any dynamic data structure that supports search and insert operations (e.g., a binary search tree). The choice of a hash table is motivated by incremental indexing requirements, which are discussed below in Section 3.2.3.3. The hash table size is fixed at 8191 slots and collisions are resolved by chaining together tokens that hash to the same slot. A batch token node stores a document count, collection frequency, and byte count for the current partial inverted list. It also points to a linked list of document entries—the "data" of the partial inverted list. The document count is equal to the number of document entries in the partial inverted list, the collection frequency is equal to the total number of term locations stored in the partial

inverted list, and the byte count is the total number bytes occupied by all of the document entries in the linked list.

The Parser flushes the document buffer to the batch buffer by traversing the document buffer in a preorder tree walk. At each document token node, the Parser either finds the corresponding node in the batch hash table, or creates a new batch token node and inserts it into the hash table. The Parser then builds a *document entry*, which contains the document identifier, term frequency, maximum term frequency for the document, and token locations list (see Figure 3.3). Document identifiers are assigned from a global document counter, which is incremented as each document is processed. The term frequency, *tf*, is obtained from the document token node. A document's maximum term frequency, *max_tf*, is the maximum of $\{tf_1, tf_2, tf_3, \ldots\}$, where tf_i is the frequency of term i in the document. *max_tf* is calculated on the fly as each tf_i is updated during document parsing. The locations list is obtained by walking the linked list of location nodes.

The Parser compresses all of the numbers in a document entry using a variable length byte encoding scheme [73]. The encoding scheme represents each integer in base 2 using the *minimum* number of bytes. The 8^{th} bit in each byte serves as a termination flag, indicating whether or not the last byte for the current integer has been processed. This leaves seven bits per byte to store the integer, such that the largest integer representable by a sequence of n bytes is 2^{n*7}. In this compression scheme, smaller integers consume less space. We will achieve better compression, therefore, if we can reduce the magnitude of the integers to be compressed. A common technique for reducing the magnitude of integers that form a sequence of nondecreasing numbers is *delta encoding* [25] (the deltas are called *gaps* by Bell et al. [2]). To delta encode a sequence of numbers, the first number is stored as an absolute value and each subsequent number is stored as the difference between itself and the previous number.

An inverted list provides two opportunities for delta encoding. The first opportunity is found in the token locations list within each document entry. The locations list is delta

encoded when the linked list of location nodes is traversed to create a document entry. The second opportunity is found in the sequence of document identifiers across the document entries in an inverted list. To delta encode the document identifier in a document entry, we must keep track of the document identifier in the last document entry that was chained onto the batch token node's linked list of document entries in the batch buffer. This information is kept in the batch token node and updated as each document entry is added.

After a document entry's locations list and document identifier have been delta encoded, the entry is compressed as described above. The compressed document entry is placed in the batch buffer and chained onto the batch term's linked list of document entries. The batch term's document count, collection frequency, and byte count are then updated to account for the new document entry. When all of the token nodes in the document buffer have been processed and added to the batch buffer, the next document in the collection is parsed.

When the batch buffer is full, it is flushed to a temporary file block. To facilitate the eventual merging of temporary files, the partial inverted lists in each temporary file block must be written in the same order. The token strings provide a natural key on which to sort the partial inverted lists and ensure a consistent ordering across temporary files. Since the batch buffer is organized as a hash table, the batch token nodes are not directly available in token string order; they must first be sorted by token string. This is accomplished using an array of pointers to the batch token nodes. The pointers are sorted based on the strings in the token nodes that they reference, and an iteration through the array yields the token nodes in sorted order.

A batch token node is written to the temporary file block in three steps. First, the token string is written with a terminating null character. Second, the statistics for the partial inverted list are compressed and written. The statistics consist of the collection frequency, document count, byte count, and document identifiers in the first and last document entries for the partial inverted list. Third, the compressed document entries are written in document identifier order.

This parsing scheme generates a large number of small main memory data structures (i.e., token nodes, location nodes, and document entries). Main memory allocation, therefore, must be fast. The Parser preallocates main memory for the document and batch buffers and manages each buffer as a heap. To allocate memory from one of the heaps, the Parser need only advance a *current* pointer and perform a limit check to ensure that the heap has enough room to satisfy the current request. This heap based buffer implementation provides fast memory allocation and simple reclamation of an entire buffer—we merely reset the *current* pointer to the beginning of the heap. If the document buffer heap cannot satisfy the current memory request during document parsing, additional main memory is temporarily allocated to the document buffer, allowing the system to finish parsing the current document. Similarly, if the batch buffer cannot satisfy the current memory request during document buffer flushing, additional main memory is temporarily allocated to the batch buffer so that the system can finish flushing the document buffer, after which the batch buffer is flushed.

3.1.2 Merging

A temporary file produced by the Parser will contain one or more blocks of partial inverted lists, where each block corresponds to a batch of documents. The partial inverted lists within a block are *complete* inverted lists for the documents indexed during the corresponding batch. To build *final* inverted lists for the entire document collection, the partial inverted lists from all of the blocks must be merged.

The merge is performed in main memory by allocating an M byte merge buffer and dividing it evenly among all of the temporary file blocks. If there are N temporary file blocks, the merge buffer can be filled using N disk reads. Ideally, each disk read will consist of a single disk seek followed by a single data transfer of M/N bytes. This behavior is encouraged by the Parser, which sequentially writes batches to their temporary file blocks. If the aggregate space occupied by the temporary file blocks is T bytes, the total number

of disk seeks required will be $\frac{TN}{M}$. For example, using a 20 MB merge buffer, 2500 disk reads are required to merge 50 temporary file blocks that occupy a total of 1 GB on disk. Assuming ideal conditions—each disk read requires one disk seek and one data transfer—the 15 millisecond average head positioning time and 5 MB per second data transfer rate cited above yield 37.5 seconds for disk seeks and 200 seconds for data transfer. Even though reading the temporary file blocks in this fashion might appear to require significant random disk I/O, this example shows that disk seek time can be limited to less than 16% of the total I/O time.

The merge buffer provides an interface to the temporary file blocks for the Merger. In the rest of this discussion, we will describe the Merger as if it were interacting directly with the temporary file blocks. Bear in mind, however, that the Merger is actually manipulating the portions of the temporary file blocks that are currently buffered in the main memory merge buffer.

Once the merge buffer has been primed from the temporary file blocks, the actual merge process can begin. Recall that the Parser sorts a batch of partial inverted lists by token string before flushing the batch to its temporary file block. This ensures that all of the blocks will present their partial inverted lists in the same order when the blocks are read by the Merger. On each iteration of the merge process, the Merger considers all of the partial inverted lists currently presented for processing by the temporary file blocks and identifies the partial inverted list with the lexicographically smallest term string. This becomes the current token. The partial inverted lists presented by all of the other blocks will either have the same token string as the current token or a larger token string, allowing the Merger to find all of the partial inverted lists that match the current token simply by inspecting the current partial inverted list in each block.

When all of the matching partial inverted lists have been found, the Merger must concatenate them such that all of the merged document entries are sorted by document identifier. The document entries in a given block pertain to the documents parsed during

the corresponding batch and are already sorted within each partial inverted list by document identifier. For any two blocks, all of the document identifiers in the first block will be less than all of the document identifiers in the second block if the first block was created before the second block. Therefore, the document identifiers across blocks will be sorted if they are concatenated in order of block creation time.

Witten et al. [90] point out that the problem of selecting the smallest token from the set of partial inverted lists currently presented for processing is similar to the problem of managing a *priority queue*. A convenient data structure for managing a priority queue is the binary min-heap [16], which allows quick extraction of the minimum element in a set. A binary min-heap consists of an array A of n elements numbered 1 through n. Each element $i > 1$ in the array satisfies the min-heap property: $A[\textbf{parent}(i)] < A[i]$, where $\textbf{parent}(i) = \lfloor i/2 \rfloor$. The min-heap property guarantees that $A[1]$ is the minimum element in the array, and $O(\lg n)$ time is required to arrange A so that it satisfies this property.

The Merger was implemented using a binary min-heap. There is one element in the min-heap for each temporary file block being merged. Each element corresponds to the next partial inverted list to be processed from the associated block. The comparison function used for the min-heap property has two components. The primary component is a string comparison of the partial inverted list tokens for the two elements being compared. The secondary component is a comparison of the creation dates for the associated temporary file blocks. The current token is readily available from the top element in the min-heap, and matching tokens from the remaining blocks are found by extracting elements from the heap until a non-matching token appears at the top of the heap. The secondary component of the min-heap comparison function causes matching tokens to be extracted from the min-heap in temporary file block creation order, which is also the concatenation order for the partial inverted lists.

The Merger builds the final inverted list for the current token by concatenating the matching partial inverted lists as they are extracted from the min-heap. When all of the

matching partial inverted lists have been processed, the final inverted list for the current token is output. Each block that contributed a partial inverted list for the current token is advanced to its next partial inverted list and the new elements are inserted into the min-heap. A new current token is then selected from the min-heap and the merge process repeats, iterating until all of the partial inverted lists in the temporary file blocks have been consumed.

As the final inverted lists are produced, they may be written to disk in a sequential fashion, adhering to our rule of favoring sequential I/O over random I/O. Storing the final inverted lists on disk and making them available for future access is the responsibility of the *Inverted File Manager*. The Inverted File Manager has a significant impact on the functionality and performance of the overall system, and its design and implementation require careful consideration of a number of important issues. In the next section, these issues are considered and the Inverted File Manager that was designed and implemented is described.

3.2 The Inverted File Manager

The Inverted File Manager is responsible for storing the inverted lists created by the document inverter and making their contents available during query evaluation. Access to the inverted lists is provided through a high-level interface that includes operations such as store a new list, modify an existing list, open a specified list for access, sequentially output the document entries from an open list, and close a list. This interface serves to shield the rest of the system from the low-level inverted file implementation details, and confines consideration of a number of important issues to just the Inverted File Manager. In particular, the problem of how to support a dynamic document collection can in large part be solved within the Inverted File Manager.

To see this more clearly, consider the process of adding new documents to an existing document collection. The documents being added will contain a combination of old and

new terms. New terms do not appear in the existing document collection and require new inverted lists to be built and added to the inverted file. Old terms already have inverted lists in the inverted file; these lists must be updated with entries for the new documents. Since document entries within an inverted list are sorted by document identifier, if new documents are always assigned increasing document identifiers, the new document's inverted list entries can simply be appended to the existing inverted lists. The functionality required to support an append operation is the ability to grow existing inverted lists. In order to add new documents, therefore, we must be able to add new inverted lists to an existing inverted file and grow existing inverted lists already in the inverted file. Both of these operations require low-level support from the Inverted File Manager.

The tasks that must be performed by the Inverted File Manager are suggestive of a traditional data management problem that can be solved using a general data management facility. In fact, inverted file modification combined with multi-user access to the overall information retrieval system introduces a host of data management issues that naturally fall within the purview of a database management system [28]. Besides the issues of data storage, modification, and access, a multi-user system must contend with issues of concurrency control, recovery, and transactions that ensure consistent and complete actions against the database.

A logical solution to satisfying this long list of data management requirements is to implement the Inverted File Manager using a relational database management system (RDBMS). An RDBMS provides a number of tools for sophisticated management of structured data, including a data definition language for describing the schema of the database, a declarative query language for populating, manipulating, and accessing the database, a powerful transaction facility for consistent multi-user access to the database, and a backup and recovery mechanism to protect the database in the event of failures. An RDBMS can easily satisfy all of the functionality requirements imposed by the Inverted File Manager, and others have shown how such a system can actually be built [67, 26].

The problem with this approach is that an RDBMS is designed to support record based data with rich structure and interesting relationships. The relational data types are tailored to this record orientation and the data access methods are optimized for selecting subsets of records and attributes and joining multiple records based on their relationships. Inverted lists, on the other hand, have no pre-determined relationships with other inverted lists and are usually accessed in a sequential fashion. This access characteristic suggests that inverted lists should be represented as strings of bytes. Although an inverted list can be decomposed into records and attributes, storing it this way in a relational database forces the use of expensive join operations in order to effect sequential processing of the overall list. Basically, an RDBMS provides too much—the general data structures and access methods are wasted when managing an inverted file. Rather than simplifying manipulation of the inverted lists, an RDBMS complicates inverted list operations and imposes unnecessary overheads.

The limited way in which inverted lists are accessed leads to consideration of a custom software implementation for the Inverted File Manager. This is the route most information retrieval system developers have chosen. Assuming we are willing to build and maintain the system, the specific functionality and performance requirements of inverted list management can be satisfied exactly. This is a big assumption. While minimum functionality requirements can be met without too much work, satisfying the demands of a large, dynamic, multi-user system requires significant effort. Concurrency control and recovery mechanisms must be built. Some form of transaction model must be implemented. Low-level storage and retrieval mechanisms must be implemented. We essentially end up duplicating much of the effort that has already gone into building a generic database system. The custom software solution suffers from high development and maintenance costs to provide functionality that is preferably obtained elsewhere.

There are other "off-the-shelf" database management systems (besides an RDBMS) that are worth considering. To decide what kind of system is most likely to satisfy our

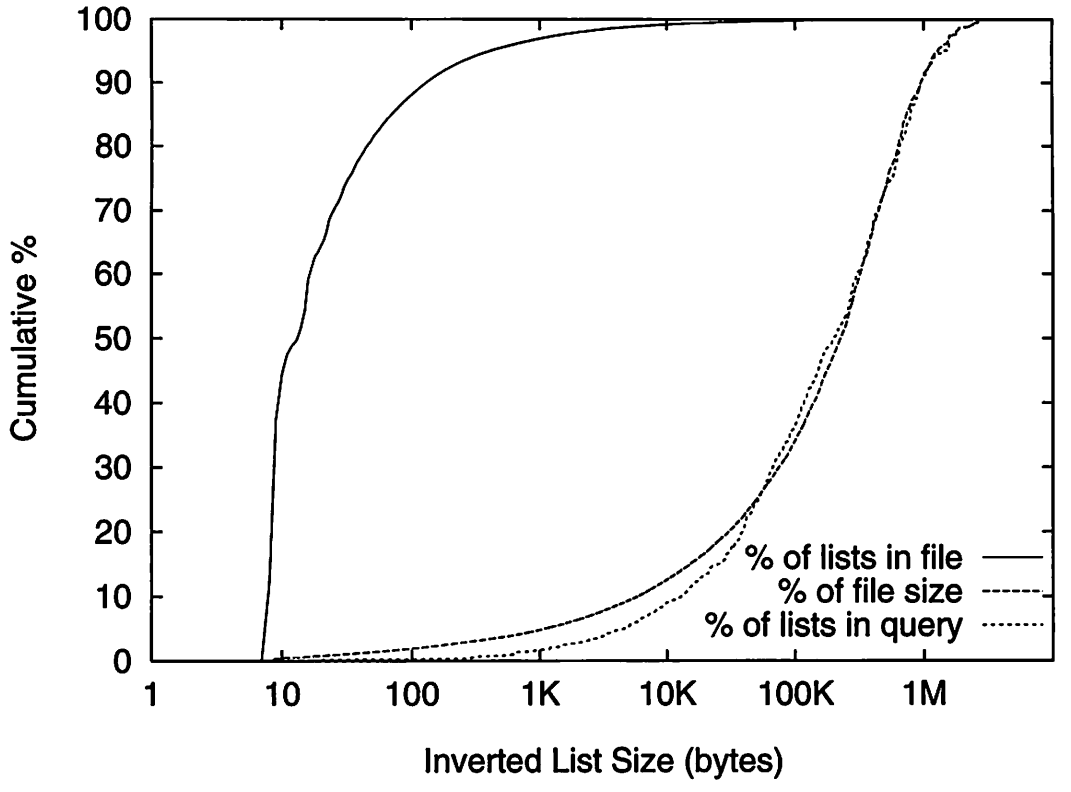


Figure 3.4 Inverted list size distributions

requirements, we need to consider further the size and access characteristics of the data we need to manage.

3.2.1 Inverted List Characteristics

The size of an inverted list depends on the number of occurrences of the corresponding term in the document collection. Zipf [94] observed that if the terms in a document collection are ranked by decreasing number of occurrences (i.e., starting with the term that occurs most frequently), there is a constant for the collection that is approximately equal to the product of any given term's frequency and rank order number. The implication of this is that most of the terms will occur a relatively small number of times, while a few terms will occur very many times.

Figure 3.4 shows the distribution of inverted list sizes for 2 GB of the TIPSTER document collection (CD-ROM disks 1 and 2) [39]. The inverted file contains 846,331 compressed inverted lists occupying a total of 720 MB. For a given inverted list size, the figure shows how many records in the inverted file are less than or equal to that size, and how much those records contribute to the total file size. As we might expect, the majority of the inverted lists are relatively small—approximately 95% of the lists are less than 1 KB. In fact, better than 50% of the lists are less than 16 bytes. It is also clear that these small lists contribute a very small amount to the total file size. Less than 5% of the total file size is accounted for by inverted lists smaller than 1 KB. In other words, better than 95% of the total file size is accounted for by less than 5% of the inverted lists in the file. The lists in this 5% can be quite large, with the largest list in the file weighing in at 2.5 MB.

If we could assume that inverted list access during query processing was uniformly distributed over the inverted lists, then supporting this activity (from a data management perspective) would be simplified, since the majority of the file accesses would be restricted to a relatively small percentage of the overall file. Unfortunately, this is not the case. Figure 3.4 also shows the distribution of sizes for the inverted lists accessed by a typical query set (produced from *TIPSTER Topics 51–100*). The majority of the records accessed are between 10 KB and 1 MB. This size range represents a small percentage of the total number of records in the file, but a large percentage of the total file size. Therefore, we must be prepared to provide efficient access to the majority of the raw data in the file.

We can, however, anticipate one access characteristic during query processing that works in our favor. It is likely that there will be non-trivial repetition of the terms used from query to query. This can be expected for two reasons. First, a user of an IR system may iteratively refine a query to obtain the desired set of documents. As the query is refined to more precisely represent the user's information need, terms from earlier queries will reappear in later queries. Second, IR systems are often used on specialized collections where every document is related to a particular subject. In this case, there will be terms

that are common to a large number of queries, even across multiple users. The implication of this is that caching inverted lists in main memory should prove beneficial.

In summary, an inverted file will display the following characteristics. Using the compression techniques described earlier, the inverted file's size will be 30–40% of the size of the raw document collection. The inverted lists contained within the inverted file will vary in size from less than 16 bytes to one or more megabytes, although the vast majority of the inverted lists will be quite small. During query processing, the longer lists will be favored and inverted list access will benefit from main memory buffering. During document additions, new inverted lists will be added to the inverted file and existing inverted lists will grow, with the longer inverted lists experiencing vigorous growth. Inverted list access must be efficient during query processing and collection modification, and mechanisms must exist to ensure that multiple users can simultaneously operate on the inverted file in a safe, consistent manner. Finally, even though inverted lists are actually built up from smaller components, at the storage management level they are best viewed as byte strings whose main operation is sequential scanning.

These requirements point to a data management system that combines a traditional database transaction facility and low-level storage management subsystem with a simple data model and low overhead. All of these features are found in a *persistent object store* (POS). A POS provides low-level storage and retrieval of objects, where an object is an identifiable unit of data. The services typically found in a POS include object creation, storage, and retrieval, disk management, buffering, transaction control, and recovery. The level of understanding possessed by the system about the contents of an object (i.e., an object's semantics) varies across different POS implementations. Usually, this understanding is limited to viewing objects as containers of bytes and references to other objects. This view eliminates the overhead associated with a more complex data model and allows the application to define the appropriate level of object semantics. The flip side of this is that the application must provide more functionality. This tradeoff, however, is appropriate

for a number of applications. For example, we can construct an object-oriented database management system using a POS as a foundation and building additional layers on top that provide a data model, data definition language, declarative query language, and other user interface applications.

The functionality and performance provided by a POS are ideally matched to the requirements of an Inverted File Manager. As such, we have used a POS to build our Inverted File Manager. In particular, we have used the Mneme persistent object store [62, 9, 8] developed under the direction of Eliot Moss at the University of Massachusetts. In the next section we consider Mneme in more detail.

3.2.2 The Mneme Persistent Object Store

The Mneme persistent object store was designed to be efficient and extensible. The basic services provided by Mneme are storage and retrieval of objects, where an object is a chunk of contiguous bytes that has been assigned a unique identifier. Mneme has no notion of type or class for objects. The only structure Mneme is aware of is that objects may contain the identifiers of other objects, resulting in inter-object references.

Objects are grouped into files supported by the operating system. An object's identifier is unique only within the object's file. Multiple files may be open simultaneously, however, so object identifiers are mapped to globally unique identifiers when the objects are accessed. This allows a potentially unlimited number of objects to be created by allocating a new file when the previous file's object identifiers have been exhausted. The number of objects that may be accessed *simultaneously* is bounded by the number of globally unique identifiers (currently 2^{28}).

Objects are physically grouped into *physical segments* within a file. A physical segment is the unit of transfer between disk and main memory and is of arbitrary size. Objects are also logically grouped into *pools*, where a pool defines a number of management policies for the objects contained in the pool, such as how large the physical segments are, how the

objects are laid out in a physical segment, how objects are located within a file, and how objects are created. Note that physical segments are not shared between pools. Pools are also required to locate for Mneme any identifiers stored in the objects managed by the pool. This would be necessary, for instance, during garbage collection of the persistent store. Since the pool provides the interface between Mneme and the contents of an object, object format is determined by the pool, allowing objects to be stored in the format required by the application that uses the objects (modulo any translation that may be required for persistent storage, such as conversion of main memory pointers to object identifiers). Pools provide the primary extensibility mechanism in Mneme. By implementing new pool routines, the system can be significantly customized.

The base system provides a number of fundamental mechanisms and tools for building pool routines, including a suite of standard pool routines for file and auxiliary table management. Object lookup is facilitated by *logical segments*, which contain 255 objects logically grouped together to assist in identification, indexing, and location. A hash table is provided that takes an object identifier and efficiently determines if the object is resident in main memory. Support for sophisticated buffer management is provided by an extensible buffering mechanism. Buffers may be defined by supplying a number of standard buffer operations (e.g., allocate and free) in a system defined format. How these operations are implemented determines the policies used to manage the buffer. A pool *attaches* to a buffer in order to make use of the buffer. Mneme then maps the standard buffer operation calls made by the pool to the specific routines supplied by the attached buffer. Additionally, the pool is required to provide a number of "call-back" routines, such as a modified segment save routine, which may be called by a buffer routine.

Mneme is particularly appropriate for the task of managing an inverted file for a number of reasons. First, an object store provides the ideal level of functionality and semantics. The data that must be managed can be naturally decomposed into objects, where each inverted list is a single object. More sophisticated mappings of inverted lists to objects can also be

easily supported with inter-object references, which allow more complex data structures to be built up. The primary function required is object retrieval, or providing access to the contents of a given object for higher level processing. Object access includes the traditional data management tasks of buffering and saving modifications. The processing of objects, however, is highly stylized and unlikely to be adequately supported within the data management system. Therefore, semantic knowledge about the contents of an object within the data management system is not only useless, but actually cumbersome. An object store that treats objects as containers of uninterpreted bytes and inter-object references provides just the right level of semantics.

Second, because Mneme is extensible, certain functions can be customized to better meet the management requirements of an inverted file. As we have seen, the objects in an inverted file come in a variety of sizes and exhibit unusual access patterns, such that a single physical storage scheme specifying clustering and physical segment layout will be inadequate. A better approach will be to identify groups of objects that can benefit from storage schemes tailored to the physical characteristics and access patterns of each group. In particular, buffer management policies should be customized for each group.

Finally, Mneme is tuned for performance and imposes a particularly low overhead along the critical path of object access. Memory resident objects are quickly located using the resident object table, and non-resident objects are faulted in with little additional processing. This can be contrasted with page mapping architectures of other object stores [49, 77] which have a fairly high penalty for accessing a non-resident object. These systems are optimized for localized processing of a large number of small objects, where the cost of faulting a page of objects can be amortized over many access to the objects in the page. This pattern of access differs from that expected in an inverted file, where large objects are accessed for sequential processing with little temporal locality.

3.2.3 The Mneme Solution

To build an Inverted File Manager using Mneme, we designed and implemented software for two layers of the system: the application interface layer and the Mneme extensibility layer. The application interface layer supplies the Inverted File Manager interface to the rest of the IR system, defines the semantics of the objects that are stored in Mneme, and translates the interface requests into Mneme operations. The Mneme extensibility layer provides hooks for extending and tailoring a number of the Mneme operations to better satisfy the specific requirements of inverted list management. Rather than address these layers individually, we will describe our implementation of the core inverted file management tasks and comment as appropriate on each task's implications for the different software layers. The core tasks include inverted list storage, inverted list lookup, document additions, and document deletions.

3.2.3.1 Inverted List Storage

The first step in the implementation process was deciding on how to map inverted lists to Mneme objects. To make this decision, we considered the basic operation that must be performed to retrieve an object from disk, namely, a disk read. A read in a typical Unix file system causes 8 KB to be read from disk. We chose to partition inverted lists into two groups: those less than or equal to 8 KB, called *short* lists, and those greater than 8 KB, called *long* lists. A short list is less than or equal to the size of an elemental file system read; it can be obtained in a single file system access. To guarantee that short lists are in fact retrieved in a single access, the low-level storage organization must align them so that they do not span file system page boundaries. Moreover, if the desired short list is less than 8 KB, the file system access will return more than just the desired inverted list. The implementation should ensure that the extra data retrieved contains useful information, such as other entire short lists.

The size distribution of inverted lists discussed in Section 3.2.1 shows that nearly 99% of the inverted lists are less than or equal to 8 KB and will be short. The remaining 1% of the inverted lists are larger than 8 KB and will be long. The long lists account for nearly 90% of the total inverted file size. Long lists, therefore, can be quite large and will require storage and access strategies substantially different from the short lists. In particular, long lists will be the most expensive lists to process during query evaluation and collection modification.

Consideration of these issues led to the following organization. Short inverted lists are stored in fixed length objects, ranging in size from 16 bytes to 8 KB by powers of 2 (i.e., 16, 32, 64, ..., 8K). When a new short list is created, an object of the smallest size large enough to contain the list is allocated. A long inverted list is stored as a linked list of 8 KB objects, requiring $\lceil \frac{l}{8192-k} \rceil$ objects, where l is the size of the long list in bytes and k is the size of the header and next pointer in the Mneme object.

The set of distinct object "types" used in this implementation is rather constrained, providing an opportunity for performance improvement via custom management of the objects. To take advantage of this opportunity, we designed and implemented three new object pools in Mneme. The new object pools constitute the modifications made at the Mneme extensibility layer. The first object pool, called the *small-object pool*, stores 16 byte objects using 4 KB physical segments. Each physical segment contains one logical segment, or 255 objects. The fixed object size and one-to-one mapping of physical and logical segments simplifies many of the pool operations, including object creation, object lookup in the file, and updates to the resident object table when transferring physical segments to and from the main memory buffer. Simplifying these tasks generally leads to smaller auxiliary tables and faster operations. The small-object pool will store approximately 50% of the inverted lists in an inverted file.

The second new object pool, called the *fixed-object pool*, stores fixed length objects ranging in size from 32 bytes to 4 KB by powers of 2. Objects are stored in 8 KB physical segments, where all of the objects in a given physical segment are the same size.

The number of objects per physical segment varies depending on the size of the objects residing in the physical segment. For example, a physical segment of 64 byte objects will contain 128 objects, while a physical segment of 512 byte objects will contain 16 objects. The fixed-object pool affords the same advantages as the small-object pool in terms of simplifying a number of the pool operations and improving storage and processing efficiency. Approximately 49% of the inverted lists will reside in the fixed-object pool.

The third object pool that we built for this application is the *page-object pool.* This pool manages page sized objects where all objects in the pool are the same size and each object is allocated in its own physical segment. The object size is specified when the page-object pool is instantiated. Although this size may be arbitrary, typically it will be some large power of 2. In this case, the object size is specified to be 8 KB. Again, the fixed object size and one-to-one mapping of objects to physical segments enables a more efficient implementation of certain pool operations, such as object creation, object lookup, and physical segment transfer to and from main memory.

The long inverted lists are stored using two separate page-object pools, with one pool storing the linked list head objects, and the other storing the remaining linked list data objects. This separation facilitates the delete operation, discussed below. Roughly 1% of the inverted lists in the inverted file will be stored this way. However, since all of the lists stored this way are long, these two object pools will account for the majority of the space in the inverted file.

This scheme efficiently allocates the large number of short inverted lists in the small and medium object pools, and provides a scalable storage structure for the long inverted lists. Physical segment sizes are sensitive to the file system transfer size, and multiple objects are efficiently packed in the physical segments that contain more than one object. Each object pool can also be attached to its own buffer manager, allowing the buffer size and management policies to be individually tuned to the requirements of each object pool. Furthermore, these policies can be adjusted depending on the current task at hand. For

example, the amount of buffer space required by the page-object pool during document indexing is substantially less than during query evaluation.

3.2.3.2 Inverted List Lookup

Once we have assigned the inverted lists to Mneme objects, we must provide some mechanism for identifying the object (or, in the case of a long list, the linked list head object) that contains the inverted list for a given term. An indexing structure commonly used for this purpose in database systems is the B+tree [15] (see [45] for additional references, and deletion pseudo-code!). A B+tree is a balanced search tree with an upper bound search time of $O(\log_b n)$ for an n node tree with branching factor b. In a disk based application, the tree nodes are typically the size of a disk page and the branching factor is relatively large, resulting in very short trees. For example, if we have one million terms and each term entry in the B+tree requires on average 20 bytes, the height of a B+tree with 8 KB nodes is 3 (counting the leaves as 1). All of the values associated with the keys are stored in the leaf nodes, simplifying scanning operations, but forcing all searches to traverse to a leaf node. With careful buffer management, however, we can keep most of the internal nodes resident in main memory and limit the number of disk reads to at most one per lookup (to obtain a leaf node).

The problem with a B+tree is that clustering of key/value pairs within a node is based on the key sort order. When a leaf node is made resident due to a search on one of its keys, the chance that we will search for another key in that same node before the node is flushed from the main memory buffer is no better than random. If instead we cluster together the key/value pairs most likely to be accessed during query evaluation, we will reduce the number of disk reads required during query evaluation and achieve a performance improvement. To accomplish this we need a method for identifying the keys most likely to be accessed and an indexing data structure that will support the clustering. The discussion in Section 3.2.1 shows that the more frequent terms are favored during query evaluation,

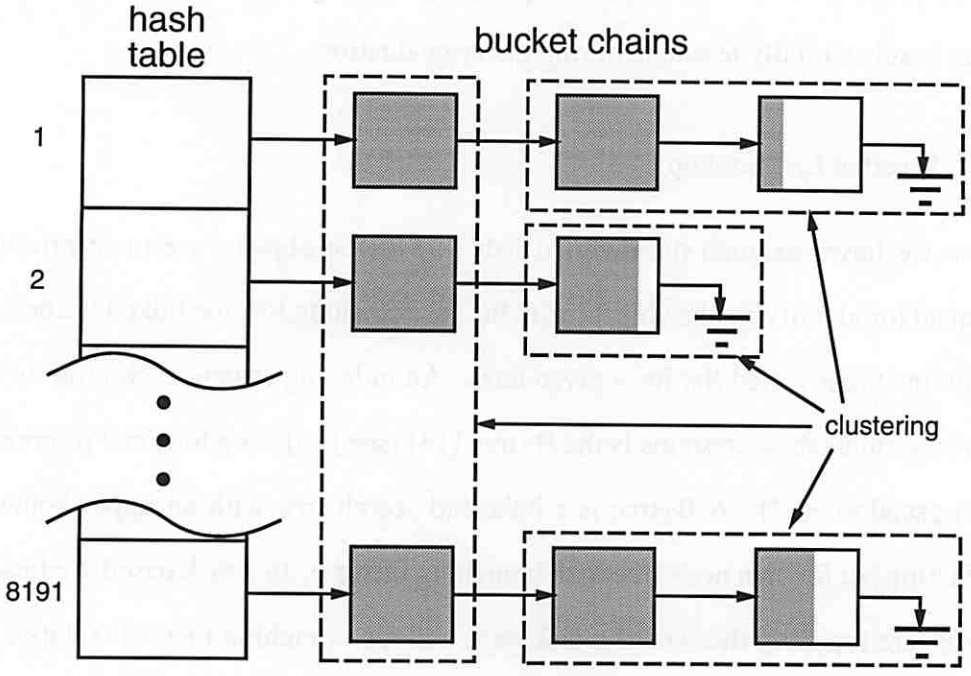


Figure 3.5 Inverted file hash table

suggesting that term frequency could be used to guide key clustering. Another, more pragmatic, approach would be to keep track of query term usage statistics over a period of time and use them to guide the clustering.

Since a B+tree cannot support arbitrary key clustering, if we want to take advantage of our clustering heuristic, we must find an alternative indexing structure. An indexing structure commonly used to store terms in an information retrieval system is a *hash table*. A hash table can incorporate an external clustering heuristic, making it the data structure of choice for this application.

We have implemented a Mneme-based hash table for our Inverted File Manager using the overall structure shown in Figure 3.5. The length of the hash table is fixed at 8191 slots. Each slot occupies 4 bytes, for a total hash table size of just under 32 KB. Rather than use a single 32 KB object to store the hash table, four 8 KB page objects are used. The motivation here is to increase concurrent access to the hash table in the event of updates. Each slot points to a linked list of buckets, which contain the key/value pairs for the keys

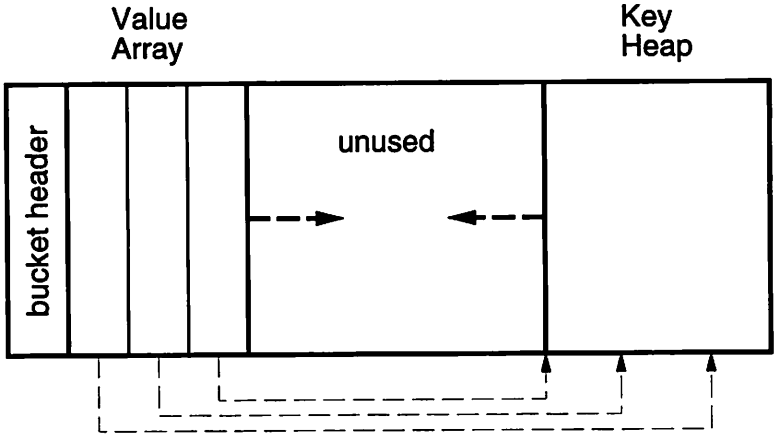


Figure 3.6 Hash table bucket

that hash to that slot. Each bucket is allocated in a 256 byte object using the fixed-object pool described above. A bucket has an array of values (object identifiers) at one end, a heap of keys (null terminated term strings) at the other, and a header containing a pointer to the next bucket in the chain, the number of entries in the bucket (*N*), and the offset of the key heap (see Figure 3.6). The value array and key heap grow towards each other, such that the maximum number of entries in a bucket is variable. The array and heap entries are paired-up from the inside out, eliminating the need for string heap offsets in the value array entries and minimizing the amount of space required by the key/value pairs (compression techniques excluded). The tradeoff is a more complex bucket search algorithm. To find a key/value pair in a bucket, we must scan the bucket's key heap from left to right, count the number of strings scanned before the key is found, and index into the value array with *N* minus count to obtain the corresponding value.

To locate the value for a given key, the hash function is applied to the key to obtain a slot index into the hash table. The appropriate hash table page object is retrieved and a chain pointer is obtained from the indexed slot. The chain pointer points to the first bucket object in the chain, which is retrieved and searched. If the key is found, its value is returned.

Otherwise, the next bucket is obtained and searched. This process is repeated until the key is found or there are no more buckets, in which case the key is not in the hash table.

The clustering heuristic is incorporated into the hash table by sorting the keys in each chain in decreasing order of term frequency. This causes within bucket clustering by placing the most frequent terms in each chain in the head bucket of that chain. We can additionally cluster across buckets by allocating all of the bucket chain heads in their own set of physical segments. Furthermore, to ensure that only a single disk read is required in the event that the desired key is not found in the head bucket, the rest of the buckets in a given chain are allocated in the same physical segment.

When the hash table is opened, the four 8 KB hash table page objects are read into their own private buffers, ensuring that they will never be swapped out by Mneme. The amount of buffer space allocated to the bucket objects is controlled by the application and should vary depending on the task at hand. When creating a new hash table from scratch, we allocate a small buffer (at least 16 KB, or enough for two physical segments) to the bucket objects. In this situation, we are sequentially allocating and filling bucket objects, and the new physical segments that contain these objects are written to disk as soon as they are full. During query processing, if our clustering heuristic is effective, we allocate a relatively modest amount of buffer space to the bucket objects. This can be tuned to a particular query environment based on observed object reference hit rates. When we are updating an existing hash table, we allocate as much buffer space as possible (up to the aggregate size of all of the buckets) to the bucket objects since every new term causes a bucket chain to be fully traversed during the initial search for the term.

3.2.3.3 Document Additions

New documents are added to an existing document collection in two steps. During the first step, complete inverted lists are created for the new document batch. In the second step, the new inverted lists are merged with the existing inverted file. The first step is executed

by the document inverter, and can proceed as described in Section 3.1 with no changes. The second step is carried out entirely within the Inverted File Manager. As the Merger outputs each final inverted list for the new document batch, the Inverted File Manager searches the existing inverted file for the term associated with the new inverted list. If the term is found, an inverted list already exists for the term and the new inverted list is appended to the existing inverted list. Otherwise, the term is new to the original document collection and the new inverted list is simply added to the existing inverted file.

The critical functionality here is the ability to grow an existing inverted list during an append operation. The inverted list storage scheme described above easily supports this operation. A short inverted list may have unused space at the end of its object and can grow to fill this space. When the list exceeds the object, a new object of the next larger size is allocated, the contents of the old object are copied into the new object, and the old object is freed. When a short list exceeds the largest object size (8 KB), it becomes a long inverted list and is stored as a linked list of 8 KB objects. Long inverted lists are grown by appending to the tail object in the linked list and adding a new object to the linked list when the tail is full.

The main advantage of this scheme is that the majority of the existing inverted file is untouched during an update, keeping the update costs more proportional to the size of the new document batch, rather than the size of the existing document collection. This behavior is provided by the long inverted list implementation. When a long inverted list is updated, only the head and tail objects in the linked list are accessed, leaving the majority of the data in the long lists untouched. Since nearly 90% of the data in an inverted file is stored in the long inverted lists, the majority of the inverted file should be untouched during an update. Note that the head object of a long list must be accessed to update the collection frequency and document count for the term and obtain the object identifier of the linked list tail. If instead this information is stored in the term hash table, accesses to the head objects can be eliminated at the expense of a larger term hash table. Increasing the size of the term hash

table, however, will cause it to demand more main memory during query evaluation. How to resolve this tradeoff depends on the frequency of updates versus the frequency of queries. The implementation described here is tuned for an environment where query evaluation is more frequent than document additions, hence a smaller term hash table is favored.

A potentially serious problem crops up during update operations on short inverted lists. Short inverted lists are stored in objects that share their physical segment with other objects. A physical segment, therefore, will contain multiple short inverted lists. When a short inverted list is retrieved for an update, all of the other short inverted lists in the same physical segment are simultaneously retrieved. It is possible that more than one inverted list in this physical segment must be updated during the batch update. It is also possible, however, that the physical segment will be swapped out of main memory before the other inverted lists have been updated, causing the same physical segment to be retrieved multiple times during the same batch update. If this thrashing behavior is extreme, performance will suffer.

One way to combat this effect is to allocate a larger main memory buffer so that more physical segments may be resident simultaneously. This is a bad solution for three reasons. First, for large inverted files the amount of space occupied by all of the short inverted lists will still be quite substantial, such that it is impossible to allocate a large enough buffer. Second, during an update, main memory is also required by the Merger (for its merge buffer) and the term hash table, making main memory a scarce resource. Third, caching modified physical segments for extended periods of time will interfere with the amount of concurrency available in the system.

A better solution to this problem is to apply the short inverted list updates in a more advantageous order. In particular, all of the short inverted lists that coexist in a physical segment should be updated simultaneously. As currently described, inverted lists are updated in sorted term string order. This order is determined by the Parser, which writes partial inverted lists in term string order. Term string order is unrelated to the assignment

of inverted lists to physical segments. Mneme object identifier order, however, *is* related to the assignment of objects to physical segments. We have implemented the small-object and fixed-object pools in such a way that a physical segment contains objects identified by a continuous range of the object identifier space. In other words, when the identifiers for the objects in a physical segment are listed out in the order in which the objects appear in the physical segment, the identifiers form the sequence $\{n_1, n_2, n_3, \ldots \mid n_{i+1} = n_i + 1\}$. Moreover, the physical segments tend to be allocated in the file in such a way that the identifiers for the objects in a physical segment earlier in the file will be less than the identifiers for the objects in a physical segment later in the file.

To take advantage of object identifier order during updates, we extended the Parser to sort partial inverted lists based on existing inverted list object identifiers. Recall that the partial inverted lists are sorted just before the batch buffer is flushed to a temporary file block. Rather than sort on term string at this point, the Parser probes the existing inverted file's term hash table for each of the partial inverted lists in the batch buffer and obtains object identifiers for the existing inverted lists. A partial inverted list associated with a new term (i.e., for which there is no existing inverted list) is assigned object identifier 0. Note that the Parser's batch buffer has the same organization as the inverted file's term hash table (Figures 3.3 and 3.5). This is by design, and is intended to improve locality of the term hash table probes as we iterate through the batch buffer.

Now the Parser can sort the partial inverted lists by object identifier. So that new partial inverted lists are added to the inverted file after any existing inverted lists have been updated, object identifier 0 is considered to be greater than all other object identifiers during the sort. Furthermore, to distinguish amongst the new terms, partial inverted lists that have been assigned object identifier 0 are sorted secondarily on term string. After the sort, the partial inverted lists are written to the temporary file block in existing inverted list object identifier order. When the temporary file blocks are merged, the final inverted lists produced are presented to the Inverted File Manager in the desired object identifier order.

Updates to existing inverted lists are performed with no physical segment thrashing, and the physical segments are retrieved in a series of scans over the inverted file. Moreover, we only need to allocate enough buffer space to hold the physical segment currently being updated. Once the last object in the segment has been updated, that segment will not be accessed again during the current batch.

3.2.3.4 Document Deletions

Document deletion is slightly more complicated. Deleting a document involves the deletion of all of the entries for that document in the inverted lists for the terms that appear in that document. There are three general approaches for accomplishing this. In the first approach, the deleted document is re-parsed (lexically analyzed, stopped, and stemmed) to identify the terms contained within the document and allow the affected inverted lists to be accessed and updated directly. This approach suffers from two problems. First, the document source must be available. This may not always be the case, especially if the inverted file is being updated to reflect the loss or unavailability of the document. Second, the parse that is performed for deletion must produce the exact same tokens as the parse that was performed when the document was originally indexed. The parser may have been upgraded or modified since the document was originally parsed, making an exact match impossible.

The second approach involves the use of an auxiliary index. For each document, the index stores a list of the terms that occur in the document. When a document is deleted, its list of terms is obtained and used to identify the inverted lists that must be updated. This eliminates the problems inherent in the first approach, but introduces an additional index that must be maintained and stored. If each term identifier in the auxiliary index requires 4 bytes of storage, then such an index for the 3.2 GB TIPSTER document collection described below would occupy 541 MB, or 17% of the space occupied by the document collection.

This estimate is based on 141,929,665 document entries in the corresponding inverted file. Of course, compression techniques could substantially reduce this overhead.

In the third approach, all of the inverted lists in the inverted file are scanned and entries for the deleted document are removed from the inverted lists as they are found. This solution is more robust than the first approach, imposes no storage overhead, and is more straightforward than the second approach; for these reasons it is the one that we have implemented. The scan of the inverted file is driven at the object level and is supported by Mneme's object scanning facility. This facility allows an object pool to iterate through its objects in order of object identifier. As we saw earlier, processing objects in object identifier order results in sequential processing of the inverted file.

Due to the high cost of scanning the inverted lists, individual document deletions are not immediately applied to the inverted file. Instead, they are buffered up in a *document delete list* and eventually applied to the inverted file in a large batch purge. In the mean time, the document delete list is used to filter query processing results. Before the final document ranking for a query is returned to the user, documents that appear in the delete list are removed from the answer. Note that management of the document delete list is external to the Inverted File Manager and falls outside the boundaries of our implementation (i.e., it was implemented by others).

The batch purge begins by scanning the small-object and fixed-object pools, which contain the short inverted lists. To process a short inverted list, we decompress the existing list and search for entries that match the documents in the document delete list. Any matching entries are deleted from the inverted list and the remaining inverted list is recompressed into the same object. The newly freed space in the list will appear at the end of the object and be available for future allocation. If no matching document entries were found, the decompressed inverted list is discarded and the object is left unmodified. Should all of the document entries be deleted from an inverted list, the list's object can be freed and the corresponding term can be deleted from the term hash table.

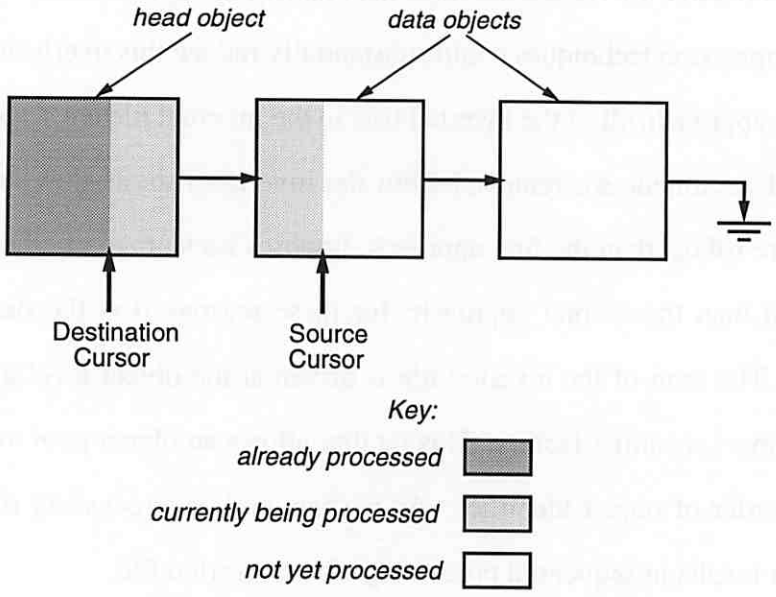


Figure 3.7 Deletion in a long inverted list

The long inverted lists are processed next. The page-object pool that contains the linked list head objects is scanned, giving us the first object for each long inverted list. A long inverted list is processed in chunks using two cursors: a source cursor and a destination cursor, shown in Figure 3.7. A portion of the inverted list (about 8 KB) is read from the source cursor and decompressed into a work buffer. We scan the work buffer and remove any entries found for documents listed in the document delete list. When the work buffer has been processed, it is re-compressed and written to the destination cursor. The destination cursor follows the source cursor and will gradually lag farther and farther behind the source cursor as more document entries are deleted. When the entire inverted list has been processed, the hole for the deleted document entries will have percolated to the end of the list. Any unused objects at the tail of the linked list can be freed.

3.3 Experimental Results

To evaluate our implementation of the Inverted File Manager, we ran a series of experiments to measure bulk indexing speed, incremental update speed, disk space utilization,

and the impact of the inverted file construction technique on query processing speed. Below we describe the experimental platform, the test collection used, and the results of our measurements.

3.3.1 Platform

All of our experiments were run as superuser with logins disabled on an otherwise idle DECSystem 3000/600 (Alpha AXP CPU clocked at 175 MHz) running OSF/1 V3.0. The system was configured with 64 MB of main memory, one DEC 1.0 GB RZ26L Winchester SCSI disk, and one Micropolis 4.3 GB M3243 SCSI disk. The executables were compiled with the DEC C compiler driver 3.11 using optimization level 2. All of the data files and executables were stored on the larger local disk, and a 64 MB "chill file" was read before each parse, merge, or query processing run to purge the operating system file buffers and guarantee that no inverted file data was cached by the file system across runs. The effectiveness of the chilling procedure was verified by measuring the number of file inputs charged to a test program that reads a 1 MB file. The test program was run 10 times, both with and without chilling between iterations. Without chilling, the number of file inputs required by each iteration after the first is 0. With chilling before each iteration, the number of file inputs required by every iteration is 133. Since the file system block transfer size is 8 KB, 128 file inputs are required to read the test file data. The remaining 5 file inputs are required by the file system to read directory and file structure data. All times reported were measured with the GNU time command.

3.3.2 Test Collection

For our experiments we used the 3.2 GB TIPSTER document collection distributed for the *Third Text REtrieval Conference (TREC-3)* [40]. The TIPSTER document collection is broken down into a number individual files containing a wide variety documents. Table 3.1 gives the size, number of documents, number of term occurrences (Postings), and number

Table 3.1 TIPSTER document collection file characteristics

File	MB	Documents	Postings	Terms
wsj87	125.6	46448	11404792	125035
wsj88	104.4	39904	9729119	53925
wsj89	35.7	12086	3247328	16739
doe	183.8	226087	17240754	118444
ziff	242.3	75180	21247322	96213
ap	254.2	84678	22386691	78330
fr_a	156.7	15640	14455792	84781
fr_b	103.0	10320	9464721	44837
wsj90	69.8	21705	6203493	19339
wsj91	139.2	42652	11853656	35432
wsj92	32.9	10163	2747163	7808
ziff2	175.5	56920	15272205	39598
ap2	237.2	79919	20607785	46125
fr2	209.2	19860	19239417	66612
ziff3_a	192.4	56398	17146002	43689
ziff3_b	152.3	104623	8830722	17795
ap3	237.5	78321	20692345	42228
patn	242.6	6711	19493312	76986
sjm_a	189.9	60399	14106777	34533
sjm_b	97.0	29858	7199499	14228
total	3181.2	1077872	272568895	1062677

of uniquely indexed terms for each file. The term count for a given file is the number of new terms added by that file to all of the files listed earlier in the table. The files contain documents from the *Wall Street Journal* (**wsj***),[1] Department of Energy abstracts (**doe**), Ziff-Davis Publishing *Computer Select* disks (**ziff***), *AP Newswire* (**ap***), *Federal Register* (**fr***), U.S. Patents (**patn**), and the *San Jose Mercury News* (**sjm***). This is one of the first publicly available large scale document collections, and has become a standard test collection in the information retrieval research community.

[1]Due to human error, the local version of **wsj89** used for these experiments was missing 294 documents from the original distribution.

Table 3.2 TIPSTER file parsing results

File	Time (sec)	msec/post	Temp Blocks	Temp Size (MB)
wsj87	800	0.070	9	42.4
wsj88	674	0.069	8	36.5
wsj89	227	0.070	3	12.3
doe	1398	0.081	14	63.3
ziff	1515	0.071	14	70.3
ap	1583	0.071	18	86.1
fr_a	889	0.061	7	39.1
fr_b	587	0.062	5	26.0
wsj90	449	0.072	5	23.4
wsj91	849	0.072	9	44.6
wsj92	201	0.073	3	10.4
ziff2	1075	0.070	10	50.4
ap2	1491	0.072	16	78.9
fr2	1241	0.064	9	51.8
ziff3_a	1145	0.067	11	56.3
ziff3_b	784	0.089	6	29.9
ap3	1494	0.072	16	79.5
patn	1214	0.062	6	42.8
sjm_a	1069	0.076	11	55.0
sjm_b	552	0.077	6	28.1
total	19237	0.071	186	927.1

3.3.3 Bulk Indexing

The first question we are interested in is how well the overall indexing scheme described in Section 3.1 works. To answer this question, we measured the elapsed (wall-clock) time required to index the entire 3.2 GB TIPSTER document collection. Using an 18 MB batch buffer, the Parser was run separately on each of the TIPSTER files. Note that for the experiments described in this chapter, INQUERY's feature recognizers were not used. The feature recognizers identify city names, company names, foreign country names (i.e., not the United States), and references to the United States, and increase the time required for parsing.

For each file, Table 3.2 gives the elapsed parsing time in seconds, the number of temporary file blocks produced, and the aggregate size of the temporary file blocks. The elapsed parsing time depends on the size of the input document file, so the table also gives a normalized parsing time in terms of milliseconds per posting. The total elapsed time required to parse the entire collection was 19237 seconds, or 5 hours 21 minutes. A total of 186 temporary file blocks were produced occupying 927 MB, or 29% of the space required by the raw document collection. On average, 0.071 milliseconds were required per posting. The table indicates, however, that parse time per posting fluctuates depending on the size of the documents being parsed. The Department of Energy abstracts (**doe**) and some of the Ziff-Davis publications (**ziff3_b**) contain relatively short documents and require more time per posting. The *Federal Register* (**fr***) and U.S. Patents (**patn**) contain relatively long documents and require less time per posting. This discrepancy is caused by the overheads associated with parsing a single document, e.g., flushing the document buffer to the batch buffer. Longer documents can amortize this overhead over more postings, resulting in lower per posting costs.

All of the temporary file blocks produced by the Parser were then merged by the Merger using a 20 MB merge buffer. Mneme was allocated 16.4 MB for its buffers, of which 14.3 MB were allocated for the term hash table objects, 2 MB were allocated for Mneme system (meta) data, and the remaining 126 KB were allocated for inverted list objects. The term hash table buffer was large enough to keep the entire term hash table memory resident throughout the merge. This is done to prevent thrashing during hash table insertions—a hash table insert typically requires access to an entire bucket chain. The inverted list objects require a relatively small amount of buffer space since the only operation we are performing here is creation. Once an object has been created, it can be flushed from main memory.

The Merger required 39 minutes to merge all of the temporary file blocks and store the new inverted lists. This gives a total time of 6 hours to index the 3.2 GB TIPSTER document collection, or an overall indexing rate of 530 MB per hour.

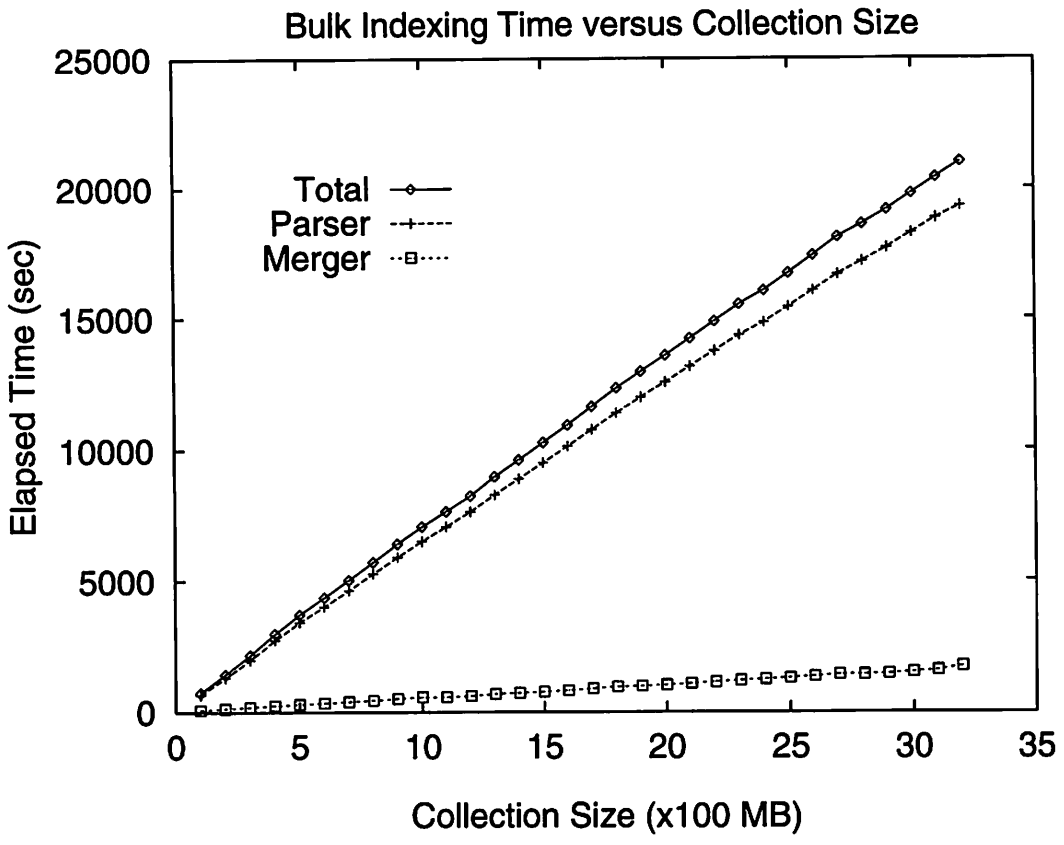


Figure 3.8 Bulk indexing times

The next question of interest is how well this indexing scheme scales. To answer this question, we divided the TIPSTER document collection into 32 batches of approximately 100 MB each.[2] The first batch contains the first 100 MB of **wsj87**, the second batch contains the remaining 25.6 MB of **wsj87** and the first 74.4 MB of **wsj88**, etc. We then indexed 32 different document collections ranging in size from 100 MB to 3.2 GB, where a document collection of size $n * 100$ MB consists of batches 1 through n. The elapsed time to index each of these document collections is plotted in Figure 3.8. The figure suggests that total indexing time scales linearly with the size of the document collection being indexed. This is due mainly to the Parser, which dominates the total running time, but maintains a constant time per posting rate (as discussed earlier).

[2]The last batch is actually only 80.7 MB.

Table 3.3 TIPSTER Inverted file object statistics

Object Size (B)	Number	Space Usage (MB)				Utilization (%)
		Total	Data	Free	Mneme	
16	569188	8.7	5.1	2.5	1.112	59.1
32	228940	7.0	4.4	2.2	0.447	62.6
64	104034	6.3	4.3	1.9	0.203	67.6
128	60849	7.4	5.1	2.2	0.119	68.8
256	35795	8.7	6.1	2.6	0.070	69.4
512	21257	10.4	7.3	3.1	0.042	70.1
1024	13654	13.3	9.4	3.9	0.027	70.7
2048	9170	17.9	12.6	5.3	0.018	70.4
4096	6536	25.5	18.0	7.5	0.013	70.5
8192	102093	797.6	749.1	47.5	1.005	93.9
total	1151516	902.9	821.4	78.5	3.054	91.0

If we change the collection size units in Figure 3.8 from bytes to postings and fit a line to the Total points using a least-squares fit linear regression, the line obtained is $y = 114.63 + 7.58 \times 10^{-5} x$. The coefficient of determination for the linear regression relationship is $r^2 = 0.99976$, suggesting a very strong linear relationship. The slope of the line indicates an overall indexing rate of 0.076 msec/posting. This is consistent with the overall parsing rate of 0.071 msec/posting reported in Table 3.2, with the difference due to the merge costs included in the Total time.

Space utilization statistics for the final inverted file created for the 3.2 GB TIPSTER collection are given in Table 3.3. For each object size, the table gives the number of objects in the file, total space occupied by the objects, amount of inverted list data stored in the objects, free space in the objects (i.e., currently unused space that may be allocated in the future), Mneme overhead (object headers and data structures), and effective space utilization (Data/Total $*$ 100). The smallest objects are poorly utilized, with less than 60% of their space occupied by inverted list data. However, they account for a very small portion of the total inverted file size. On the other hand, most of the 8 KB objects are fully utilized since they are in the middle of a long inverted list. The overall object utilization is quite

Table 3.4 Indexing variations for 3.2 GB TIPSTER collection

Variation	A	B	C	D
Stemming	yes	no	yes	yes
Stopping	yes	yes	yes	no
Proximity	yes	yes	no	no
Parser (sec)	19297	17909	18744	17327
Merger (sec)	1726	2219	1029	1426
Total (sec)	21023	20128	19773	18753
Temp Blocks (MB)	929	1043	541	673
Inv File (MB)	913	1003	513	634
Vocab Size	1062667	1229847	1062667	1062690
Term Hash Tbl (MB)	13.8	16.5	13.8	13.8

high—better than 90%. Mneme system data and free space in the object file add a negligible 9.7 MB to the object space total, for a total inverted file size of 913 MB. The term hash table requires an additional 13.8 MB, such that the overall inverted index requires 927 MB, or 29% of the space occupied by the original document collection.

Given that the Parser accounts for nearly 90% of the total indexing time, a closer look at how the Parser spends that time is in order. Of the 19237 seconds spent parsing, 18076, or nearly 94%, are charged to user CPU time. Since the Parser appears to be CPU bound, it was profiled using the gprof profiler. The resultant profile report indicates that only 16% of the CPU time is spent assembling and handling inverted lists, i.e., adding entries to the document buffer, flushing the document buffer to the batch buffer, and flushing the batch buffer. The rest of the CPU time is spent as follows: 61% is spent scanning and parsing, 14% is spent checking for stop words, 8% is spent updating the document catalog, and 1% is spent stemming. From this profiling data we conclude that our efforts at improving the efficiency of inverted list assembly have successfully eliminated the bottlenecks imposed by that portion of the indexing system. Scanning and parsing are now the most expensive components of the Parser and have the greatest need for future performance tuning.

To provide a complete picture of the performance of our indexing system, we evaluated a number of variations on the original indexing process described above. Each variation was run on the 3.2 GB TIPSTER collection. Using an 18 MB batch buffer, the collection was parsed in 32 batches of approximately 100 MB each. The temporary file blocks produced were then merged in a single step using a 20 MB merge buffer. Results for the different variations are shown in Table 3.4. Note that variation A is the original indexing process.

First, the Parser profile suggests that stemming is a relatively insignificant component of the overall cost. To verify this, we measured the time required to index the 3.2 GB TIPSTER collection without stemming, shown as variation B in Table 3.4. Compared to the original indexing process (variation A), parse time decreased by 7%, merge time increased by 29%, and total indexing time decreased by 4%. The measured effect of stemming on parse time is actually larger than the profile suggests, although the parse time savings obtained by eliminating stemming is still modest and is offset somewhat by an increase in the time required to merge.

The increase in merge time is due to a 12% increase in the size of the temporary file blocks, a 10% increase in the size of the final inverted file, a 16% increase in the size of the vocabulary (the number of unique terms indexed), and an overall increase in the string length of the indexed terms. The temporary file blocks and final inverted file are larger in the absence of stemming because many of the inverted lists in the stemmed version are now split into multiple inverted lists for terms that would otherwise stem to the same term. This increases the average distance between two occurrences of the same term, eliminating some of the benefits of delta encoding and reducing the effectiveness of the inverted list compression algorithm. The larger vocabulary and term string length additionally contribute to the increase in temporary file block size. More importantly, they increase the size of the term hash table by 19%, which is created and written during the merge.

Next, we explored the cost of storing term occurrence locations (i.e., proximity information) in the inverted file. In variation C, term occurrence locations were *not* stored when indexing the 3.2 GB TIPSTER collection. Compared to variation A, parse time is reduced by 2.8%, merge time is reduced by 40%, and total indexing time is reduced by 5.9%, for an overall indexing rate of 580 MB per hour. The temporary file blocks produced by the Parser occupy a total of 541 MB, and the final inverted file produced by the Merger occupies 513 MB, or just 16% of the size of the raw document collection. Compared to the original indexing process, temporary file block and final inverted file space requirements are reduced by 42% and 44%, respectively. Viewed another way, storing term occurrence locations increases the size of the final inverted file by 78%.

When term occurrence locations are not stored, little time is saved during parsing. This is expected given that the savings are confined to inverted list assembly and handling, which account for only 16% of the CPU time spent in the Parser. Scanning, parsing, stopping, stemming, and document cataloging are unchanged. The substantial reduction in the size of the temporary file blocks, however, yields a large savings at merge time, where the amount of data that must be merged is nearly halved. The reduction in total indexing time is rather modest since merging accounts for only 10% of the total time. In the indexing system described here, the extra processing cost of indexing and storing term occurrence locations is minimal. The most noticeable expense is an increase in the size of the inverted file. We should note that this comparison was made using an inversion algorithm originally designed to store term occurrence locations. It is likely that the algorithm could be better tuned for the case where term occurrence locations are not stored, resulting in a more significant savings in parse time.

Finally, the Parser profile suggests that stopping is a relatively expensive operation. The last variation measured, variation D in Table 3.4, does no stopping and does not store term occurrence locations (but does stemming). Compared to variation C, eliminating stopping reduces parse time by 8%, increases merge time by 39%, reduces total indexing time by

5%, and increases both temporary file block and final inverted file space requirements by 24%. Since the terms eliminated by stopping are highly frequent, indexing those terms (by not stopping) increases the size of the temporary file blocks and final inverted file. If we additionally stored term occurrence locations in this variation, the size increases would be even more substantial—the number of postings indexed increases by 66% when stopping is turned off. In spite of the increased file sizes and merge time compared to variation C, variation D is the fastest variation we measured, with an overall indexing rate of 614 MB per hour.

3.3.4 Incremental Update

We evaluated the ability of our Inverted File Manager to accommodate document additions by indexing the 3.2 GB TIPSTER document collection in a series of *incremental updates*. In an incremental update, a new batch of documents is added to an existing document collection and the necessary updates to the inverted file are performed in-place. We use the term *incremental* to distinguish this process from the traditional method of adding new documents, which simply re-indexes the entire document collection from scratch, building a whole new inverted file.

Using the 100 MB document batches described earlier, we incrementally indexed the TIPSTER collection by successively adding each document batch. Figure 3.9 shows the elapsed time required to add each successive batch, where the x-axis enumerates the 100 MB batch updates in the order that they were applied. For example, at batch update 5, we have already indexed 400 MB in the first 4 batches, and are now adding 100 MB of new documents to the existing 400 MB document collection. The figure shows a rapidly increasing cost per update when the existing document collection is small. However, as the existing document collection becomes larger, the cost per update starts to level off. This is an encouraging result, indicating that the overall technique will scale well. The cumulative

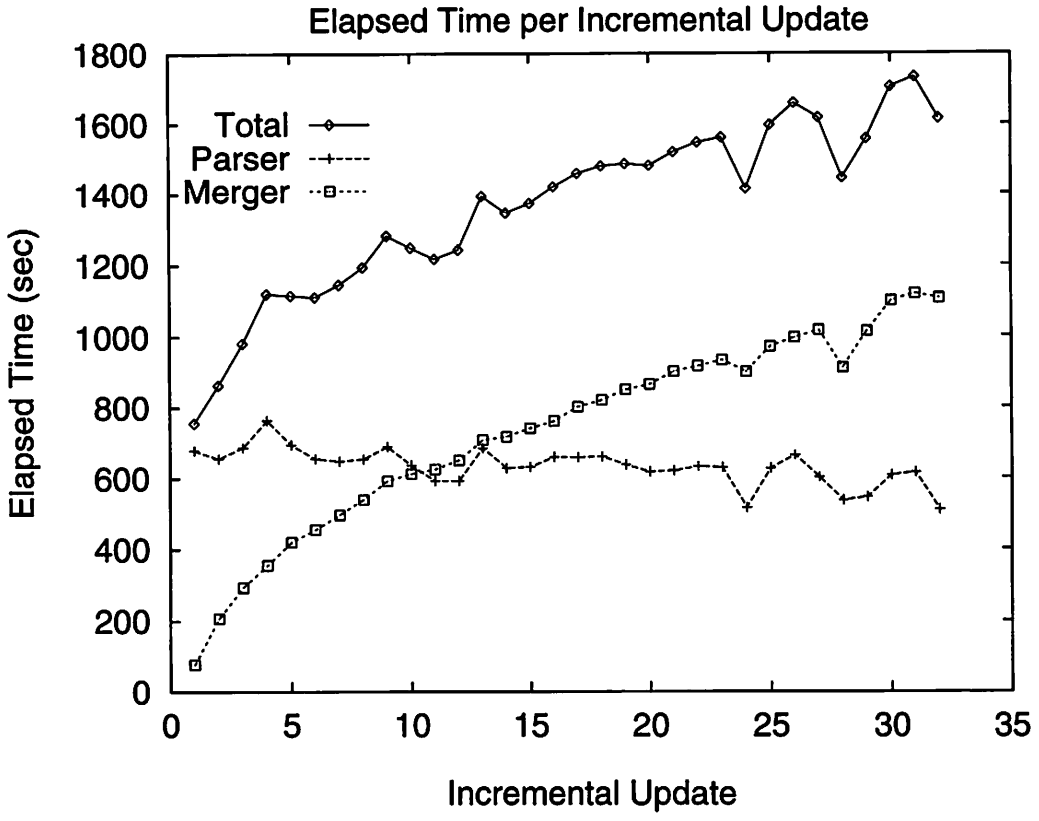


Figure 3.9 Incremental update times

elapsed time required to incrementally index the entire 3.2 GB collection in 32 batches is just over 12 hours, giving an overall indexing rate of 265 MB per hour.

Consistent with the results reported above for bulk indexing, the parsing rate is essentially constant with minor variations depending on the size of the documents in the particular batch. In the case of an incremental update, however, we now must access the term hash table during parsing so that partial inverted lists are written in inverted list object identifier order. The extra cost of this operation is shown in Figure 3.10, which compares bulk and incremental parsing costs for a series of collection sizes. The bulk parsing costs are the same costs reported earlier in Figure 3.8. The incremental parsing costs are the cumulative parsing costs accrued when incrementally indexing in 100 MB batches. The

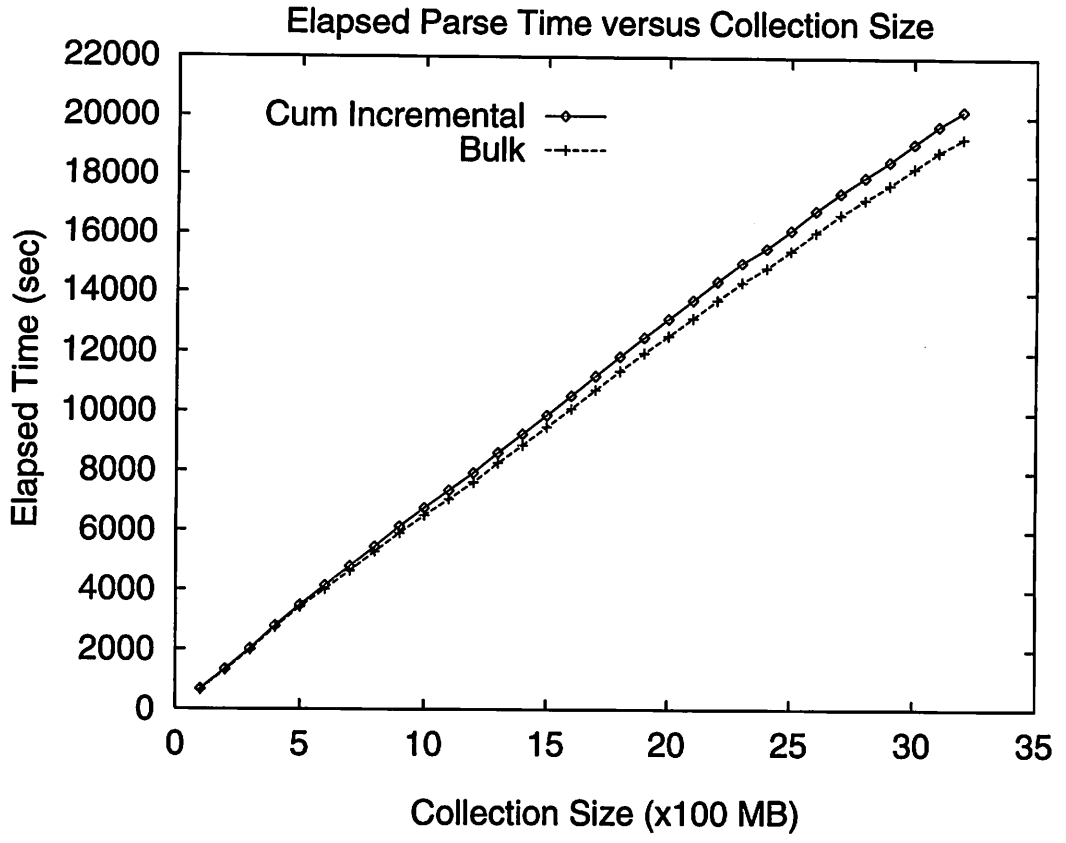


Figure 3.10 Parse time comparison

figure shows that incremental parsing costs are only slightly higher than bulk parsing costs, and the incremental version scales well with collection size.

Unlike the bulk indexing version, the Merger eventually dominates running time in the incremental version. The new cost is not actually due to the merge process itself, but rather the extra work that must be performed by the Inverted File Manager in the form of reading existing inverted file data. When an existing inverted list is updated, it must first be retrieved from the inverted file. Although we have taken pains to make this retrieval sequential and efficient, the fact remains that a certain portion of the existing inverted file must be read from disk. Figure 3.11 shows the amount of existing inverted file data read during each incremental update. When merge time is plotted versus bytes of existing inverted file read for each incremental update, we see a strong linear relationship. Figure 3.12 shows each

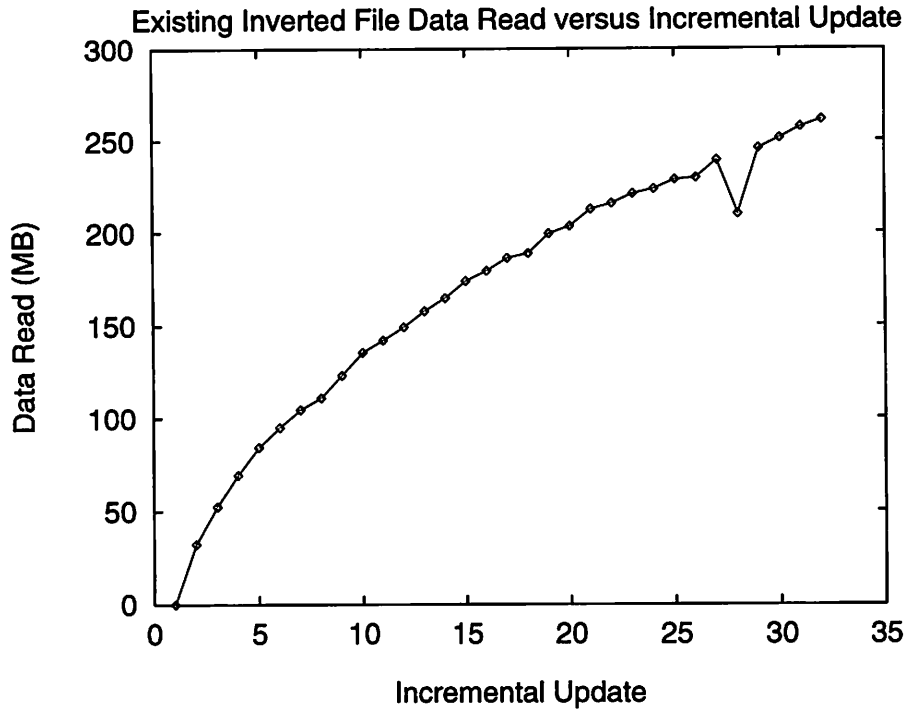


Figure 3.11 Inverted file data read per update

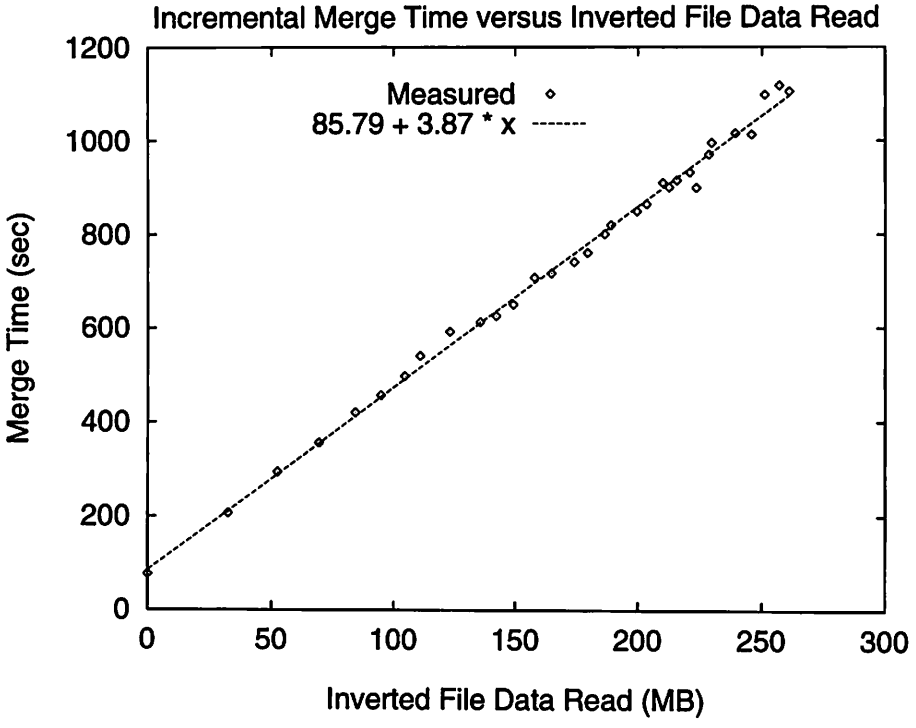


Figure 3.12 Incremental merge time versus data read

of these data points, along with the curve $y = 85.79 + 3.87x$, obtained via a least-squares linear regression. The coefficient of determination for the linear regression relationship is $r^2 = 0.9956$, suggesting a very strong linear relationship and an incremental merge time that, for a given batch size, is entirely dependent on the amount of existing inverted file data read.

Our inverted file organization allows us to bound the amount of inverted file data that must be read when updating a given inverted list. A long inverted list is the most expensive list to update, requiring two 8 KB objects to be read from the existing inverted file. All other lists (i.e., short lists) are less than 8 KB; at most one disk access is required to read these lists. In practice, less than one disk access is required per short inverted list update since short lists are clustered in segments and our update algorithm takes advantage of this clustering. In the worst case, therefore, the total amount of data that must be read is bounded by a constant times the size of the vocabulary.

Heaps [43] suggests that vocabulary size can be estimated with $V = aN^b$, where V is the size of the vocabulary and N is the number of term occurrences (postings) in the document collection. Using least squares fitting, we fit this function to the vocabulary sizes measured when indexing the 3.2 GB TIPSTER collection. The constant and exponent obtained from the fitting are $a = 2.693395$ and $b = 0.664163$, giving a sub-linear function. Figure 3.13 shows this function plotted along with the actual vocabulary sizes measured.

The function $V = 2.693395N^{0.664163}$ is linear when plotted using a log-log scale. If we take the log of both sides, we get $\ln(V) = \ln(2.693395) + 0.664163 \ln(N)$. Substituting y for $\ln(V)$ and x for $\ln(N)$, we get the linear function $y = 0.990803 + 0.664163x$. This is plotted in Figure 3.14, along with the logs of the measured vocabulary growth data points (from Figure 3.13). The coefficient of determination for the linear relationship is $r^2 = 0.9979$, suggesting a very strong linear relationship, and a function that models vocabulary growth quite well.

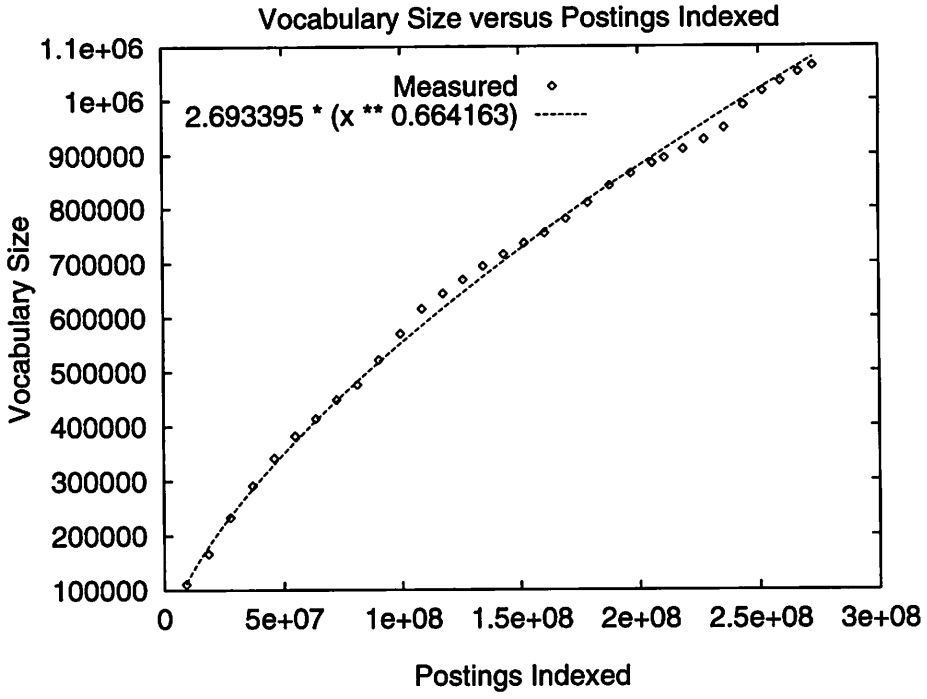


Figure 3.13 TIPSTER vocabulary growth

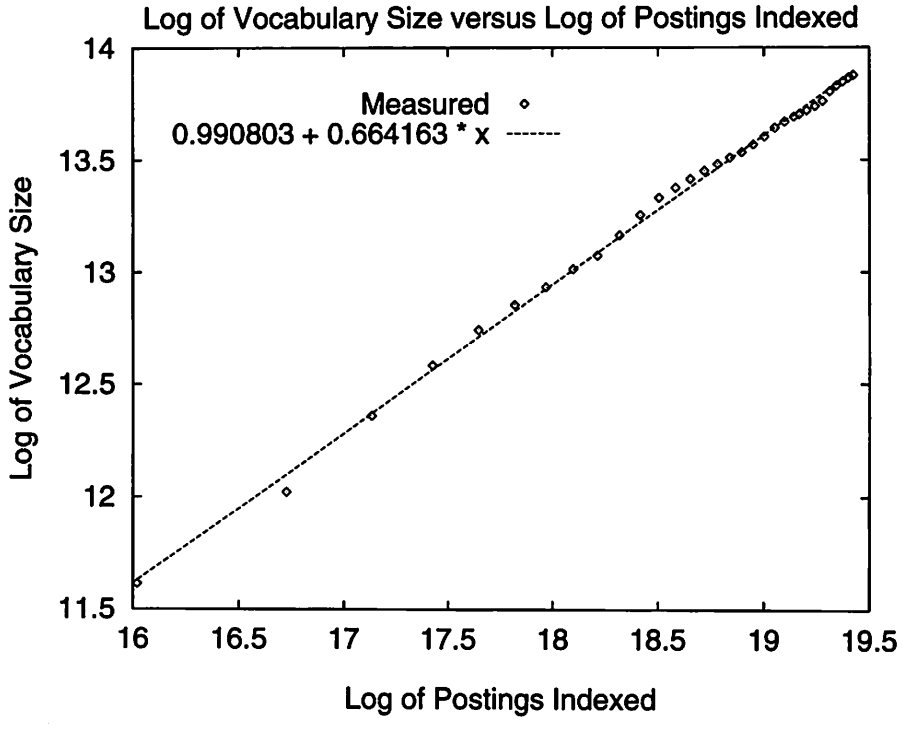


Figure 3.14 Log of TIPSTER vocabulary growth

There are some noticeable systematic variations of the measured data from the function, but they are due to the way in which we assembled the document collections that were used to obtain each of the data points. Since a collection of a given size was created by indexing the files listed in Table 3.1 from top to bottom, a relatively homogeneous set of documents (i.e., documents that come from the same file or kind of files) represents the difference between two data points plotted in Figure 3.13. This set of documents will have slightly different vocabulary growth characteristics than the rest of the collection, causing the occasional systematic variations. If the different document collections were assembled by randomly drawing documents from all of the different files, this variation would disappear.

Since vocabulary growth is sub-linear in terms of the size of the document collection, the amount of existing inverted file data that must be read during an update grows sub-linearly with the size of the existing document collection. Therefore, incremental update merge costs (and total update costs) grow sub-linearly with the size of the existing document collection. Bounding this cost with vocabulary size is very conservative, however, and we are working on a better model for estimating the amount of data read during an incremental update.

An inverted file produced by a series of incremental updates will have the exact same object utilization as if the inverted file had been built in a single bulk indexing operation. The inverted file produced by the above incremental procedure, therefore, has the same object characteristics as the inverted file described in Table 3.3. The only possible difference is the addition of vacant objects created by inverted list relocations in the incrementally produced version. Because we add new objects last in a batch update, however, vacant objects have high likelihood of being reused immediately, and there is no noticeable impact on the size of the incrementally produced inverted file. In fact, the inverted file produced incrementally above occupies 906 MB of disk space, or 7 MB *less* than the inverted file produced in a single bulk indexing operation. This reduction is somewhat misleading—it is caused by an

inefficiency in Mneme's low-level file allocation mechanism. Recall that 16 byte objects are allocated in 4 KB physical segments, while all other objects are allocated in 8 KB physical segments. Mneme aligns the 8 KB physical segments on 8 KB file boundaries, such that a 4 KB hole will be created when an 8 KB physical segment is created immediately after a 4 KB physical segment that was aligned on an 8 KB boundary. These holes can be allocated to 4 KB physical segments in the future, but it may take a while before the file free space search algorithm (circular next fit) finds them. When the file is built incrementally, 4 KB physical segments are created in bursts, reducing the chances for holes to be created.

One final issue that we must address in evaluating the ability of our Inverted File Manager to support document additions is the impact of incremental updates on query processing speed. The danger with our implementation is that the objects that make up a long inverted list will be allocated far apart from each other during the different batch updates. In contrast, a long inverted list created in a single bulk indexing procedure will have all of its objects allocated contiguously in the inverted file. Accessing an incrementally built long inverted list during query processing will potentially require additional disk seeks, increasing the time required to process queries.

To determine if this is a factor, we measured the time to process **Query Set 1** (see Section 4.4.3) using both the bulk indexed inverted file and the incrementally built inverted file. For each version, we ran the query set 6 times and measured the elapsed time for each run. In both versions, the range between the best and worst times recorded for the 6 runs was less than 1% of the average for the 6 runs (i.e., there was no noticeable variation across runs). The average elapsed time for the bulk indexed inverted file was 2741 seconds. The average elapsed time for the incrementally indexed inverted file was 2694 seconds. The incrementally indexed version actually *reduces* query processing time by nearly 2%. This pleasant result can be explained by considering the query evaluation strategy employed by INQUERY (discussed in more detail in Chapter 4). Queries are evaluated document-at-a-time, such that all of the inverted lists for the terms in the query

are processed simultaneously. Rather than sequentially process each list one by one, we cycle through all of the lists for each document in document identifier order. The long inverted list objects in the incrementally indexed inverted file will be clustered based on update batch, and a batch corresponds to a range of document identifiers in the document collection. Therefore, access to the inverted file will be localized for each range of document identifiers, actually reducing the amount of disk seeking.

There are a number of alternatives to the technique described here for supporting document additions. One alternative was mentioned briefly at the beginning of this subsection, and is the simple strategy of re-indexing the entire document collection whenever a batch of new documents must be added. This scheme has two serious problems. First, if the document collection is to remain available for query evaluation during the indexing process, there must be sufficient disk space to hold two complete versions of the inverted file plus the temporary files required by the indexing process. For large existing document collections, this may be impractical. Second, the cumulative bulk indexing costs will quickly exceed the cumulative incremental indexing costs, making this alternative much more expensive.

If we can afford to save the temporary file blocks, we can avoid the redundant parsing of the existing document collection required by the previous scheme. The processing costs to add a batch of new documents will now be limited to parsing the new batch and merging the entire collection. Even in this scheme, the cumulative bulk merge costs will eventually exceed the cumulative merge costs of the incremental version. This is demonstrated in Figure 3.15, which shows the cumulative merge costs for each scheme when the document collection is indexed in 100 MB batches. Merge costs in the bulk indexing scheme are proportional to the size of the existing collection and will continue to grow with each new batch update. Overall, this modified bulk indexing scheme is a poor solution. It requires more elapsed time than the incremental solution and wastes significant disk resources. Moreover, document deletions will make the saved temporary file blocks obsolete and useless.

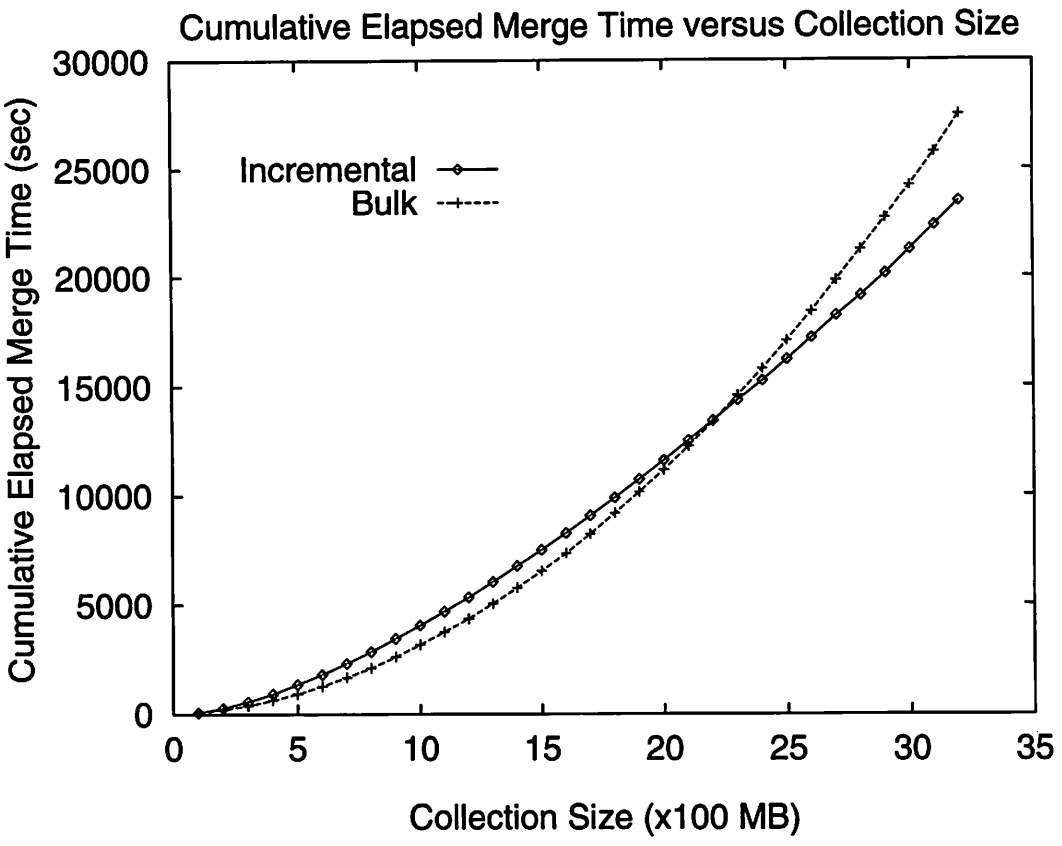


Figure 3.15 Cumulative merge time comparison

One last alternative is to parse just the new document batch and use the existing inverted file as input to the merge process, rather than the temporary file blocks used to build the inverted file. The entire existing inverted file is scanned and a new complete inverted file is written. This scheme is similar to the incremental version. However, the incremental version performs updates in-place and, as we saw in Figure 3.11, must read a relatively small proportion of the existing inverted file. We would expect, therefore, that this last alternative scheme will have higher merge costs due to its complete scan of the existing inverted file. Furthermore, this last scheme will certainly have higher storage costs, requiring enough disk space to hold two complete copies of the inverted file.

3.4 Conclusions

The inverted file index is a critical component in an information retrieval system, determining to a large extent the performance and functionality available from the system. A number of issues must be considered in the management of an inverted file, including efficient inverted file construction, support for inverted file modification, and efficient access to the contents of an inverted file. We have discussed a number of these issues in detail and presented a comprehensive solution to the inverted file management problem.

Our first hypothesis with respect to indexing is that fast, scalable document indexing can be achieved by localizing sort and insertion operations, building intermediate results in main memory, minimizing I/O, and favoring sequential I/O over random I/O. We have presented an inversion scheme that adheres to these principles. Documents are processed using a document based main memory buffer that localizes the inversion of each document. The document buffer is then flushed to a main memory batch buffer, delaying the output of intermediate results to disk as long as possible. Furthermore, the batch buffer stores compressed data, increasing its effective capacity. When the batch buffer must be flushed, it is written to disk in a sequential fashion.

The temporary file blocks written during parsing incur a disk space overhead of approximately 100% of the size of the final inverted file—only 30% to 40% of the size of the raw document collection. The temporary file blocks are efficiently merged using a large main memory merge buffer. The merge process can keep disk seek time down to as little as 16% of the total I/O time required to read the temporary file blocks, and the final inverted lists produced by the merge process can be sequentially written. The overall system was shown to index documents at a rate of 530 MB per hour on a current, midrange workstation, and experiments over a wide range of collection sizes indicate excellent scalability, all of which lead us to accept our first indexing hypothesis.

Without implementing alternative algorithms or measuring other systems on the same platform, it is difficult to compare the indexing system described here with previously

proposed solutions in terms of speed. Possibly the best indexing speeds reported in the literature have been obtained by Witten et al. [90], who achieve an overall indexing rate of 430 MB per CPU hour when indexing the 2 GB TIPSTER collection on a Sun SPARC 10 Model 512 using one processor. In their implementation, Witten et al. do not store term occurrence locations (proximity information) in their inverted files and do not use a stop words list. The particular inversion scheme used to obtain these results performs two passes over the document collection. The first pass gathers statistics for a parameterized compression algorithm, a minimal perfect hash function for the terms, and the final size of each inverted list. Using these statistics, an inverted file skeleton is laid out in main memory marking the start of each inverted list. During the second pass over the document collection, compressed inverted list entries are entered directly into the main memory inverted file at the appropriate locations, avoiding the use of linked lists or sorting. For large document collections, the text is partitioned into chunks. An inverted file for each chunk is is built in main memory as before. A skeleton of the final inverted file is laid out on disk. At the end of each chunk, the main memory inverted file is flushed to disk, filling in the final inverted file skeleton at the appropriate locations.

If we do not store term occurrence locations and do not use stopping, our indexing system requires 5 hours 13 minutes of wall-clock time to index the 3.2 GB TIPSTER collection on a DECSystem 3000/600 (a rate of 614 MB per hour). Of this wall-clock time, 4 hours 54 minutes is CPU time, for a rate of 654 MB per CPU hour. For a very rough comparison to Witten et al.'s system, we can project the indexing speed they might obtain on the machine we used by scaling their reported time based on the difference in machine performance as measured by the SPEC (Standard Performance Evaluation Corporation) benchmark. The SPECint92 numbers for the DECSystem 3000/600 and Sun SPARC 10 Model 512 (one processor) are 114.1 [81] and 65.2 [80], respectively, suggesting that the DECSystem is 1.75 times as fast as the Sun. We speculate, therefore, that Witten et al.'s

scheme would index at a rate of 753 MB per CPU hour on our platform, or 15% faster than our indexing system.

This comparison suggests that our indexing system still has room for improvement. Recall that a profile of the Parser showed that most of the time is spent scanning and parsing. System tuning efforts aimed at scanning and parsing promise to close the gap between the speed of our system and that of Witten et al.'s. Even if a performance gap remains, our system offers other advantages. First, as we have already seen, the modular design of the Parser and Merger allow them to be used "as is" in a system that supports dynamic document collections. Second, if multiple processors or machines are available, the parsing process can be parallelized by partitioning the document collection into sub-collections and parsing all sub-collections simultaneously (followed by a single merge of all temporary file blocks). Third, we can easily extend our indexing system to handle a "real-time" stream of new documents augmented with specific indexing deadlines and availability constraints (this is pursued further in Section 5.1.1).

Our second indexing hypothesis is that document additions can be efficiently supported by an inverted list data structure that minimizes access to the existing inverted file during the update. Support for such an inverted list data structure was obtained by using the Mneme persistent object store as a foundation for our inverted file implementation. The object data model provided by Mneme allowed us to create an inverted file organization that met the functionality requirements specified in the hypothesis. By continuing to adhere to the design principles stated in the first hypothesis, an incremental indexing scheme was designed and implemented that can add new documents to an existing large document collection by accessing less than 30% of the existing inverted file and requiring temporary disk space equal to 30% to 40% of the size of the new document batch. We obtained an overall indexing rate of 265 MB per hour when indexing a 3.2 GB document collection in 100 MB batches. While the incremental indexing costs of this scheme are not entirely independent of the existing document collection, they are significantly better than the

alternative schemes considered, and the trends observed in our experiments indicate good potential for scale. Furthermore, the impact of incremental indexing on query evaluation speed was shown to be negligible. In fact, document-at-a-time style query evaluation can actually benefit from the inverted file locality created by an incremental indexing scheme. These results led us to accept our second hypothesis.

Our last indexing hypothesis is that a general, "off-the-shelf" data management system can be used to manage an inverted file if the system provides the appropriate data model and extensibility mechanisms. We conducted an in-depth exploration of the functionality requirements of an inverted file and concluded that a persistent object store could best satisfy these requirements. A full design and implementation was described, and experimental results related to indexing were presented to validate the feasibility of the implementation. Results pertaining to query evaluation are presented in the following chapter. All of these results combined led us to accept this last hypothesis. While the full potential of this architecture in terms traditional database functionality (e.g., concurrency control, recovery, transactions) is yet to be explored, the work described here lays a strong foundation for the pursuit of a comprehensive information management system.

CHAPTER 4

QUERY EVALUATION

In this chapter we turn to the second main topic addressed in this dissertation—improving execution performance during query evaluation. Query evaluation speed has always been an important factor in the success and acceptance of information retrieval systems. If an information retrieval system is too slow it will be intolerable to use, regardless of its ability to identify relevant documents. Recent trends in the volume and availability of information suggest that system speed will continue to become more important. Commercial document collections already contain tens of gigabytes of data, and projects involving digital libraries forecast document collections containing hundreds of gigabytes of data. Recall the conflicting system goals depicted in Figure 1.1. As document collections become larger, document retrieval inevitably becomes more expensive. Moreover, more sophisticated retrieval techniques are necessary to identify relevant documents. Unfortunately, more sophisticated retrieval typically implies more expensive retrieval, compounding the problem of providing answers quickly and efficiently.

Resolution of the conflict between these competing system goals can be found through the use of query optimization techniques. A query optimization can be targeted at reducing computation, I/O, or both, and is generally intended to result in an overall reduction in running time. A number of query optimization techniques have been proposed for the Boolean, vector-space, and probabilistic retrieval models. Optimizations for the Boolean retrieval model focus on identifying an evaluation order that will constrain the result set as quickly as possible. This is accomplished by considering the collection frequency of each term and distributing any conjunctive operators such that potential result set sizes are

minimized. Optimizations for the vector-space and probabilistic retrieval models generally focus on identifying term weights that can be eliminated from the final document score calculation, saving computation and possibly the I/O that would otherwise be required to retrieve the term weights. Selection of term weights for elimination is often done in such a way that guarantees can be made about the quality of the final document ranking.

While a number of query optimization results have been published for the three retrieval models just mentioned, comparatively little has been published on optimizations for a fourth class of retrieval models, namely statistical ranking retrieval models that support structured queries. These retrieval models are characterized by a statistical or probabilistic term weighting function and a query language that provides a variety of query operators for combining term weights, proximity information, and the results of nested operators. Although many of the optimization techniques proposed for other retrieval models are applicable to retrieval models in this fourth class, the extent to which they can be applied and the effectiveness of their application has not been thoroughly evaluated. Moreover, few optimization techniques have been suggested specifically for statistical ranking retrieval models that support structured queries. We begin to address this situation with the work presented here. We consider a number of issues related to reducing the cost of evaluating structured queries in a ranking retrieval model and present a new optimization technique that yields a dramatic reduction in evaluation time with no noticeable impact on retrieval effectiveness.

Our exploration of structured query optimization techniques uses INQUERY as an experimental framework. INQUERY supports a rich, structured query language and has been shown to produce good levels of retrieval effectiveness [39, 40]. Moreover, the inference network-based retrieval model provides a general framework for representing a variety of retrieval strategies, suggesting that the results reported within this experimental framework will have applicability beyond that of just the inference network-based model.

We begin with an overview of structured queries, including background information on the probabilistic retrieval model, its generalization in the inference network-based retrieval model, and INQUERY's implementation of this model. We then consider the issues involved in optimizing structured queries and present our new optimization technique. This is followed by implementation details and a performance evaluation of the technique. Extensions to the basic optimization technique are considered, and its effectiveness on short, unstructured queries is explored. Finally, the chapter ends with conclusions.

4.1 Structured Queries

A retrieval model supports structured queries if its query language provides a variety of operators that can be nested to create a query tree[1]. This definition includes the Boolean retrieval model, but excludes the vector-space model, which supports flat queries only. We further restrict the retrieval models of interest by requiring support for statistical ranking. This last restriction eliminates the simple Boolean retrieval model from consideration.

The best example of a statistical ranking model that supports structured queries is the inference network-based retrieval model as implemented by INQUERY. The inference network-based retrieval model is rooted in the probabilistic retrieval model.

4.1.1 Probabilistic Retrieval

Maron and Kuhns [56] first suggested the probabilistic retrieval model in 1960. The basic idea is to rank the documents in a collection based on their probability of being relevant to the current information need. This is expressed as $P(relevant \mid d)$, or the probability that the information need is met given document d. A user's information need is something internal to the user and cannot be expressed exactly to the system, so this probability must

[1] In fact, the query can form a DAG, although a tree structure can be obtained by duplicating nodes or subtrees as necessary.

be estimated using the terms supplied by the user in a query. The estimation is simplified using a version of Bayes' theorem to rewrite the probability as

$$P(relevant \mid d) = \frac{P(d \mid relevant)P(relevant)}{P(d)}$$

Document d can be represented as a binary vector $\mathbf{x} = (x_1, x_2, \ldots, x_v)$, where $x_i = 1$ if term i appears in document d, $x_i = 0$ otherwise, and the terms are (typically) limited to those that appear in the query. Now the estimation task amounts to estimating the probability of the terms appearing in a relevant document, $P(\mathbf{x} \mid relevant)$, and the *a priori* probability of a document, $P(\mathbf{x})$. $P(relevant)$ will be constant for a given query and so may be ignored.

Robertson and Sparck Jones [71] revised the probabilistic model into its current form. They observed that a document should be retrieved if its probability of being relevant is greater than its probability of being not relevant, $P(relevant \mid d) > P(not\ relevant \mid d)$. For the purposes of ranking the documents in a collection, this can be restated as a cost function

$$g(\mathbf{x}) = \log \frac{P(\mathbf{x} \mid relevant)}{P(\mathbf{x} \mid not\ relevant)} + \log \frac{P(relevant)}{P(not\ relevant)}$$

where document d is expressed as the binary vector $\mathbf{x}$, Bayes' theorem has been used, and the logs have been introduced to linearize the function.

If we assume that terms appear independently in the relevant documents, we can rewrite $P(\mathbf{x} \mid relevant)$ as $P(x_1 \mid relevant)P(x_2 \mid relevant) \cdots P(x_v \mid relevant)$, and similarly for the not relevant case. Let $p_i = P(x_i = 1 \mid relevant)$ and $q_i = P(x_i = 1 \mid not\ relevant)$, then

$$P(\mathbf{x} \mid relevant) = \prod_i p_i^{x_i}(1 - p_i)^{(1-x_i)}$$

and

$$P(\mathbf{x} \mid not\ relevant) = \prod_i q_i^{x_i}(1 - q_i)^{(1-x_i)}$$

Our cost function can now be rewritten as

$$g(\mathbf{x}) = \left(\sum_i x_i \log \frac{p_i(1 - q_i)}{(1 - p_i)q_i}\right) + \left(\sum_i \log \frac{1 - p_i}{1 - q_i}\right) + \frac{P(relevant)}{P(not\ relevant)}$$

The last two terms will be constant for a given query (since x_i does not appear in them), so we are left with the first term as our ranking function. This is known as the *binary independence* model.

We are still faced with the problem of estimating p_i and q_i. The solution is to use some other technique to return an initial set of documents to the user and obtain feedback about the relevant and non-relevant documents in the set. The distribution of query terms in the relevant and non-relevant documents in this sample is then used to estimate p_i and q_i, and the query is re-evaluated probabilistically. Croft and Harper [22] showed how the probabilistic model could also be used for the initial search. They assume that p_i is the same for all terms and q_i can be estimated with n_i/N, where n_i is the number of documents in which term i occurs and N is the number of documents in the collection. The ranking function now becomes

$$g(\mathbf{x}) = C\sum_i x_i + \sum_i x_i \log \frac{N - n_i}{n_i} \tag{4.1}$$

This is referred to as the *combination match*, which applies the constant factor C times the number of matches between the terms in the query and the terms in the document, plus what is essentially the inverse document frequency of each query term that appears in the document.

Equation 4.1 assumes that a term is either fully assigned to a document, or not at all. The mere appearance of a term in a document, however, does not necessarily mean that the term is indicative of the contents of the document. Rather than make such extreme judgments, we would prefer to use a finer granularity when expressing the degree to which a term should be assigned to a document. This was accomplished by Croft [19, 20] who expressed this degree as the probability of a term being assigned to a document, $P(x_i = 1 \mid d)$, such that documents should now be ranked by the *expected* value of Equation 4.1, or

$$g(\mathbf{x}) = \sum_i \left[P(x_i = 1 \mid d)\left(C + \log \frac{N - n_i}{n_i}\right)\right]$$

$P(x_i = 1 \mid d)$ is then estimated using the normalized within document frequency of the term, $ntf_{id} = tf_{id}/max\_tf_d$, where tf_{id} is the number of occurrences of term i in document d,

and max_tf_d is the maximum of $\{tf_{1d}, tf_{2d}, \ldots\}$. To increase the significance of even a single occurrence of a term in a document, a constant K in the range 0 to 1 is applied to yield the final probabilistic ranking function

$$g(\mathbf{x}) = \sum_i \left[(K + (1 - K) ntf_{id}) \left(C + \log \frac{N - n_i}{n_i} \right) \right] \tag{4.2}$$

4.1.2 Inference Network-based Retrieval

The Bayesian inference network model generalizes the probabilistic retrieval model by treating retrieval as an evidential reasoning process where documents are used as evidence to estimate the probability that a user's information need is met. An inference network consists of nodes and directed edges between the nodes forming a directed acyclic graph (DAG). The nodes represent binary valued (i.e., true or false) propositional variables or constants and the edges represent dependencies between the nodes. If the proposition represented by a given node p implies the proposition represented by node q, then a directed edge is drawn from p to q. Node q will also contain a *link matrix* that specifies the probability of q given p, $P(q \mid p)$, for all possible values of p and q. Since p and q may each be either true or false, this link matrix will contain four entries. If q has multiple parents (π_q), the link matrix will specify the conditional probability of q on the set of parents, $P(q \mid \pi_q)$. Typically the network is large such that storing the entire link matrix for a node is impractical. Instead, the link matrix is represented in a canonical form and we store only the information required to compute each matrix entry from the canonical form.

If the probabilities of the root nodes in the network are known, Bayesian inference rules can be used to condition these probabilities over the rest of the network and compute a probability, or belief, for each of the remaining nodes in the network. Moreover, if our belief in any given proposition should change, its probability can be adjusted and the network can be used to update the probabilities at the rest of the nodes.

The application of Baysien inference networks to information retrieval was advanced by Turtle and Croft [86, 88, 87]. The inference network used for information retrieval is

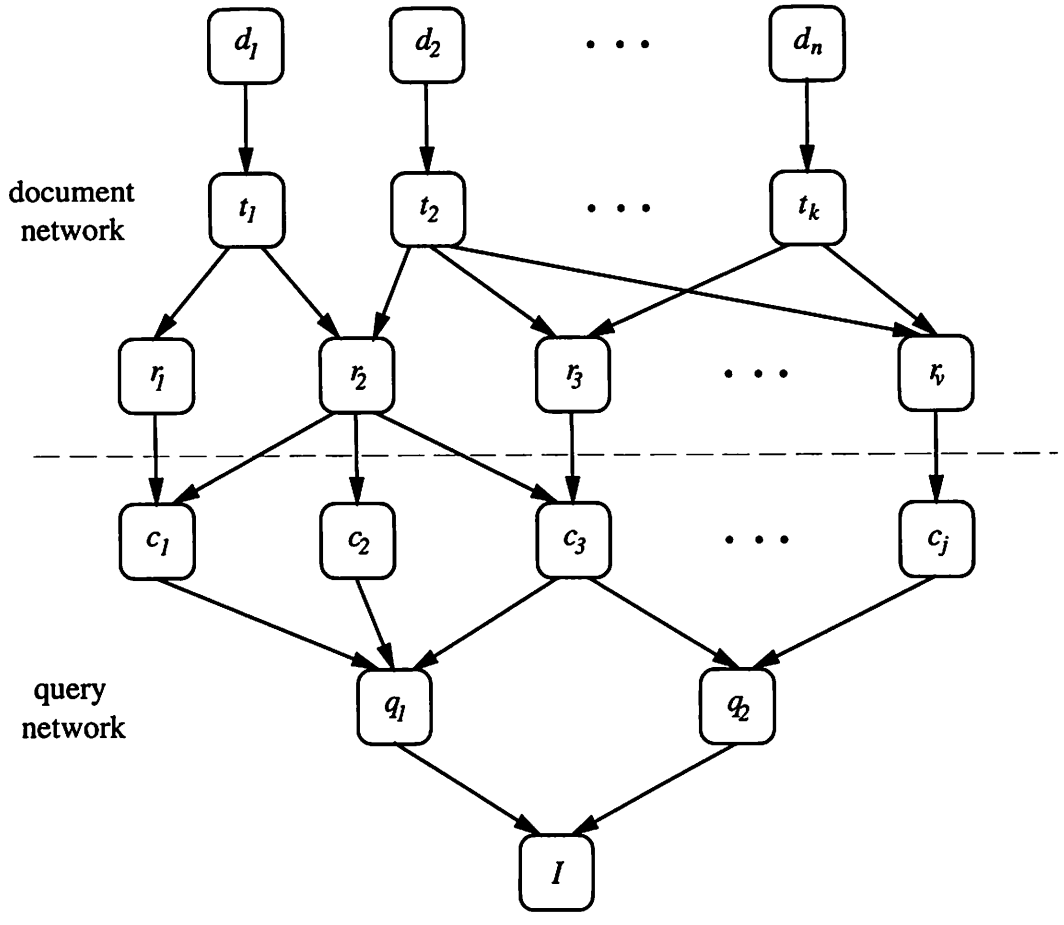


Figure 4.1 Inference network for information retrieval

divided into two parts, a document network and a query network, shown in Figure 4.1. The document network consists of document nodes (d_i's), text representation nodes (t_i's), and concept representation nodes (r_i's). A document node represents the event that a document has been observed at an abstract level, while a text node represents the event that the actual physical content of a document has been observed. This distinction is made to support complex documents which may have multiple physical representations (e.g., multimedia documents with text and video), and sharing of the same physical text by multiple documents (e.g., if two documents are merely different published forms of the same text). In the first case, a document node will have multiple children text nodes, while in the second case, a text node will have multiple parent document nodes. Typically, each document has only

one text representation and the text representations are not shared by multiple documents, such that the document network may be simplified by eliminating the text nodes.

A concept representation node represents the event that a document concept has been observed. Document concepts are the basic concepts identified in the document collection. Commonly these are the terms in the document collection, but they may also be more semantically meaningful concepts extracted from the text by sophisticated indexing methods. The conditional probability $P(r_i \mid d_j)$ stored in a concept representation node quantifies our estimate of the degree to which the concept should be assigned to the document, as well as the ability of the concept to describe the information content of the document. This estimate can be borrowed from the probabilistic retrieval model, using Equation 4.2 as the foundation of the estimate.

The query network consists of query concept nodes (c_i's), query nodes (q_i's), and a single information need node (I). Node I represents the event that a user's information need has been met. Query nodes are a representational convenience that allow the information need to be expressed in multiple query forms. They represent the events that particular query forms have been satisfied, and could be eliminated by using more complicated conditional probabilities at node I. Query concepts are the basic concepts used to represent the information need. A query concept node describes the mapping between the concepts used in the document representation and the concepts used in the query representation, and will have one or more document concept representation nodes for parents. In the common case, each query concept node will have a single parent.

The document network is constructed once at indexing time. The links between the nodes and the link matrices stored within the nodes never change. The query network is constructed when the query is parsed. The link matrix stored in a query node will be based on the query operator represented by the node. Such operators might include the boolean operators, simple sums, or weighted sums where certain query concepts have been identified as being more significant and consequently given more weight. The link matrix

in the information need node will describe how to combine the results from the different query representations. Unlike the document network, the conditional probabilities in the query network may be updated given additional information from the user, as might occur during relevance feedback.

The inference network is used by attaching the roots of the query network to the leaves of the document network. To produce a score for document d_j, we assert d_j = *true* and d_k = *false* for all $k \neq j$, and condition the probabilities through the network to obtain $P(I \mid d_j)$. If a document provides no support for a concept (i.e., it doesn't contain that term), a default belief is assigned to that concept node when conditioning over the network. A score is computed in this way for all documents in the collection, which are then ranked based on their scores. In practice, we need only compute scores for documents which contain at least one of the query concepts. As the query is evaluated, a default document score is computed which is then assigned to all documents that contain none of the query terms.

4.1.3 INQUERY

In INQUERY, a user's information need is satisfied by expressing that need as a query and evaluating the query against a collection of documents. Evaluating the query for a given document produces an estimate of the probability of that document satisfying the information need, expressed as a *final belief score*. After all of the documents in the collection have been evaluated, they are ranked based on their final belief scores. A ranked document list is then returned to the user.

A query consists of *indexed concepts*, *belief operators*, and *proximity operators*. These elements are combined in a tree structure with indexed concepts at the leaves and operators at the internal nodes. An example query is shown in Figure 4.2, where operators are prefixed with a hash mark (#). An indexed concept is a term or other special object identified at indexing time. A proximity operator produces *constructed concepts* by combining indexed

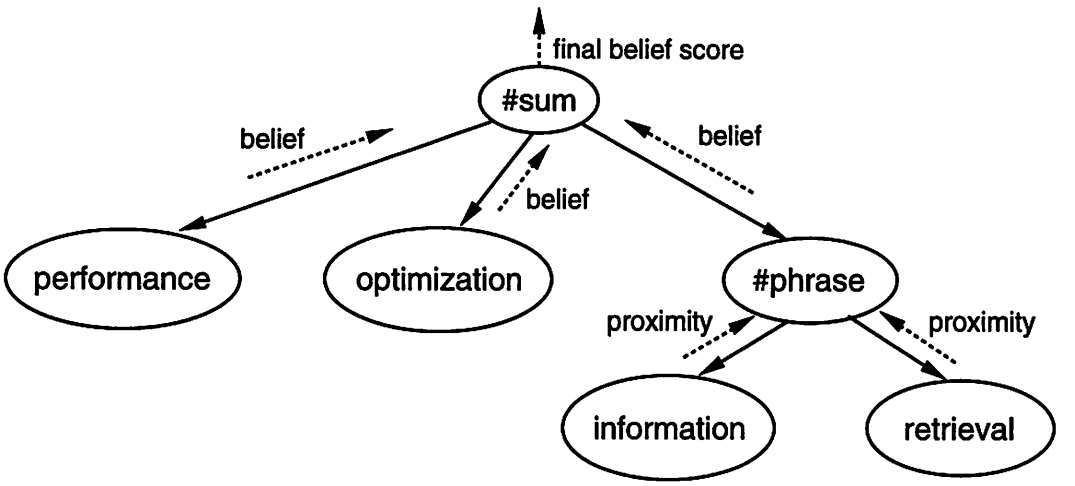


Figure 4.2 Example query in internal tree form

concepts and other constructed concepts at query processing time.[2] Concepts contribute *belief values* for every document in which they appear. Belief operators describe how to combine these belief values to produce the final belief score.

Belief operators operate on belief values and return belief values. The belief operators include **and, or, not, sum, weighted sum,** and **maximum**. The first three are probabilistic implementations of the traditional boolean operators. The next two return the average and weighted average, respectively, of their children's belief values. The last operator returns the maximum of the belief values from its children.

Proximity operators operate on *proximity lists* and return either a new proximity list or a belief value. A proximity list contains the locations where its associated concept occurs in a given document. For example, in Figure 4.2 the proximity list for the term "information" in document *j* would contain the locations of each occurrence of "information" in document *j*. When the #phrase operator combines that proximity list with the proximity list for the term "retrieval" in document *j*, a new proximity list for the phrase "information retrieval" is constructed that contains the locations where "information retrieval" appears in document

[2]This definition of concept is a slight departure from the formal definition in the inference network [88]. The distinction between indexed and constructed concepts is emphasized here to facilitate discussion from an implementation perspective.

j. This may be returned to a parent proximity operator, or a belief value may be computed from the proximity list and returned to a parent belief operator.

The proximity operators include **phrase**, **ordered distance n**, **unordered window n**, **synonym**, and **passage sum**. The **ordered distance n** operator identifies documents that contain all of the operator's child concepts $\{c_1 \ldots c_k\}$ with the constraint that the concepts must appear in order and be spaced such that the distance between c_i and c_{i+1} is less than or equal to n. The **unordered window n** operator is similar except that all of the child concepts must appear within a window of size n and they may appear in any order. The **phrase** operator is initially evaluated as an **ordered distance n** with $n = 3$. However, depending on the quality of the resultant phrase, the operator may ultimately be evaluated as an **ordered distance n** with $n = 3$, a **sum**, or a **maximum** of these two.

The **synonym** function combines two or more proximity lists into a single proximity list by taking the union of the locations for each document in the lists. The new proximity list represents a constructed concept that occurs anywhere any of the child concepts occur.

The last function, **passage sum**, calculates a belief for a document as follows. First, the document is divided into fixed size overlapping passages, where the last half of each passage overlaps the first half of the subsequent passage. Next, a belief score for each passage is calculated based on the number of occurrences of each of the child concepts within the passage and any weights associated with the child concepts. Finally, the maximum passage belief is returned as the belief for the document. Proximity lists are required from the children to determine concept occurrences within each passage, and a belief list is returned from the passage operator itself.

The belief value contributed by a concept for a given document is calculated using a probabilistic version of the $tf \cdot idf$ score. The tf weight is directly proportional to the within document frequency of the concept, such that the more times the concept appears in the document, the greater the belief value. The idf weight is inversely proportional to the concept's document count (the number documents in which the concept appears), such that

the greater the document count, the smaller the belief value. Specifically, the belief value for concept i in document j is calculated with the following formula:

$$belief_{ij} = C + (1 - C)\, ntf_{ij}\, nidf_i \tag{4.3}$$

where

$$ntf_{ij} = KH + (1 - K)\left(\frac{\log(tf_{ij} + 0.5)}{\log(max\_tf_j + 1.0)}\right)$$

$$nidf_i = \frac{\log((N + 0.5)/n_i)}{\log(N + 1.0)}$$

ntf_{ij}	is the normalized within document frequency
$nidf_i$	is the normalized inverse document frequency
tf_{ij}	is the within document frequency
max_tf_j	is the maximum of $\{tf_{1j}, tf_{2j}, \ldots\}$
N	is the # documents in the collection
n_i	is the # documents in which concept i appears

The constants C and K both default to 0.4 in INQUERY, although they may be specified by the user. C is the default belief value returned for documents that do not contain the given concept. K acts to increase the significance of even a single occurrence of a concept in a document. H is used to reduce the influence of document length for long documents. If max_tf_j is greater than 200, then H is set to $200/max\_tf_j$. Otherwise, H is set to 1.0. Additionally, if tf_{ij} is equal to max_tf_j, then ntf_{ij} is set to 1.0. Note that a belief value will always be between 0 and 1.0 inclusive.

The document counts, within document frequencies, and proximity lists for indexed concepts are extracted and stored in an inverted file when the document collection is indexed (see Chapter 3). An inverted file consists of a record, or inverted list, for every indexed concept that appears in the document collection. A concept's inverted list contains its document count and an entry for every document in which that concept appears, identifying the document and giving the within document frequency and proximity list of the concept within the document.

To facilitate locating information about a particular document in an inverted list, the document entries are stored in document id order. This naturally leads to the following

query processing strategy. First, each node in the query tree is initialized with the next document id (NID) to be processed at that node. For indexed concept (leaf) nodes, this is simply the id of the first document that appears in the inverted list for that concept. Operator (internal) nodes are classified as either union or intersection style operators. Union style operators calculate a result for the current document if *at least one* of its children contributes a result for that document (e.g., **weighted sum**). Intersection style operators calculate a result for the current document only if *all* of its children contribute a result for that document (e.g., **ordered distance n**). A union style operator is initialized with the *minimum* of its children's NIDs, while an intersection style operator is initialized with the *maximum* of its children's NIDs.

Processing is performed document-at-a-time with the current document to process determined by the NID at the query tree root. The query tree is evaluated in a depth-first fashion for the current document. When a node representing a concept is encountered, a belief value for the current document is computed using Equation 4.3. The belief values flow from the leaves to the root, being combined according to the belief operators along the way. In addition, as each node is evaluated the node's NID is updated appropriately from its children. When the root node returns the final belief score for the current document, it is saved in a list for later ranking. This process repeats until the NID at the root node indicates that all documents have been processed. The list of final belief scores can then be sorted and the ranked listing returned. Note that the only documents evaluated are those that appear in the inverted lists for the indexed concepts in the query. All other documents receive a default final belief score.

It turns out that an extra query processing step is required. In order to calculate a belief value for a constructed concept (e.g., a phrase), we need the concept's *idf* weight. The *idf* weight depends on the number of documents in which the concept occurs. This is unknown until the constructed concept has been evaluated for all of the documents. Therefore, a preprocessing step is needed to fully evaluate the constructed concepts and determine their

idf weights. The results of this preprocessing step are saved in temporary inverted lists, allowing proximity lists and belief values to be obtained immediately from constructed concepts during the final query evaluation phase.

4.2 Structured Query Optimization

Optimization techniques for information retrieval systems that support statistical ranking may be classified as either *safe* or *unsafe*. Safe techniques have no impact on retrieval effectiveness, while unsafe techniques may trade retrieval effectiveness for execution speed. We consider a number of safe optimizations and introduce a new unsafe optimization below.

4.2.1 Safe

The first safe technique is intended to improve execution performance by eliminating unnecessary I/O. In the traditional inverted list organization, an inverted list document entry stores its term weight and proximity list together. We saw in the last section, however, that proximity lists are not required when processing a belief operator. The traditional inverted list organization results in unnecessary I/O when processing a belief operator. To remedy this situation, we can use an inverted list organization that separates term weights from proximity lists and allows selective access to one or the other. If belief operators no longer need to read proximity lists from disk, they will be less expensive to process and execution performance will improve.

The next safe technique can generally be called an intersection optimization, and is borrowed from the Boolean retrieval model. In that model, a query consisting of a conjunction of terms can be evaluated in the following fashion. First, a candidate document set is created consisting of the set of documents in which one of the terms appears. Then, for each of the remaining terms, the set of documents in which that term appears is intersected with the set of candidate documents. After all terms have been processed, the candidate document set will consist of the documents which satisfy the conjunction. This process can

be improved by starting with the term that appears in the smallest number of documents and processing the remaining terms in increasing order of document frequency. At each intersection, it is only necessary to check if the current term appears in the documents in the candidate set, since the candidate set can only shrink or stay the same. Therefore, savings can be realized if we can access just the portions of the inverted list for the current term that might contain an entry for a candidate document. Furthermore, if the candidate set should become empty, processing can stop immediately.

Unfortunately, the conjunction operation in the probabilistic retrieval model is not a strict intersection, so this optimization is not applicable to the **and** operator. However, the proximity operations described above *are* strict intersections in the sense that every term in a proximity must appear in a document (and satisfy any ordering and window constraints) in order for the document to satisfy the proximity. Therefore, the exact same technique can be used to improve execution performance for proximity operations. This technique requires the ability to access just that portion of an inverted list that might contain an entry for a given document, i.e., selective access to the contents of an inverted list.

The final safe technique for improving execution performance is inverted list compression. Assume that we have u bytes of data that can be compressed down to z bytes, $z < u$. If the cost of decompressing z bytes of data is less than the cost of reading $u - z$ bytes from disk, then execution performance will improve. Note also that if the cost of decompressing z bytes exceeds the cost of reading the $u - z$ extra bytes in an uncompressed inverted list, then execution performance will deteriorate.

Compression techniques for inverted lists have received a fair amount of attention in the literature [90, 57, 51, 2, 6, 95]. We do not claim anything novel with respect to compression. Rather, for completeness we merely describe how it fits into an overall optimization strategy and give a necessary condition for providing benefit with respect to execution performance. Note that compression clearly has other desirable side effects, e.g. reduced disk space requirements, whose benefits may outweigh any additional execution costs.

4.2.2 Unsafe

While the previous techniques will always guarantee a correct answer to a query, they generally depend on the particular operators used in the query. We now introduce a more general technique that attacks the evaluation costs inherent in any structured query [7]. There are two factors that determine the cost of query evaluation. First, there is the complexity of the query. The discussion in Section 4.1.3 suggests that queries may be quite complex. The more complex the query, the more processing required for each document in order to evaluate the document's final belief score. The second factor is the size of the set of documents that must be evaluated, or the *candidate document set*. This set may be quite large. Moffat and Zobel [58] found that for queries containing around 40 terms, using the terms' inverted lists to populate the candidate document set caused nearly 75% of the documents in the collection to be placed in the candidate document set. This is consistent with our results reported below, where our unoptimized candidate document set typically contained over half of the documents in the collection.

Given the relatively small number of top documents a user might actually review in an interactive system, such a large candidate document set seems exorbitant. If our document collection contains one million documents, the system may have to evaluate over five hundred thousand documents, while the user will rarely consider more than the top one thousand documents. Therefore, the goal of our optimization technique is to constrain the set of candidate documents. If we can reduce the size of the candidate document set, we will reduce the number of per document evaluations of the query tree, reducing overall query processing time. Moreover, if we are no longer processing every document that appears in the inverted lists, we may be able to skip portions of inverted lists [60]. If the skipped portions are large enough and our inverted list implementation provides the necessary functionality, the overall number of disk I/Os might be reduced.

To constrain the set of candidate documents, we want to add just those documents that have a strong chance of satisfying the user's information need. Without actually evaluating

the query, the best we can do to estimate this chance for a given document is to consider the belief contributions from the indexed concepts in the query. Recall that the belief value for concept *i* in document *j* is a product of the *idf* weight for concept *i* and the *tf* weight for concept *i* in document *j*. This leads to the following two observations and corresponding rules:

1. Due to their large *idf* weights, rarely occurring concepts are likely to make large contributions to a document's final belief score. Therefore, they will identify highly ranked candidate documents. For a concept whose *idf* weight exceeds some threshold, add to the candidate document set all documents that contain the concept (i.e., all documents that appear in the concept's inverted list).

2. More frequently occurring concepts may still contribute significant belief values for the documents in which they appear frequently (i.e., have a large *tf* weight). For a concept that does not exceed the *idf* weight threshold, add to the candidate document set the documents associated with the concept's top *n tf* weights.

An indexed concept's *idf* weight is inversely proportional to the length of its inverted list. Rather than establish an *idf* weight threshold for candidate set population, we use an inverted list length threshold. An inverted list is *short* if it can be obtained in a single disk read, otherwise it is *long*. From our first rule, all of the documents that appear in a short list will be used to populate the candidate document set. The cost associated with this activity is a single disk read per short inverted list. Since one disk read is required anyway to access an inverted list for later processing, populating the candidate document set with a short list will incur no extra I/O costs.

From our second rule, we need to obtain the documents associated with the top *n tf* weights in the long inverted lists. This suggests that the inverted lists should be sorted by *tf* weight. However, query evaluation is document driven and requires that the inverted lists be sorted by document identifier. Instead, if *n* is defined to be relatively small, we can

maintain a separate list of the documents associated with the top *n tf* weights for each long inverted list. Zipf's Law [94] suggests that there will be relatively few long inverted lists, but they will consume the majority of the space in the inverted file. If each top document list is constrained to be smaller than a disk page, then the overhead associated with the top document lists will be a small percentage of the total space occupied by the long inverted lists. Furthermore, obtaining the top document list for a long inverted list will require a single disk read.

Using our two rules, the candidate document set is created in a final preprocessing pass over the query tree, after the constructed concepts have been built. When an indexed concept with a short list is encountered, all of the documents in that list are added to the candidate set. When an indexed concept with a long list is encountered, the documents with the top *n tf* weights from that list are added to the candidate set. When a constructed concept built by a proximity operator is encountered (e.g., a phrase), it could be handled in the same way as an indexed concept. However, for simplicity in the current implementation, constructed concepts are treated like short lists and all of the documents in a constructed concept's inverted list are added to the candidate set.

One special case is the **not** operator. In this case, we ignore the subtree below the **not** altogether. The **not** operator returns $1 - belief_c$, where $belief_c$ is the belief value returned by c, the child of the **not** operator (i.e., the negated concept). $belief_c$ will be greater than or equal to the default belief value at c, such that the largest possible belief value returned by the **not** operator will be for documents that *do not* contain the negated concept. In other words, documents identified by inverted lists in the subtree below the **not** can only have their final belief scores reduced by the **not**. Therefore, it is sufficient to ignore the **not** when establishing the candidate set and simply evaluate the **not** on the candidate set established from the rest of the query tree.

The final candidate document set is used to drive the document evaluation process. Rather than choose the current document to evaluate based on the NID at the root of the

query tree, we simply evaluate each of the documents in the candidate set. Otherwise, query evaluation proceeds as described in Section 4.1.3. Each document in the candidate set is fully evaluated and receives an accurate final belief score. The final relative ranking of the documents in the candidate set will be the same as if no optimization had been used. The only difference will be that documents that were not added to the candidate set will receive the default document score and may appear lower in the final ranking than they would have had they been evaluated.

4.3 Implementation

The optimization techniques described above place certain functionality requirements on the inverted file implementation. The safe optimizations require separation and isolated access of proximity and belief information and the ability to skip portions of an inverted list when reading the list from disk. The new unsafe optimization requires storage of the top document lists for the long inverted lists and the ability to distinguish between the different types of lists and handle them accordingly at indexing time, query processing time, and collection modification time.

Fortunately, we can easily extend the Mneme-based inverted file implementation described in Chapter 3. Recall that short lists are defined to be 8 KB or less. Since a single file read will obtain an entire short inverted list, it is not profitable to support disk access of short lists in granularities smaller than the entire inverted list. Short lists, therefore, are stored as before using fixed length objects, ranging in size from 16 bytes to 8 KB by powers of 2 (i.e., 16, 32, 64, ..., 8K). The same small-object, fixed-object, and page-object pools are used to create and manage these objects.

Long inverted lists must satisfy all of the functionality requirements stated above. The simple linked list implementation for long inverted lists described in Section 3.2.3 is inadequate. Instead, long inverted lists are stored as shown in Figure 4.3. A long inverted list is split into two distinct lists: a frequencies list and a locations list. The frequencies

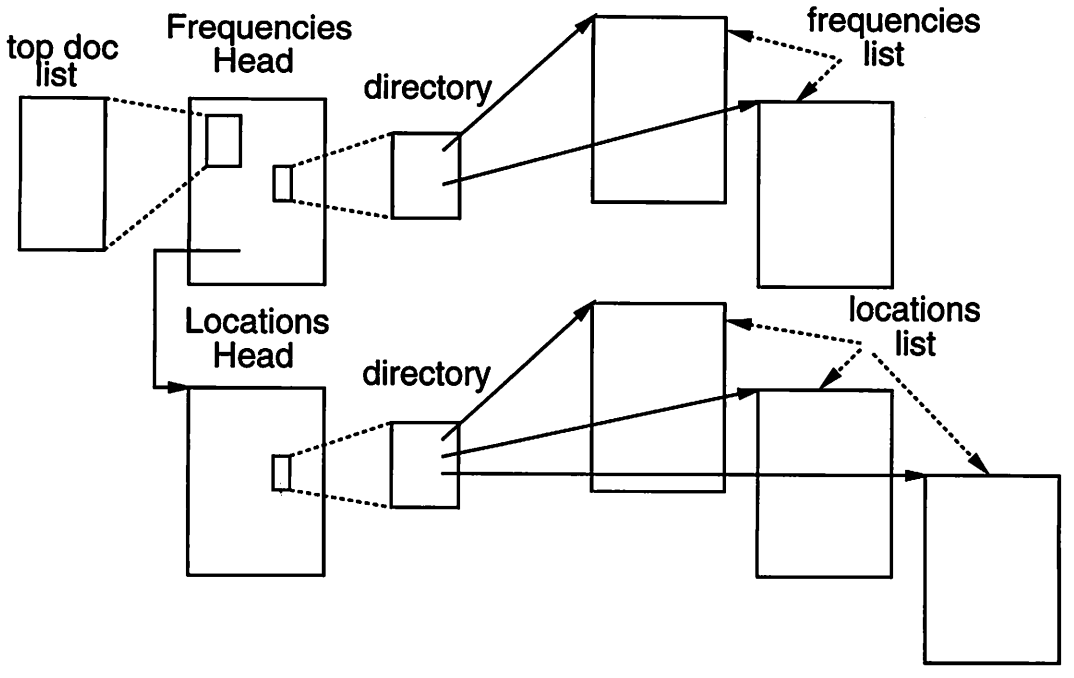


Figure 4.3 Long inverted list structure

list contains the document id and frequency statistics from each of the document records in the original inverted list. The locations list contains the locations (proximity lists) from the document entries. Each of these new lists is stored in 8 KB objects accessed through a directory. A directory entry contains a pointer to an object, along with the document id for the first list entry in the object. To obtain the information for a specific document, the directory is used to identify and directly access the objects that contain the desired information.

The directory for the frequencies list is compressed and stored in a special 8 KB object called the Frequency Head. When the inverted list is first accessed, the Frequency Head is obtained and the directory is decompressed. This is all that is needed to access the frequencies list and satisfy requests for belief values from parent belief operators. If a proximity list is required, the Locations Head must be obtained. The Locations Head is another special 8 KB object that contains the compressed directory for the locations list.

Both the frequencies list and the locations list are accessed simultaneously to return the desired proximity list.

The Head objects will store the tails of their respective lists if there is enough room. In addition, the Frequencies Head contains the top document list stored in a compressed format. For our initial implementation, we set the number of top documents n to 1000. Within inverted list i, documents are ranked based on their tf weights, calculated as the normalized term frequency ntf_{ij} (see Equation 4.3). This produces a floating point number between 0.0 and 1.0. To increase the amount of compression possible on the top document list, each document's normalized term frequency was multiplied by 16383 (i.e., $2^{14} - 1$) to produce an integer guaranteed to fit in two bytes or less using our variable length compression technique. This reduces the precision of our within list ranking function, but yields a significant space savings. The lost precision is seen only at the boundary score for the worst document in the top document list, where we may not be sure that we have the best document mapped to that integer. All documents with larger integer scores are guaranteed to have a larger ntf_{ij}.

Our use of normalized term frequency to rank documents within an inverted list has one drawback. Recall that if term i is the most frequent term in document j (i.e., $tf_{ij} = max\_tf_j$), then ntf_{ij} is set to 1.0. All of the documents in which term i is the most frequent term will have a normalized term frequency of 1.0 for term i. These documents will be arbitrarily ranked relative to each other within the inverted list for term i. If term i is the most frequent term in more than 1000 documents, the top document list for i may not contain i's "best" 1000 documents. In cases such as this, however, term i will have a very low idf weight; i is less likely to identify relevant documents and more likely to act as a fine tuning adjustment on final document scores, reducing the need for accuracy in i's top document list. If the calculation for normalized term frequency were modified to differentiate between documents in which term i is the most frequent term, then we would expect our optimization technique to perform even better.

This inverted file implementation furnishes all of the functionality necessary to support the safe optimizations described above as well as our new unsafe optimization technique. The split long inverted lists allow the selective access of inverted list contents required by the first safe optimization described above. The directory based access into the long inverted lists supports skipping through the lists due to a reduced candidate document set or from application of a safe intersection style optimization in a proximity operator. The long inverted lists provide storage of the top document lists. Finally, the customized Mneme object support described in Section 3.2.3 facilitates the distinction between short and long inverted lists and simplifies appropriate handling of each.

4.4 Performance Evaluation

We now evaluate the effectiveness of the optimization techniques considered above. For safe optimization techniques, it is sufficient to merely measure their impact on execution speed. For unsafe optimizations, we must additionally assess the impact of the optimization technique on the system's retrieval effectiveness. We describe our evaluation below, including the platform on which we ran our experiments, the test collections and query sets used, the performance measured, and the levels of retrieval effectiveness observed.

4.4.1 Platform

All of our experiments were run as superuser with logins disabled on an otherwise idle DECSystem 3000/600 (Alpha AXP CPU clocked at 175 MHz) running OSF/1 V3.0. The system was configured with 64 MB of main memory, one DEC 1.0 GB RZ26L Winchester SCSI disk, and one DEC 2.0 GB RZ28B Winchester SCSI disk. The executables were compiled with the DEC C compiler driver 3.11 using optimization level 2. All of the data files and executables were stored on the larger local disk, and a 64 MB "chill file" was read before each query processing run to purge the operating system file buffers and guarantee that no inverted file data was cached by the file system across runs (see Section 3.3.1 for

Table 4.1 Test collection statistics

Collection	Size (MB)	Docs	Terms	Postings
Tip1	1206	510343	639914	112812693
Tip12	2069	741562	859121	191742705
Tip123	3181	1077872	1090896	281417622

verification of the chill procedure). In all cases we allocated 15 MB of Mneme buffer space to cache memory resident inverted list objects.

4.4.2 Test Collections

For our experiments we used three test collections drawn from the three volume *TIPSTER* document collection used in the *TREC* [39] evaluations. This is the same test collection described in Chapter 3, although here it is divided into three separate volumes. Statistics for the test collections can be found in Table 4.1, where *Terms* is the number of unique indexed concepts and *Postings* is the total number of occurrences of the indexed concepts. **Tip1** is volume 1, **Tip12** is volumes 1 and 2, and **Tip123** is all three volumes.

The test collections were indexed automatically, using stemming to reduce words to common roots and a stop words list to eliminate words too frequent to be worth indexing. Feature recognizers were also used to identify city names, company names, foreign country names (i.e., not the United States), and references to the United States.[3] Statistics for the inverted files generated during the indexing process can be found in Table 4.2. For each file the table gives the size of the inverted list data after compression, the overheads in the file, and the total file size. *Top Docs* is the space required for the top document tables, *Free Space* is unused space at the end of an object that could be allocated in the future, and *Other* is data structure and Mneme overhead. Most of the free space appears in the Head objects of long inverted lists, indicating that a better implementation could be more space efficient.

[3]Note that feature recognizers were not used during the indexing experiments in Chapter 3, which explains why the posting and term counts reported there do not reconcile with those reported here.

Table 4.2 Inverted file space requirements (MB)

Collection	IL Data	Overheads (% of IL data)			Total
		Top Docs	Free Space	Other	
Tip1	338	22 (6.5)	89 (26.4)	9 (2.7)	458
Tip12	574	30 (5.3)	122 (21.2)	11 (1.9)	737
Tip123	836	39 (4.6)	154 (18.4)	14 (1.7)	1043

Regardless, the overall inverted files are still only 33%–38% of the size of their respective document collections.

The more complicated long inverted list structure described in Section 4.3 does impose an additional time overhead when the inverted file is being built. The top document tables must be built, inverted list entries must be separated into frequencies and locations lists, and directories must be created for both of these lists. Fortunately, the overhead is restricted to the merge phase of indexing; the dominant cost of indexing—parsing—is the same regardless of the final inverted list structure.

For comparison to the results presented in Section 3.3.3, we measured the time required to build an inverted file employing the complex long inverted list structure for the 3.2 GB TIPSTER collection using the temporary file blocks produced during our bulk indexing experiment (see Table 3.2). On the platform described in Section 3.3.1, the Merger required 76 minutes to merge the temporary file blocks and build the final inverted file employing the complex long inverted list structure. This is nearly twice the time required by the Merger when the simpler linked list long inverted list structure is used (described in Section 3.2.3). The Parser requires 5 hours 21 minutes in either case, however, so the overall increase in indexing time is just 10%. Even with our more complex long inverted list implementation, we achieve a bulk indexing rate of 484 MB per hour.

4.4.3 Query Sets

The query sets used in these experiments were generated locally from topics provided for the *TREC* evaluations. The first query set, **Query Set 1**, was generated from *TIPSTER topics 51–100* using automatic and semi-automatic methods. The resultant fifty queries consisted primarily of weighted sums of terms, phrases, and ordered proximities, with an average of 39 terms per query.

The second query set, **Query Set 2**, was generated from *TIPSTER topics 151–200* in a series of steps. First, a base query set was created using automatic methods. Next, each base query was run against a *PhraseFinder* [46] database built from *TIPSTER* volumes 1 and 2. *PhraseFinder* returns a set of phrases extracted from the supporting database based on the given query. Thirty new phrases were automatically added to each query, forming an augmented query. The augmented queries were then interactively modified to simulate changes an end user might make to automatically generated queries. The changes were limited to the deletion of words judged spurious by the user, changes in weighting based on perceived relative importance, and the addition of proximity constraints. Approximately five minutes was spent on each query. Finally, each modified query was duplicated and one copy was placed inside a passage sum operator with a passage size of 200, which in turn was added to the other copy in a weighted sum. The final set of fifty queries contained an average of 105 terms per query.

The third query set, **Query Set 3**, was generated from *TIPSTER topics 51–100* by taking the text of the description section from each topic and placing it inside a sum operator. Rudimentary manual processing was performed to remove stop phrases, resulting in short, flat (i.e., unstructured) queries with an average of 8 terms per query. **Query Set 3** is essentially a simplified version of **Query Set 1**.

4.4.4 Performance Results

4.4.4.1 Safe

To evaluate the impact of the safe optimization techniques on execution speed, a comparison was made between the original linked list list implementation for long inverted lists described in Section 3.2.3 and the split list implementation for long inverted lists described in Section 4.3. The linked list implementation does *not* support the safe optimizations that are based on selective access of inverted list contents, while the split list implementation does. Two versions of the inverted file for **Tip12** were constructed, one using the linked list implementation, the other using the split list implementation. The three query sets were then run against each of these inverted file implementations and the execution time was measured using the GNU time command. Each query set was run five times and the average of the five runs is reported below. In all cases, the range between the best and worst times recorded for a given query set/implementation configuration was less than 1.3% of the average. Note that **Query Sets 1** and **2** were measured on the platform described in Section 4.4.1, while **Query Set 3** was measured on the platform described in Section 3.3.1.[4]

Figure 4.4 shows the wall-clock times broken down into CPU and I/O time for each of the configurations.[5] The labels in the figure are interpreted as follows: **Q***n* stands for **Query Set** *n*, **Linked** stands for the linked list implementation, and **Split** stands for the split list implementation. The impact of the safe optimizations is generally disappointing. In **Query Set 1**, the split list implementation is able to skip a total of 1,254 long list objects when "intersecting" proximity operators. This, combined with selective access of term weights and proximity information, leads to a reduction in *object faults* of 29% over the linked list implementation. An object fault occurs when a non-memory resident object is accessed and must be read from disk. The reduction in object faults translates into a reduction in I/O time of 14%. This would be notable, except that the increased complexity of the split list

[4]Note that no comparisons are made across platforms

[5]CPU time is the sum of the user and system CPU times returned by the GNU time command. I/O time is estimated by subtracting total CPU time from wall-clock time.

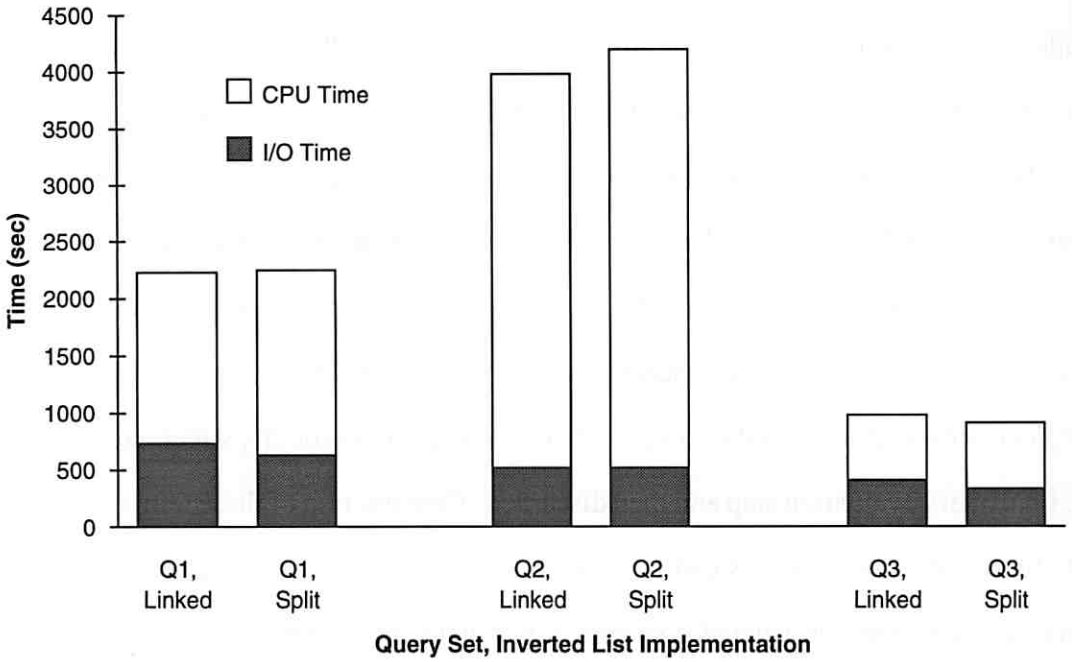


Figure 4.4 Linked versus Split inverted lists wall-clock time

implementation causes an increase in CPU time of 8% over the linked list implementation, for a net wall-clock time reduction of 0. In **Query Set 1**, the safe optimizations are a wash.

In **Query Set 2**, the situation is even worse. The split list implementation is able to skip 2,302 long list objects when "intersecting" proximity operators. However, recall that each query in **Query Set 2** has a core component that is duplicated, with one copy placed inside a passage operator. Since the passage operator requires proximity lists, and every term in a query will appear inside a passage operator, proximity lists are required for every term. No gains will be made from selective access of term weights and proximity information. This is reflected in the number of object faults recorded for each implementation—58,746 for the linked list implementation versus 58,471 for the split list implementation. Similarly, the I/O time required by both implementations is the same. The split list implementation, however, requires 6% more CPU time, resulting in a 5% increase in wall-clock time.

In **Query Set 3** we finally see a benefit to the split list implementation. There are no proximity operators in this query set, so no proximity lists are required during query

processing. The selective access of term weights provided by the split list implementation leads to a reduction in I/O time of 20%. The increase in CPU time caused by the split list implementation is only 2%, leading to an overall reduction in wall-clock time of 7%.

There is one more safe optimization technique that is specific to evaluation of proximity operators but independent of the inverted list implementation. In our discussion of query evaluation in INQUERY (Section 4.1.3), we noted that a preprocessing step is required to fully evaluate constructed concepts and compute their *idf* values. The results of this preprocessing step are saved in inverted lists constructed on the fly, which are used during the final query evaluation step and then discarded. Construction of these temporary inverted lists for constructed concepts can be viewed as a safe optimization. If these inverted lists were not built, the constructed concepts would have to be redundantly evaluated in full during the final query evaluation phase.

To see the effect of this optimization, we measured the wall-clock time required to evaluate **Query Sets 1** and **2** on **Tip12** both with and without temporary inverted lists for constructed concepts.[6] For **Query Set 1**, 2336 seconds are required to evaluate the query set without using temporary inverted lists. 36% of the terms appear inside proximity operators, and 255 seconds (11% of the total time) are spent in the preprocessing step. When temporary inverted lists are used, an average of 6 temporary inverted lists occupying a total of 188 KB are built per query, reducing the total evaluation time by 137 seconds (6%). This entire savings is due to a reduction in CPU time, indicating that when temporary inverted lists are not built, the inverted list data read during the preprocessing step is cached until the final evaluation step (i.e., no I/O is required during the redundant evaluation of the constructed concepts).

For **Query Set 2**, 4923 seconds are required to evaluate the query set when temporary inverted lists are not used. 54% of the terms appear inside proximity operators, and 594 seconds (12% of the total time) are spent in the preprocessing step. When temporary

[6]For these results, all experiments were run on the platform described in Section 3.3.1.

inverted lists are used, an average of 27 temporary inverted lists occupying a total of 449 KB are built per query, reducing the total evaluation time by 662 seconds (13%). Here, roughly 12% of the total savings is due to reduced I/O, while the remainder is due to reduced CPU time. The large number of terms and proximity operators per query in **Query Set 2** makes it impossible to cache all of the inverted list data read during the preprocessing step, such that, in addition to the CPU savings, building temporary inverted lists yields a noticeable savings in I/O during the final evaluation phase.

Curiously, the optimization reduces total evaluation time by more than the cost of the preprocessing step to evaluate the constructed concepts. This is unexpected, given that the optimization replaces the redundant evaluation of the constructed concepts (the equivalent of the preprocessing step) in the final evaluation phase with another computation—belief calculation from the temporary inverted lists. Belief calculation, however, is substantially less complex than proximity evaluation and causes much less data to be processed during evaluation. For the very large queries in **Query Set 2**, we speculate that this leads to better cache locality and a further reduction in execution time.

4.4.4.2 Unsafe

To evaluate our new unsafe optimization technique, we leave the linked list implementation behind and focus on the split list implementation. The new optimization was evaluated using a variety of experimental configurations, where each configuration involved three variables: query set, document collection, and level of optimization. **Query Set 1** was run against all three document collections, while **Query Set 2** was run against just the first two document collections (relevance judgements were not available for topics 151–200 on volume 3). (**Query Set 3** is evaluated separately in Section 4.6 below.) For a given query set and document collection, performance was measured at three levels of optimization: **all**, **1000**, and **100**. **all** is the unoptimized baseline, where the candidate document set is defined by the original query processing strategy described in Section 4.1.3. **1000** is the

Table 4.3 Number of documents evaluated

Collection	Qry Set	Documents (% change)		
		All	**1000**	**100**
Tip1	**1**	13436637	1057900 (−92)	382841 (−97)
	2	11131087	977694 (−91)	419740 (−96)
Tip12	**1**	21207958	1263141 (−94)	559012 (−97)
	2	17384562	1181650 (−93)	611787 (−96)
Tip123	**1**	29763641	1439024 (−95)	710976 (−98)

most conservative level of optimization we considered, where the candidate document set is populated from constructed concepts, short inverted lists, and the top 1000 documents from long inverted lists. **100** is a more aggressive level of optimization, where the candidate document set is populated from constructed concepts, short inverted lists, and the top 100 documents from long inverted lists. The level of optimization is controllable with a run-time switch allowing the same inverted file to be used for all optimization levels within a given configuration.

Our first metric of interest is the size of the candidate document set. Table 4.3 gives the total number of documents evaluated in each query set configuration. For example, when **Query Set 1** was run against **Tip1** with no optimization, scores were calculated for a total of 13,436,637 documents, or an average of 268,733 documents per query. This is over half of the documents in the entire collection. However, when only the top 1000 documents from long inverted lists are used to populate the candidate document set, scores were calculated for a total of 1,057,900 documents, or an average of 21,158 documents per query. We have reduced the number of documents being evaluated by over 90%. The more aggressive level of optimization reduces the number of documents being evaluated even further. From this table it is clear that we have met our first goal of reducing the size of the candidate document set.

The more important question is how this translates into a reduction in query processing time. To answer this question, we measured the real (wall-clock) time required to run

Table 4.4 Wall-clock times

Collection	Qry Set	Seconds (% change)		
		All	**1000**	**100**
Tip1	**1**	1364	632 (−54)	569 (−58)
	2	2530	938 (−63)	806 (−68)
Tip12	**1**	2258	1054 (−53)	980 (−57)
	2	4195	1535 (−63)	1394 (−67)
Tip123	**1**	3300	1518 (−54)	1445 (−56)

each query set configuration. Real time was measured using the GNU time command and includes all time from start to finish of the query set batch run, including the processing of relevance judgements. constant overhead, regardless of the optimization configuration. For example, relevance judgement processing requires 306 wall-clock seconds (41 CPU seconds, 265 I/O seconds) when evaluating **Query Set 1** on **Tip12**, and 247 wall-clock seconds (51 CPU seconds, 196 I/O seconds) when evaluating **Query Set 2** on **Tip12**. Elimination of relevance judgement processing would make query evaluation more CPU bound and would increase the percent improvement obtained with query optimization (the same constant reduction would occur in both unoptimized and optimized times, increasing the percentage difference between the two). While an interactive system does not have this overhead, it does have other overheads (e.g., document title lookup for display to the user). Therefore, we include relevance judgement processing in our measurements as a substitute for these other overheads. ten separate runs for each configuration. In all cases the range between the best and worst times recorded for a given configuration was less than 3.3% of the average for the configuration.

The query processing speedup realized even with our most conservative level of optimization is quite dramatic. In all cases, query processing time is cut at least in half. Moreover, most of the improvement is realized in the more conservative **1000** configuration. Optimizing more aggressively in the **100** configuration yields just an additional

2%–5% improvement over the baseline. Clearly we have achieved our ultimate goal of reducing query processing time.

With the candidate document set considerably reduced, we would expect to be able to skip significant portions of the long inverted lists during query evaluation. To measure this, we counted the number of whole objects skipped during long inverted list processing. Perhaps surprisingly, in all **1000** configurations there was no increase in the number of long list objects skipped. In fact, even at more aggressive optimization levels, the number of additional objects skipped was minimal. Moreover, the real impact of any skipping was measured in terms of a reduction in the number of object faults. Even when there was an increase in skipping, the reduction in object faults was insignificant, indicating that we were skipping memory resident objects which wouldn't have required a disk read anyway. An object will be memory resident if it was referenced during evaluation of a previous query and not purged from the buffer, or the associated term is used more than once in the current query, causing multiple references to the same inverted list.

The reason for the limited skipping is twofold. First, the information in the long inverted lists is very densely packed in order of document id. Second, the membership of the candidate set is independent of document id, meaning the entries in a long inverted list that must be accessed during query processing should be arbitrarily distributed over the entire list. Therefore, even though we are in fact skipping large portions of the long lists, we still end up accessing at least one document entry in nearly every object in the lists.

To investigate this effect further, we built our inverted files using 2 KB objects in the frequencies and locations lists. In this version skipping was more noticeable (especially at more aggressive optimization levels), but again the number of object faults was reduced by less than 2%. Moreover, since disk reads are 8 KB, we wouldn't expect to see any reduction in the number of raw disk I/Os when compared with the version that used 8 KB objects.

The question remains as to where the reduction in wall-clock time is coming from. The answer can be found by examining the CPU and I/O time components of the wall-clock

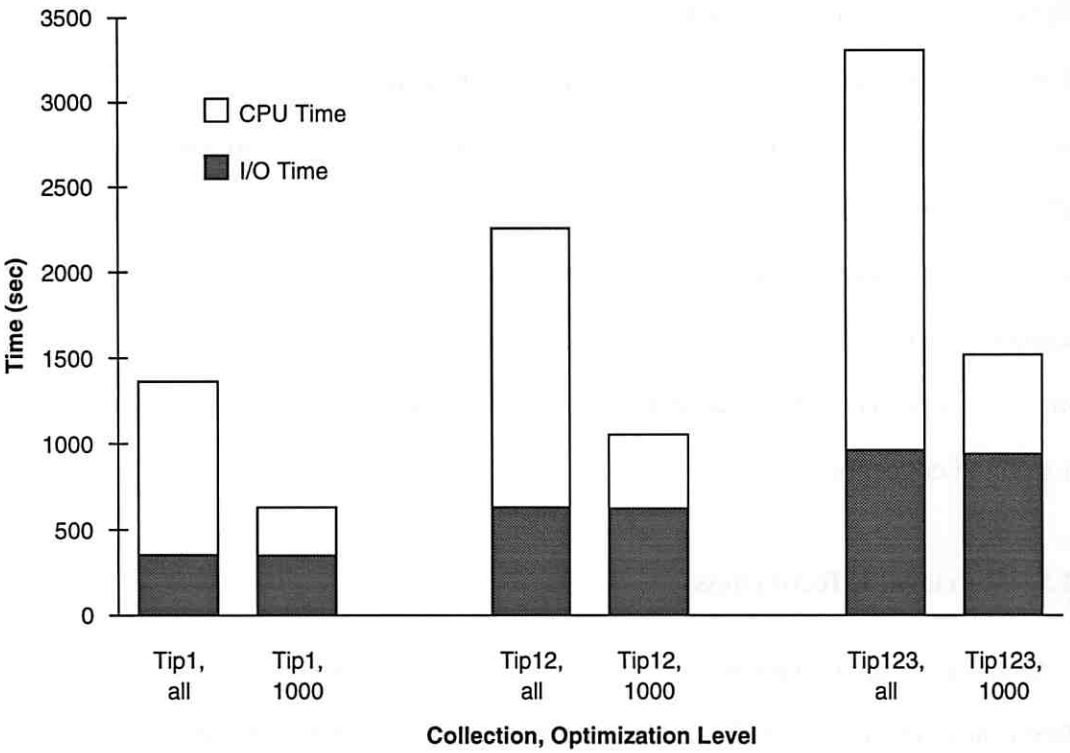


Figure 4.5 Query Set 1 wall-clock time breakdown

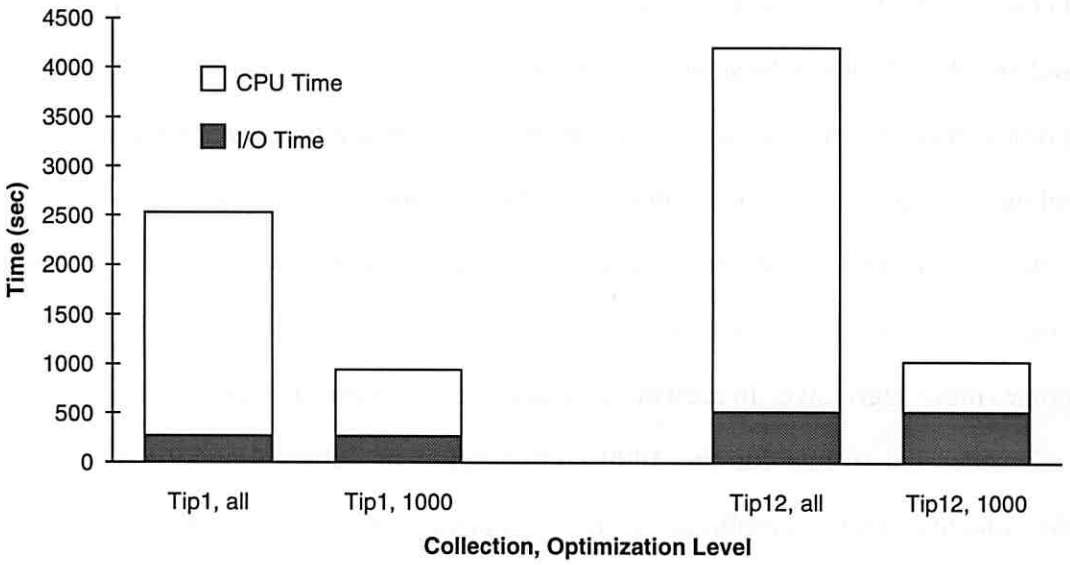


Figure 4.6 Query Set 2 wall-clock time breakdown

time. Figures 4.5 and 4.6 give the wall-clock time broken down into CPU and I/O time for baseline (**all**) and optimized (**1000**) versions of the two query sets on each of the three test collections. The figures show that the optimization reduces CPU time 70% to 75%, but has essentially no impact on I/O time. CPU time, however, is the dominant component of the wall-clock time, such that the CPU savings translates into a significant wall-clock savings. The rate of reduction in CPU time is still less than the rate of reduction in candidate document set size due to query evaluation overheads common to both the baseline and optimized versions, with the largest overhead being the preprocessing step to fully evaluate constructed concepts.

4.4.5 Retrieval Effectiveness

Along with query processing speed, we must also look at the impact on retrieval effectiveness in order to fully evaluate our unsafe optimization technique. Precision at standard recall points obtained with different levels of optimization for each of our five query set/document collection combinations is reported in Tables 4.5–4.9 (the corresponding Recall-Precision curves are shown in Figures 4.7–4.9). The relevance judgements used to generate these tables came from the *TREC* evaluations. We show interpolated precision based on full rankings at the standard 11 recall points and the 11pt average. As before, **all** is the unoptimized baseline version, while **1000** through **50** are optimized versions where the label indicates the number of top documents taken from long inverted lists to populate the candidate document set. We show a broader range of optimization levels here than in our timing test to give a better feel for the impact on retrieval effectiveness as the optimization becomes more aggressive. In each of the tables, percent change is from the baseline version.

Consider the results for the **1000** configuration in Tables 4.5–4.9. For all query set/document collection combinations, retrieval effectiveness is remarkably good. At recall levels up to 70%, there is no noticeable degradation in precision. The implication here is that the high end of a document ranking returned by the optimized system, or the docu-

Table 4.5 Precision at standard recall pts for Tip1, Query Set 1

	Precision (% change) – 50 queries					
Recall	**all**	**1000**	**500**	**300**	**100**	**50**
0	83.5	83.7 (+0.2)	83.9 (+0.5)	83.9 (+0.5)	83.9 (+0.5)	83.9 (+0.5)
10	60.3	60.5 (+0.2)	60.9 (+1.0)	60.8 (+0.8)	61.4 (+1.7)	61.6 (+2.1)
20	52.7	53.0 (+0.6)	53.3 (+1.2)	53.4 (+1.3)	53.5 (+1.5)	53.1 (+0.8)
30	46.8	47.1 (+0.6)	47.0 (+0.4)	46.7 (−0.3)	46.1 (−1.7)	44.3 (−5.5)
40	40.6	40.9 (+0.7)	40.9 (+0.8)	41.0 (+1.0)	38.9 (−4.2)	38.4 (−5.5)
50	34.9	35.2 (+1.0)	35.1 (+0.8)	35.1 (+0.7)	33.1 (−5.0)	32.4 (−7.1)
60	30.4	30.6 (+0.6)	30.7 (+1.1)	30.1 (−1.0)	28.1 (−7.6)	27.2(−10.4)
70	25.3	25.7 (+1.7)	25.6 (+1.3)	24.0 (−5.1)	20.9(−17.1)	20.4(−19.4)
80	19.9	19.8 (−0.1)	18.3 (−7.9)	17.7(−10.9)	15.8(−20.7)	14.6(−26.4)
90	12.1	11.6 (−4.6)	11.3 (−6.9)	9.6(−20.9)	8.6(−29.4)	7.6(−37.1)
100	2.4	1.7(−29.2)	1.5(−38.7)	1.6(−36.2)	1.6(−36.1)	1.6(−33.3)
average	37.2	37.3 (+0.2)	37.2 (−0.1)	36.7 (−1.2)	35.6 (−4.2)	35.0 (−5.8)

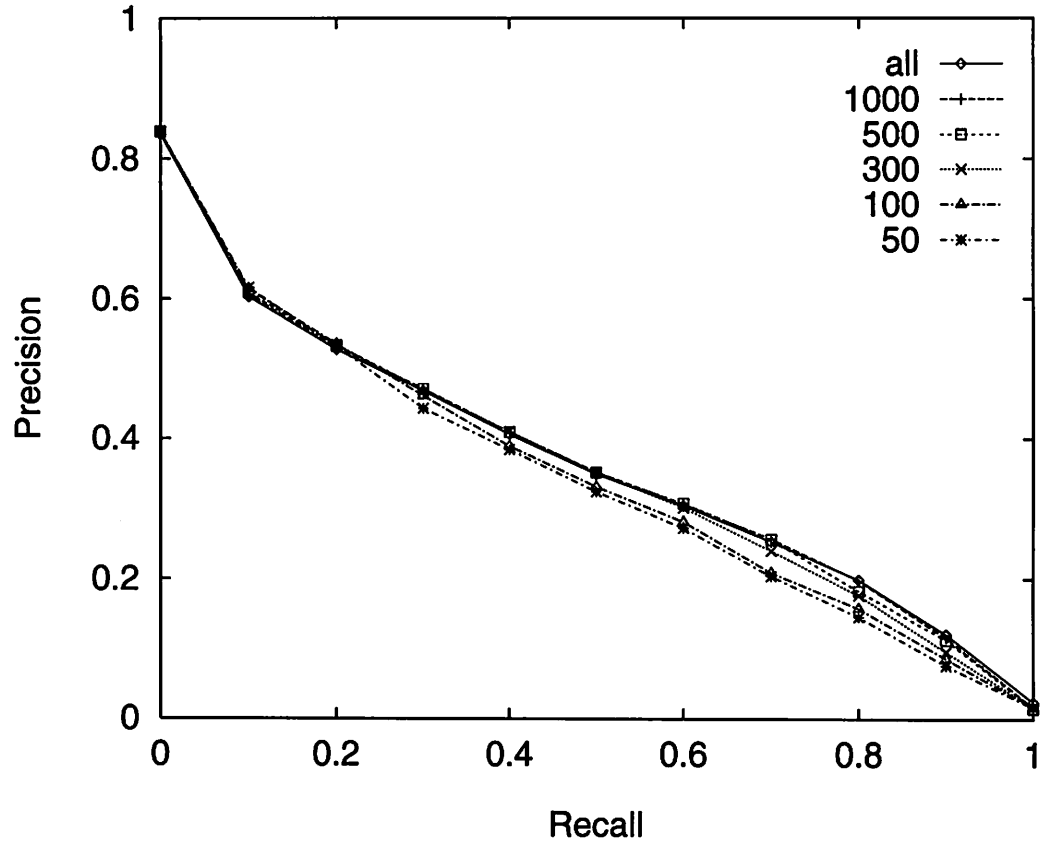


Figure 4.7 Recall-Precision curves for Tip1, Query Set 1

Table 4.6 Precision at standard recall pts for Tip12, Query Set 1

	Precision (% change) – 50 queries					
Recall	all	1000	500	300	100	50
0	83.6	83.7 (+0.1)	83.5 (−0.1)	83.3 (−0.4)	83.3 (−0.3)	83.6 (+0.0)
10	57.2	57.5 (+0.6)	57.7 (+0.9)	57.7 (+0.9)	56.8 (−0.6)	56.5 (−1.2)
20	49.0	49.5 (+1.0)	49.7 (+1.4)	49.6 (+1.1)	48.7 (−0.7)	48.1 (−1.9)
30	43.1	43.4 (+0.8)	43.5 (+0.9)	43.2 (+0.4)	42.0 (−2.5)	40.3 (−6.4)
40	37.7	38.1 (+1.0)	38.0 (+0.9)	37.3 (−1.0)	34.9 (−7.5)	34.4 (−8.8)
50	32.4	32.9 (+1.5)	32.5 (+0.3)	32.0 (−1.3)	29.2 (−9.8)	28.7(−11.3)
60	27.7	27.9 (+0.6)	27.2 (−1.8)	26.0 (−6.1)	23.9(−13.6)	23.2(−16.5)
70	22.5	22.8 (+1.4)	21.7 (−3.8)	20.2(−10.4)	17.7(−21.5)	17.2(−23.7)
80	17.3	17.0 (−1.6)	15.0(−13.4)	13.8(−20.0)	12.2(−29.3)	12.1(−29.9)
90	11.2	10.0(−10.6)	8.5(−24.2)	7.8(−30.3)	7.8(−30.7)	7.8(−30.1)
100	1.2	0.5(−59.3)	0.6(−54.0)	0.6(−55.4)	0.7(−47.5)	0.7(−42.2)
average	34.8	34.9 (+0.1)	34.3 (−1.3)	33.8 (−3.0)	32.5 (−6.7)	32.1 (−7.9)

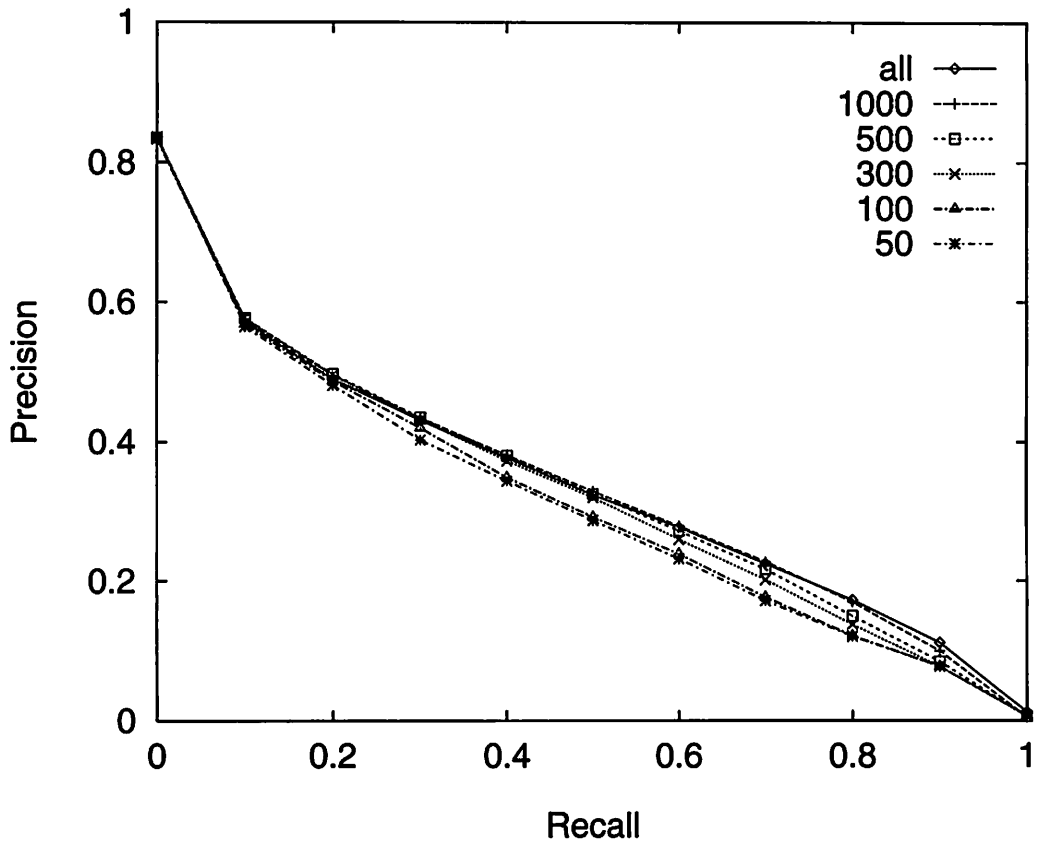


Figure 4.8 Recall-Precision curves for Tip12, Query Set 1

Table 4.7 Precision at standard recall pts for Tip123, Query Set 1

	Precision (% change) – 50 queries					
Recall	**all**	**1000**	**500**	**300**	**100**	**50**
0	84.3	84.3 (+0.0)	84.2 (−0.1)	84.1 (−0.2)	84.7 (+0.5)	84.5 (+0.3)
10	54.6	54.8 (+0.4)	55.0 (+0.8)	54.8 (+0.4)	53.6 (−1.8)	53.2 (−2.5)
20	47.1	47.3 (+0.5)	47.3 (+0.4)	47.0 (−0.2)	45.4 (−3.5)	43.9 (−6.6)
30	40.6	40.9 (+0.7)	40.5 (−0.3)	39.7 (−2.3)	37.2 (−8.4)	36.4(−10.3)
40	35.3	35.5 (+0.5)	34.9 (−1.3)	33.4 (−5.4)	30.7(−12.9)	30.0(−14.9)
50	30.3	30.4 (+0.6)	29.5 (−2.6)	27.9 (−7.9)	25.7(−15.0)	25.7(−15.2)
60	25.7	25.7 (+0.1)	24.0 (−6.7)	22.4(−13.0)	21.1(−18.0)	20.9(−18.5)
70	20.7	20.0 (−3.6)	18.1(−12.6)	17.2(−17.2)	16.7(−19.2)	16.6(−19.9)
80	15.5	13.3(−14.3)	11.8(−24.1)	11.3(−27.0)	10.2(−34.0)	10.2(−34.2)
90	9.1	7.2(−20.9)	6.1(−32.6)	5.8(−35.9)	6.0(−34.2)	6.1(−33.4)
100	0.5	0.2(−67.2)	0.1(−73.6)	0.1(−73.2)	0.1(−72.7)	0.1(−72.6)
average	33.1	32.7 (−1.1)	31.9 (−3.4)	31.2 (−5.5)	30.1 (−8.8)	29.8 (−9.9)

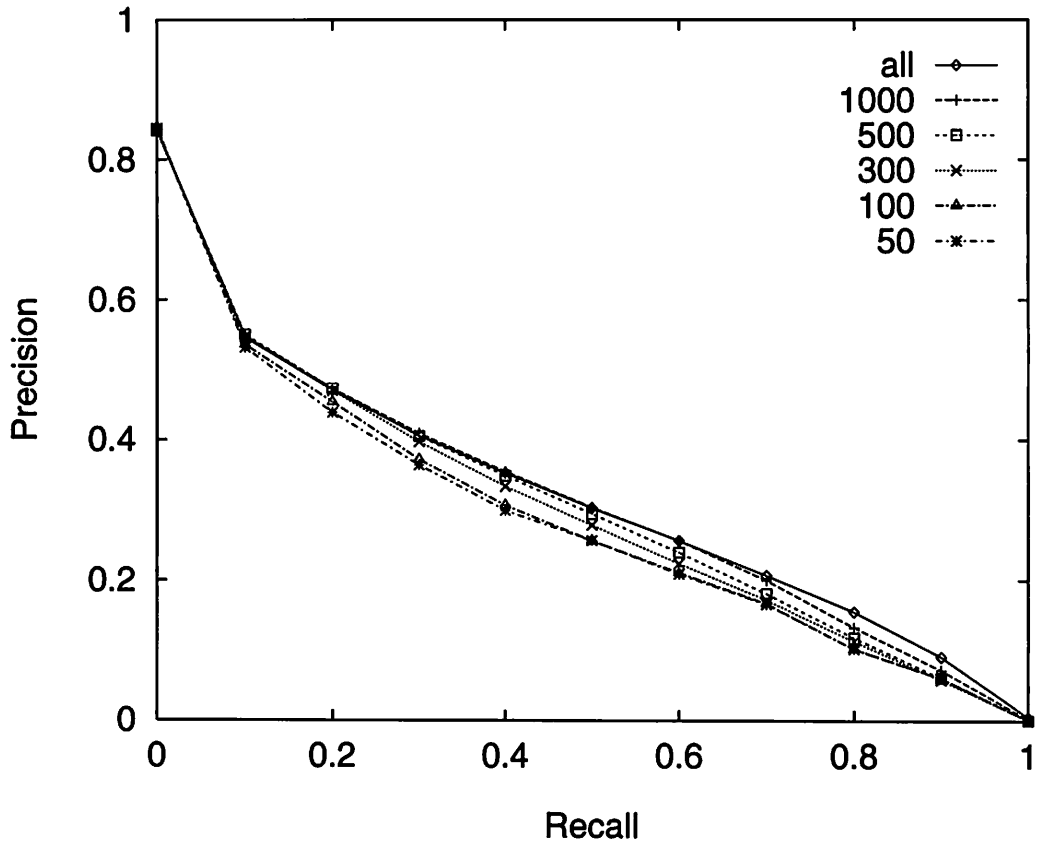


Figure 4.9 Recall-Precision curves for Tip123, Query Set 1

Table 4.8 Precision at standard recall pts for Tip1, Query Set 2

	Precision (% change) – 50 queries					
Recall	**all**	**1000**	**500**	**300**	**100**	**50**
0	91.1	91.1 (+0.0)	91.1 (+0.0)	91.1 (+0.0)	91.1 (+0.0)	91.1 (+0.0)
10	75.9	75.9 (−0.0)	75.8 (−0.1)	75.8 (−0.1)	75.8 (−0.1)	75.8 (−0.1)
20	66.0	66.0 (−0.0)	65.9 (−0.1)	65.8 (−0.3)	65.8 (−0.3)	65.8 (−0.3)
30	55.6	55.6 (+0.1)	55.2 (−0.7)	55.2 (−0.7)	55.0 (−1.1)	54.9 (−1.3)
40	47.4	47.4 (+0.1)	47.0 (−0.8)	47.0 (−0.8)	46.6 (−1.6)	46.4 (−2.1)
50	41.4	41.3 (−0.2)	41.1 (−0.7)	40.8 (−1.3)	40.3 (−2.7)	40.2 (−2.9)
60	35.1	35.0 (−0.1)	34.8 (−0.8)	34.5 (−1.7)	32.8 (−6.4)	32.4 (−7.5)
70	27.4	27.3 (−0.4)	26.6 (−2.9)	26.1 (−5.0)	24.7 (−9.8)	24.6(−10.2)
80	22.1	21.8 (−1.2)	21.4 (−3.1)	21.4 (−3.1)	19.4(−12.3)	18.6(−15.5)
90	15.5	15.3 (−1.5)	14.6 (−6.1)	13.1(−15.9)	11.0(−29.4)	10.6(−31.8)
100	3.7	2.7(−27.4)	2.3(−37.0)	1.8(−51.9)	1.4(−62.8)	1.3(−64.6)
average	43.7	43.6 (−0.4)	43.3 (−1.1)	43.0 (−1.8)	42.2 (−3.6)	42.0 (−4.0)

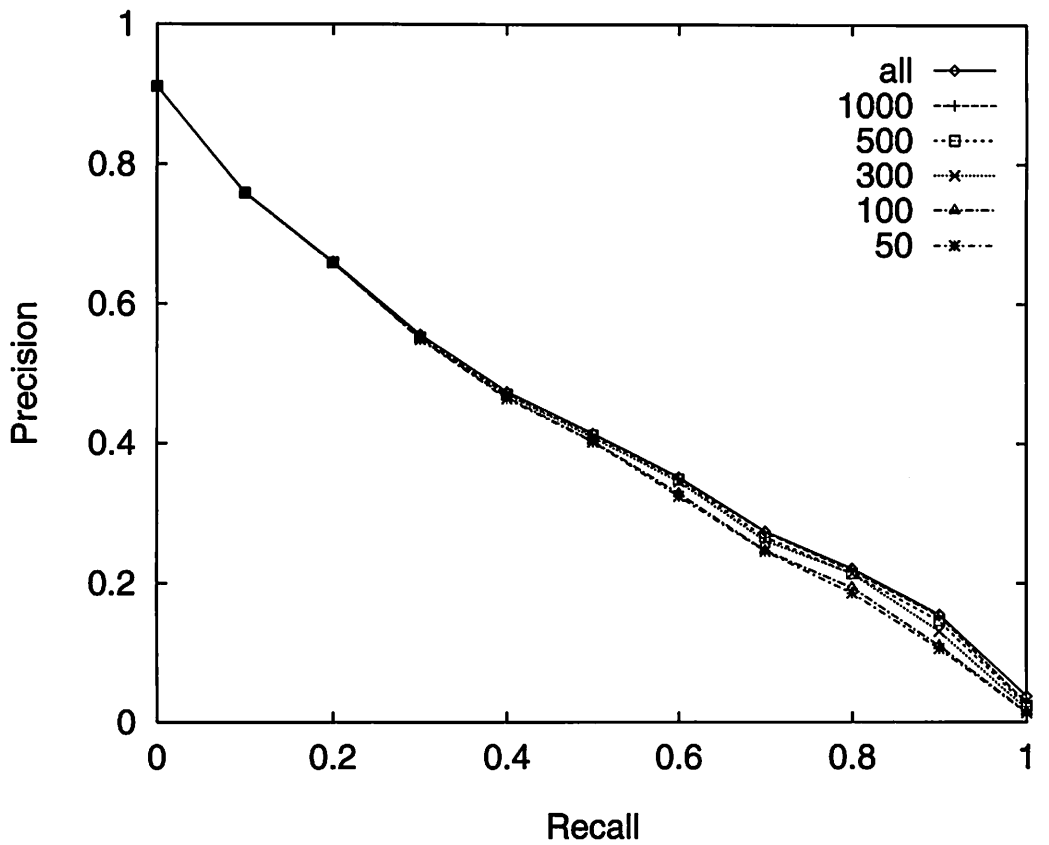


Figure 4.10 Recall-Precision curves for Tip1, Query Set 2

Table 4.9 Precision at standard recall pts for Tip12, Query Set 2

	Precision (% change) – 50 queries					
Recall	**all**	**1000**	**500**	**300**	**100**	**50**
0	89.4	89.4 (+0.0)	89.4 (+0.0)	89.4 (+0.0)	89.4 (+0.0)	89.4 (+0.0)
10	73.8	73.8 (−0.0)	73.7 (−0.1)	73.7 (−0.1)	73.7 (−0.1)	73.7 (−0.2)
20	64.3	64.2 (−0.1)	64.1 (−0.3)	64.1 (−0.2)	64.1 (−0.3)	64.1 (−0.3)
30	56.6	56.5 (−0.0)	56.4 (−0.2)	56.2 (−0.5)	56.0 (−1.0)	55.9 (−1.2)
40	49.6	49.6 (−0.0)	49.6 (−0.1)	49.5 (−0.4)	49.0 (−1.2)	48.9 (−1.4)
50	43.6	43.5 (−0.3)	43.2 (−0.9)	43.0 (−1.5)	41.9 (−3.9)	41.8 (−4.2)
60	36.9	36.4 (−1.2)	35.7 (−3.2)	34.7 (−5.8)	33.3 (−9.7)	33.2 (−9.8)
70	30.1	29.5 (−2.1)	28.8 (−4.3)	28.1 (−6.6)	25.8(−14.5)	25.3(−16.0)
80	24.7	24.0 (−3.2)	23.4 (−5.2)	21.1(−14.8)	20.1(−18.8)	20.0(−19.4)
90	16.5	15.4 (−6.7)	13.0(−21.3)	12.7(−22.9)	12.5(−24.2)	12.5(−24.2)
100	2.3	1.4(−37.9)	1.4(−39.4)	1.5(−37.6)	1.0(−56.4)	1.0(−55.9)
average	44.4	44.0 (−0.8)	43.5 (−1.8)	43.1 (−2.8)	42.4 (−4.3)	42.3 (−4.5)

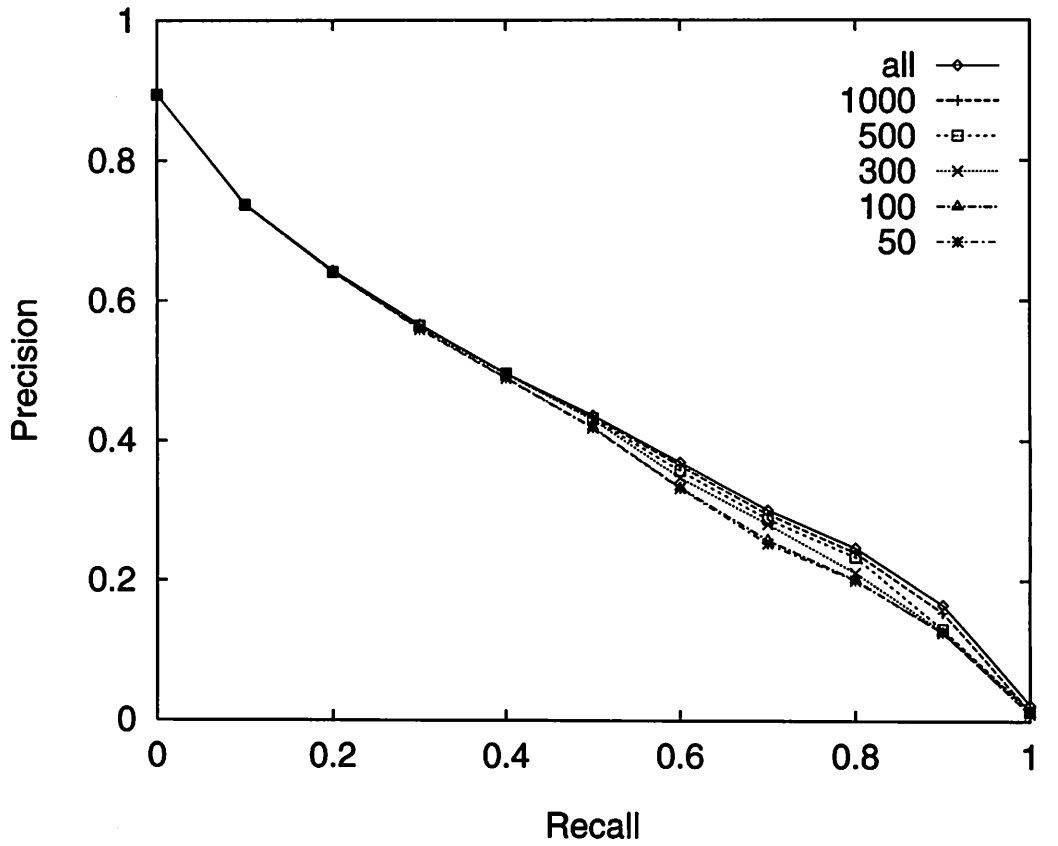


Figure 4.11 Recall-Precision curves for Tip12, Query Set 2

ments most likely to be considered by a user in an interactive system, will be just as rich in relevant documents as in the unoptimized version. Furthermore, the 11pt averages are not significantly different from those for the unoptimized version.

Now consider the results in Table 4.5. As the optimization becomes more aggressive (from **1000** to **50**), we see two trends. First, at low recall, precision actually improves a tiny amount and then falls off. This indicates that the technique is doing a good job of identifying the very best candidate documents, and is consistent with other results using similar techniques [65, 58]. Second, at high recall, precision becomes significantly worse as the optimization becomes more aggressive. This is because we are not considering documents which have a strong combined belief from all of the query terms, but lack a single query term belief strong enough to place the document in the candidate set.

In Tables 4.8 and 4.9 we do not see any improvement in precision at low recall as the optimization becomes more aggressive. This is due to the use of the passage operator in **Query Set 2**. The calculation of belief for concept i in document j is slightly modified inside a passage operator since it is based on a passage of the document, rather than the entire document. Thus, our ranking of document j within the inverted list for concept i is slightly inaccurate with respect to the passage operator. This suggests that our retrieval performance could even be improved.

4.5 Extensions

The optimization technique described above has a large impact on CPU time, but very little impact on I/O. In the baseline query sets considered above, CPU time accounts for 70% to 90% of the overall running time, such that reducing CPU costs is an appropriate goal. After the optimization has been applied, however, I/O becomes a larger component of overall running time. A natural question that arises here is whether or not the amount of I/O that must be performed during query evaluation can be reduced. Two approaches

for explicitly reducing I/O are considered below (from here on the unsafe optimization described above will be referred to as the *original* optimization).

The first approach explores the effects of ignoring the bulk of a long inverted list and using just the term weighting information stored in the inverted list's top document list. In the original optimization, a long inverted list l contributes belief scores for *all* of the documents in the candidate document set that contain l's associated term. In other words, during final query evaluation, l will contribute belief scores not only for the documents added to the candidate set by l, but also for other documents added to the candidate set by other parts of the query, where those documents happen to contain the term associated with l. These other documents appear in l, just not in l's top document list. The result is that large parts of l must still be retrieved during final query evaluation to obtain belief scores for these other documents.

If instead l contributes belief scores for documents in its top document list only, the rest of l can be ignored and significant I/O savings should be realized. As with the original optimization, constructed concepts are fully built during the preprocessing step. Moreover, this extended optimization is applied only to *selected* long inverted lists in the query. The selection is made by identifying all of the terms in the query tree reachable from the root along a path that includes only sum, weighted sum, and, or, and max operators—the other query operators are either proximity operators that were evaluated in the preprocessing step anyway, or operators where this approach is inappropriate. The identified terms, called the *optimization candidates*, are then sorted in increasing order of weighted *idf* score (the weighting is based on any weighted sum operators encountered on the path from the query root to the term). A percentage of the lowest scoring terms are then selected for application of the extended optimization, such that the optimization is applied to the terms with the lowest estimated impact on final document score. This approach is called *top-docs-only*.

The second approach is a more aggressive version of the first approach. Rather than obtain belief scores from a selected long inverted list's top document list, the list is ignored

altogether. The lists to ignore are selected in the same way as above—a percentage of the terms are chosen based on weighted *idf* score from the optimization candidates. This approach is similar to the optimization of Buckley and Lewit [10], where entire inverted lists are ignored during query evaluation. The approaches differ in the way the inverted lists to ignore are chosen. Buckley and Lewit use upper bound thresholds to decide when an inverted list can be ignored without affecting the top ranked documents. In our case, query structure complicates the computation and maintenance of similar upper bounds. Instead, we ignore an arbitrary percentage of the inverted lists with the lowest estimated impact on final document score. This approach is called *term-elimination.*

A preliminary investigation of these two approaches revealed that the retrieval effectiveness obtained with top-docs-only is the same as or inferior to the retrieval effectiveness obtained with term-elimination. This is shown in Tables 4.10 and 4.11 (the corresponding Recall-Precision curves are shown in Figures 4.12 and 4.13) for **Query Sets 1** and **2** on **Tip12**. Each table gives the precision at standard recall points for the baseline version (**all**), the original optimization using 1000 top documents from long lists (**1000**), the original optimization extended with top-docs-only on 50% of the terms (**1000-50a**), and the original optimization extended with term-elimination on 50% of the terms (**1000-50b**). Term-elimination provides a greater execution savings than top-docs-only because selected terms are completely ignored, rather than evaluated using their top document list. Given the relative retrieval effectiveness of the two approaches, term-elimination is deemed superior to top-docs-only, and top-docs-only is not considered further.

Term-elimination is a general optimization technique by itself; it can be applied directly to the baseline (**all**) configuration, as well as in combination with the original optimization. To determine how these optimizations compare and interact, an evaluation of the performance of different optimization configurations was conducted using both query sets on **Tip12**. The experiments were run on the same platform described in Section 3.3.1. Note that the large disk drive used in that platform is different from the one used in the

Table 4.10 Precision at standard recall pts for Tip12, Query Set 1, extended

	Precision (% change) – 50 queries							
Recall	**all**	**1000**		**1000-50a**		**1000-50b**		
0	83.6	83.7	(+0.1)	85.5	(+2.2)	85.8	(+2.6)
10	57.2	57.5	(+0.6)	59.4	(+3.9)	58.0	(+1.4)
20	49.0	49.5	(+1.0)	49.5	(+1.0)	49.2	(+0.4)
30	43.1	43.4	(+0.8)	44.1	(+2.3)	44.2	(+2.7)
40	37.7	38.1	(+1.0)	38.4	(+1.8)	38.9	(+3.1)
50	32.4	32.9	(+1.5)	33.1	(+2.1)	33.5	(+3.5)
60	27.7	27.9	(+0.6)	27.5	(−0.7)	28.1	(+1.2)
70	22.5	22.8	(+1.4)	21.8	(−3.0)	22.9	(+1.9)
80	17.3	17.0	(−1.6)	16.5	(−4.3)	17.3	(−0.2)
90	11.2	10.0	(−10.6)	10.3	(−7.8)	10.8	(−3.7)
100	1.2	0.5	(−59.3)	0.6	(−53.4)	0.7	(−39.9)
average	34.8	34.9	(+0.1)	35.2	(+1.0)	35.4	(+1.7)

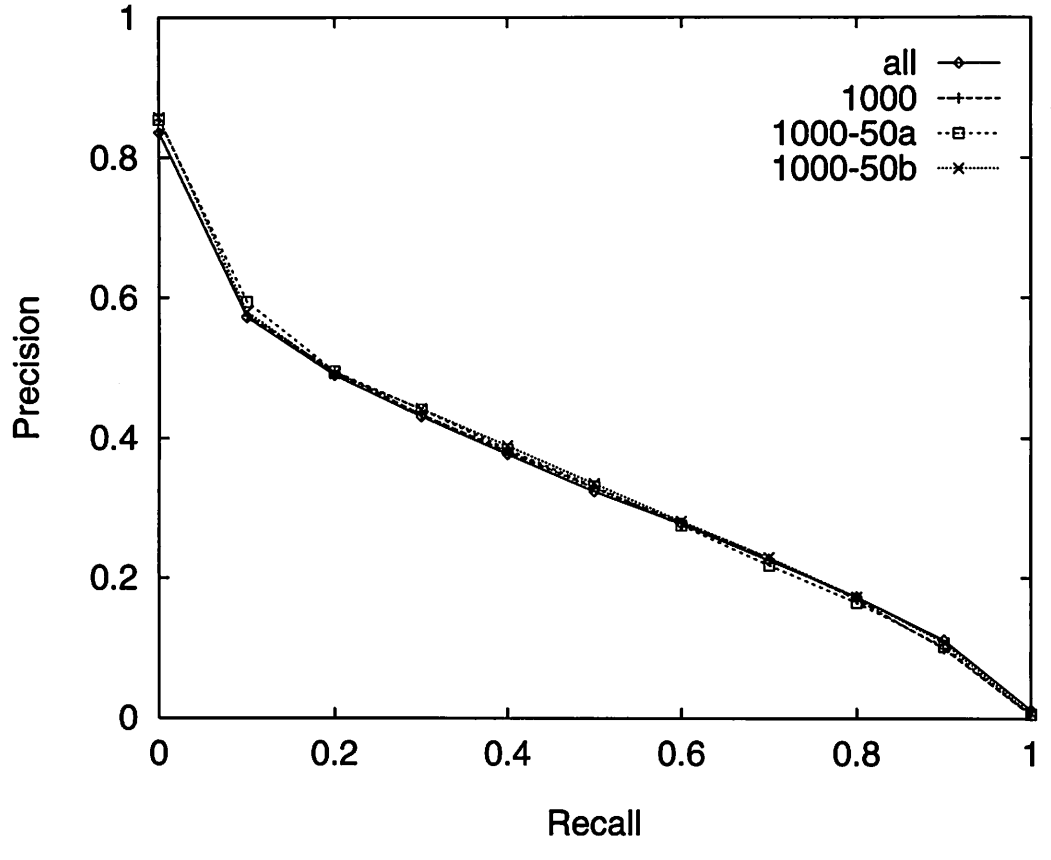


Figure 4.12 Recall-Precision curves for Tip12, Query Set 1, extended

Table 4.11 Precision at standard recall pts for Tip12, Query Set 2, extended

	Precision (% change) – 50 queries			
Recall	**all**	**1000**	**1000-50a**	**1000-50b**
0	89.4	89.4 (+0.0)	89.1 (−0.4)	89.3 (−0.1)
10	73.8	73.8 (−0.0)	73.6 (−0.3)	73.6 (−0.2)
20	64.3	64.2 (−0.1)	63.9 (−0.6)	63.9 (−0.6)
30	56.6	56.5 (−0.0)	56.4 (−0.3)	56.3 (−0.4)
40	49.6	49.6 (−0.0)	49.5 (−0.3)	49.7 (+0.1)
50	43.6	43.5 (−0.3)	43.7 (+0.1)	43.6 (+0.0)
60	36.9	36.4 (−1.2)	36.3 (−1.5)	35.9 (−2.6)
70	30.1	29.5 (−2.1)	29.5 (−2.1)	29.3 (−2.7)
80	24.7	24.0 (−3.2)	24.0 (−2.9)	24.0 (−3.1)
90	16.5	15.4 (−6.7)	15.4 (−6.7)	15.3 (−7.0)
100	2.3	1.4 (−37.9)	1.5 (−36.0)	1.5 (−35.8)
average	44.4	44.0 (−0.8)	43.9 (−1.0)	43.9 (−1.1)

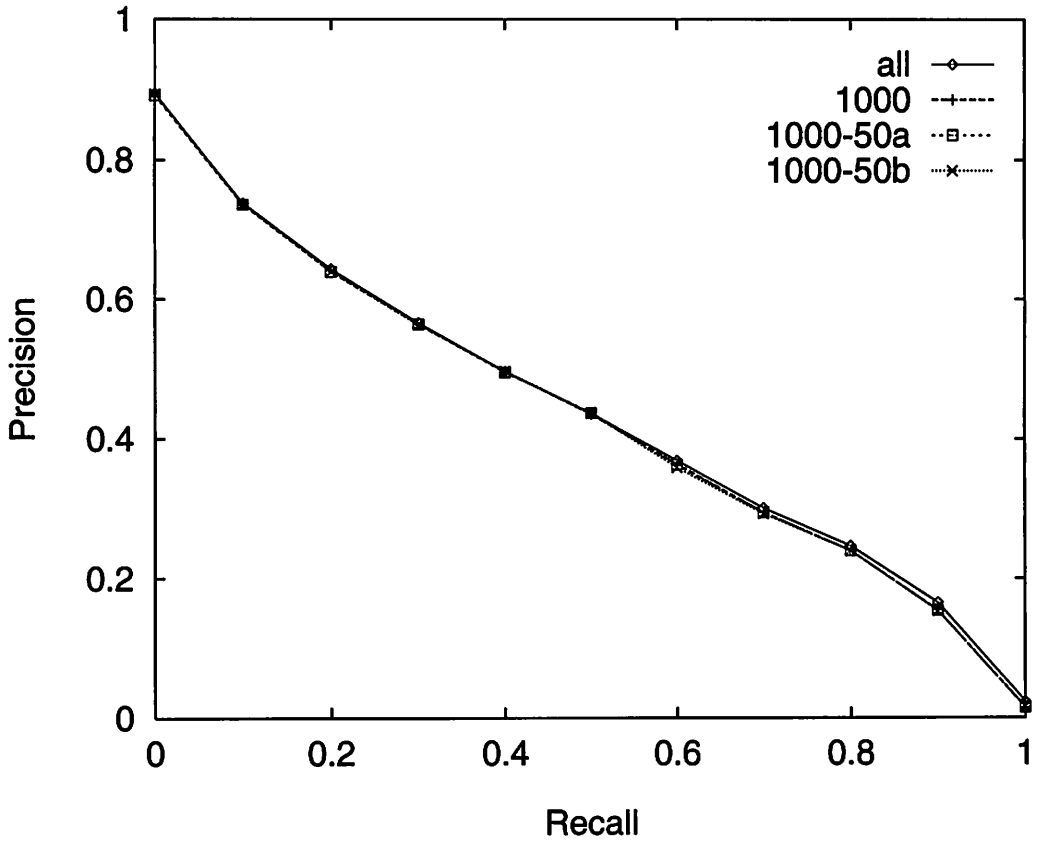


Figure 4.13 Recall-Precision curves for Tip12, Query Set 2, extended

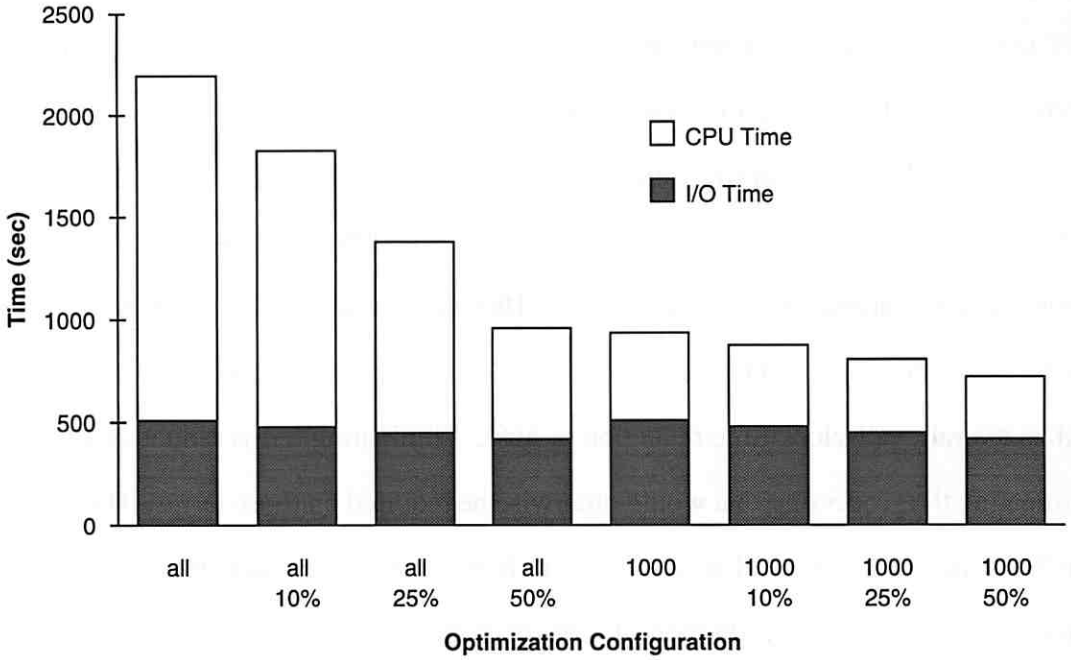


Figure 4.14 Extended optimization wall-clock times for Tip12, Query Set 1

platform described in Section 4.4.1, so the timing results presented below are *not* directly comparable to those presented in Section 4.4.4.

As before, all of the data files and executables were stored on the larger local disk, and a 64 MB "chill file" was read before each query processing run to purge the operating system file buffers and guarantee that no inverted file data was cached by the file system across runs. In all cases 15 MB of Mneme buffer space was allocated to cache memory resident inverted list objects. The timing results were measured with the GNU time command and the average of 5 runs is reported for each configuration. In all cases the range between the best and worst times recorded for a given configuration was less than 3% of the average for the configuration.

The execution performance for **Query Set 1** on **Tip12** is shown in Figure 4.14. Each bar gives the wall-clock time broken down into CPU and I/O components for a given configuration (raw timing figures for all of the query sets considered throughout the rest of this Chapter are summarized in Table 4.19). The bar label on the x-axis identifies the

configuration. For example, **all** is the unoptimized baseline, **all 50%** is the baseline plus 50% term-elimination, and **1000 50%** is the original optimization using 1000 top documents from long lists plus 50% term-elimination.

The term-elimination optimization is quite effective when used by itself. Ignoring 50% of the terms identified as candidates for elimination (**all 50%**) produces a reduction in wall-clock time comparable to that achieved in the **1000** configuration. Compared to **all**, **all 50%** produces a reduction in I/O time of nearly 18%, a reduction in CPU time of nearly 68%, and an overall wall-clock time reduction of 56%. Eliminating terms reduces CPU time by eliminating the processing that would otherwise be required on those terms. The size of the candidate document set is also reduced. In **all 50%**, 5,941,239 documents are evaluated across the 50 queries—a reduction of 72% from the 21,207,958 documents evaluated in **all**. This is still substantially less than the 94% reduction in candidate document set size afforded by **1000** (see Table 4.3), explaining why the reduction in CPU time obtained with **1000** is better than that obtained with **all 50%**. In **1000**, CPU time is reduced by 75%, compared to 68% for **all 50%**.

Recall that the original motivation for this optimization was to reduce I/O. While the 18% reduction in I/O time is notable, it is not exceptional. Since 50% of the optimization candidate terms are not processed, and these are the terms with the largest inverted lists, we might expect a much larger reduction in I/O. The reason for this less-than-expected reduction in I/O is revealed by looking at the object fault rates. Compared to **all**, **all 50%** reduces the number of object faults by only 20%. The number of object references is actually reduced by 30%, indicating that the optimization is eliminating references to objects that were already resident in main memory—eliminating these references nets no savings in I/O. Furthermore, the optimization is eliminating 50% of the *optimization candidate* terms only. The selection algorithm does not consider for elimination terms that participate in a constructed concept (i.e., proximity operator), so the optimization is actually eliminating less than 50% of the total terms in the query. It is also possible that a term appears more

than once in the query with different weighted *idf* scores, causing it to be selected for elimination in one part of the query but not the other. No I/O will be saved in this case since only one copy of the term's inverted list would have been read in the unoptimized version, and this copy must still be read in the optimized version.

When term-elimination is combined with the original optimization, the execution performance improvement is even better. **1000 50%** produces a reduction in I/O time of 18%, a reduction in CPU time of 82%, and a reduction in total wall-clock time of 67%. Compared to **all 50%**, **1000 50%** reduces overall wall-clock time by an additional 25%. This suggests that term-elimination is complementary to the original optimization and the best execution performance will be obtained by combining the two techniques.

The term-elimination optimization is unsafe; we must assess its impact on retrieval effectiveness. Table 4.12 and Figure 4.15 give precision at standard recall points for the baseline case and selected configurations of the optimization. Surprisingly, precision at nearly all levels of recall improves up to a certain point as a larger percentage of the high frequency terms are eliminated. The best precision is found at 50% term-elimination. At 75% term-elimination, precision has substantially deteriorated. Moreover, adding 50% term-elimination to the original optimization improves its precision at nearly all recall levels as well. Although improving retrieval effectiveness is never frowned upon, obtaining the improvement by removing evidence from the query suggests that the evidence is being improperly incorporated into the final document belief scores. Rajashekar and Croft [68] show that retrieval effectiveness generally improves as more evidence is added to the query. It is likely, therefore, that the query can be expressed better, either with improved term weighting, different query operators, or an improved retrieval model. We will return to this issue later in the context of the other query sets.

1000 50% produces the best precision at low recall of all of the configurations listed in Table 4.12, albeit by an insignificant margin. Low recall corresponds to the top end of the ranked listing returned to the user; it is the more important end of the recall spectrum

Table 4.12 Precision at standard recall pts for Tip12, Query Set 1, optimized

	Precision (% change) – 50 queries					
Recall	**all**	**all 50%**	**all 75%**	**1000**	**1000 50%**	**1000 75%**
0	83.6	85.8 (+2.6)	87.4 (+4.5)	83.7 (+0.1)	85.8 (+2.6)	87.4 (+4.5)
10	57.2	57.8 (+1.2)	54.8 (−4.1)	57.5 (+0.6)	58.0 (+1.4)	54.8 (−4.1)
20	49.0	49.0 (−0.1)	45.3 (−7.6)	49.5 (+1.0)	49.2 (+0.4)	45.4 (−7.4)
30	43.1	44.2 (+2.6)	38.1(−11.5)	43.4 (+0.8)	44.2 (+2.7)	38.2(−11.2)
40	37.7	38.8 (+2.9)	32.4(−14.1)	38.1 (+1.0)	38.9 (+3.1)	32.5(−13.8)
50	32.4	33.5 (+3.2)	27.5(−15.2)	32.9 (+1.5)	33.5 (+3.5)	27.8(−14.1)
60	27.7	28.0 (+0.9)	22.7(−18.2)	27.9 (+0.6)	28.1 (+1.2)	23.0(−17.0)
70	22.5	23.0 (+2.1)	17.7(−21.4)	22.8 (+1.4)	22.9 (+1.9)	18.4(−18.4)
80	17.3	17.8 (+2.9)	13.0(−24.9)	17.0 (−1.6)	17.3 (−0.2)	12.4(−28.3)
90	11.2	12.0 (+7.2)	8.2(−27.1)	10.0(−10.6)	10.8 (−3.7)	8.0(−29.0)
100	1.2	1.3 (+7.2)	0.6(−48.8)	0.5(−59.3)	0.7(−39.9)	0.3(−74.7)
average	34.8	35.6 (+2.1)	31.6 (−9.2)	34.9 (+0.1)	35.4 (+1.7)	31.7 (−9.1)

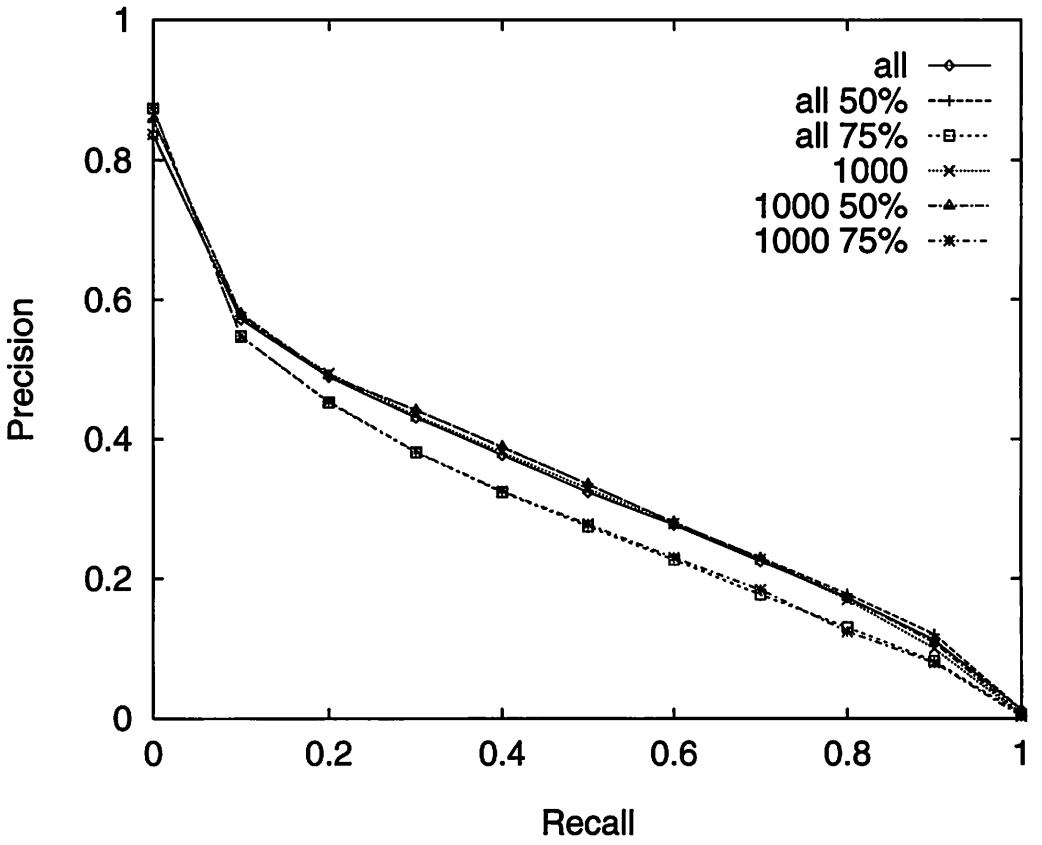


Figure 4.15 Recall-Precision curves for Tip12, Query Set 1, optimized

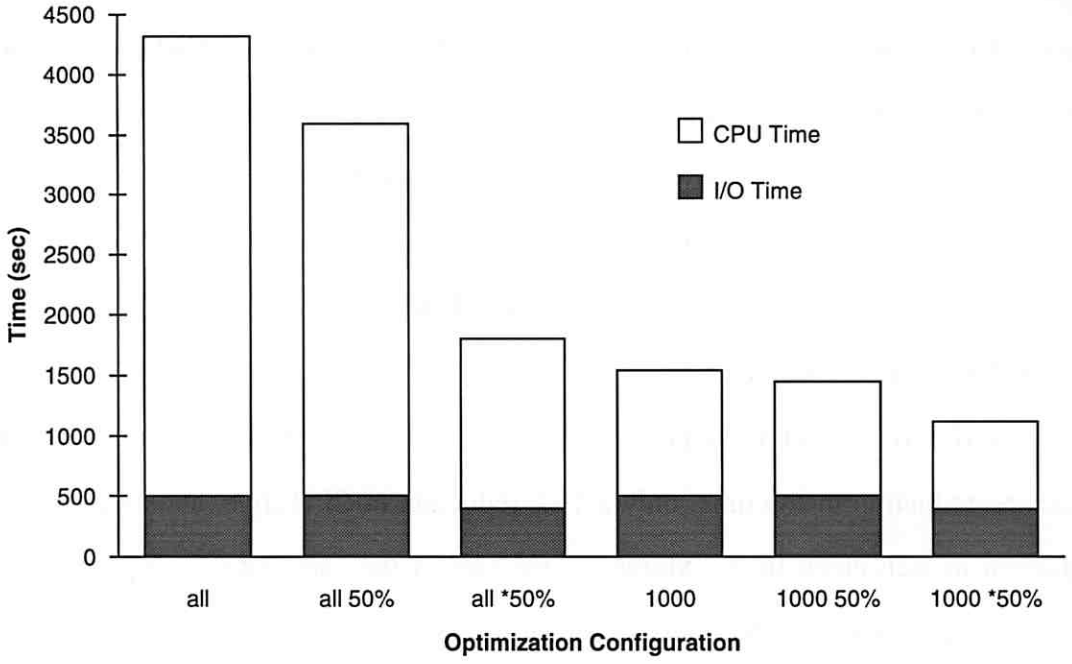


Figure 4.16 Extended optimization wall-clock times for Tip12, Query Set 2

when considering an interactive system. The precision produced at low recall by **1000 50%**, combined with its superior execution performance, indicate that **1000 50%** is the configuration of choice for processing queries like those in **Query Set 1** in an interactive information retrieval system.

Term-elimination was also evaluated for **Query Set 2**. Recall that each query in this query set is created by duplicating a core query, placing one copy inside a passage operator, and combining that with the first copy in a weighted sum. The passage operator presents a dilemma when identifying the optimization candidate terms because it works to localize application of the query within the document, changing the impact of high frequency query terms. When this happens, eliminating high frequency query terms will most likely degrade retrieval effectiveness.

On the other hand, if term-elimination is applied only to the portion of the query outside of the passage operator, the reduction in I/O is certain to be insignificant—any term outside of the passage operator that is eliminated will still appear inside the passage operator and

its inverted list will still be read. The optimization, therefore, was applied both ways. Figure 4.16 shows the execution performance of **Query Set 2** on **Tip12** using various optimization configurations, where a star (*) indicates that term-elimination was applied *inside* the passage operator as well as outside, e.g., **all *50%** is the baseline plus 50% term-elimination applied both inside and outside of passage operators. Note that in **all *50%**, twice as many terms are eliminated as in **all 50%**, since twice as much of the query is considered for term-elimination.

As predicted, term-elimination applied only outside of the passage operator (**all 50%**) yields no reduction in I/O time, only a 19% reduction in CPU time, and an overall 17% reduction in wall-clock time. Moreover, the size of the candidate document set is not reduced at all. Applying 50% term-elimination inside the passage operator (**all *50%**), however, reduces I/O time by 21%, CPU time by 63%, wall-clock time by 58%, and the size of the candidate document set by 44%. While the 21% reduction in I/O time is better than the 0% reduction obtained in **1000**, the overall improvement is inferior. **1000** reduces the candidate document set size by 93%, CPU time by 73%, and wall-clock time by 64%. Again, combining the two optimizations yields the best overall improvement. **1000 *50%** reduces I/O time by 21%, CPU time by 81%, and wall-clock time by 74%. The reduction in the candidate document set size is the same as in **1000**.

Table 4.13 and Figure 4.17 show the retrieval effectiveness for the various optimized versions of **Query Set 2** on **Tip12**. Unlike the results seen for **Query Set 1**, term-elimination in **Query Set 2** leads to a deterioration in precision at most recall levels. The deterioration is even worse when term-elimination is applied inside the passage operator (the starred versions). Returning to the point considered earlier regarding the removal of evidence from a query, the behavior observed in **Query Set 2** suggests that the user's information need is better expressed in these queries and removing evidence will produce the expected degradation in retrieval effectiveness.

Table 4.13 Precision at standard recall pts for Tip12, Query Set 2, optimized

	Precision (% change) – 50 queries					
Recall	**all**	**all 50%**	**all *50%**	**1000**	**1000 50%**	**1000 *50%**
0	89.4	89.3 (−0.1)	87.5 (−2.1)	89.4 (+0.0)	89.3 (−0.1)	87.5 (−2.1)
10	73.8	73.7 (−0.2)	72.4 (−1.9)	73.8 (−0.0)	73.6 (−0.2)	72.4 (−1.9)
20	64.3	63.9 (−0.6)	63.6 (−1.1)	64.2 (−0.1)	63.9 (−0.6)	63.6 (−1.1)
30	56.6	56.3 (−0.4)	55.4 (−2.0)	56.5 (−0.0)	56.3 (−0.4)	55.3 (−2.1)
40	49.6	49.7 (+0.1)	48.0 (−3.3)	49.6 (−0.0)	49.7 (+0.1)	48.0 (−3.4)
50	43.6	43.8 (+0.4)	41.6 (−4.8)	43.5 (−0.3)	43.6 (+0.0)	41.4 (−5.1)
60	36.9	36.4 (−1.4)	34.2 (−7.2)	36.4 (−1.2)	35.9 (−2.6)	34.1 (−7.5)
70	30.1	30.0 (−0.6)	28.8 (−4.2)	29.5 (−2.1)	29.3 (−2.7)	28.5 (−5.2)
80	24.7	24.8 (+0.1)	23.5 (−4.9)	24.0 (−3.2)	24.0 (−3.1)	22.7 (−8.1)
90	16.5	16.5 (+0.1)	15.5 (−6.1)	15.4 (−6.7)	15.3 (−7.0)	14.7(−10.8)
100	2.3	2.4 (+4.2)	2.3 (+0.4)	1.4(−37.9)	1.5(−35.8)	1.5(−34.3)
average	44.4	44.2 (−0.2)	43.0 (−3.1)	44.0 (−0.8)	43.9 (−1.1)	42.7 (−3.7)

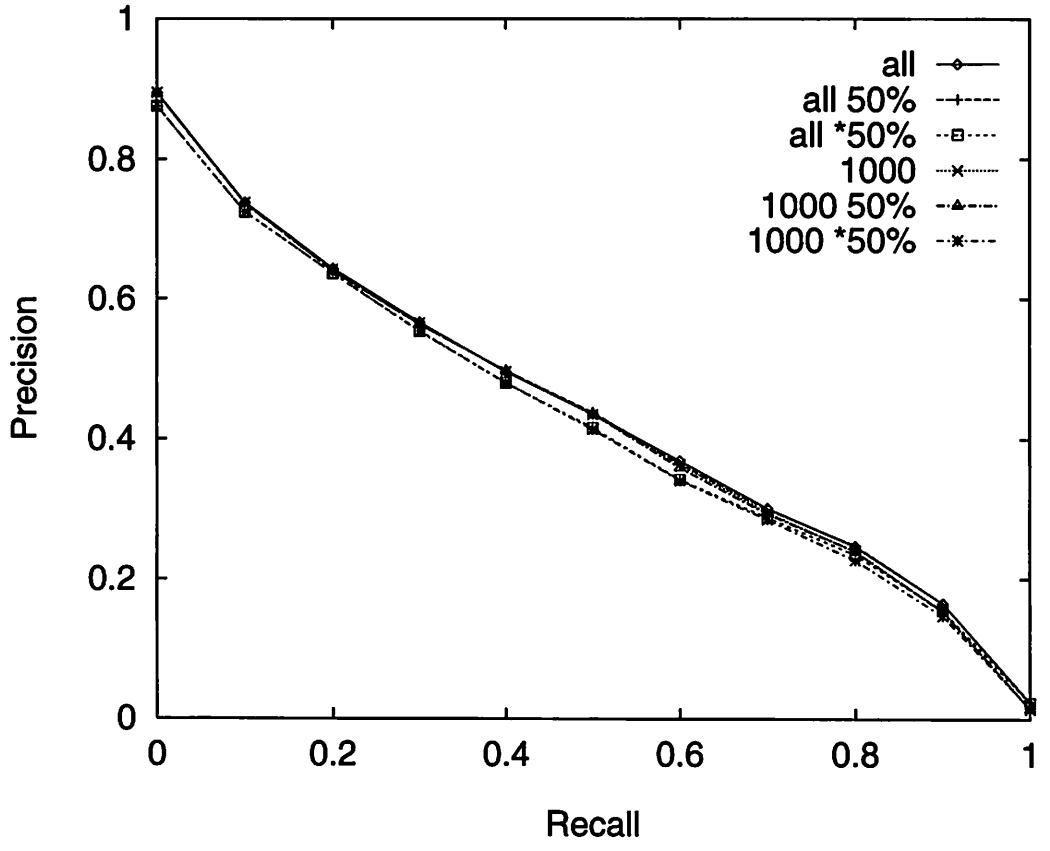


Figure 4.17 Recall-Precision curves for Tip12, Query Set 2, optimized

At lower recall levels, the deterioration in precision is less marked using just the original optimization (**1000**). Adding term-elimination to the original optimization (**1000 *50%**) causes a further deterioration in precision at all recall levels, although precision at low recall in **1000 *50%** is still about the same as in **all *50%**. Compared to term-elimination alone, **1000** produces the same or slightly better precision at low recall and better execution performance, making it the optimization of choice for evaluating queries like those in **Query Set 2** in an interactive IR system. Furthermore, if we are willing to sacrifice some precision, combining the two optimizations in **1000 *50%** produces an additional 28% improvement in wall-clock time over **1000**.

4.6 Short Unstructured Queries

Although we are primarily concerned with improving the execution performance of structured query evaluation, it is worthwhile to investigate how well the optimization techniques described here perform on short, unstructured queries. Recall that the queries in **Query Set 3** are short and flat, containing an average of 8 unique terms combined in a sum or weighted sum operator. Using these short, unstructured queries and the same experimental platform and methodology as in the previous section, the impact of the various optimization techniques was evaluated. Again, the timing results were measured with the GNU time command and the average of 5 runs is reported for each configuration. In all cases the range between the best and worst times recorded for a given configuration was less than 3% of the average for the configuration.

Figure 4.18 shows the execution performance obtained for **Query Set 3** on **Tip12** using a variety of optimization configurations. The trends observed in **Query Set 1** generally hold in this query set as well. Here, 50% term-elimination (**all 50%**) slightly outperforms the original optimization (**1000**), producing a reduction in I/O time of 25%, a reduction in CPU time of 65%, and a reduction in overall wall-clock time of 51%. The reduction in CPU time can again be traced to a reduction in the size of the candidate document set. The baseline

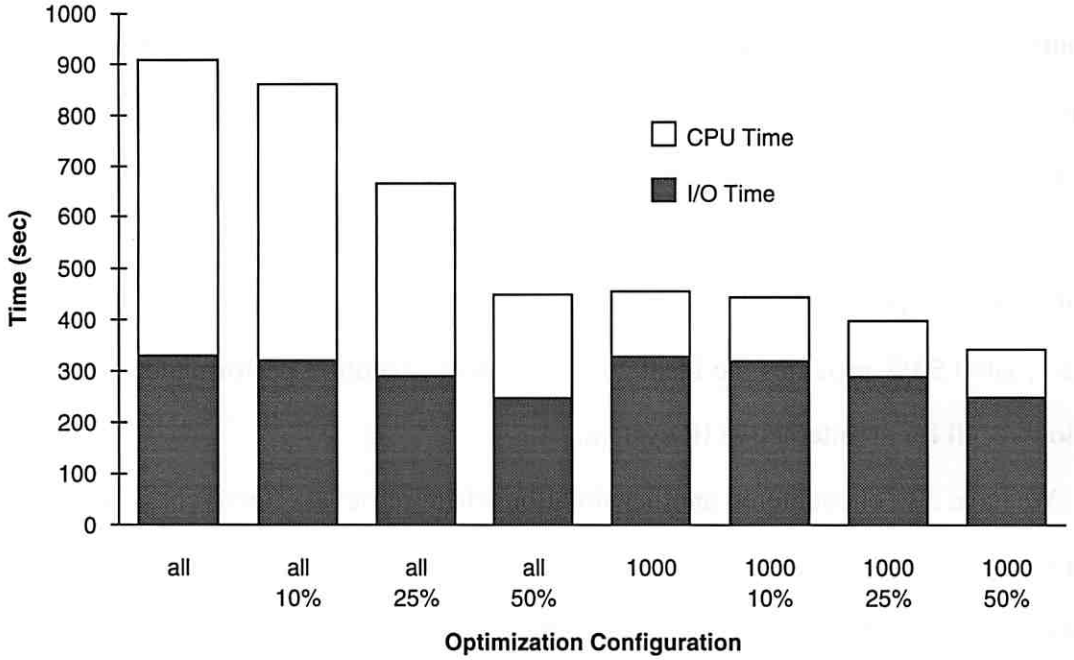


Figure 4.18 Extended optimization wall-clock times for Tip12, Query Set 3

query set evaluates scores for 12,931,770 documents, or an average of 258,635 documents per query (35% of the documents in the collection). 50% term-elimination reduces the number of documents scored by 69%, to 80,210 per query.

The original optimization stays competitive by producing a more substantial reduction in the number of documents scored. In **1000**, 7,561 documents are evaluated per query—a reduction of 97% in the size of the candidate document set. This translates into a reduction in CPU time of 78% and an overall reduction in wall-clock time of 50% (**1000** yields no reduction in I/O time). Combining 50% term-elimination with the original optimization (**1000 50%**) produces the best overall performance, leading to a reduction in I/O time of 25%, a reduction in CPU time of 84%, and a reduction in wall-clock time of 62%. These results show that both the original optimization and term-elimination can produce a substantial execution performance improvement even on relatively short queries.

Given that these queries are so small, we might expect retrieval effectiveness to suffer considerably when the unsafe optimizations are applied. Table 4.14 and Figure 4.19 show

the retrieval effectiveness obtained with **Query Set 3** on **Tip12** for various optimizations. Contrary to expectations, application of the optimizations can actually improve retrieval effectiveness. Term-elimination of up to 50% dramatically improves precision at all levels of recall. The original optimization improves precision at low recall, but displays its characteristic deterioration in precision at high recall levels. Adding 50% term-elimination to the original optimization, however, produces the largest improvement at low recall. Once again, **1000 50%** provides the ideal combination of execution performance and precision at low recall for an interactive IR system.

We have also encountered another situation where retrieval effectiveness has improved via the removal of evidence from the queries. To investigate this phenomenon further, an attempt was made to duplicate this improvement in retrieval effectiveness using a technique other than optimization. The hypothesis here is that the high frequency query terms are polluting the final document scores because they have a greater likelihood of occurring many times within a document. Term-elimination removes this pollution, improving precision. An alternative is to focus the contribution of high frequency query terms by placing them in a passage operator. Using the technique proposed by Callan [11] (the same technique used for **Query Set 2**), a new query set—**Query Set 4**—was created from **Query Set 3** by duplicating each core query, placing one copy inside a passage operator, and combining the passage operator with the first core copy in a weighted sum, where the passage operator's weight is twice the weight of the first core copy.

The retrieval effectiveness obtained with the new query set is shown in Table 4.15 and Figure 4.20. It is compared with the baseline version of **Query Set 3** (**Q3 all**); **Query Set 3** plus 50% term-elimination (**Q3 all 50%**) is shown for reference. The new query set (**Q4 all**) provides a substantial improvement in retrieval effectiveness over the baseline **Query Set 3**, supporting the hypothesis that retrieval effectiveness will improve when the high frequency terms are focused in a passage operator. The improvement in retrieval

Table 4.14 Precision at standard recall pts for Tip12, Query Set 3, optimized

	Precision (% change) – 50 queries					
Recall	**all**	**all 50%**	**all 75%**	**1000**	**1000 50%**	**1000 75%**
0	59.9	64.6 (+7.9)	59.2 (−1.1)	60.6 (+1.2)	65.1 (+8.6)	60.3 (+0.7)
10	33.8	36.6 (+8.5)	33.5 (−0.9)	35.8 (+6.1)	37.3 (+10.6)	34.0 (+0.6)
20	28.6	31.5 (+10.2)	28.4 (−0.9)	30.1 (+5.2)	32.2 (+12.5)	28.2 (−1.5)
30	24.9	27.8 (+11.7)	24.4 (−2.0)	25.0 (+0.5)	27.7 (+11.0)	23.0 (−7.6)
40	22.3	25.2 (+12.9)	21.1 (−5.2)	20.3 (−9.1)	21.7 (−2.6)	17.2(−23.0)
50	18.7	22.0 (+17.2)	18.1 (−3.5)	16.5(−11.8)	17.4 (−6.9)	14.4(−22.9)
60	15.6	18.3 (+17.4)	14.7 (−6.1)	10.6(−32.1)	11.7(−25.0)	9.7(−38.1)
70	12.8	14.6 (+14.0)	10.5(−17.7)	4.9(−61.7)	5.3(−58.3)	4.0(−68.7)
80	9.7	10.8 (+12.0)	7.8(−19.0)	2.4(−75.1)	2.7(−71.8)	2.8(−70.8)
90	5.9	7.2 (+22.0)	5.2(−12.0)	1.1(−81.9)	1.5(−74.5)	1.8(−70.2)
100	0.8	0.9 (+21.5)	0.4(−47.0)	0.1(−82.3)	0.1(−81.4)	0.2(−74.7)
average	21.2	23.6 (+11.4)	20.3 (−4.2)	18.9(−10.9)	20.3 (−4.3)	17.8(−16.1)

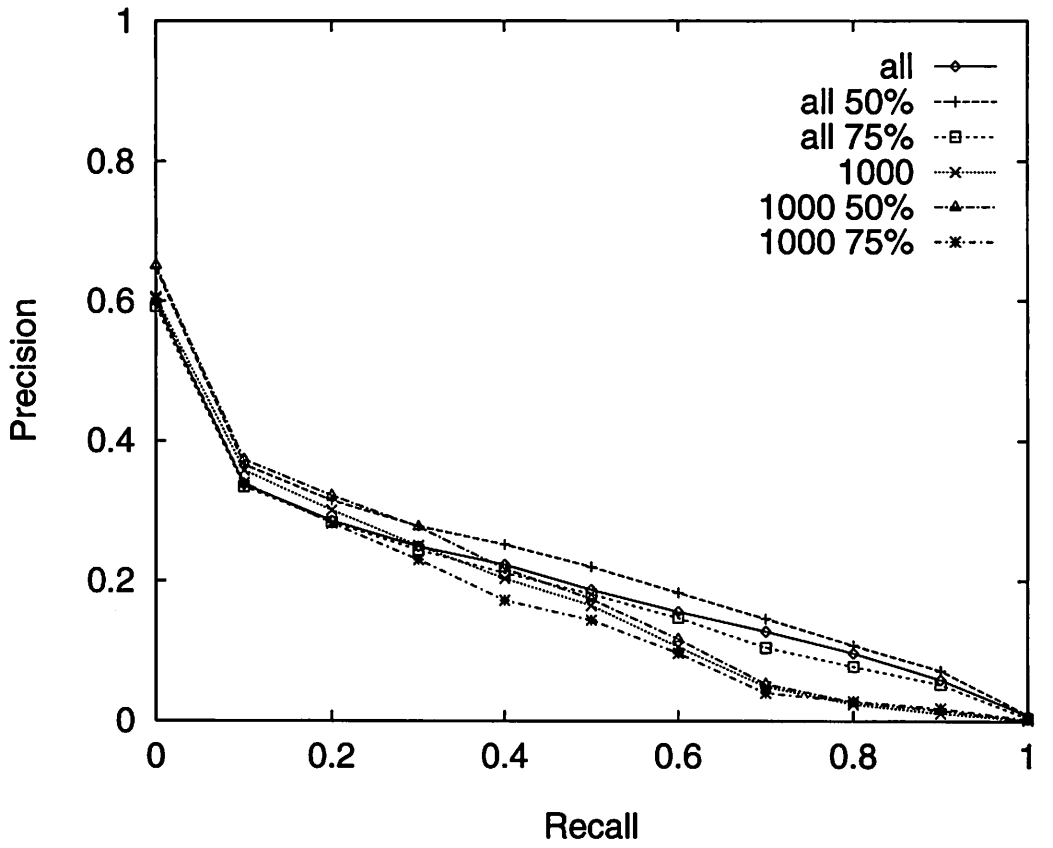


Figure 4.19 Recall-Precision curves for Tip12, Query Set 3, optimized

Table 4.15 Precision at standard recall pts for Tip12, Query Sets 3 and 4

	Precision (% change) – 50 queries		
Recall	**Q3 all**	**Q3 all 50%**	**Q4 all**
0	59.9	64.6 (+7.9)	70.3 (+17.4)
10	33.8	36.6 (+8.5)	37.6 (+11.3)
20	28.6	31.5 (+10.2)	31.8 (+11.0)
30	24.9	27.8 (+11.7)	27.5 (+10.3)
40	22.3	25.2 (+12.9)	24.7 (+10.5)
50	18.7	22.0 (+17.2)	21.2 (+13.0)
60	15.6	18.3 (+17.4)	17.5 (+11.8)
70	12.8	14.6 (+14.0)	14.3 (+11.4)
80	9.7	10.8 (+12.0)	11.0 (+13.5)
90	5.9	7.2 (+22.0)	6.9 (+17.3)
100	0.8	0.9 (+21.5)	0.9 (+16.5)
average	21.2	23.6 (+11.4)	23.9 (+13.1)

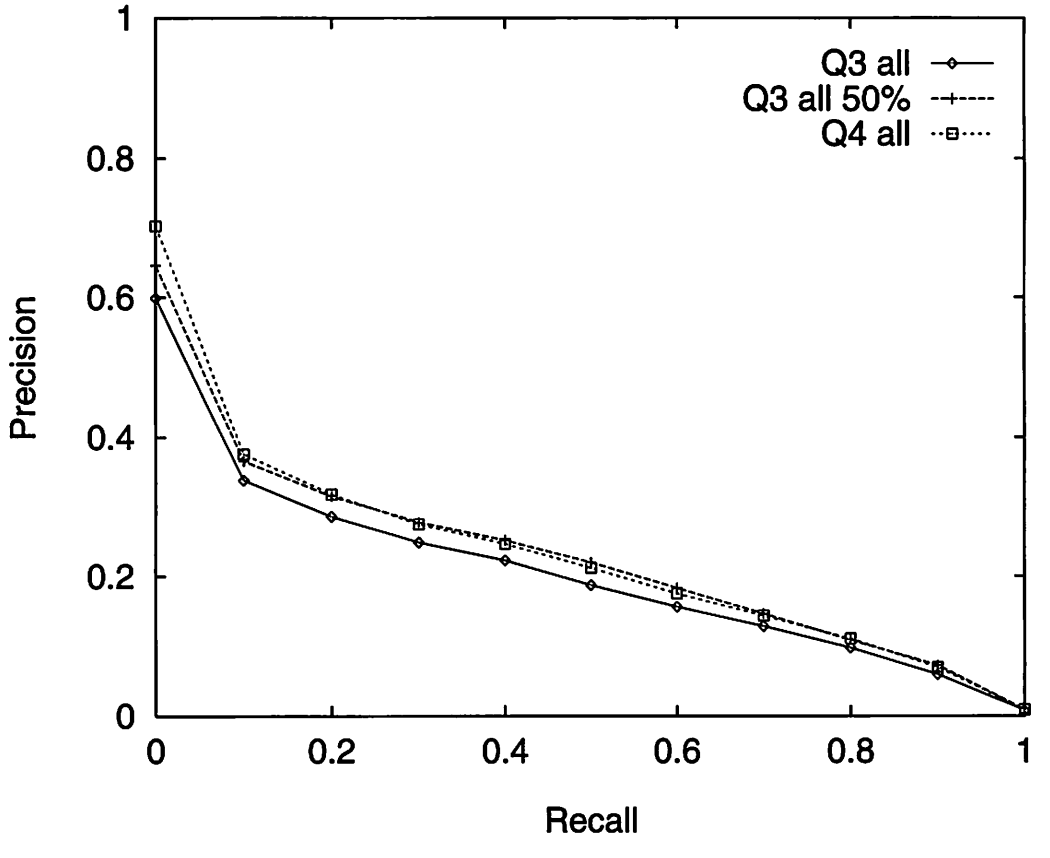


Figure 4.20 Recall-Precision curves for Tip12, Query Sets 3 and 4

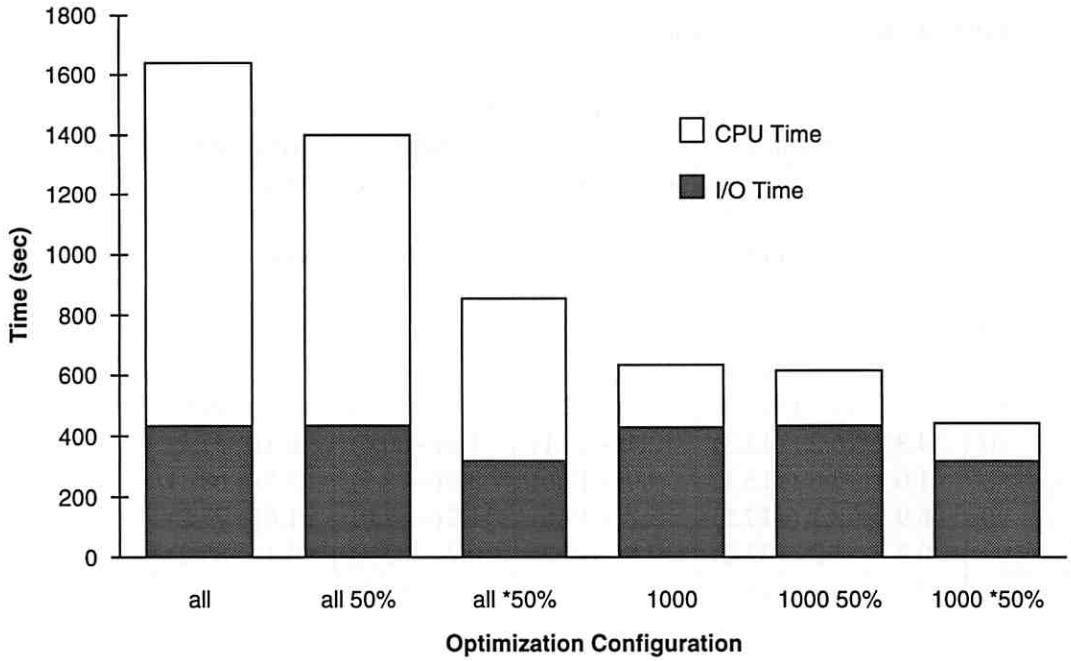


Figure 4.21 Extended optimization wall-clock times for Tip12, Query Set 4

effectiveness in **Q4 all** is similar to that obtained in **Q3 all 50%**—slightly better at low recall, slightly worse at high recall.

Of course, we can apply our optimizations to **Query Set 4** as well. Execution times for various optimization configurations of **Query Set 4** are shown in Figure 4.21 and retrieval effectiveness is shown in Table 4.16 and Figure 4.22. Under application of the various optimization techniques, **Query Set 4** behaves similarly to **Query Set 2** in terms of both execution performance and retrieval effectiveness. The changes in precision seen across various optimization configurations of **Query Set 4** are essentially "magnified" versions of those seen in **Query Set 2** (compare Table 4.16 with Table 4.13). In **Query Set 4**, however, **all 50%** provides a notable improvement in retrieval effectiveness, while **all *50%** is markedly worse. For these relatively short queries (with no structure other than the passage operator), eliminating high frequency terms outside of the passage operator improves precision, while eliminating high frequency terms inside the passage operator worsens precision.

Table 4.16 Precision at standard recall pts for Tip12, Query Set 4, optimized

	Precision (% change) – 50 queries					
Recall	**all**	**all 50%**	**all *50%**	**1000**	**1000 50%**	**1000 *50%**
0	70.3	73.5 (+4.6)	61.4(−12.6)	71.2 (+1.3)	73.4 (+4.4)	63.5 (−9.6)
10	37.6	40.5 (+7.7)	36.3 (−3.5)	37.9 (+0.9)	40.8 (+8.4)	38.2 (+1.7)
20	31.8	34.1 (+7.3)	30.3 (−4.5)	32.9 (+3.6)	34.6 (+9.0)	31.7 (−0.2)
30	27.5	29.7 (+7.9)	26.1 (−5.0)	26.2 (−4.6)	27.9 (+1.4)	25.3 (−7.8)
40	24.7	26.8 (+8.7)	22.5 (−8.8)	20.9(−15.3)	22.3 (−9.8)	19.7(−20.2)
50	21.2	23.6 (+11.4)	18.9(−10.9)	17.5(−17.5)	18.8(−11.4)	15.9(−24.8)
60	17.5	19.5 (+11.8)	15.2(−12.8)	11.2(−35.8)	12.4(−29.0)	10.0(−43.0)
70	14.3	16.2 (+13.3)	12.4(−13.1)	5.6(−60.6)	6.1(−57.2)	4.7(−67.0)
80	11.0	12.6 (+15.1)	9.0(−17.6)	3.0(−72.5)	3.5(−68.1)	2.3(−79.3)
90	6.9	8.1 (+17.5)	5.7(−17.8)	1.2(−82.0)	1.6(−77.2)	1.0(−86.1)
100	0.9	1.2 (+33.2)	0.8(−15.2)	0.2(−83.3)	0.1(−85.2)	0.1(−84.8)
average	23.9	26.0 (+8.5)	21.7 (−9.4)	20.7(−13.5)	21.9 (−8.4)	19.3(−19.4)

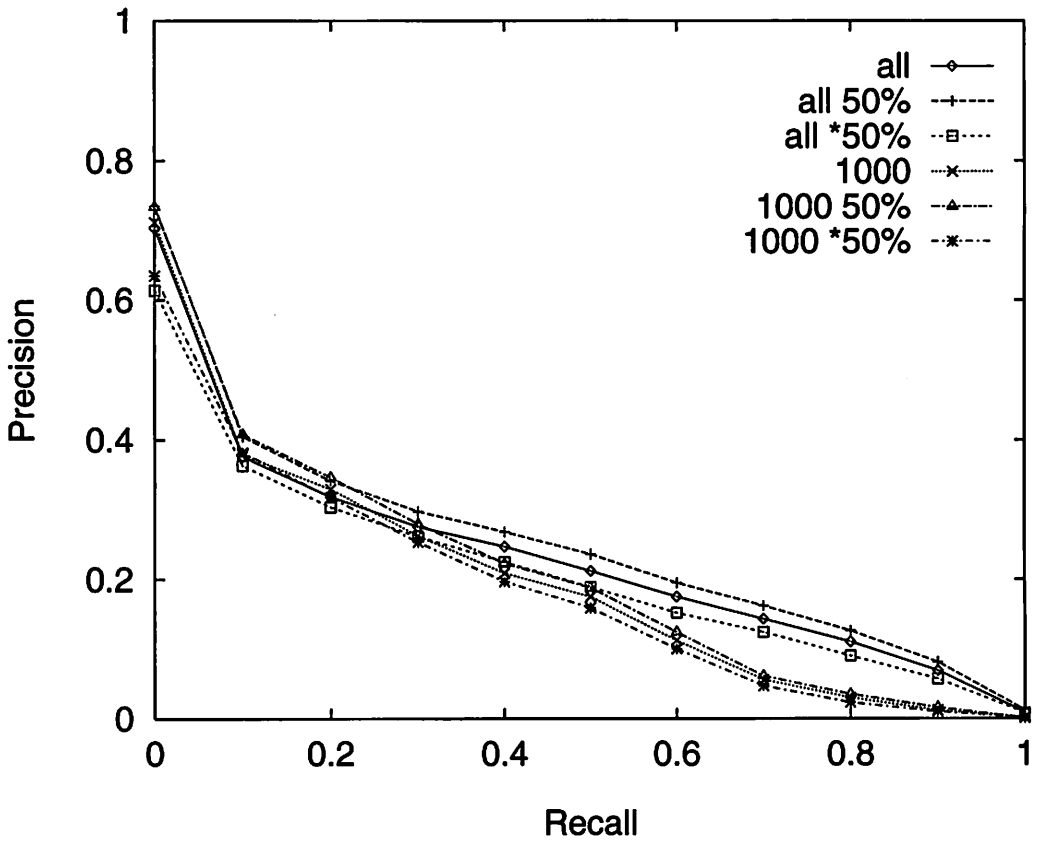


Figure 4.22 Recall-Precision curves for Tip12, Query Set 4, optimized

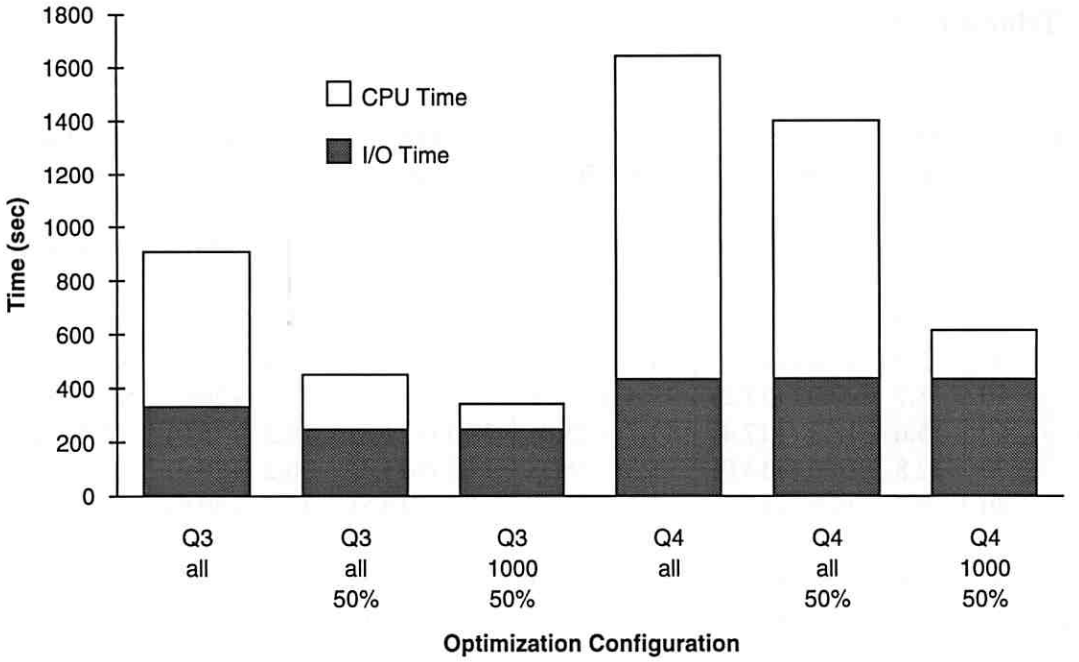


Figure 4.23 Extended optimization wall-clock times for Tip12, Query Sets 3 and 4

Although improving retrieval effectiveness through query modification rather than optimization is perhaps more "theoretically sound," the bottom line is which version gives the best combination of retrieval effectiveness and execution performance. Figure 4.23 compares the execution performance of selected configurations of **Query Sets 3** and **4**, and Table 4.17 and Figure 4.24 compare their retrieval effectiveness using the unoptimized configuration of **Query Set 3** (**Q3 all**) as the baseline. The best execution performance is obtained in **Q3 1000 50%**, while the best overall retrieval effectiveness is obtained in **Q4 all 50%**. The best compromise is achieved by **Q4 1000 50%**, which matches the best precision obtained at low recall and provides the third best overall execution performance. Although **Q3 1000 50%** provides an additional 44% reduction in wall-clock time over **Q4 1000 50%**, the substantially better precision at low recall obtained in **Q4 1000 50%** makes it the better choice for an interactive IR system.

While eight term queries are certainly small compared to the much larger queries in **Query Sets 1** and **2**, novice information retrieval system users are likely to enter even

Table 4.17 Precision at standard recall pts for Tip12, Query Sets 3 and 4, optimized

	Precision (% change) – 50 queries					
Recall	**Q3 all**	**Q3 all 50%**	**Q3 1000 50%**	**Q4 all**	**Q4 all 50%**	**Q4 1000 50%**
0	59.9	64.6 (+7.9)	65.1 (+8.6)	70.3 (+17.4)	73.5 (+22.7)	73.4 (+22.5)
10	33.8	36.6 (+8.5)	37.3 (+10.6)	37.6 (+11.3)	40.5 (+20.0)	40.8 (+20.7)
20	28.6	31.5 (+10.2)	32.2 (+12.5)	31.8 (+11.0)	34.1 (+19.1)	34.6 (+20.9)
30	24.9	27.8 (+11.7)	27.7 (+11.0)	27.5 (+10.3)	29.7 (+19.1)	27.9 (+11.9)
40	22.3	25.2 (+12.9)	21.7 (−2.6)	24.7 (+10.5)	26.8 (+20.1)	22.3 (−0.3)
50	18.7	22.0 (+17.2)	17.4 (−6.9)	21.2 (+13.0)	23.6 (+26.0)	18.8 (+0.2)
60	15.6	18.3 (+17.4)	11.7(−25.0)	17.5 (+11.8)	19.5 (+25.0)	12.4(−20.6)
70	12.8	14.6 (+14.0)	5.3(−58.3)	14.3 (+11.4)	16.2 (+26.2)	6.1(−52.3)
80	9.7	10.8 (+12.0)	2.7(−71.8)	11.0 (+13.5)	12.6 (+30.6)	3.5(−63.8)
90	5.9	7.2 (+22.0)	1.5(−74.5)	6.9 (+17.3)	8.1 (+37.8)	1.6(−73.3)
100	0.8	0.9 (+21.5)	0.1(−81.4)	0.9 (+16.5)	1.2 (+55.2)	0.1(−82.8)
average	21.2	23.6 (+11.4)	20.3 (−4.3)	23.9 (+13.1)	26.0 (+22.7)	21.9 (+3.6)

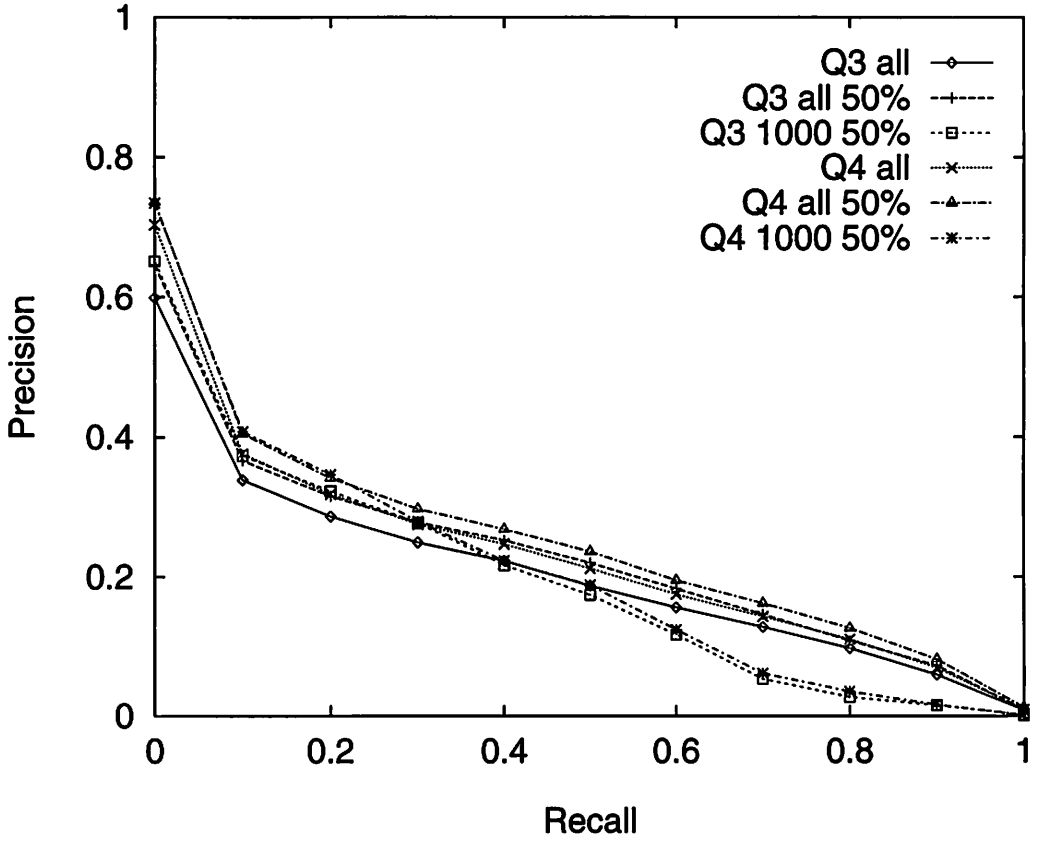


Figure 4.24 Recall-Precision curves for Tip12, Query Sets 3 and 4, optimized

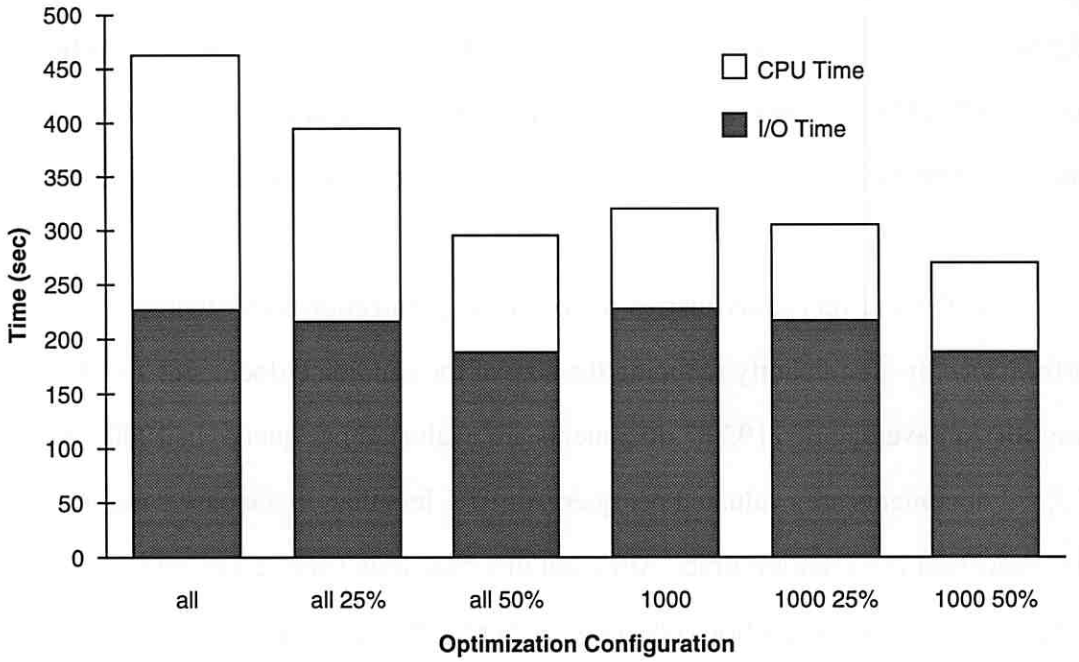


Figure 4.25 Extended optimization wall-clock times for Tip12, Query Set 5

smaller queries. For completeness, we evaluated our optimization techniques on a fifth query set, **Query Set 5**, generated from the title fields of *TIPSTER topics 51–100*. Each query is simply a sum of the terms in the corresponding title field, with an average of 3 terms per query. Measurements were made using the same platform and experimental methodology as above.

Execution performance results are shown in Figure 4.25. With 25% term elimination, I/O time is reduced by 5%, CPU time is reduced by 24%, and overall time is reduced by 15%. These improvements are modest because only queries with at least 4 terms are affected by the optimization. Only 22 of the 50 queries comprise 4 or more terms. 50% term elimination causes a 17% reduction in I/O time, a 55% reduction in CPU time, and an overall wall-clock time reduction of 36%. All queries with more than 1 term are affected by 50% term elimination, and all but 4 of the 50 queries consist of more than 1 term.

When our original optimization is applied (**1000**), I/O is unchanged, CPU time is reduced by 61%, and overall time is reduced by 31%. All queries are affected by this

optimization, regardless of the number of query terms. Adding term elimination to our original optimization yields the same reduction in I/O as that obtained with term elimination alone. **1000 25%** provides a 63% reduction in CPU time and a 34% reduction in overall time, and **1000 50%** provides a 65% reduction in CPU time and a 42% reduction in overall time.

Even with these very short queries, our original optimization is able to improve execution performance by significantly reducing the size of the candidate document set. In the base case (**all**), an average of 119,787 documents are evaluated per query. In **1000**, an average of 3,172 documents are evaluated per query, or 97% less than in the base case, leading to a 31% reduction in execution time. Although this execution time reduction is noticeable, it is less than the 50+% reductions observed earlier for the larger query sets. Note, however, that the base query set here takes relatively little time to evaluate in the first place, such that relevance judgement processing becomes a substantial component of the overall cost. Relevance judgement processing accounts for 53% of the total time in **all** and 82% of the total time in **1000**. If this time is factored out in both cases, the reduction in total time provided by **1000** is actually 73%.

The impact on retrieval effectiveness when optimizing **Query Set 5** is shown in Table 4.18 and Figure 4.26. Unlike the results obtained earlier, optimization never causes retrieval effectiveness to improve. 25% term elimination incurs the smallest degradation in retrieval effectiveness, although less than half of the queries in the query set are affected by the optimization. **1000** provides the next best level of retrieval effectiveness, including good precision up to 30% recall. Adding term elimination to **1000** causes retrieval effectiveness to deteriorate further with relatively little payback in terms of execution performance. With its good precision at low recall and 31% reduction in execution time (73% if relevance judgement processing is excluded), **1000** offers the best combination of retrieval effectiveness and execution performance.

Table 4.18 Precision at standard recall pts for Tip12, Query Set 5, optimized

	Precision (% change) – 50 queries					
Recall	**all**	**all 25%**	**all 50%**	**1000**	**1000 25%**	**1000 50%**
0	66.6	66.6 (+0.0)	58.8(−11.7)	66.2 (−0.6)	64.8 (−2.7)	58.8(−11.6)
10	43.5	42.4 (−2.6)	36.4(−16.2)	42.6 (−2.1)	41.5 (−4.6)	35.2(−19.0)
20	37.0	36.1 (−2.5)	30.2(−18.3)	35.8 (−3.3)	35.2 (−4.9)	28.6(−22.6)
30	32.6	32.0 (−1.9)	25.3(−22.4)	29.4 (−9.7)	28.6(−12.1)	22.2(−31.8)
40	28.6	27.9 (−2.3)	22.3(−22.1)	21.9(−23.2)	21.7(−24.1)	17.0(−40.4)
50	24.5	24.0 (−1.9)	18.9(−22.8)	18.3(−25.2)	18.2(−25.6)	13.5(−44.8)
60	20.9	20.4 (−2.3)	16.0(−23.6)	11.3(−46.0)	11.3(−45.9)	7.3(−65.0)
70	15.6	15.2 (−3.1)	12.0(−23.6)	5.3(−66.1)	5.3(−65.9)	3.8(−75.9)
80	12.1	11.8 (−2.7)	9.0(−25.9)	3.1(−74.3)	3.1(−74.3)	2.1(−82.3)
90	7.8	7.4 (−4.8)	5.5(−29.3)	1.7(−77.6)	1.7(−77.6)	1.4(−82.4)
100	0.9	0.9 (−0.8)	0.5(−41.1)	0.2(−78.7)	0.2(−78.7)	0.0(−95.1)
average	26.4	25.9 (−1.9)	21.4(−19.0)	21.4(−18.7)	21.1(−20.1)	17.3(−34.5)

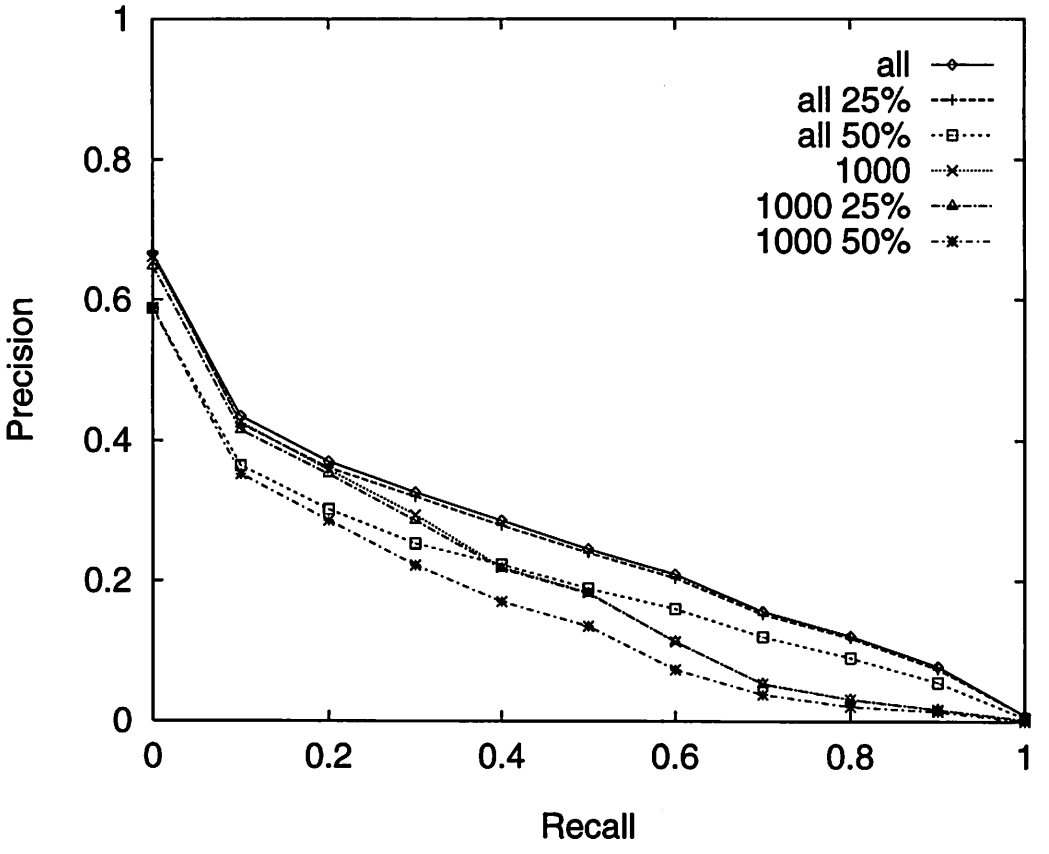


Figure 4.26 Recall-Precision curves for Tip12, Query Set 5, optimized

Table 4.19 Wall-clock time summary for Tip12 (seconds)

Optimization Configuration	Query Set 1			Query Set 2			Query Set 3			Query Set 4			Query Set 5		
	I/O	CPU	Total	I/O	CPU	Total	I/O	CPU	Total	I/O	CPU	Total	I/O	CPU	Total
all	509	1684	2193	500	3817	4317	331	578	909	433	1208	1641	227	236	463
all 10%	477	1353	1830	n/a	n/a	n/a	321	540	861	n/a	n/a	n/a	n/a	n/a	n/a
all 25%	449	932	1381	n/a	n/a	n/a	290	375	665	n/a	n/a	n/a	216	179	395
all 50%	418	542	960	505	3088	3593	248	202	450	435	965	1400	188	107	295
all *50%	n/a	n/a	n/a	397	1404	1801	n/a	n/a	n/a	318	536	854	n/a	n/a	n/a
1000	508	428	936	499	1043	1542	329	127	456	428	205	633	228	92	320
1000 10%	477	398	875	n/a	n/a	n/a	320	124	444	n/a	n/a	n/a	n/a	n/a	n/a
1000 25%	448	356	804	n/a	n/a	n/a	289	109	398	n/a	n/a	n/a	217	88	305
1000 50%	419	302	721	504	945	1449	249	94	343	434	181	615	188	82	270
1000 *50%	n/a	n/a	n/a	394	722	1116	n/a	n/a	n/a	317	124	441	n/a	n/a	n/a

4.7 Conclusions

In this chapter we have examined a variety of techniques for improving the execution speed of structured queries, including both safe and unsafe optimizations. The safe techniques explored here generally depend on the inverted file implementation satisfying certain functionality requirements. It was hypothesized that two inverted list features in particular would lead to reductions in I/O and execution time. First, separating term weights from proximity lists would free belief operators from the overhead of accessing proximity lists and result in better execution performance. This was shown to be the case in Section 4.4.4.1, where **Query Set 3** (which contains no proximity operators) experienced a 20% reduction in I/O time and a 7% reduction in wall clock time when evaluated using an inverted file implementation that provides the requisite functionality.

This meager reduction in wall-clock time, however, is barely sufficient to justify the optimization, especially when we consider the following. In Section 4.6 it was shown that the retrieval effectiveness obtained with **Query Set 3** could be significantly enhanced through the use of the passage operator. This is consistent with other results [11, 21] which show that using a passage operator, or proximity operators in general, can improve retrieval effectiveness. This suggests that the kind of query that will benefit from this safe optimization is one that should be augmented with proximity operators to improve its retrieval effectiveness, making the safe optimization no longer applicable. It might be better, therefore, to simplify the implementation and store term weights and proximity lists together, since the proximity lists should be used in evaluating the query anyway. A final answer to this question requires more work in the area of proper query formulation, and is beyond the scope of this investigation.

The second feature that was hypothesized to be useful is the ability to skip portions of an inverted list. This comes into play when evaluating an intersection style operator where one of the terms in the intersection is infrequent and can be used to constrain the intersection process. While skipping opportunities were found in **Query Sets 1** and **2** (see

Section 4.6), they only amounted to an average of 25 to 46 long list objects per query. The small improvement obtained in I/O was generally overshadowed by the extra CPU costs incurred in the more complex split list implementation. However, given that the more complex inverted list implementation generally pays for itself, it is still worthwhile to support this optimization. There will inevitably be situations where an intersection can be significantly constrained and this optimization will produce a large payback.

We explored a third safe optimization that eliminates redundant evaluation of constructed concepts during the final query evaluation phase. As with the other safe optimizations, the benefit derived from this optimization depends on the makeup of the query. A greater improvement was obtained in **Query Set 2** than in **Query set 1** because **Query Set 2** has a larger proportion of proximity operators. In this case, the improvement is significant—we measured a 13% reduction in total wall-clock time. This optimization is independent of the inverted file implementation and is always worthwhile. It can also be extended as follows. The temporary inverted lists built during the preprocessing phase can be cached across queries, potentially eliminating the need to evaluate the same constructed concept in the future. Furthermore, frequently accessed temporary inverted lists can eventually be written to the inverted file, treating the corresponding constructed concepts as if they were terms. To fully support this, the document indexing system must be modified to recognize the saved constructed concepts and appropriately update the corresponding inverted lists. The net result is automatic indexing of frequently used phrases.

The unsafe optimization introduced in this thesis generated much more rewarding results. Our experimental results show that for highly structured queries (e.g., **Query Sets 1** and **2**), our optimization will reduce query processing time by over 50% with no noticeable degradation in precision until better than 70% recall. The basic hypothesis here was that the candidate document set could be significantly constrained with minimal effort, which in turn would produce a significant savings in query evaluation execution time. Using the heuristics developed in Section 4.2.2 and the inverted list implementation described in

Section 4.3, we were able to efficiently reduce the size of the candidate document set by over 90%. This was shown to produce a significant savings in CPU time and a substantial improvement in overall execution performance, leading to the acceptance of the hypothesis.

We also applied our optimization technique to short, unstructured queries. The results in this case were very rewarding as well. On queries comprising an average of 8 terms, the candidate document set reduction was still better than 90%, leading to a 50% reduction in wall-clock time. The impact on retrieval effectiveness was somewhat more noticeable. For example, in Table 4.14, our top 1000 optimization (**1000**) produces worse precision than the unoptimized version after 40% recall. Note, however, that the precision at 30% recall is 25%. Since this result was generated using a query set with an average of 328 relevant documents per query, 40% recall is over 400 documents down in the ranked listing. In an interactive system, this level of retrieval effectiveness will still be quite acceptable.

Our optimization technique mainly attacks the CPU costs of evaluating a query. We considered additional techniques specifically aimed at reducing I/O. In particular, term-elimination was evaluated. This technique was originally introduced in the context of the vector-space retrieval model [10]. We have described a novel application of term-elimination to structured queries, including an adjustable selectivity based on estimated contribution to final document score. 50% term-elimination alone was found to reduce I/O time by 17% to 25%. Moreover, in queries without passage operators, it produced reductions in wall-clock time comparable to our top 1000 optimization. The best execution performance, however, was obtained by combining our original top 1000 optimization with 50% term-elimination, which reduced wall-clock time by an additional 8% to 38% over either optimization alone.

Perhaps surprisingly, term-elimination of up to 50% generally caused precision to improve in query sets without passage operators. The improvement was quite dramatic in 8 term, unstructured queries. We suspected that obtaining such an improvement via the removal of evidence is actually indicative of a problem in either the query formulation

or the retrieval model. This was pursued further by augmenting the 8 term, unstructured queries with passage operators. The new queries were able to better the improvement in retrieval effectiveness produced by the optimization, confirming our suspicions. In the very short, 3 term unstructured queries, optimization never caused an improvement in retrieval effectiveness, suggesting that when queries are sufficiently short, all terms in the query must be considered to achieve adequate recall.

In the case of passage operators, term-elimination provides little improvement in execution performance unless applied within the passage operator. Doing so generally caused retrieval effectiveness to suffer. Our top 1000 optimization was much more robust with respect to the passage operator, providing at least a 61% reduction in wall-clock time with no impact on precision at low recall. Applying 50% term-elimination just on the outside of the passage operator was found to actually improve precision markedly on short queries (**Query Set 4**). This was less true on larger queries (**Query Set 2**). Adding 50% term-elimination just on the outside of the passage operator to the top 1000 optimization improved its precision as well.

In general, the best combination of execution performance and precision at low recall was found by combining 50% term-elimination with the top 1000 optimization. In an interactive system, this optimization is unlikely to impact the user's perception of the effectiveness of the system. However, the reduction in query processing time by more than half is certain to impact the user's perception of the usefulness of the system. Moreover, the level of aggressiveness for both of these optimizations is tunable at run-time, allowing the user to control the tradeoff between speed and precision.

The execution performance improvements obtained with the optimization technique introduced here compare favorably with results reported by others. The optimization proposed by Buckley and Lewit [10] produces reductions in CPU time ranging from 37% to 84%, where the greatest savings are obtained when only the top document in the final ranking is guaranteed to be correct. These results were obtained using the relatively small

CACM and INSPEC document collections, which contain 3,204 and 12,684 documents, respectively. Buckley and Lewit do not present standardized retrieval effectiveness results, making it difficult to fully assess the effect of this optimization on retrieval effectiveness. Their optimization is similar to the term-elimination optimization evaluated here, however, and we obtained very good retrieval effectiveness with term-elimination on sufficiently large queries. It is notable that this optimization does not work well on very short queries.

The pruning optimization proposed by Harman and Candela [41] produces a 62% reduction in candidate set size and a 29% reduction in total search time with essentially no reduction in average precision. These results were obtained using the Cranfield test collection of 1,400 abstracts. On the CACM, INSPEC, and 1,033 document MEDLARS collection, Smith's [79] list pruning optimization reduces query evaluation time 11% to 51%. The effect of list pruning on retrieval effectiveness is similar to that obtained with our optimization—precision at low recall is unchanged while precision at higher recall levels degrades. Smith does not report final candidate document set sizes after pruning.

Using the 2 GB TIPSTER document collection, Moffat and Zobel [58] reduce the size of the candidate document set nearly 99% by fixing the number of document score accumulators. This in turn produces a 55% reduction in CPU time during query evaluation, with no loss in average retrieval effectiveness. Using just the *Wall Street Journal* documents in the TIPSTER collection (532 MB), Persin's [65] optimization reduces the candidate document set size by 98%, yielding an 80% reduction in query evaluation CPU time with no loss in average retrieval effectiveness.

Turtle and Flood [89] use a 254 MB document collection to evaluate their *max-score* optimization in terms of the number of postings and intermediate document scores read or written. With document-at-a-time evaluation, max-score reduces the total number of read and write operations 25% to 74%. The amount of savings depends on the number of final document scores guaranteed to be correct and whether or not term occurrence location information is used. The largest savings (74%) occurs when term occurrence locations are

used and the top 20 documents in the final ranking are guaranteed. With term-at-a time evaluation, the savings range from 53% to 85%, with the greatest savings occurring under the same conditions as before. While the savings in number of postings and intermediate document scores read or written is clear, it is difficult to infer the real savings in terms of CPU time, I/O time, and total execution time. Charging the same cost to each posting read is misleading if the postings are compressed, and elimination of individual posting reads does not necessarily translate into I/O savings—if the amount of data skipped within an inverted list does not exceed the file access granularity, then no I/O savings are realized. Admittedly, elimination of the need to process these postings should result in comparable CPU savings.

The main characteristic that distinguishes our optimization technique from the ones discussed above is the way in which the candidate document set is populated. Our optimization uses information created and stored at indexing time (the top document lists) to establish the candidate document set before final evaluation of the query. The other optimizations above establish the candidate document set as query evaluation proceeds, using either upper bound document scores or candidate document set insertion and modification thresholds to control the population process. The use of upper bound document scores has the advantage that guarantees can be made about the final document ranking. Adapting a similar strategy to a structured query environment is not straightforward, since each query operator will require a different upper bound calculation. In particular, the not operator is troublesome and may require that upper *and* lower bounds (i.e., a range) be calculated. This is an interesting area for future work. The bottom line here, however, is that our optimization technique produces competitive reductions in execution time, causes no noticeable degradation in precision at low recall, and works for structured queries.

One topic not directly addressed in the performance evaluation above is the impact of our implementation and optimization techniques on main memory usage. Optimization techniques that reduce the size of the candidate document set provide an immediate main

memory savings opportunity for systems that allocate a document score accumulator for every member of the candidate document set [58, 61]. If the size of the candidate document set is reduced, the number of document score accumulators can be reduced accordingly. This main memory savings, however, is advantageous only to systems that evaluate queries term-at-a-time. For systems that evaluate queries document-at-a-time (such as INQUERY), the number of document score accumulators can be reduced trivially to the number of final document scores that will be returned to the user. For example, assume that the user will be shown n final document scores. The first n final document scores calculated are saved in the accumulators. Then, as each remaining final document score is calculated, it replaces one of the saved scores only if it exceeds the smallest currently saved score. Otherwise it is discarded. This scheme has an additional computational cost to update and maintain the n final document scores, although the additional cost is similar to that incurred by a system that uses term-at-a-time evaluation and must locate and update document scores in a constrained accumulator space. Furthermore, a sorted search structure (e.g., binary search tree) allows the document-at-a-time implementation to bypass the final sorting step required by the term-at-a-time implementation to present the final document scores in ranked order.

Another potential main memory savings is provided by our Mneme-based inverted file implementation. In this implementation, even if a term appears more than once in a query, only *one* copy of the inverted list for that term will be read into main memory. Each occurrence of the term in the query tree will refer to the term's inverted list using the object identifier for the Mneme object that contains the inverted list (or the head of the inverted list). Mneme always resolves multiple references for the same object to a single copy of that object in main memory. This is a basic but central concept in a persistent object store. A simpler inverted file implementation might very well retrieve as many copies of an inverted list as there are occurrences of the term in the query.

Finally, while our implementation and experimental evaluation have been carried out in the context of an inference network-based retrieval model, the techniques described here are

generally applicable to any statistical retrieval model that supports structured queries. As these retrieval models are applied to larger and larger document collections, optimization techniques such as these will become ever more crucial to the success of these systems.

CHAPTER 5

CONCLUSIONS

The goal of this thesis was to provide solutions to the challenges in information retrieval created by large, dynamic document collections, and to lay the foundation for a comprehensive information management system that incorporates sophisticated document management. We have addressed a number of issues related to indexing and modifying a document collection in Chapter 3, including the design, implementation, and evaluation of an inverted file architecture based on a persistent object store. Our design and implementation was guided by the following principles:

- Localize sort and insert operations.
- Build intermediate results in main memory.
- Minimize I/O.
- Favor sequential I/O over random I/O.
- Minimize access to the existing inverted file.

We described an indexing system that provides a bulk indexing rate of 530 MB per hour, provides an incremental indexing rate of 265 MB per hour, and supports a fully dynamic document collection. These results lead us to conclude that our indexing system design principles are sound and a persistent object store provides an effective solution for inverted file management.

In Chapter 4, we addressed the problem of providing fast evaluation of structured queries in information retrieval and presented a new, unsafe optimization technique that

returns a 50% reduction in execution time with no noticeable degradation in retrieval effectiveness. We explored a variety of other optimization techniques and presented a comprehensive evaluation of the tradeoffs between optimization aggressiveness, speed, and retrieval effectiveness.

Our safe optimizations dealt primarily with the overheads imposed by storing and processing term occurrence locations. Storing term occurrence locations increases the size of the inverted file by 78% and adds an additional 6% to the total indexing time. Proximity operators that use term occurrence locations require an extra preprocessing step during query evaluation, although this was measured to be only 12% of the total query evaluation time for queries with 36% to 54% of their terms appearing inside proximity operators. Moreover, the temporary inverted lists built during the preprocessing step can be cached for future use.

The additional indexing and retrieval overhead imposed by proximity operators appears to be small in terms of computation. Attempts at further reducing these overheads through the use of more complex inverted list structures were generally ineffective. Compared to a simple linked list implementation for long inverted lists, a directory based split list implementation served only to increase indexing time with little payback during retrieval for the queries that we considered. The greatest cost of storing term occurrence locations is a near doubling in the size of the inverted file. However, given the generally better retrieval effectiveness obtained through the use of operators that require term occurrence locations (e.g., the passage operator), the additional overheads of storing and processing term occurrence locations are well worthwhile.

Our results lead us to conclude that:

- Safe optimizations are generally ineffective, although they provide contingency solutions.

- Candidate set reduction is a general, robust optimization for structured queries.

- Term-elimination is almost as good.
- A combination of candidate set reduction and term-elimination is best.

The contributions of this thesis work are primarily practical in nature, with implications for information retrieval system implementation. The main contributions are summarized below:

- Implementation and evaluation of a fast, scalable indexing system.
- Design and implementation of an inverted file management architecture using "off-the-shelf" data management technology, providing opportunities for all aspects of an information retrieval system to benefit from traditional database management features, such as buffer management and efficient low-level storage management.
- Development and evaluation of an incremental indexing strategy enabled by the above architecture.
- Ground work for a comprehensive information management system where information retrieval is a full-featured component.
- Development and evaluation of a structured query optimization that reduces execution time by over 50% with no noticeable impact on retrieval effectiveness.
- An investigation of the impact of high frequency query terms in short, unstructured queries and how to handle them for best retrieval effectiveness and execution performance.

5.1 Future work

A significant contribution of this thesis work was the integration of INQUERY and Mneme. The product of this integration is an information retrieval system positioned to

explore a number of new issues in scalable, multi-user information retrieval. We consider possible future directions here.

5.1.1 Small updates

The solution presented in Chapter 3 for supporting document additions works best when the new batch of documents is relatively large. If small batch updates are more common, the solution must be extended. The first possible extension is straightforward. If the partial inverted lists for the entire batch of new documents can be buffered in a single batch buffer (Figure 3.3), the Merger can be bypassed and the partial inverted lists can be handed directly to the Inverted File Manager for addition to the existing inverted file. In this case, there will only be a single temporary file block, obviating the merge step. This also eliminates the I/O that would otherwise be required to write the temporary file block after the batch is parsed and read the temporary file block during the merge.

If the batch buffer is large, it may take a while before a document presented to the system for indexing actually becomes visible in the inverted file. To accommodate environments where documents must be indexed and available immediately, the system can be extended to take advantage of the following observation: once a document has been parsed and flushed to the batch buffer, partial inverted lists for that document are available in the batch buffer. The batch buffer in main memory is essentially an extension of the existing inverted file on disk. Documents that have been parsed and added to the batch buffer can be made visible during query processing by modifying the Inverted File Manager to check the batch buffer for relevant inverted list information.

For example, suppose a query is being processed involving the term "cat". The Inverted File Manager will first retrieve the inverted list for "cat" from the existing inverted file. When the end of that list is reached, the batch buffer is checked to see if it contains a partial inverted list for "cat". If it does, query processing continues with this additional information. Since the partial inverted list in the batch buffer will eventually be appended

to the existing inverted list on disk, the sequential processing of inverted list contents during query processing transitions smoothly from the on-disk version to the batch buffer version.

This solution assumes that document indexing and query processing can run simultaneously and share main memory. A reasonable way to support this configuration is with *threads*. A document indexing thread handles requests to add new documents to the document collection. The new documents are parsed, inverted, and their partial inverted lists are added to the batch buffer. When the batch buffer is full, it is flushed directly to the existing inverted file, as described in the first extension above. Meanwhile, a query evaluation thread handles requests to process queries and interacts with the Inverted File Manager as usual. The main issues are coordination of the threads with suitable concurrency control mechanisms and proper interaction with the Inverted File Manager.

5.1.2 Multi-user support

Although the current implementation does support concurrency control, recovery, and transactions, these issues have yet to be explored in a systematic fashion. In particular, an information retrieval system offers opportunities to relax the consistency and coherency constraints typically imposed by traditional data processing applications. For example, since query results are actually estimates based on the information available in the inverted file, as long as a result is internally consistent, it is reasonable to return this result to the user. If the underlying inverted file management system can identify (and retrieve) the last consistent version of the database before the current write transaction began, then during query evaluation the retrieval engine need never block on inverted list data that is locked for update [69]. This scheme must be tailored to suit the particular visibility requirements of the system (i.e., how soon new documents must be available in the system) and the rate of new document additions.

Other issues worth pursuing involve log file management for transactions and recovery. The kind of updates that an inverted file will experience are relatively constrained, and

include append operations to existing objects, creation of new objects, and rewriting of the majority of the objects in the database. The first two activities occur during document additions and are the most common. The last activity occurs during the occasional purge of deleted documents. This characterization of modification behavior combined with a predictable occurrence rate can be used to customize transaction and recovery log management and improve performance.

5.1.3 Hardware based optimization

The final area for future work that we propose here is based on our relative lack of success at significantly reducing I/O costs during query evaluation. In fact, since our optimization technique provides such a significant reduction in CPU time, it shifts the query evaluation cost model from being CPU bound to being I/O bound. Our attempts at attacking these I/O costs with sophisticated data structures and algorithms were minimally successful.

This suggests that lower-level hardware support must be pursued to obtain true scalability. Techniques such as disk striping [14], parallel processing, and distributed computing need to be investigated more thoroughly. Previous work has been done in this area [83], but not in the context of the sophisticated retrieval models considered here. Using the architecture that has been built for this thesis work, substantial insights can be gained into the efficacy of these hardware techniques.

Improvements in I/O speed, either through the approaches suggested above or through advances in magnetic disk speed, will shift the dominant cost back to CPU time and increase the importance of optimization techniques such as ours. Reductions in I/O speed, on the other hand, decrease the benefit derived from these optimization techniques. In particular, optical disk environments introduce a substantially different query evaluation cost model and require a reevaluation of inverted file implementation and query optimization decisions. While an investigation of these issues is beyond the scope of this dissertation, we note that our inverted file architecture is well suited to such an investigation. The ability to customize

the inverted file implementation and control the low level storage and access mechanisms will greatly facilitate an exploration of appropriate file organizations and optimizations for optical disk environments.

BIBLIOGRAPHY

[1] Belkin, N. J. and Croft, W. B. Retrieval techniques. In M. E. Williams, editor, *Annual Review of Information Science and Technology*, volume 22, pages 109–145. Elsevier Science Publishers, New York, 1987.

[2] Bell, T. C., Moffat, A., Nevill-Manning, C. G., Witten, I. H., and Zobel, J. Data compression in full-text retrieval systems. *J. Amer. Soc. Inf. Sci.*, 44(9):508–531, Oct. 1993.

[3] Biliris, A. An efficient database storage structure for large dynamic objects. In *Proc. 8th IEEE Inter. Conf. on Data Engineering*, pages 301–308, Tempe, AZ, Feb. 1992.

[4] Biliris, A. The performance of three database storage structures for managing large objects. In *Proc. of the ACM SIGMOD Inter. Conf. on Management of Data*, pages 276–285, San Diego, CA, June 1992.

[5] Blair, D. C. An extended relational document retrieval model. *Inf. Process. & Mgmnt.*, 24(3):349–371, 1988.

[6] Bookstein, A., Klein, S. T., and Ziff, D. A. A systematic approach to compressing a full-text retrieval system. *Inf. Process. & Mgmnt.*, 28(6):795–806, 1992.

[7] Brown, E. W. Fast evaluation of structured queries for information retrieval. In *Proc. of the 18th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 30–38, Seattle, WA, July 1995.

[8] Brown, E. W., Callan, J. P., and Croft, W. B. Fast incremental indexing for full-text information retrieval. In *Proc. of the 20th Inter. Conf. on Very Large Databases (VLDB)*, pages 192–202, Santiago, Sept. 1994.

[9] Brown, E. W., Callan, J. P., Croft, W. B., and Moss, J. E. B. Supporting full-text information retrieval with a persistent object store. In *Proc. of the 4th Inter. Conf. on Extending Database Technology (EDBT)*, pages 365–378, Cambridge, UK, Mar. 1994.

[10] Buckley, C. and Lewit, A. F. Optimization of inverted vector searches. In *Proc. of the 8th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 97–110, June 1985.

[11] Callan, J. P. Passage-level evidence in document retrieval. In *Proc. of the 17th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 302–310, Dublin, July 1994.

[12] Callan, J. P., Croft, W. B., and Harding, S. M. The INQUERY retrieval system. In *Proc. of the 3rd Inter. Conf. on Database and Expert Systems Applications*, Sept. 1992.

[13] Carey, M. J., DeWitt, D. J., Richardson, J. E., and Shekita, E. J. Object and file management in the EXODUS extensible database system. In *Proc. of the 12th Inter. Conf. on Very Large Databases (VLDB)*, pages 91–100, Kyoto, Aug. 1986.

[14] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Paterson, D. A. RAID: High-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, June 1994.

[15] Comer, D. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[16] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction fo Algorithms*. The MIT Press/McGraw-Hill Book Company, Cambridge, MA, 1990.

[17] Crawford, R. G. The relational model in information retrieval. *J. Amer. Soc. Inf. Sci.*, 32(1):51–64, 1981.

[18] Crawford, R. G. and MacLeod, I. A. A relational approach to modular information retrieval systems design. In *Proc. of the 41st Conf. of the American Society for Information Science*, 1978.

[19] Croft, W. B. Document representation in probabilistic models of information retrieval. *J. Amer. Soc. Inf. Sci.*, 32(6):451–457, Nov. 1981.

[20] Croft, W. B. Experiments with representation in a document retrieval system. *Inf. Tech.: Res. Dev.*, 2(1):1–21, 1983.

[21] Croft, W. B., Cook, R., and Wilder, D. Providing government information on the internet: Experiences with THOMAS. In *Proc. of the 2nd Annual Conf. on the Theory and Practice of Digital Libraries (Digital Libraries '95)*, Austin, TX, June 1995.

[22] Croft, W. B. and Harper, D. J. Using probabilistic models of document retrieval without relevance information. *J. Documentation*, 35(4):285–295, Dec. 1979.

[23] Croft, W. B. and Savino, P. Implementing ranking strategies using text signatures. *ACM Trans. Office Inf. Syst.*, 6(1):42–62, Jan. 1988.

[24] Cruden, A., Adams, A. D., Irwin, C. H., and Waters, S. A. *Complete concordance to the Holy Scriptures of the Old and New Testaments*. Holt, Rinehart and Winston, New York, 1949.

[25] Cutting, D. and Pedersen, J. Optimizations for dynamic inverted index maintenance. In *Proc. of the 13th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 405–411, 1990.

[26] DeFazio, S., Daoud, A., Smith, L. A., Srinivasan, J., Croft, B., and Callan, J. Integrating IR and RDBMS using cooperative indexing. In *Proc. of the 18th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 84–92, Seattle, WA, July 1995.

[27] Deogun, J. S. and Raghavan, V. V. Integration of information retrieval and database management systems. *Inf. Process. & Mgmnt.*, 24(3):303–313, 1988.

[28] Elmasri, R. and Navathe, S. B. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1989.

[29] Faloutsos, C. Access methods for text. *ACM Comput. Surv.*, 17:50–74, 1985.

[30] Faloutsos, C. and Christodoulakis, S. Signature files: An access method for documents and its analytical performance evaluation. *ACM Trans. Office Inf. Syst.*, 2(4):267–288, Oct. 1984.

[31] Faloutsos, C. and Jagadish, H. V. Hybrid index organizations for text databases. In *Proc. of the 3rd Inter. Conf. on Extending Database Technology (EDBT)*, pages 310–327, 1992.

[32] Faloutsos, C. and Jagadish, H. V. On b-tree indices for skewed distributions. In *Proc. of the 18th Inter. Conf. on Very Large Databases (VLDB)*, pages 363–374, Vancouver, 1992.

[33] Fox, C. A stop list for general text. *SIGIR Forum*, 24(1-2):19–35, 1990.

[34] Fox, C. Lexical analysis and stoplists. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, chapter 7, pages 102–130. Prentice Hall, Englewood Cliffs, NJ, 1992.

[35] Fox, E. A. and Lee, W. C. FAST-INV: A fast algorithm for building large inverted files. Technical Report TR-91-10, VPI&SU Department of Computer Science, Blacksburg, VA, March 1991.

[36] Frakes, W. B. Stemming algorithms. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, chapter 8, pages 131–160. Prentice Hall, Englewood Cliffs, NJ, 1992.

[37] Graefe, G. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, June 1993.

[38] Grossman, D. A. and Driscoll, J. R. Structuring text within a relational system. In *Proc. of the 3rd Inter. Conf. on Database and Expert Systems Applications*, pages 72–77, Sept. 1992.

[39] D. Harman, editor. *The Second Text REtrieval Conference (TREC-2)*, Gaithersburg, MD, 1994. National Institute of Standards and Technology Special Publication 500-215.

[40] D. Harman, editor. *The Third Text REtrieval Conference (TREC-3)*, Gaithersburg, MD, 1995. National Institute of Standards and Technology Special Publication 500-225.

[41] Harman, D. and Candela, G. Retrieving records from a gigabyte of text on a mini-computer using statistical ranking. *J. Amer. Soc. Inf. Sci.*, 41(8):581–589, Dec. 1990.

[42] Harman, D., Fox, E., Baeza-Yates, R., and Lee, W. Inverted files. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, chapter 3, pages 28–43. Prentice Hall, Englewood Cliffs, NJ, 1992.

[43] Heaps, H. S. *Information Retrieval, Computational and Theoretical Aspects.* Academic Press, Inc., New York, 1978.

[44] Hessel, A. *A History of Libraries.* The Scarecrow Press, New Brunswick, NJ, 1955. Translated by Reuben Peiss.

[45] Jannink, J. Implementing deletion in B^+-Trees. *SIGMOD RECORD*, 24(1):33–38, Mar. 1995.

[46] Jing, Y. and Croft, W. B. An association thesaurus for information retrieval. In *Proc. of RIAO 94 Conf.*, pages 146–160, New York, Oct. 1994.

[47] Knaus, D. and Schäuble, P. Effective and efficient retrieval from large and dynamic document collections. In Harman [39], pages 163–170.

[48] Kohlenberger, J. R. *The NRSV concordance unabridged: including the apocryphal / deuterocanonical books.* Zondervan, Grand Rapids, MI, 1991.

[49] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. The ObjectStore database system. *Commun. ACM*, 34(10):50–63, Oct. 1991.

[50] Lehman, T. J. and Lindsay, B. G. The starburst long field manager. In *Proc. of the 15th Inter. Conf. on Very Large Databases (VLDB)*, pages 375–383, Amsterdam, Aug. 1989.

[51] Linoff, G. and Stanfill, C. Compression of indexes with full positional information in very large text databases. In *Proc. of the 16th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 88–95, Pittsburgh, PA, June 1993.

[52] Lucarella, D. A document retrieval system based on nearest neighbour searching. *J. Inf. Sci.*, 14(1):25–33, 1988.

[53] Lynch, C. A. and Stonebraker, M. Extended user-defined indexing with application to textual databases. In *Proc. of the 14th Inter. Conf. on Very Large Databases (VLDB)*, pages 306–317, 1988.

[54] MacLeod, I. A. SEQUEL as a language for document retrieval. *J. Amer. Soc. Inf. Sci.*, 30(5):243–249, 1979.

[55] MacLeod, I. A. and Crawford, R. G. Document retrieval as a database application. *Inf. Tech.: Res. Dev.*, 2(1):43–60, 1983.

[56] Maron, M. E. and Kuhns, J. L. On relevance, probabilistic indexing and information retrieval. *J. ACM*, 7(3):216–244, July 1960.

[57] Moffat, A. and Zobel, J. Compression and fast indexing for multi-gigabyte text databases. *Australian Comput. J.*, 26(1):1–9, February 1994.

[58] Moffat, A. and Zobel, J. Fast ranking in limited space. In *Proc. 10th IEEE Inter. Conf. on Data Engineering*, pages 428–437, Feb. 1994.

[59] Moffat, A. and Zobel, J. Self-indexing inverted files. In *Proc. Australasian Database Conf.*, Christchurch, New Zealand, January 1994.

[60] Moffat, A. and Zobel, J. Self-indexing inverted files for fast text retrieval. Technical Report 94/2, Collaborative Information Technology Research Institute, Department of Computer Science, Royal Melbourne Institute of Technology, Australia, Feb. 1994.

[61] Moffat, A., Zobel, J., and Sacks-Davis, R. Memory efficient ranking. *Inf. Process. & Mgmnt.*, to appear.

[62] Moss, J. E. B. Design of the Mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, Apr. 1990.

[63] Panagopoulos, G. and Faloutsos, C. Bit-sliced signature files for very large text databases on a parallel machine architecture. In *Proc. of the 4th Inter. Conf. on Extending Database Technology (EDBT)*, pages 379–392, Cambridge, UK, Mar. 1994.

[64] Perry, S. A. and Willett, P. A review of the use of inverted files for best match searching in information retrieval systems. *J. Inf. Sci.*, 6(2-3):59–66, 1983.

[65] Persin, M. Document filtering for fast ranking. In *Proc. of the 17th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 339–348, Dublin, July 1994.

[66] Pfeifer, U. and Fuhr, N. Efficient processing of vague queries using a data stream approach. In *Proc. of the 18th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 189–197, Seattle, WA, July 1995.

[67] Putz, S. Using a relational database for an inverted text index. Technical Report SSL-91-20, Xerox Palo Alto Research Center, Jan. 1991.

[68] Rajashekar, T. B. and Croft, W. B. Combining automatic and manual index representations in probabilistic retrieval. *J. Amer. Soc. Inf. Sci.*, 46(4):272–283, May 1995.

[69] Ridgway, J. Personal communication. Mneme design team, University of Massachusetts, 1995.

[70] Robertson, S. E. The probability ranking principle in IR. *J. Documentation*, 33(4):294–304, 1977.

[71] Robertson, S. E. and Sparck Jones, K. Relevance weighting of search terms. *J. Amer. Soc. Inf. Sci.*, 27(3):129–146, May 1976.

[72] Salton, G., Fox, E. A., and Wu, H. Extended boolean information retrieval. *Commun. ACM*, 26(11):1022–1036, Nov. 1983.

[73] Salton, G. and McGill, M. J. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

[74] Saxton, L. V. and Raghavan, V. V. Design of an integrated information retrieval/database management system. *IEEE Trans. Know. Data Eng.*, 2(2):210–219, June 1990.

[75] Schäuble, P. SPIDER: A multiuser information retrieval system for semistructured and dynamic data. In *Proc. of the 16th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 318–327, Pittsburgh, June 1993.

[76] Shoens, K., Tomasic, A., and Garcia-Molina, H. Synthetic workload performance analysis of incremental updates. In *Proc. of the 17th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, Dublin, July 1994.

[77] Singhal, V., Kakkad, S. V., and Wilson, P. R. Texas, an efficient, portable persistent store. In *Proc. of the 5th Inter. Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy, Sept. 1992.

[78] Smeaton, A. F. and van Rijsbergen, C. J. The nearest neighbour problem in information retrieval. An algorithm using upperbounds. In *Proc. of the 4th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 83–87, Oakland, CA, 1981.

[79] Smith, M. E. Aspects of the p-norm model of information retrieval: Syntactic query generation, efficiency, and theoretical properties. Technical Report TR 90-1128 (Ph.D. Thesis), Department of Computer Science, Cornell University, May 1990.

[80] Standard Performance Evaluation Corporation (SPEC). SPARCstation 10 model 51. *SPEC Newsletter*, 5(2), June 1993.

[81] Standard Performance Evaluation Corporation (SPEC). DEC 3000 model 600. *SPEC Newsletter*, 6(1), June 1994.

[82] Stanfill, C. Parallel computing for information retrieval: Recent developments. Technical Report TR-69 DR88-1, Thinking Machines Corporation, Cambridge, MA, Jan. 1988.

[83] Tomasic, A. and Garcia-Molina, H. Caching and database scaling in distributed shared-nothing information retrieval systems. In *Proc. of the ACM SIGMOD Inter. Conf. on Management of Data*, pages 129–138, Washington, D.C., May 1993.

[84] Tomasic, A., Garcia-Molina, H., and Shoens, K. Incremental updates of inverted lists for text document retrieval. In *Proc. of the ACM SIGMOD Inter. Conf. on Management of Data*, pages 289–300, Minneapolis, MN, May 1994.

[85] Turtle, H. R. Natural language vs. boolean query evaluation: A comparison of retrieval performance. In *Proc. of the 17th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 212–220, Dublin, July 1994.

[86] Turtle, H. R. and Croft, W. B. Inference networks for document retrieval. In *Proc. of the 13th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 1–24, Sept. 1990.

[87] Turtle, H. R. and Croft, W. B. Efficent probabilistic inference for text retrieval. In *Proc. of RIAO 91 Conf.*, pages 644–661, Barcelona, Apr. 1991.

[88] Turtle, H. R. and Croft, W. B. Evaluation of an inference network-based retrieval model. *ACM Trans. Inf. Syst.*, 9(3):187–222, July 1991.

[89] Turtle, H. R. and Flood, J. Query evaluation: Strategies and optimizations. *Inf. Process. & Mgmnt.*, to appear.

[90] Witten, I. H., Moffat, A., and Bell, T. C. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Van Nostrand Reinhold, New York, 1994.

[91] Wolfram, D. Applying informetric characteristics of databases to IR system file design, Part I: informetric models. *Inf. Process. & Mgmnt.*, 28(1):121–133, 1992.

[92] Wolfram, D. Applying informetric characteristics of databases to IR system file design, Part II: simulation comparisons. *Inf. Process. & Mgmnt.*, 28(1):135–151, 1992.

[93] Wong, W. Y. P. and Lee, D. L. Implementations of partial document ranking using inverted files. *Inf. Process. & Mgmnt.*, 29(5):647–669, 1993.

[94] Zipf, G. K. *Human Behavior and the Principle of Least Effort.* Addison-Wesley Press, 1949.

[95] Zobel, J. and Moffat, A. Adding compression to a full-text retrieval system. In *Proc. 15th Australian Computer Science Conf.*, pages 1077–1089, Hobart, Australia, Jan. 1992.

[96] Zobel, J., Moffat, A., and Ramamohanarao, K. Inverted files versus signature files for text indexing. Technical Report CITRI/TR-95-5, Collaborative Information Technology Research Institute, Department of Computer Science, Royal Melbourne Institute of Technology, Australia, July 1995.

[97] Zobel, J., Moffat, A., and Sacks-Davis, R. An efficient indexing technique for full-text database systems. In *Proc. of the 18th Inter. Conf. on Very Large Databases (VLDB)*, pages 352–362, Vancouver, 1992.