INFORMATION RETRIEVAL USING SIGNATURES IN AN OFFICE ENVIRONMENT

By

Glenn N. Paulley

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada

ISBN 0-315-63229-1

INFORMATION RETRIEVAL USING SIGNATURES IN AN OFFICE ENVIRONMENT

BY

GLENN N. PAULLEY

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Table of Contents

	Page
Abstract	iv
Acknowledgements	v
Dedication	vi
Introduction	1
Chapter 1: Characteristics of the Model Office Environment	4
1.1 Schools of the Pine Creek School Division	7
1.2 Requirements of a Document Retrieval System	8
1.3 Choice of a Signature File Organization	11
Chapter 2: A Survey of Access Methods for Text	13
2.1 Terminology	13
2.2 Document Characteristics and Retrieval Performance	17
2.3 Text Retrieval in an Office Environment	22
2.4 Automatic Term Extraction	24
2.5 Access Methods for Text	27
2.5.1 Full Text Scanning	27
2.5.2 Inversion	28
2.5.3 Multi-attribute Hashing	34
2.6 Signature Representation of Text	46
2.7 An Outline of Some Signature Methods	53

Page

Chapter 3: The Doubly-Compressed Bit Slices (DCBS) Method 72

3.1 Bit-Sliced Signature Files (BSSF) 73

3.2 The Compressed Bit Slices (CBS) Method 75

3.3 The Doubly-Compressed Bit Slices (DCBS) Method 78

3.3.1 Mathematical Analysis of DCBS 80

3.3.2 The "No False Drops" (NFD) Modification 90

Chapter 4: Partial Match Query Application 93

4.1 Introduction 94

4.2 Logical Structure of Documents in PMQ 96

4.3 User Interface Component 99

4.4 Indexing Functional Interface 104

4.5 Retrieval Functional Interface 111

4.6 PMQ Implementation of DCBS 119

4.6.1 File Structure 119

4.6.2 Indexing Algorithm 125

4.6.3 Search Algorithm 127

4.7 Possible Enhancements to PMQ 130

Chapter 5: Performance Analysis and Conclusions 133

5.1 Model Office Document Set Characteristics 133

5.2 Experimental Results 139

5.3 Conclusions and Future Research Directions 149

5.4 Summary 151

Page

Appendix A: Abbreviations of Signature Methods 153

Appendix B: Enable Non-Disclosure Agreement 154

References 155

Abstract

An implementation of an information retrieval application for an office environment is described. The application uses a relatively new signature method developed by Faloutsos (Doubly-Compressed Bit Slices, or DCBS). DCBS was originally intended for use with Optical Disk technology, but may also be used with standard magnetic media. The implementation of the retrieval application is discussed from a number of perspectives, including a comparison of known signature techniques and implementation trade-offs. Characteristics of the office environment are discussed, and comparisons are made between theoretical search results and actual results in the model office. An evaluation of the use of this application in the model office shows a high degree of success in meeting the requirements of its users. Finally, ideas for future enhancements are presented along with topics for future research.

Acknowledgements

The author is indebted to Mr. Ron McFadyen of the Computer Science Department, University of Manitoba, for his effort spent on my behalf during my Master's program.

I would like to express my appreciation to Dr. David Scuse of the Computer Science Department, and Dr. Tom Berry of the Applied Mathematics Department for acting as members of my Thesis committee.

Thanks is also due to Mrs. Irene Mellick and the Great-West Life Assurance Company[1] for their enthusiastic support, and for permission to use the computer resources of Great-West Life for the publishing of this thesis.

I would like to thank Mr. Ron Brown, Principal of the Langruth Elementary School, Langruth, Manitoba, for his support and patience during development of the retrieval application, and for providing his time and School resources for the model office.

Acknowledgments and thanks are extended to my Uncle, Mr. Michael Achtemichuk of Yorkton, Saskatchewan, for the Polish-Ukrainian-English translation examples in Section 2.2.

[1] This work was sponsored partially by an Continuing Education grant from the Great-West Life Assurance Company, Winnipeg, Manitoba, Canada.

Dedication

To my wife, Leslie

This thesis is dedicated to my wife Leslie for her support, encouragement, and love over what has been a rather long three years of full-time employment and part-time study.

Introduction.

Text retrieval systems facilitate the search of a document file (database) for documents that are relevant to a particular query. In most implementations, document relevance is measured by the presence in the document of the words, or partial words, in the query (term detection). This thesis discusses the implementation of an text retrieval application for an office environment.

Computer systems that support a document retrieval capability have many applications. These applications include:

a) Office document management systems, supporting the various aspects of document creation and retrieval in an office environment.

b) Computerized libraries [Sal83]. A major benefit in placing library materials on a computer system is that there is no need to preserve old, deteriorating documents on paper media.

c) Storage and retrieval of newspaper and magazine articles, offering a reference system on a broad range of subjects.

d) Electronic encyclopedias.

e) Online dictionaries (e.g. the Oxford English Dictionary).

f) Medical Information Systems.

g) Geographic databases.

h) Consumer catalogues.

i) Online reference materials.

j) Statutes, Court rulings, and other government documents.

The handling of free-format document text (as opposed to attributes) is typically characterized by techniques for text analysis and automatic indexing [Sal83]. These techniques attempt to provide a fast and accurate retrieval method with minimal cost. The cost associated with such methods includes additional media storage over and above the storage necessary for the documents themselves, and large amounts of computer resources (typically CPU) necessary in searching.

We typically use the terms index or inverted list to identify the data structure used in the mapping of keys to record numbers in a file [Har88] and the terms indexing or inversion to identify the process of building the index. Though not all text retrieval methods require the use of such a structure, it is common to use the terms index and indexing to describe the building of any external data structure used to speed retrieval. We follow this convention in this thesis, and we will be more specific when there is a danger of being ambiguous.

Text retrieval methods which use an index include the full document-term indexing methods STAIRS from IBM and Infodata's Inquire/Text. Methods not requiring an index include multi-attribute hashing [Rot74] (also, good surveys of these methods are presented by Rivest [Riv76] and Knuth [Knu73]), hardware-assisted methods (such as Memex Information Systems' Textract [Mem89]) and the use of signature files [Fal85a].

In this thesis we will limit the discussion to that of text retrieval methods that support electronic office filing. The application (named Partial Match Query, abbreviated PMQ for the rest of this

thesis) presented here was developed for the Pine Creek School Division (located in Central Manitoba) to provide retrieval capability for word processing files kept on each School's IBM-PC[1] compatible microcomputer. The requirements of the application, among other factors, led to the use of signature techniques, as their qualities were highly desirable (good retrieval speed, with (typically) low space overhead).

The thesis begins with an introduction to the office environment within the Schools of the Pine Creek School Division. Chapter 2 introduces problems in text management in general, and discusses various information retrieval methods that have been discussed in the literature. These include signature methods of document representation. Chapter 3 discusses the Doubly-Compressed Bit Slices (DCBS) signature method in detail. DCBS is the bit-sliced technique that was selected for this application. Interestingly, the data structures used with DCBS are also suitable for Optical Disk storage. Chapter 4 describes the implementation of the Partial Match Query application, its data structures (based on DCBS), and the implementation trade-offs that were made during its development. We end Chapter 4 with some ideas for extending the application. Chapter 5 provides a discussion of the system from various perspectives (usability, retrieval speed, overhead) along with a comparison of theoretical results versus actual results from the model office. In conclusion, we present an overview of the results and a discussion of topics for future research.

[1] IBM is a registered trademark of International Business Machines Corporation, Inc.

1. Characteristics of the Model Office Environment.

In this chapter, we define what is meant by the term "document", and define its properties, to give a framework for the discussion in the following chapters. We discuss characteristics of office documents in general, and in Section 1.1 we describe the model office. The sections that follow deal with the requirements of the text retrieval application, and the software and hardware environments in the model office. Section 1.3 discusses our proposed solution.

It is perhaps best to begin discussing the problems and environment of a "paper-less" office in terms of an office that is not automated.

In every office there are large amounts of memoranda, documents, and papers. We will use the term document to identify these items, and the term document set to identify all of the documents in the office. In general, each document has properties which define its purpose and contents. Each document typically contains two types of data: attributes, which are normally fixed-format and represent items such as author, destination, date/time drafted, date/time received, etc.), and the text of the document itself, merely the set of all the words (or terms) present in the document [Chr84][2]. Properties of textual documents include:

a) document type: identifies the broad category of the document (paper, memorandum, letter) and which also typically defines the set of attributes for that document;

[2] In this thesis we will restrict our discussion to textual documents only. We acknowledge that multimedia document management systems capable of handling image, voice, and text exist, but are outside the scope of discussing signature retrieval methods for text. An introduction to a prototype of such a system, called MINOS, can be found in [Chr86a] and [Chr86b].

b) document subject area(s): broad subject area(s) to which the document pertains (an example being the subject areas listed in a United States Library of Congress "Cataloging in Publication Data" entry for a book);

c) document size: the length of the document text in bytes;

d) document vocabulary: the number of distinct, non-common[3] words in the document text;

e) document state: either "editing state" (the document may be modified), or "archived state" (where the document may only be referenced) [Chr86a];

f) access pattern: the technique(s) used to search for a given document (example: office memos are often untitled, making search by document title impractical);

g) document access frequency: the number of times the document is accessed over a period of time;

h) document update frequency: the number of times a document is altered in an editing state, and moved to an archived state, over a period of time.

Such paper documents are typically stored in a file, within drawers of file cabinets. Each file can contain documents of different types (memos, reports, etc.). Documents are filed together according to their intrinsic meaning (i.e. memos sent by a particular person relating to a particular subject may be stored together in the same file). Documents are searched for by determining the file that contains the relevant document, and sequentially searching the file's contents until the document is found[4].

[3] Strictly speaking, the vocabulary of a document is the set of all distinct words in the text; however, for the purposes of text retrieval common words (or stop words) such as "the", "where", and "but" can be (and are usually) ignored.

[4] It is acknowledged that the documents may, in fact, be stored in some order within the file to speed the manual retrieval process (e.g. temporal ordering).

Let us discuss these document properties in a little more detail. It should be apparent that documents in an office environment may be of several types: memos, letters, papers, reports, forms, etc. It can also be assumed that access patterns differ by type of document [Chr84]; memos, for example, may be more often searched by sender name than by the content of the particular memo.

Searching for a list of documents relevant to a particular subject in the above scenario is difficult; documents are stored in a specific, physical file. By this we mean that a document may only be filed under one subject area. As longer documents may deal with several different subjects, such a filing scheme can prove unworkable. A possible solution is to duplicate the document, and store it in different files, one for each relevant subject area. However, this may introduce new problems: the volume of documents will now grow much more quickly, and this can lead to consistency problems among the (possibly many) copies if a document's update frequency is greater than zero.

The sizes of the documents can also vary considerably, from very short memoranda to reports in the hundreds of pages. Document vocabulary differs widely as well; indeed, the overall vocabulary of the entire file system may change with each new document created.

Another problem of the "paper" office is that of sheer volume. Any enterprise of a substantial size can generate enormous amounts of paper. In addition, documents of widely varying subject matter can be produced by an enterprise made up of several distinct departments.

A final point concerning documents in an office: it has been shown [Fal85a] that office documents are typically not updated once they have been drafted (stated another way, average document update frequency is close to zero). Also, document access frequency has also been

shown [Chr84] to decrease exponentially with time (i.e. the older the document, the less frequently it is viewed).

1.1 Schools of the Pine Creek School Division.

The Pine Creek School Division is located in West-Central Manitoba, north of the city of Portage la Prairie and west of Lake Manitoba. Seven schools belong to the Division; five are elementary (Grades K to 8), and there are two high schools (Grades 9-12), one each in the towns of Macgregor and Gladstone. A total of 95 teachers in the various schools are responsible for approximately 1500 students. The smallest school (in the town of Langruth) has 5 teachers with about 70 students; the largest school is Macgregor High School with 15 teachers and 350 students.

For the past three years, as part of a program sponsored by the Manitoba Department of Education, each of the schools in the Division has used an IBM-PC compatible microcomputer to automate school office functions. Typically, each school has one secretary/administrator responsible for managing the office under the direction of the school principal. This individual uses the microcomputer to perform a variety of tasks. Some representative documents created by the secretary include:

a) student examinations
b) Division correspondence
c) letters to parents
d) timetables
e) minutes of staff meetings

f) class worksheets and assignments

g) class notes for students, prepared in advance

h) student handbook

These documents are created on behalf of any staff member, and are rarely, if ever, deleted. In addition:

a) not every school uses the microcomputer to the same degree; some schools continue to use typewriters for letters and other small documents.

b) the smallest school, Langruth Elementary, now has approximately 400 documents online, with an annual growth rate of about 20%. An average document consists of approximately 3000 bytes of text.

c) most of the documents are composed in English, though some are in French (all Schools in the Division have at least some classes in French).

1.2 Requirements of a Document Retrieval System.

Before discussing detailed requirements of the application, it is necessary to describe the hardware and software environment within each of the School offices in the Pine Creek School Division.

Software:

The Pine Creek School Division has standardized on the use of a software package called Enable[5], a unified productivity tool that includes word processing, spreadsheet, graphics,

[5] "Enable" is a registered trademark of The Software Group, Northway Ten Executive Park, Ballston Lake, New York 12019 USA.

database management, and telecommunications applications under a single user interface component (the "Master Control Module", or MCM). MCM provides windowing, file management tools, macros, profiles, a graphics mode, an MS-DOS[6] interface, and a menu generator for constructing user-defined scripts and command shells.

Documents on a School's microprocessor are created and edited using Enable's word processing component. This product functions in the same way as other, perhaps more familiar word processing software products (such as Microsoft[7] Word or Multimate[8]) and uses a proprietary MS-DOS file format for document storage (see Appendix B). A non-disclosure agreement between Langruth Elementary School and The Software Group was required before the Enable file format documentation could be received from the vendor.

Unfortunately, Enable lacks any type of sophisticated document management functions. This lack of function includes any type of document search capability; that is, the ability to search for documents previously created that contain a keyword or phrase. If the eight-character MS-DOS filename of the Enable document cannot be recalled, searching for a file can be a tedious process. This is the problem the PMQ application was expected to solve.

---

[6] "MS-DOS" is a trademark of Microsoft Corporation.

[7] "Microsoft" and "Word" are trademarks of Microsoft Corporation.

[8] "Multimate" is a trademark of Ashton Tate Corporation.

Hardware.

Each of the School offices in the Division has a Commodore PC10-II microcomputer (IBM PC-XT compatible), which has an Intel 8088[9] microprocessor CPU. Some have Nippon Electric Company (NEC) V-20 replacement 8088 chips for improved performance.

The "standard" configuration of these machines is 640K RAM (Random Access Memory), green graphics-capable monochrome monitor, 20 megabyte hard disk, and one floppy drive. Three of the machines now have a pointing device (mouse) installed.

Requirements.

The Partial Match Query application was to establish a document retrieval capability for Enable and ASCII files, under the following constraints:

a) the application had to be able to support the installed hardware, i.e. lack of a pointing device, monochrome monitors, and limited memory due to the 640K limit under MS-DOS.
b) the retrieval technique had to limit disk overhead to as great an extent as possible, yet provide acceptable retrieval performance. Document insertion performance was deemed to be less critical.
c) the application had to support searching by document content and document author, at a minimum.

[9] "Intel", "8088", "80286", and "80386" are trademarks of Intel Corporation.

d) to ensure compatibility with future hardware purchases, the application should be able, at a minimum, to support BIOS (Basic Input-Output System) screen I/O operations. Optimally, memory-mapped screen I/O should be used for speed.

e) no additional expenses, such as program royalties, were to be incurred. It was also hoped that no software licenses would be required.

f) only the English ASCII character set (French documents are composed using a standard English keyboard) would be supported.

1.3 Choice of a Signature File Organization.

The document searching requirements of the model office immediately suggested use of a signature technique, for a number of reasons. In general, signature techniques provide very low overhead compared to other methods such as inversion [Fal85b,Fal87c,Chr86] so media constraints could be avoided. The information loss inherent in signature methods [Fal87d] could be controlled by scanning the text of the documents to determine an exact match. Scanning could be considered due to the small average document size.

The DCBS method was chosen specifically for a number of reasons:

a) the method is very simple, and easy to implement using standard sequential files;

b) analytical results from [Fal87c] showed that DCBS exhibited excellent retrieval performance, with a low space overhead, over other signature techniques;

c) the DCBS file organization (discussed in detail in Chapter 3) can support a dynamic document set, with no theoretical limit on file sizes (physically restricted by MS-DOS to 32 megabytes);

d) the PMQ application presented an opportunity to implement and assess a functional, production system using a relatively new signature technique, which no known software at the time had used. Its use in the workplace could lead to further studies on retrieval performance once the aggregate size of all the documents became greater than the amount for which the index was originally designed.

To address the hardware support requirements, a documented, "freeware" menuing system had to be acquired, one that supported both BIOS and memory-mapped screen I/O under MS-DOS, had a C language API (Application Programming Interface), and did not have any licenses or royalties. Such a system was found in [Roc88]. Choice of BIOS or memory-mapped I/O was deemed a requirement due to questionable levels of full IBM-PC compatibility with the various hardware configurations in use in the Pine Creek School Division.

2. A Survey of Access Methods for Text.

In this chapter we review the access methods applicable to information retrieval that have been discussed in the literature. In the first section, we introduce common terminology regarding these methods. The terminology will assist in the discussion of the different access methods.

In Section 2.2, we discuss document (and document set) characteristics that affect retrieval performance. Here, problems studied in the field of Library Science are discussed from an information retrieval perspective. In Section 2.3, the particular characteristics of text retrieval in an office environment are introduced. Section 2.4 describes Salton and McGill's [Sal83] work concerning automatic indexing and term extraction.

Access methods for text that are not based on signature techniques are very briefly described in Section 2.5. The methods described include full text scanning, inversion, and multi-attribute hashing. This is followed in Section 2.6 by a more detailed discussion of signature techniques, giving an overview of the literature. Some examples of these methods, including some using compression techniques, are given in Section 2.7. This discussion will set the stage for an introduction to bit-slice signature techniques, and a detailed description of the DCBS (Doubly-Compressed Bit Slice) signature method in Chapter 3.

2.1 Terminology.

In a typical automated office, users create, send, and archive documents using an interactive computer (PC, minicomputer, or even mainframe based) system. Formally, a document **d** consists of the tuple (a_0, a_1, a_2, ... a_n, **b**) where a_i is an attribute associated with the particular

document (e.g. document sender, date/time created, etc.) and **b** is the free-format text of the document itself, consisting of the (ordered) occurrences of terms drawn from the document's vocabulary [Tsi83,Chr84]. Queries should be allowed to specify either attributes, terms that should be relevant to the particular document (usually based on the term's existence within a document, or term detection), or a combination of both.

Each document is defined to be of a particular document type. These document types correspond to typical documents found in paper environments: memos, letters, reports, etc. A document may be of only one document type. All of the documents in the system are referred to as a document set.

The text of a document (**b**) contains an arbitrary number of words (or terms). The document's vocabulary is dependent upon the vocabulary of the author and intended audience. Documents can, therefore, greatly differ in depth, detail, clarity, and precision. Fortunately, office documents are typically small, compared to other types of documents, such as academic papers. Salton [Sal75], quoting from Mandelbrot (Mandelbrot, B., "An Information Theory of the Statistical Structure of Language", Proceedings of the Symposium on Application of Communication Theory, London, Butterworth, 1953), states that small document size implies the document vocabulary is also small. Larson [Lar83] states that a small vocabulary implies better search retrieval performance.

In most implementations, queries against the document set use some form of Boolean algebra [Fal85a]. Search terms specified by the user are typically embraced by Boolean operators such as AND, OR, and NOT. Some commercial text management systems (such as Lotus' Magellan[10]) also permit queries based on the relative ordering (also known as proximity

[10] "Magellan" is a trademark of Lotus Corporation.

searching) of words within a passage of text; i.e. a search for a document relevant to "information retrieval" may give quite different results than a query made up of the two separate terms, "information" and "retrieval".

Uncontrolled indexing allows the entire individual vocabulary of each document in the document set to be used in defining the bibliographic terms (specific keywords, or author information, that pertains to a document) to be used. In contrast, controlled indexing uses a pre-determined set of bibliographic terms to control spelling and synonym usage.

The use of single terms in a document indexing system implies that the index words are made up of independent, individual words that characterize each document. Terms in context (another expression for proximity searching) allows the indexing system to form word phrases that identify a document. To do this, an indexing system may rely on either of two methods: pre-coordination, or post-coordination. Pre-coordination is the grouping of terms (phrases) in advance, to be treated as one bibliographic item. Pre-coordination implies that suitable term groups can be determined by the software and treated as a single unit (typically, it serves little purpose to index all groups of n consecutive words). Post-coordination involves the indexing of single terms, then combining each of the bibliographic terms given in a user's query in order to find the relevant documents based on each term group. In this way, terms within a document are indexed separately; on retrieval, relative query word position in each document is determined (perhaps using full text scanning) to ensure the document contains the term group.

Exhaustivity and specificity describe the characteristics of an indexing method. Exhaustivity describes how detailed and complete the index is for a given document; the more exhaustive an

index, the higher the proportion of matching documents that can be retrieved. Specificity refers to "the generic level of the index terms used to characterize the document content" [Sal83]. An example of this would be the use of the general phrase "Computer Science", instead of the more specific term "cryptography" in describing a paper discussing data encryption. If specific index terms are used, then there is a greater likelihood that non-relevant documents will be (correctly) rejected by the indexing system [Sal83].

The two terms recall and precision are used for relevance testing for any document search, and are the most important terms to remember when discussing search performance. These two measurements attempt to provide a common approach to evaluating distinct indexing techniques. Recall is a measurement of the proportion of relevant documents retrieved in a response to a query to the number of relevant documents that exists in the database. Precision measures the number of false drops that occur during a search; that is, how many *non-relevant* documents were shown as *relevant* in response to a given query. It should be apparent that these two properties are inversely related to each other. Often, the two properties are restated as "deep" and "shallow" indexing. Deep indexing describes an index that is both exhaustive and specific; shallow indexing implies the use of a more general method of indexing, on a broader, subject level.

A specific measurement for precision is called the false drop probability. This is the probability that a document *appears* to be relevant (is returned during a search), but the document is, in fact, not relevant to the query. The procedure for calculating the false drop probability is as follows [Fal84]:

a) the user submits a query.

b) from the total number of documents M in the database we discard the actually qualifying documents M_a.

c) from the remaining $M-M_a$ nonqualifying documents, we determine the number of false drops M_f, documents which *appear* to be relevant.

d) the false drop probability is the ratio of M_f to $M-M_a$ for the given query.

2.2 Document Characteristics and Retrieval Performance.

Three important characteristics of a document set in most office environments have already been mentioned: one, that the size of the document set (and, consequently, the size of the document database) is quite large, often in the gigabyte range; two, that the access frequency of documents decreases over time; and three, the performance of a document retrieval system is not typically dependent on the size of the documents in the set, but on the size of their vocabulary.

Salton [Sal83] agrees with Larson [Lar83] regarding vocabulary size. Salton also makes the point that indexing an entire document often gives very small increases in search efficiency over indexing only an abstract of the document. Often enough, the set of important bibliographic terms appear in the abstract of a document; performing analysis on the document as a whole provides few benefits, and (perhaps obviously) increases the cost of indexing (simply due to index size).

Another problem which contributes to the problems of recall and precision in text retrieval systems is that of handling the various forms of basically the same word; that is, treating the words "inform", "information", "informed", "informative", etc. as different instances of the verb "inform" (word truncation). This is a necessity since it is not desirable to have the user enter the exact form of the word to be searched for. Programs that handle the various prefixes and suffixes that occur in the English language have been written [Sal83, McI82] (a familiar example

is the spelling checker of commercial PC word processing systems). [Ben85] contains an excellent discussion of these problems. In some cases, however, the manipulation of a given word may involve a complete change of meaning. This is critical if the question being asked is, "What documents are relevant to this query?".

One example of such a linguistic problem is the word "expression", and its word stem, "express". The former deals with the subjects of meanings, ideas, or actions; but the latter will also be present in documents involving shipments of goods! The English language is, of course, very ambiguous; however, it is desirable that text retrieval systems should not exacerbate the problem by artificially adding to the ambiguity of a document index.

Other linguistic problems that exist with text retrieval systems include [Ten89]:

a) handling document sets that contain documents of more than one language. Searching without translation is very difficult; yet the translation process itself typically results in lost meaning. This can be easily demonstrated with metaphoric phrases, such as "throw the baby out with the bath water" which becomes nonsensical in French. Other French examples include words copied from other languages (e.g. "le weekend"). Table 1 presents some actual translation problems in Ukrainian, Polish, and English.

b) spelling variations and dialects. Examples here include British/American spelling variations ("labor" and "labour"), and the sometimes subtle (and sometimes dramatic) differences among various Slavic languages (see Table 1).

c) appropriately handling acronyms and abbreviations.

Example	Ukrainian[11]	Polish
1	maketpa	makutra
2	pokiplatuti	pogotowatz[12]
3	reedniti	spokrevaniatz
4	pulootornia	pulltorni
5	polootorka	pulltornonofka
6	pedstiati	podschilatch
7	perilazvati	wiligivitz or pschlzaich

Meanings (no direct English translation):

1. Earthen vessel in which poppy seeds are ground with a wooden roller.
2. to boil "for a while".
3. to make related; to become related to.
4. increased by fifty percent; of one and a half.
5. one and a half ton truck.
6. to spread under.
7. to lie for some time; to be numb by long lying.

Table 1
Examples of Translations between English and Polish
or Ukrainian languages.

[11] Due to technical limitations in printing the Ukrainian and Polish languages, the words listed here are presented in the English alphabet.

[12] In Polish this word has two different (but similar) spellings.

To make matters worse, authors in the field of Library Science [Mal83] state that a typical user of a text retrieval application is limited by other factors:

a) users are often under the misconception that the system "contains 'all knowledge' rather than a variety of discrete indices which cover individual fields well, partially, or not at all".[13]

b) or, the user assumes that the computer can "pick" the "good" ones, and reject the "bad" ones, on any given subject.

c) most document sets only contain data from the last 20 to 30 years; older information is usually not available in machine-readable form.

d) not all users have the personality and background to be productive at using these complex tools[14]. Such factors include a logical, analytical mind, good communication skills (particularly interviewing techniques), "economic attitude" (a "feel" for search cost), ability to make quick decisions, and (certainly not least) typing ability.

Two basic approaches can be used to "index" a document set (the word "index" here is used loosely, and is not meant to be confused with inversion). These two approaches are automatic indexing and manual indexing. In the automatic approach the computer system automatically determines the relevance of each document to a particular subject area. With manual indexing methods, a person is required to read each document, determine each document's relevance to the subject areas, and add indexing keywords that can be used to retrieve that particular document.

[13] [Mal83]: Thesing, Jane I. - "Online Searching in Perspective - Advantages and Limitations"

[14] [Mal83]: Hock, Randolph E. - "Who Should Search? The Attributes of a Good Searcher"

As English can be ambiguous (as previously mentioned), it would seem that automatic indexing techniques based on document content could not compete with manual methods in regard to accuracy. Salton, however, has shown that when suitable algorithms are used, automatic techniques can be used with great confidence, and produce results that are as correct as achieved by manual methods using experimental databases [Sal83].

In contradiction, other studies appearing in the literature [Ten89,Vig88,Bla85] have, to a great extent, not yet successfully proven some of the earlier theories. In fact, there seems to be considerable disagreement over some aspects of precision and recall when using various search strategies. Blair and Maron's six month study [Bla85] of a 40,000 document legal application (consisting of 350,000 pages of text) clearly shows that full-text retrieval alone, especially when search terms are combined in a conjunctive (AND) manner, affects recall quite negatively. A similar (but less thorough) study in [Ten89] shows improved recall using full-text retrieval and the conjunctive (AND) operator. Both the above studies also brought to light a very interesting characteristic of such systems: that recall is anywhere from 20% to 80% below the recall level anticipated by the user.

What is clear is that searching by full text only will yield less than optimal results, depending upon a variety of factors. These include document set vocabulary, the user's familiarity of the material, the size of the document set, and the average size of each document. Even so, Blair and Maron state that merely the depth of the index hinders the search performance (in terms of

recall and precision) of full-text systems, over manual indexing (shallow) techniques:

> "...in addition, it has only recently been observed that information retrieval systems do not scale up. That is, retrieval strategies that work well on small systems do not necessarily work well on larger systems (primarily because of output overload)... A full-text retrieval system bears the burden of retrieval failure because it places the user in the position of having to find (in a relatively short time) an impossibly difficult combination of search terms."[15]

2.3 Text-Retrieval in an Office Environment.

Online searching can be described as a classic example of the scientific research process, involving three key elements [Vig88]:

a) hypothesis (query formulation)

b) submission (query execution)

c) evaluation (evaluating the result with heuristics)

The performance of these three steps, using a myriad of different search criteria in combination (full text, document titles, keywords, author information, synonyms, thesauri, and word stems) takes strategy and planning on the part of the user. Searching is often governed by the law of diminishing returns [Vig88]; more search criteria do not necessarily provide a better search result (new terms increase recall, but usually decrease precision in that the new terms will create more (new) false drops).

[15] [Bla85], Page 298.

In an office environment, retrieval of documents by attribute(s) alone will occur; but almost certainly queries for documents pertaining to a particular subject will prevail. This exemplifies differences in search strategy in office environments, as opposed to more conventional applications, such as bibliographic search in an automated library. In an office environment, "abstracts" of documents are usually not available, nor are document titles normally useful. Search by document author is much more common. These differences in search strategy, along with the types of documents found in an office environment, dictate the indexing technique possibilities. Not every indexing algorithm is suitable for every application.

Given that retrieval of documents by their content (in addition to attributes) is desired, then a basic technique comes immediately to mind: full text scanning. This method is not usually considered, however. This is due to (normally) poor performance as the size of textual databases is usually very large. If the text of a document was transformed into a series of additional attributes, then database search techniques could then be applied (multi-attribute hashing, signature files, B-tree indexing).

An important characteristic of such an office system is that updates and deletions of documents typically occur very infrequently. In such an environment, the text retrieval method can take advantage of this characteristic and concentrate on insertion and retrieval techniques. Here, the use of an inversion technique to facilitate a document search may be practical; the (relatively major) expense of updating the index when a document is changed or deleted will not often occur. However, the storage overhead of using an inverted index (in the 50-300% range) [Fal85b] suggests that other alternatives, such as signature files, be considered.

2.4 Automatic Term Extraction.

This section introduces terminology and identifies problems of automatic term extraction to support full text retrieval, taken mostly from [Sal83]. The intent here is to contrast methods of automatic indexing against the important retrieval measurements of recall and precision. The following material will be referred to in the discussion of the PMQ application in Chapter 4.

There are two major tasks associated with automatic term extraction. Firstly, terms must be selected that are capable of representing a document's content. This selection process is important, as only useful terms should be candidates for retrieval (e.g. included in an index); terms which are too general or ambiguous should be ignored. Secondly, a value or weight is assigned to each term to represent its relative importance in indexing that particular document. This weight, or rank, provides a means of ranking selected documents in their order of relevance to a query, a highly desireable feature of a retrieval system. Two points should be made concerning ranking:

a) such a scheme requires a complete analysis of the document set vocabulary prior to any documents being inserted;

b) due to their simplicity, signature methods lack facilities to embed vocabulary weights in their file structure; thus signature techniques do not permit ranking of documents in retrieval.

In the area of defining bibliographic (representative) terms that represent the content of a particular document, most (if not all) automatic indexing schemes use Zipf's constant rank-frequency law [Sal75] which states:

$$\text{Frequency} * \text{Rank} = \text{Constant}$$

This implies that the frequency of a given word multiplied by the rank order of that word will be approximately equal to the frequency of another word multiplied by its rank (stated by Salton in referring to Zipf's paper entitled "Human Behavior and the Principle of Least Effort"). In [Sal75] Salton discusses Mandelbrot's (Mandelbrot, B., "An Information Theory of the Statistical Structure of Language", Proceedings of the Symposium on Application of Communication Theory, London, Butterworth, 1953) adaptation, which modifies the formula to:

$$\text{Frequency}_i = \text{Constant} * (\text{Rank}_i - a)^{-S}$$

where a and s are small real constants. "a" improves the fit of Zipf's hyperbola function for common words of low rank; s improves the fit for low-frequency terms of higher rank. Sacks-Davis [Sac87] uses the values $s = 1$, $a = 0$, Constant = 0.08137; the values are taken from Zipf's paper, which also discusses how to estimate these values.

Zipf's law has an interesting corollary; given some statistics concerning the frequency of commonly occurring words in the English language, it should be possible to predict the size of any given document, based on the size of that document's vocabulary.

What Zipf's law attempts to show is that term distribution in the document set is skewed. A search using a term that exists in each and every document will merely return the entire document set, which in all likelihood is not desirable (excellent recall, but poor precision). In the same way, a query using a word that exists in only one or two documents will probably result in excellent precision, but poor recall. Recall will be low because, in all probability, there are other relevant documents in the document set that do not contain that particular term (or term

group). Thus, whatever measurement is used to represent the rank of any particular term should use that term's relative frequency vs. that of any other word.

This method identifies terms that occur with substantial frequency in some documents, but with a (relatively) low frequency within other documents in the document set. This means, essentially, that one can assume that the frequency of useful terms in a document set vocabulary follow a normal distribution; high-frequency words that appear in too many documents are as undesirable as those words that appear in too few.

Three basic methods are outlined by Salton and McGill [Sal83] that incorporate the relative frequency of terms in the document set:

a) Inverse Document Frequency Weight (IDF)

b) Signal-Noise Ratio

c) Term Discrimination Value

These methods (among other, less complex/efficient ones, such as basic term frequency) are discussed in detail in various references ([Sal83], [Les68], [Fal85a]) and will not be repeated here. Notably, in all cases the generation of appropriate word phrases (pre-coordination of index terms) provided only marginal better query performance than that of automatic index using single terms. The use of word phrases did, statistically, generate better results, but the improvement was not substantial. It was also determined that using only document titles in the indexing process performed dismally in terms of precision (somewhat of an obvious result). However, the use of only document abstracts, rather than the entire text of the document, did not substantially alter the search results (and, of course, reduced the amount of resources required to store/search a document).

Furthermore, the use of sophisticated linguistic methods did not seem to offer any better results than those obtained with straightforward, single term techniques. Salton states that, in practice, a simple, "heterogeneous" document retrieval system may very well serve the needs of a broad spectrum of user requirements, as opposed to a complex technique that may, to some degree, be document set specific.

2.5 Access Methods for Text.

Before discussing signature techniques, we will briefly introduce other access methods suitable for text retrieval so that comparisons between these methods and signature techniques can be made.

In this section, we discuss non-signature access methods in a more general way than in the narrow context of document text retrieval. To this end, we define the term record as a group of related data elements (also termed attributes) (normally stored in a contiguous fashion) and usually identified by one or more data elements termed a primary key. A file is a set of records with the same characteristics (in particular, the format of each record is identical, though their contents may differ).

2.5.1 Full Text Scanning.

Full text scanning has already been introduced in previous sections. This is the most straightforward way of locating a particular string (a sequence of characters) in a document set. Typically scanning is not a suitable method for systems of any size, since this (rather brute force) method is inherently slow. Indeed, the whole point of indexing is to avoid this expense.

Several different algorithms have been studied to speed up full text scanning. These techniques fall under the general term pattern matching. Boyer and Moore [Boy77] discuss a technique where the pattern matching starts at the *end* (i.e. the last character) of the string to be searched; this method is acknowledged in the literature as the most efficient method discovered to date. Faloutsos [Fal85a] gives a good introduction to a range of full text scanning techniques, including hardware-based methods.

Despite its problems, full text scanning can be useful when used in combination with other techniques (such as signature techniques) where the scope of the documents to be searched can be reduced to a manageable number.

2.5.2 Inversion.

The most common access method for use in database systems (particularly relational database systems) is inversion [Dat86]. As stated previously, the generic term "index" is often used as a synonym for the term inverted list, which more accurately describes the technique.

With a non-indexed file, each record lists, in physical order, the data elements that make up the record. By contrast, an inverted list lists, for each value of the indexed data element, the records containing that value. To identify the records, a number of possible techniques may be used, including the primary key or the record's physical media address. An introduction to inversion can be found in [Dat86,Knu73,Car79, and Wie83].

Applying an inversion method to information retrieval, each document is represented by an inverted list of redundantly stored (key) words. The inverted list contains "pointers" (symbolic or

physical), which identify those documents that contain the particular term, and therefore describe the contents of the actual document.

Various, more sophisticated methods of storing the inverted lists are usually used, the most common being forms of B-trees [Dat86]. However, before explaining B-trees we must explain the notion of a multi-level, or tree-structured, index.

A tree index is one consisting of a hierarchy of indexes. Conceptually we treat the lowest level index as not only an inverted list, but as a file in its own right, such that the next higher level is an index to the immediately lower level. This can be recursively applied to as many levels as necessary.

Common terms for these levels are sequence set for the lowest level index, and index set for the aggregate of the higher levels [Wie83]. The top level (or node) of the tree is referred to as the root, and nodes at the lowest level are referred to as leaves. Each node contains a fixed number of pointers (tree pointers) to nodes in the next lowest level (termed children). Each tree pointer points to a lower-level node whose key values (a word in this context) are bounded by the key values on each side of the pointer (keys are stored inorder in the tree). The number of levels of the tree is referred to as the depth of the tree. Physically, the nodes of the tree are combined into pages, to save in I/O operations. Balanced trees are ones in which the longest path from the root to any leaf node, and the shortest such path, differ by at most one level.

Tree indexes are classified as either heterogeneous or homogeneous [Har88]. In a heterogeneous tree index, each node of the tree only contains one type of pointer, but the pointers are different for the lowest level, or leaf, nodes than for the higher-level nodes. The leaf nodes point to data records; higher-level nodes point only to lower-level nodes. Homogeneous

tree indexes contain one type of node for all levels. With this type of index, the leaf nodes all contain empty tree pointers, and have valid data pointers. Higher-level nodes may contain both types of pointers. With homogeneous trees, it is usually required to have one more tree pointer than data pointers in each node [Har88].

A B-tree of order m is defined as a balanced, homogeneous index tree with these characteristics [Har88]:

a) Every node has at most 2m keys and 2m + 1 tree pointers.

b) No node, except for the root, may have fewer than m keys.

c) The root has at least 2 children (unless the file consists of a root only).

d) All leaves (records with no children) appear on the same level.

e) A non-leaf node with k children contains k-1 keys.

A common modification to B-trees, that improves searching, results in a more refined structure known as B*-trees. B*-trees differ from B-trees in how full pages of records are split; we change note (b) above to ensure that a minimum of (4/3)m keys are kept in each node (not m). This improves the average occupancy rate of each node, saving index space.

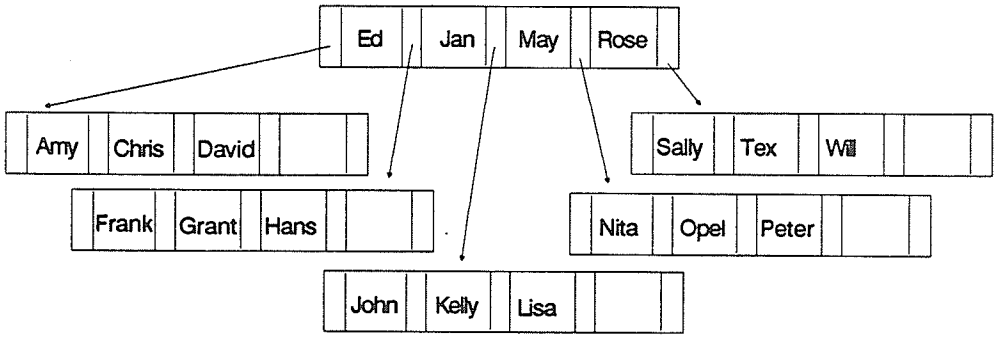
A further variant of B-trees is the B+-tree, which is used as the basis for many commercial implementations (IBM's Virtual Storage Access Method (VSAM)). The B+-tree is quite different from the B-tree in that:

a) a B+-tree is heterogeneous, not homogeneous. All data pointers are in the lowest-level, leaf nodes.

b) Keys are duplicated in the tree (as all keys are in the lowest level). A typical implementation duplicates the rightmost leaf key of the left subtree in the next higher index level.

c) the nodes are threaded in that each leaf node points to the next leaf node (in key sequence). Thus the index may be traversed by merely navigating through the sequence set alone.

Two-level Standard B-tree of order d=2
as described by [Har88].



Three-level B+-tree of order d=2
as described by [Har88].

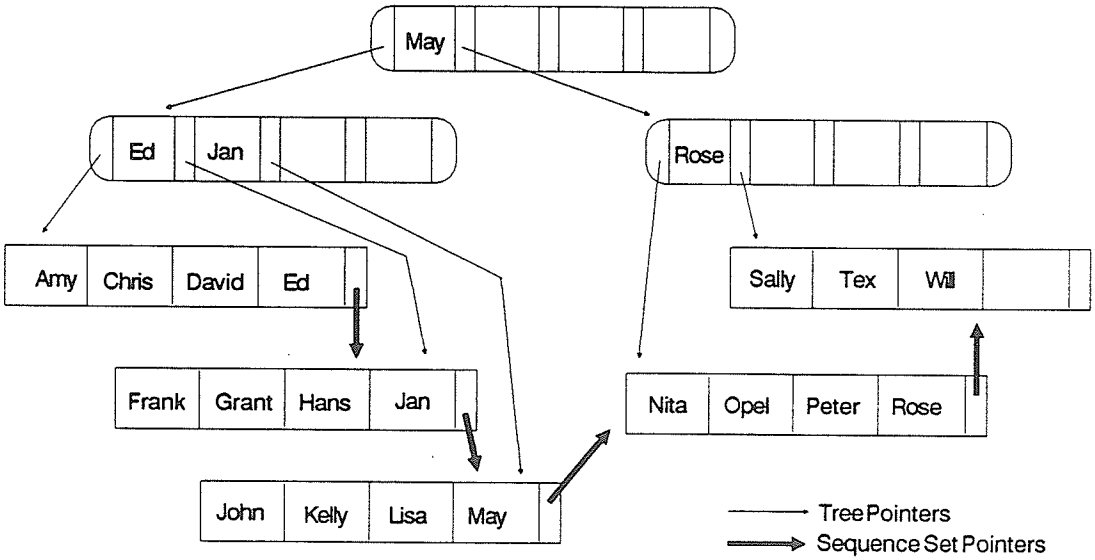


Figure 1
Sample B-tree and B+-tree Organizations

For text retrieval systems, B+-trees typically contain the terms that exist in the document set. Each entry in the tree points to the document that contains the term, and may even store additional information (STAIRS), such as the exact location of each term in the document (down to the chapter, paragraph, sentence, and term-within-sentence level). Unfortunately, this level of detail frequently results in a space overhead of 50-300% of the size of the document set [Fal85a]. Also, modification of an indexed document results in the possible re-calculation of each term position in the document, and the consequent updates to much of the index. Finally, a major disadvantage of inversion techniques are that the more attributes specified in a query, the *larger* the amount of work has to be performed (simply because of the numbers of false drops generated in the merging of the lists) [Ram83].

2.5.3 Multi-attribute Hashing.

Multi-attribute hashing (also known in the literature as multi-key hashing, or MKH) methods present an alternative to inversion, and can be used in many situations to provide random access to particular records where the overhead of a B-tree is not desired. Hashing techniques are very similar to those used in signature methods. Chapter 3 will show marked similarities between hashing and the DCBS signature method used by PMQ.

One of the difficulties encountered in using traditional hashing techniques for external files is that typically the size of the file must be fixed; the address space of the hashing function must be known beforehand, and if ever altered the entire file must be reorganized. What is required is a technique to handle a dynamic file structure. In this subsection we discuss four such methods:

a) Z-ordering;

b) extendible arrays;

c) linear hashing;

d) extendible hashing.

For a survey of various multi-attribute hashing methods, the reader is directed to [Knu73][16] and [Tre84][17].

[16] [Knu73], pp. 562-567.

[17] [Tre84], pp. 764-797.

Z-ordering.

Multi-attribute hashing refers to the transformation of a record's attribute values into binary representations using a series of hashing functions. Z-ordering refers to the interleaving of the bits formed by the hashing functions into a single value representing the entire record; as such it can be used as a key and stored in a conventional index (B-tree) and used for retrieval purposes. The interleaving function attempts to cluster records with similar attributes so that they are stored in the same page of the file (or, at least, in a near page).

With clustering, the number of records retrieved in a partial match query is the same as without clustering, but performance is improved. The improved performance is due to fewer random I/O operations to different parts of the file. In fact, if the query profile is known in advance, asymmetric interleaving [Fal88] of the bits may substantially improve best-case performance. If the query profiles are not known, symmetric interleaving [Fal88] (treating all attribute values in the same manner) gives optimal worst-case performance.

[Ore82] is credited with the first use of the term Z-ordering. Originally, the intent was to use a type of k-d tree to store the combined hash values, but the concept of Z-ordering is applicable to many different file organizations. Faloutsos [Fal88] applies the use of Z-ordering using Gray Codes to extendible hashing techniques.

Suppose that each of the domains from which **k** attribute values are drawn contain 2^d elements. If the **k** attributes of tuple **t** are transformed into a binary coding scheme, then we have $\{\mathbf{P}_{0,0}, \mathbf{P}_{0,1}, \cdots \mathbf{P}_{0,d-1}, \mathbf{P}_{k-1,0}, \mathbf{P}_{k-1,1}, \cdots \mathbf{P}_{k-1,d-1}\}$ where $\mathbf{P}_{ij}$ is the jth bit of $\mathbf{P}_i$'s binary value. A "shuffle" function is defined (shuffle(X)) which interleaves the bits. X represents the order in which bits

are selected from each attribute (often done cyclically amongst the attributes in turn). For example, shuffle(1,2,1,2) is a cyclic system for two attributes where bits P_{11}, P_{21}, P_{12}, P_{22} are interleaved to form the combined hash value. Figure 2 shows a shuffle function of (2,2,1,1). An "unshuffle" function can also be defined to determine the binary representation of a given attribute value when the combined hash value is known.

Z-ordering represents a useful way to encode attribute values for any set of records such that primary key file organization techniques like linear hashing and extendible hashing may be applied, since Z-ordering maps a multi-dimensional key to a single value. It is particularly appropriate in text retrieval applications for attribute values. However, for full text indexing the number of attributes **k** is very large (e.g. the document set's vocabulary). This makes hashing methods in general unsuitable for full text indexing.

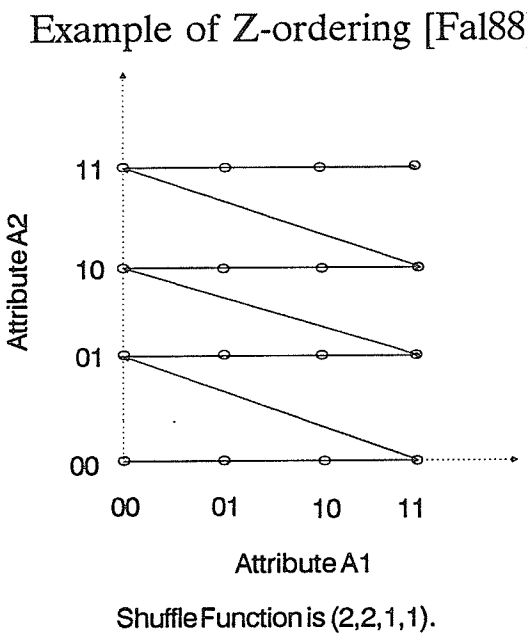


Figure 2
Example of Z-ordering.

Extendible Arrays.

Extendible Arrays are **k**-dimensional arrays of buckets (pages each containing **n** blocks) organized on physical media in such a fashion that the bounds of the array can be expanded to suit dynamic files. Such an array is defined to be orthant [Ros74]. In a text-retrieval application, each dimension would represent an attribute of a document. The "value" stored in the array position corresponding to the attribute values would be a document identifier.

Conceptually, the array **A** is expanded by adding slices of **k**-1 dimensions whenever one particular dimension is expanded (see Figure 3). With standard, sequential file organizations it is difficult to expand all but one dimension; for instance, for a 2-dimensional array the familiar "store-by-row" scheme [Ros75] permits easy extension of the number of rows but not the number of columns. Again, like Linear Hashing, a series of overflow buckets from each array bucket is required to handle more than **n** elements per bucket. In fact, [Ros74] shows that the goals of efficient array traversal and easy extension of any array dimension are incompatible; however a reasonable compromise can be made at the cost of a more inefficient file structure.

The file organization for extendible arrays permits the expansion of any dimension. Three ways of determining the location of any element in the array have been defined in the literature [Oto83]: a) computed access, where the location can be defined using an algorithm; b) linked allocation, where the array buckets are stored using multiple linked lists to link the rows and columns; and c) hashing.

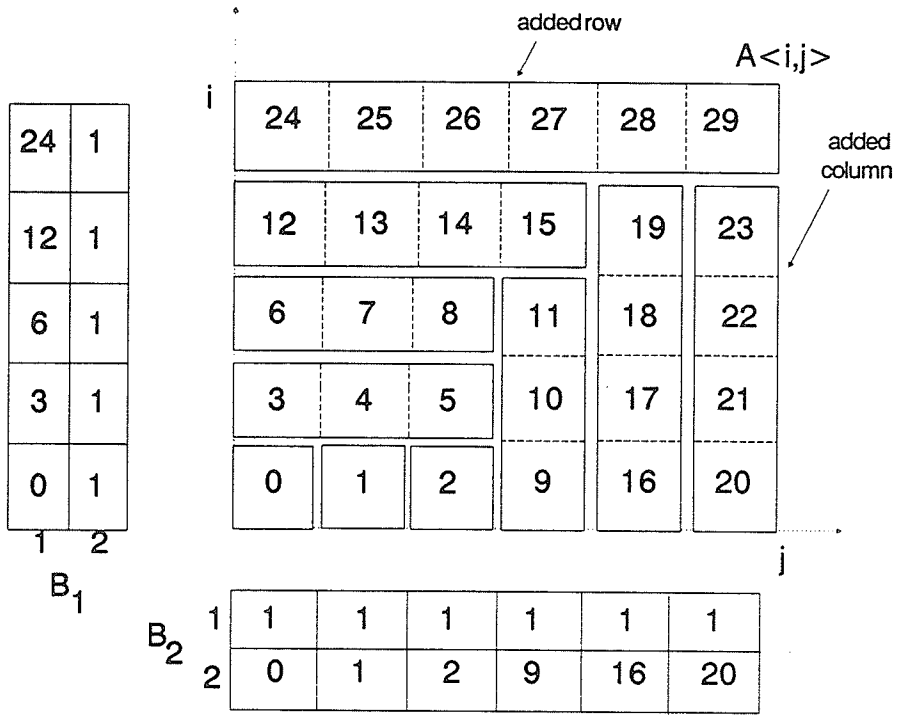
Otoo [Oto83] presents a computed access scheme (termed the Index-Array Technique) that keeps track of the starting bucket number for each row and column in a separate data structure

(k arrays denoted $\mathbf{B}_i$ which are themselves extendible in a single dimension, i.e. store-by-row). This scheme is illustrated in Figure 3. As the file expands, the starting buckets of the added slices are stored in the $\mathbf{B}_i$ index array corresponding to the expanded dimension.

As an example, to find the bucket corresponding to coordinates (1,5) in Figure 3, we proceed as follows. First, we must determine if the element A(1,5) was allocated as part of row 1 or column 5 (i.e. which dimension was expanded to store the element). If array element A(1,5) was added as part of row 1, $\mathbf{B}_2(5,2)$ (denoting the 5th position in dimension 2) would have a value less than that of $\mathbf{B}_1(1,1)$. In this instance we have $\mathbf{B}_2(5,2) = 20$, $\mathbf{B}_1(1,1) = 3$. So we must assume that A(1,5) was added as part of column 5, and the bucket number is $20 + 1 = 21$.

Use of this technique with full text retrieval is not recommended, as the array will be very sparse (and hence the method will incur a large space overhead) since the k dimensions will map to the document set vocabulary. However the approach is a very reasonable one for document attributes.

Example of Two-dimensional 6x5 Extendible Array from [Oto83].



A<i,j> is the extendible array; the other arrays marked B are Index Arrays in Otoo's scheme.

Figure 3
Extendible Array of Two Dimensions (from [Oto83]).
Note A has been extended to 5 rows and 6 columns.

Linear Hashing.

The credit for the design of the basic linear hashing algorithm is given to Litwin[18] [Ram84,Tre84]. Litwin's file structure can handle dynamic files as the file is gradually expanded by splitting the buckets (pages) of records in order until the size of the file has doubled. Linear hashing is adaptable to multi-attribute keys using Z-ordering or applying the techniques for extendible arrays.

To obtain a hashing function that can vary with the size of the file, we require a series of hashing functions $\mathbf{h}_i(\mathbf{k}), i = 1,2,...$ These functions map the key space of the file (denoted by **K**) into the integers $\{0,1,2, ... 2^i-1\}$ such that (called Litwin's condition [Tre84]) for any key:

$$\mathbf{h}_i(\mathbf{k}) = \mathbf{h}_{i-1}(\mathbf{k}) \qquad \text{or,}$$
$$\mathbf{h}_i(\mathbf{k}) = \mathbf{h}_{i-1}(\mathbf{k}) + 2^{i-1}.$$

These two rules ensure that when the file size is doubled, the hashing function maps a key to the same location as before (first equation), or maps the key in the same relative position in the second half of the file (second equation). For example, a key mapped to bucket 154 in a 256-bucket file will, after doubling, either remain in bucket 154 or be stored in bucket 154 + 256 = 410 in a 512-bucket file.

Typically, the i hashing functions are defined by first defining a hashing function **H** for the entire key space, mapping to a bit string of **m** bits (**m** sufficiently large for the maximum size of the

[18] Litwin, W. - Linear Hashing, A New Tool for Files and Table Addressing. In Proceedings 6th International Conference on Very Large Databases (Montreal, 1980), pp. 212-223.

file). Then, $\mathbf{h_i(k)}$ are merely defined as substrings of the **m** bits (and therefore satisfy the two equations above).

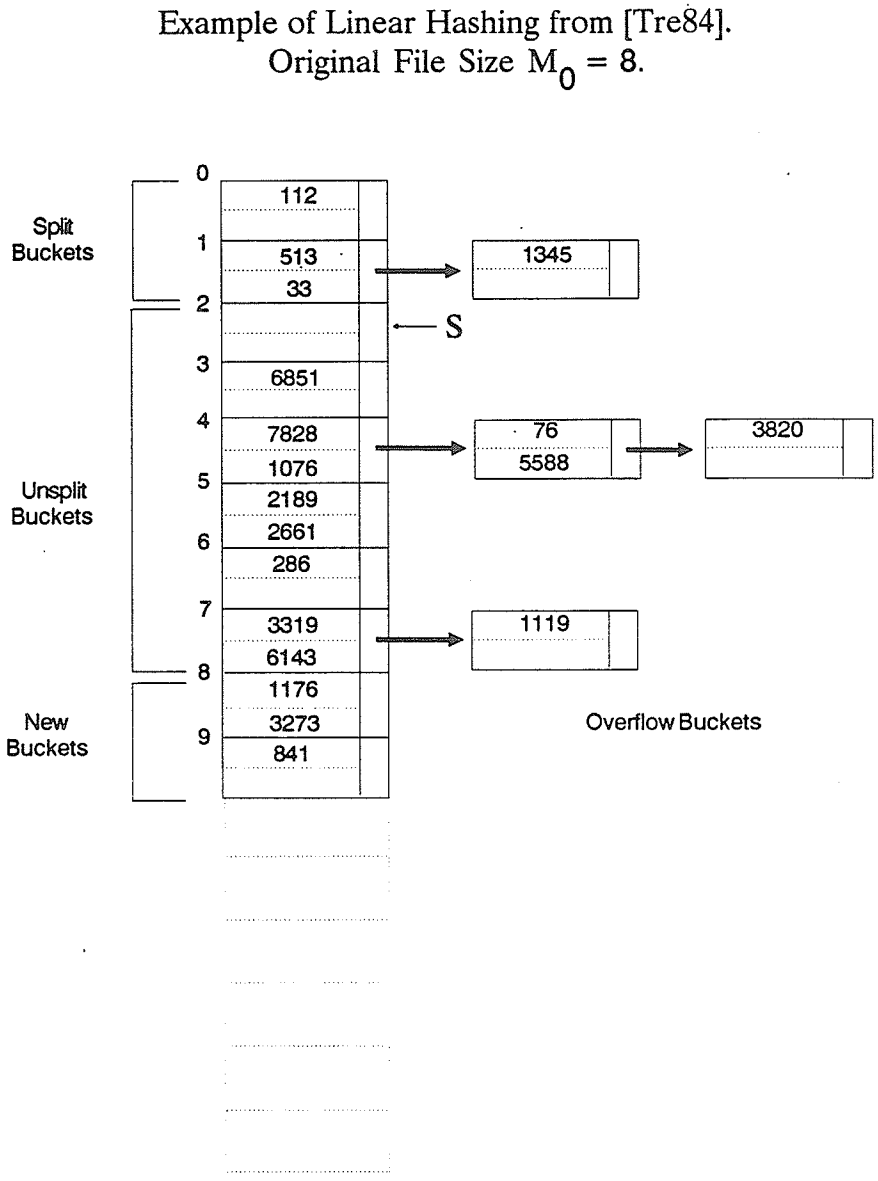


Figure 4
Linear Hashing Scheme.

The file structure to support linear hashing is as follows [Ram84]. Groups of records are combined into buckets (termed home pages in his paper); all records within a bucket must hash to the same value. A chain of overflow buckets from each home page handles extra records that cannot fit into a home page.

With linear hashing, only three variables must be known: M_0, **d**, and the number of the bucket to be split next, **s**. When the file is created, it will consist of $s = 2^d M_0$ pages, where **d** is initially 0 and M_0 is the initial size of the file. The value **d**, then, represents the number of times the file has been doubled in size. The hashing function for a key **k** being inserted into these 2^d pages is $h_d(k)$. The rate at which new pages are appended is determined by the load factor, a design parameter.

Consider a file of size $2^d M_0$. The file consists of three parts. Buckets 0,1,2 ... **s**-1 are split buckets, buckets **s**, **s**+1, ... $2^{d-1}M_0$ -1 are unsplit buckets, and buckets $2^{d-1}M_0$, $2^{d-1}M_0+1$, ... $2^{d-1}M_0+$s-1 are new buckets. The hashing function h_d is used with split and new buckets, and h_{d-1} is used with unsplit buckets. As pages are added, **s** is incremented to split pages as required until the file size is doubled. **s** is then reset to the first bucket (0), and the process is repeated for the next doubling (see Figure 4).

Extendible Hashing.

Another technique that is described in a number of references [Har88,Tre84,Dat86,Fal88] is extendible hashing. Here, a series of transformations of the record's key is used to first map the attribute values onto a fixed address space of hash values **H**(key), followed by a table look-up of the first few digits of the hash value to determine the bucket address (again the buckets contain **n** records). Figure 5 gives an example of an extendible hashing scheme.

Z-ordering can be used to create a single value from transformations of P attribute values for each record in the file [Fal88]. In a text retrieval application, a record would represent a document. The value of the Z-ordered hash value is then partitioned into 2^d equal ranges, typically by partitioning the hash values using the first **d** bits of each value. The 2^d ranges are represented by a directory file which has 2^d pointers. Note that **d** can increase as the file gets larger, though 2^d may not exceed the size of the initial hashing address space. Some algorithms do exist to adapt extendible arrays to extendible hashing to circumvent this restriction.

The buckets pointed to by the directory store the actual records. With each bucket is kept the number of digits of **H**(key) whose value is common to all records in the bucket (denoted in Figure 5 by **d**'). Though rather complex, this two-level organization allows the file to expand and contract gracefully as records are added and deleted.

Briefly, the expansion of any bucket due to overflow results in a new bucket being created, and (typically) half of the records in the original bucket moved to the new bucket (and half of the directory entries pointing to the old bucket are changed to point to the new one). The number of digits of **H**(key) **d**' "recognized" for each bucket is altered accordingly. If only one directory entry

points to a bucket that is overflowing, then the directory size is doubled (becomes 2^{d+1}) and each pointer in the original directory is duplicated. Once there are two directory pointers to the candidate bucket, the bucket may be split as above. Contraction of the buckets (and the directory) follows in reverse to expansion.

Extendible Hashing from [Har88].

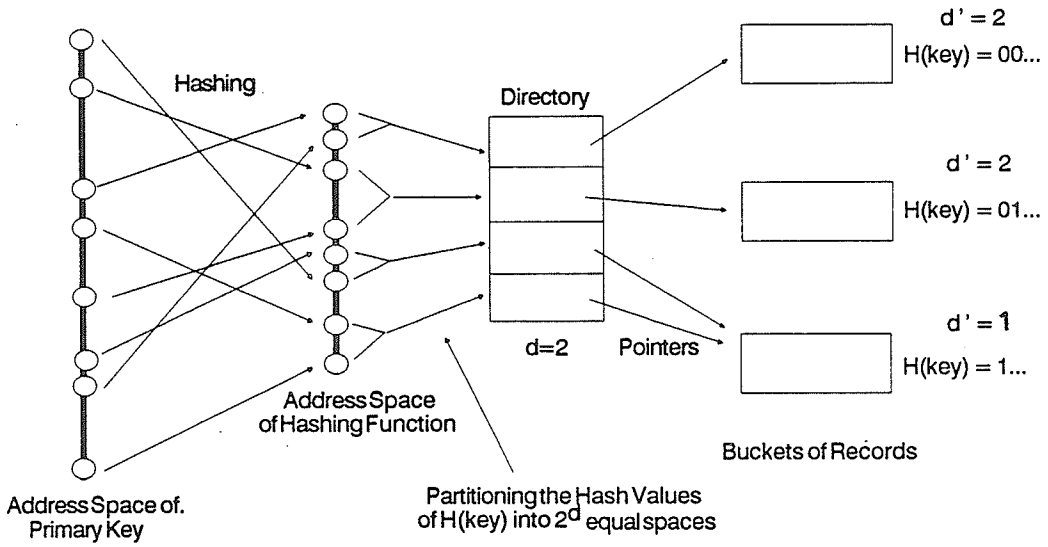


Figure 5
Extendible Hashing (from [Har88]).

This concludes our review of alternative access methods for text. From this point onwards, we shall discuss signature methods.

2.6 Signature Representation of Text.

In this section, we discuss several methods of representing the content of a document using signatures. For the time being, we will ignore the fact that our definition of a document includes the existence of attributes (sender name, date/time sent, date/time received, etc.). As these attributes are fixed in terms of definition and data type, usual database search techniques can be used for them (inversion, multi-attribute hashing, etc.). Thus, the two different types of data, attributes and text, can be searched using completely different techniques; the results can be combined in order to determine the relevant documents.

In the literature, the term signature is often used interchangeably with the term descriptor. A definition from Pfaltz et al. [Pfa80] defines a descriptor **D** to be a string of **w** bits, and each record (document) in the file **R** has associated with it a descriptor **D** that represents the actual contents of the data record. It should be noted that the value of a descriptor bears no relation to where the record may actually be stored in **R**; this is where signatures differ from multi-attribute hashing techniques.

There are two basic ways of forming descriptors:

a) The first method, superimposed coding, uses a hash function h on the attributes of each record, and inclusive-OR's the hash values together to form a single hash value (the record's signature) which then represents the contents of the record as a whole. To check if a signature matches a query, the query terms are also transformed to a bit string using the identical hashing algorithm. If every bit set in the query signature is also set in a record signature, then the record is a potential match (only potentially because 1:1 hashing functions are rare, and

expensive). If the record does not actually match the query (but its signature does), then the record is termed a false match, or false drop. If the record does explicitly contain the query terms then the record is termed a true drop.

b) The other method, called disjoint coding, uses a slightly different technique. Like the former method, each attribute in the record is hashed, using some hashing algorithm. With disjoint coding, however, the individual attribute hash values are concatenated to form the record signature.

A Brief Survey of Signature Techniques.

Discussions in the literature regarding use of signatures first appeared between 1945 and 1950 (in the application of notched-edge card filing systems), and has continued through to the present day [Rob79,Fal85a]. Many of the discussions up to the late 1960's concentrated on coding and information theory; in the late 1970's and early 1980's we begin to see references to practical computer applications.

Since the two basic techniques were studied, many efforts have been made in attempting to minimize the search time for signature files, but keep space overhead to a minimum. Four basic strategies to do this have been studied recently [Cha89b], namely:

a) special schemes for encoding high-usage terms and attributes;

b) multi-level signatures files;

c) bit-slice representations;

d) compression techniques.

Table 2 [Cha89b] describes the progress of research into the various methods. Some of these investigations are noted below.

Signature Files - Classification of the Literature

		Bit String		Bit Slice	
		No Special Encoding	Special Encoding	No Special Encoding	Special Encoding
One Level	No Comp.	[Fil69]	[Fal87b]	[Rob79]	N/A
	Comp.	[Fal85a]	N/A	[Fal87c]	N/A
Multi-level	No Comp.	[Sac82]	[Sac87]	[Sac82]	[Sac87]
	Comp.	N/A	N/A	[Fal87c]	N/A

Table 2
Classification of the Literature [Cha89b].

[Lar83] and [Tsi83] describe the use of disjoint coding techniques (coined the Word Signature (WS) method) in the creation of a "surrogate" signature file that contains a coded representation of the words in each document. The advantage to disjoint coding is that the order of the words in the free-format text is retained in the signature file, making searching by terms in context a relatively easy matter.

Files and Huskey [Fil69] are credited with the introduction of the superimposed coding (SC) signature method [Cha89b]. Here, the document signature is created by inclusive-OR'ing all the word signatures for each word contained in the document. To handle varying-length documents, each document is broken up into "logical blocks" of a fixed size (a parameter to the method). [Fal84] gives a thorough description of the method, and compares the SC method to the WS method of [Lar83] and [Tsi83].

Shepherd et al. [She88] describes a method based on SC, but their method does not require the use of logical blocks; their method is based on Bloom Filters [Blo70]. In simplistic terms, a Bloom Filter is a (usually) large signature made up of superimposed signature values for each non-common word in the document set (thus the document set is treated as a single entity). The "filter" can then discriminate search values, for if a signature match between a query word and the filter does not exist, then the document set is known to not contain the query word. Thus, Bloom Filters are an excellent algorithm for use in situations where an unsuccessful search is desired (an example being a credit-card authorization system). Shepherd et al.'s method uses a distribution of numbers of words per document (as opposed to a fixed number in SC) to determine the size of the filter. Knuth [Knu73] provides a brief description of Bloom Filters. An experiment regarding actual versus theoretical probabilities of filter error performance using Bloom Filters for text retrieval can be found in [Ram89].

Other attempts at managing skewed distributions of query terms (in particular, attributes and text) have been given by [Rob79,Fal87b,Cha89b]. Roberts gives suggestions concerning hashing of frequently occurring keywords. Faloutsos [Fal87b] takes this one step further and gives formulae for optimal parameters when indexing both text and attributes of documents, using the SC method. [Cha89b] contains an empirical study of using other, non-signature techniques to store attributes. However, little work has been done in this area in regard to bit-slice signature organizations (bit-slice methods are discussed later in this section).

Pfaltz et al. [Pfa80] discussed the use of a multi-level signature file, an instance of the Indexed Descriptor Access Method (IDAM) organization described in [Fre85]. A signature at level n-1 is created by superimposing the signatures at level n. Signatures at level n were created using disjoint coding of the attributes given for each document (full-text was not considered). Another general description of the method can be found in [Tha90].

[Du89] gives a new approach to the [Pfa80] method, using a hierarchical file structure of dynamically allocated "buckets". Attributes and document keywords form separate signature values, the former using an extendible-hashing technique, the latter using superimposed coding. Again, full-text is not considered in the analysis. However, the major contribution of this paper was the comparison of signature methods to inversion techniques. Pfaltz et al. [Pfa80] showed that, with their method, as the number of attributes specified in a query increases, the number of blocks accessed decreases.

A similar approach was used by Ramamohanarao, Lloyd and Thom [Ram83]. Like the technique of Pfaltz et al., [Ram83] uses disjoint coding to form the descriptors. The descriptors

are used to aid in the search of attributes with a high cardinality. The main method of retrieval relies on a multi-attribute hashing algorithm.

[Cha89a], [Sac82] and [Sac87] describe two other approaches to multi-level signature files. They attempt to solve the two major problems that exist with multi-level signature files:

a) the density of the higher-level signatures becomes greater than the optimal density of 0.5 (as all lower-level signatures are inclusive-OR'd).

b) the block (high-level) signature now represents the Cartesian product of all the words that are represented by the lower-level signatures, which will result in a higher false drop probability.

[Cha89a] describes a method of combining block signatures using n different algorithms for the n levels in the signature file, in an attempt to avoid problem (b). Problem (a) is avoided by increasing the signature size as more levels are created in the signature file. Multi-level signature files are easily adaptable to other (well-known) file structures, such as B-trees. [Cha89a] describes how to incorporate these multi-level signatures into Starburst B-tree indices, giving text retrieval capability to a relational database manager. [Sac82] uses multi-level signature files, but also incorporates a bit-slice approach; the method (and it's extension for handling special encoding discussed in [Sac87]) is described in Section 2.6.

[Cha89b] modifies the method of Sacks-Davis [Sac82] in a different way. Recognizing that query distributions are normally not equi-probable, Chang et al. use a multi-level bit-slice signature file to index full text, but describe three different methods to index attributes: a bit-slice block signature as for text, inversion using inverted lists, and a hashed organization. [Cha89b]

describes performance improvements up to 30%, with an increase in space overhead of only 9.3%.

Roberts [Rob79] gives one of the first discussions of bit-slicing techniques in his implementation of a telephone directory application. If the set of signatures developed from the document set is (conceptually) considered as a signature *matrix*, typically the signatures are stored row-wise on the storage media (each signature behaves in the same manner as records in a file). Bit-slicing involves storing the record signatures *column-wise* on the storage media (the terms bit-wise and column-wise are synonymous). With this organization, typically each column of the signature matrix is stored as its own file, allowing additional signatures to be added to the end of each "column" of the matrix. This method typically provides superior retrieval performance, with a penalty of increased I/O operations for signature insertion (now each of the files (one per column) must be updated).

A significant amount of research in the last several years has been done in the area of compressing signatures, such that the same false drop probability (performance) is maintained, but less media storage overhead is required.

[Fal85b,Fal87a] give an overview and performance comparison of compression techniques based on superimposed coding methods. These methods are discussed in some detail below. Briefly, the attempt here is to use techniques such as run-length encoding to reduce the size of each signature. However, the entire signature file must (still) be read in order to find all possible matches.

[Fal87c] describes several approaches to compression using bit-slice techniques. Here, the method is applied to optical disk (WORM) [Fuj84] technology. This implies that the file

structure used to store the document signatures can permit only the re-writing of a disk block that changes 0's to 1's. Further references to the use of WORM disks in document retrieval systems can be found in [Chr86a,Chr86b].

2.7. An Outline of Some Signature Methods.

We now present a brief analysis of several signature methods for representing text. The methods are not overly complex; all rely on some hashing algorithm (transformation) that generates the **m** bits for each signature. The methods discussed here include[19]:

a) the "Word Signature" (WS) method [Tis83 and Lar83];

b) the "Superimposed Coding" (SC) method [Fal84, Fal85b];

c) the "Run-length Encoding (RL) method [Fal85b];

d) the "Bit-block Compression" (BC) method [Fal85b];

e) the "Variable Bit-block Compression (VBC) method [Fal85b];

f) "Multi-level Block Signatures" (ML) methods, from [Pfa80, Sac82, Sac87].

In terms of relevance to a query, each of these methods offers varying degrees of recall. Only the Multi-level signature file method (ML) makes an attempt at vocabulary analysis of the document set; with all the other methods, each non-common word in the document is indexed. Non-common terms are defined as those words that are not "stop words" ("filter words"). These "common" words include many of the pronouns and prepositions found in everyday language, such as "and", "not", "where", etc. The usage frequency of these words typically renders them unsuitable for indexing and retrieval purposes.

[19] In the next chapter, we will focus on a set of bit-slice signature techniques, which are the basis for the "Doubly-Compressed Bit Slice" (DCBS) method.

As a consequence of not performing vocabulary analysis, the methods other than ML suffer somewhat in terms of precision. Precision is lessened in two ways. Firstly, if the vocabulary is not analyzed to index only useful terms, then the probability that two distinct words will be represented by the same bit pattern (signature), termed a collision, is increased. Secondly, the existence of a given term in a document does not in itself mean that the document is relevant to the query (see Section 2.4).

Again in this section, we use the expressions "term" and "word" synonymously to describe individual items in the free format text of a document.

The "Word Signature" (WS) method.

This method was originally published by Tsichritzis and Christodoulakis in 1983 [Tis83] and further documented by Larson [Lar83]. The signature file is termed a "surrogate" database in which the document signatures are kept; this database is kept synchronized with the actual text data.

In the WS method, each significant word (or term) of a document is mapped into a bit string of length w; 2^w possible word signatures may be generated. Using disjoint coding, the signatures of each individual term in the document are concatenated together to form the document signature. Searching for a particular document is straightforward: the query terms are hashed using the same hashing function, and the surrogate database is sequentially scanned (that is, the full document signature of each document must be analyzed), looking for matching hash values.

As an example, suppose we have the following document, consisting of six words:

Document: The cow jumped over the moon.
Words to be indexed: cow, jumped, over, moon
Word Signatures: 0001,0110,1100,0101
(assuming 4 bits per word)
Document Signature: 0001011011000101

Figure 6
Word Signature Encoding Method
with **m** = 4 bits per word, filter words excluded.

The two major advantages of this technique are that it is very simple, and the order of the words in the document is maintained in the signature file. The ordering can be useful in simplifying the search algorithm for terms in context queries. The size of the surrogate database containing the signatures should be much smaller (typically an order of magnitude or more) than the actual text database. However, as the entire surrogate database must be scanned in response to any query, a very high number of I/O operations may take place (for a 1 gigabyte document set, the surrogate database would be in the 100 megabyte range) which effectively rules out any implementation of this method in a real-time environment.

The "Superimposed Coding" (SC) Method.

This method [Fal84, Fal85b] uses the superimposed coding technique to create signatures for a "logical block" of text. In using this method, we must arbitrarily divide the document into "logical blocks" of D distinct, non-common words, as the method does not work well if the number of words represented by one "signature" is not constant.

The method is as follows. Each term of a logical block of text is hashed into a bit string of length F. These bit strings are inclusive-OR'ed together to come up with the signature of the entire logical block. The concatenation of the block signatures form the signature of the entire document (hence the other name for the SC method, that of Sequential Signature File, or SSF [Fal87c]).

The hashing of each word generates **m** one-bits in each word signature of size **F** (**m** and **F** are parameters to the method, and are based on the vocabulary of the document set). These two values also directly affect the number of "false drops", or the precision factor, of the method. m should be chosen such that for a block signature, the number of 1-bits is half of F. This is because the greatest number of combinations of signature values is

$$\binom{F}{F/2}$$

An advantage of this method is that it easily supports the handling of parts of words, so that the "search engine" can handle queries that specify incomplete words. Each word to be included in a block signature can be split into k triplets of characters, each triplet being hashed to one of m bit

positions (m must be chosen accordingly). If fewer than m triplets make up the word, then the other (k-m) bit positions are randomly set (to ensure that half the bits in the signature are '1's). The random number generator could use, as its seed, a numerical hash value of the entire word. If k > m, then m is ignored and k bits are set.

To handle a query, the query words (or word triplets) are hashed in the same manner as the document text, and are inclusive-OR'ed together. If the "on" bit positions of the query are also "on" in the block signature, that block is retrieved and its contents checked for the actual occurrence of the query words. If the block signature does not match, then that block can be ignored. Using Boolean logic, if the query signature is AND'ed with the block signature, and the result equals the query signature, then the block is a "match".

An example of the SC (or SSF) method:

Word	Signature
free	001000110010
text	000010101001
block signature:	001010111011

Figure 7
Superimposed Coding Method
with **D** = 2 (number of words per logical block is 2)
and with a signature size **F** = 12 and **m** = 4 bits per word.

One of the disadvantages of this method is that since the text is arbitrarily divided into logical blocks of words, complex queries are made more difficult. This is because the "search engine" must "remember" those matching bit patterns found in previous blocks when searching for keywords in an entire document.

The Run-Length Encoding (RL) Modification.

The run-length encoding modification [Fal85b] of the SC method compresses the signatures of each document to save storage space (and thus I/O) when performing the search.

With the RL modification, the signature is defined as in the SC method (the text is divided into logical blocks), but in this case we use a very large bit string of **B** bits (typically much larger than a typical signature size for SC). Here **m** is set to 1 (or, at least, a very small number). This gives the effect of creating a very large bit string, being made up of almost entirely "0"s. As a result, the bit string of **B** bits can be highly compressed using a run-length (e.g. Huffman) encoding technique, and the compressed bit string gives the signature size, **F**.

Note that if $\mathbf{m} = 1$, then an obvious compression technique would be to store the offset of each 1-bit set in the bit string. The number of bits required to identify any bit position is $\log_2 \mathbf{B}$ bits. For example, if the size of the bit string was 1024, 10 bits would be required to identify the position of each 1-bit in the 1024 bit string. If $\log_2 \mathbf{B}$ is sufficiently large, then it too can be compressed using the same technique.

A better compression technique would be to apply run-length encoding [Gol66,Gal75], that is to store the number of consecutive '0's between each '1' bit using a sophisticated encoding algorithm. A disadvantage of this method, though it does offer very high compression, is that in attempting to determine whether or not a given bit is a "1", the lengths of each previous "run" must be calculated. As a corollary, the next signature cannot be found unless the length of the current signature is calculated. Faloutsos [Fal85b] presents formulae for calculating the number

of bits required for the (compressed) signature, and the encoding scheme; they are not presented here.

text	0001 0000 0000 0000 0000 0000 0000 0000
retrieval	0000 0000 0000 0000 0000 0100 0000 0000
	---
block signature:	0001 0000 0000 0000 0000 0100 0000 0000

compressed signature (simple offset): 00011 10101
Run-length encoding: 0011 100001

Figure 8
The Run-Length Encoding method
with $\mathbf{B} = 32$, $\mathbf{D} = 2$ (number of words/block) and $\mathbf{m} = 1$

The "Bit-block Compression" (BC) method.

This method was developed by Faloutsos [Fal85b] in an attempt to deal with the computational overhead of the RL method. One of the problems with the RL method is that to determine if a given bit in a signature is a '1', the encoded lengths of all the preceeding intervals have to be decoded and summed. To solve this problem, both the BC method and the VBC method (following) rely on splitting the signature into several parts, with each successive part containing more and more detail about the contents of the document.

As in the RL method, the document is divided into logical blocks of **D** words. The bit string size is very large (**B**), and, again, **m** is very small (usually 1). This sparse bit string is now split into bit-blocks (groups of consecutive **b** bits). The value of **b** is important, as it directly affects the efficiency of the method. The formulae for determining the correct value of **b** follows.

To begin, the probability **w** (for each of the compression methods) that a bit in the *sparse* bit string is a '1' is:

$$w = 1 - \left[1 - \frac{1}{B}\right]^{mD}$$

(I)

Given **w**, the expected number of '1's in the *sparse* bit string is **Bw**. Once **b** has been determined, the method proceeds as follows: for each bit-block of size **b**, a signature consisting of three parts is created:

Part 1. The first part, consisting of 1 bit, defines whether or not any bits are "on" in the bit-block (i.e. a '0' if no bits are on, a '1' if at least one bit is a '1'). Note that if Part 1 of the signature is 0, the remaining two parts are unnecessary, and need not be stored. The parameter **b** should be chosen such that half of the parts are '0', and half are '1', as it is the parts that make up the signature size **F** (and not **B**).

Part 2. The second part defines the number (denoted by **s**) of '1's present in the bit-block. For simplicity, Part 2 is represented by **s**-1 '1's, followed by a trailing '0'.

Part 3. The third part defines exactly where in the block signature the '1's reside. This is achieved by storing, for each '1' bit, the $\log_2 \mathbf{b}$ bits of the offset from the beginning of the bit-block (note that since $\log_2 \mathbf{b}$ must be an integer, it follows that **b** must be a power of 2).

To illustrate, we use the following example:

free	0000 0000 0000 0010 0000
text	0000 0001 0000 0000 0000
retrieval	0000 1000 0000 0000 0000
methods	0000 0000 0000 0000 1000
block signature:	0000 1001 0000 0010 1000

If the bit groupings are as above (i.e. $\mathbf{b} = 4$) then the three parts of the block signature can be

formed as follows:

sparse vector:	0000	1001	0000	0010	1000
Part 1	0	1	0	1	1
Part 2		10		0	0
Part 3		00 11		10	00

Figure 9
The BC method, with bit-block size **b** = 4

This example is unnaturally small, as it saves only 4 bits over storing the entire (uncompressed) block signature. In a typical environment, the size of **b** would be large, and the size of **B** would be very large. The formula for the derivation of the signature size **F** is:

$$F_{BC} = \left\lceil \frac{B}{b} \right\rceil + Bw + Bw \log_2 b$$

(II)

Note that the signature size **F** dictates the average number of collisions, and hence the average number of false drops, that may occur.

Once the signatures for each logical block have been determined, two different methods for their storage present themselves:

a) the three Parts can be stored consecutively;

b) the Parts can be stored in three <u>different</u> files.

The latter technique speeds searching, as half of all of the Part 1's can be ignored (half of all the Part 1's should contain no 1's) and thus no Parts 2 or 3 exist.

The VBC (Variable Bit-Block Compression) Method.

Since the BC method arbitrarily divides a document into logical blocks (which must be treated as separate documents in their own right) it is desirable that the method be modified to remove such a constraint. With the VBC method, this is accomplished by fixing the size of the sparse vector (**B**) across all documents, but allowing the bit-block size (**b**) to vary for each document.

By having a different bit-block size for each document, two advantages are gained:

a) complex queries are easier, as there is only one block signature for the entire document (in the BC method, there are more block signatures to search for the larger documents).

b) different document sizes and vocabularies are accommodated and are optimally compressed on a case-by-case basis.

Once the optimal bit-block size $\mathbf{b}_{opt}$ is determined for the document (which depends on its vocabulary):

$$b_{opt} = B \frac{\ln 2}{w}$$

(III)

the signature is built as for the BC method ($\mathbf{m} = 1$). Since the format of the three parts depend solely on the document itself, the bit-block size **b** must be stored with the block signature (for each document).

To exemplify the VBC method, suppose we have two documents, named Q and V, with substantially different vocabularies (Q having the larger). As Q has a larger vocabulary, the value of **w** will be greater (remember **w** is the probability that a certain bit is a 1-bit). If **w** is greater, $\mathbf{b}_{opt}$ will be smaller. As **B** is constant for both Q and V, it takes more bit-blocks to represent Q than for V.

In summary, the signature for Q contains more Part 1's, a larger Part 2 (in proportion to Q's larger vocabulary), and Part 3 will be made up of more (but smaller) offsets into the bit-block (remember that each offset is stored as $\log_2 \mathbf{b}_{opt}$).

Multi-level Block Signatures (ML).

This method has been proposed by a number of authors including Pfaltz and Bergman [Pfa80] and Sacks-Davis et al. [Sac82]. They have been proposed to offset the disadvantages of the compression methods, namely:

a) Compression methods add computational overhead (though they do result in less storage being used).
b) There exists some difficulty in using commercial DBMS systems to store the compressed bit-strings.
c) Even with compression, the number of I/O operations required may be prohibitive to the development of real-time retrieval applications.

The ML method is basically as follows. Record signatures (denoted $\mathbf{F_r}$) are formed from the document using the SC method ("logical blocks" of text, consisting of **D** words), as described above. Block signatures are formed using a different (and typically larger) number of bits (denoted as $\mathbf{F_b}$). The block signature is stored in a bit-slice fashion, and represents the entire contents of the block of text. Optimally, all of the record signatures for any given block reside on the same physical page of secondary storage.

In order to answer a query using this two-level scheme, two signatures are formed from the query: one (of $\mathbf{F_r}$ bits) to match against record signatures, and another (of $\mathbf{F_b}$ bits) to match against the block signatures. For one-word queries, the algorithm is straightforward. Each block signature is compared to the query signature, and only if the block signature matches are the

record signatures for that block compared. A sample layout of the signature files is presented below:

Multi-Level Block Signatures (from Sacks-Davis et al.)

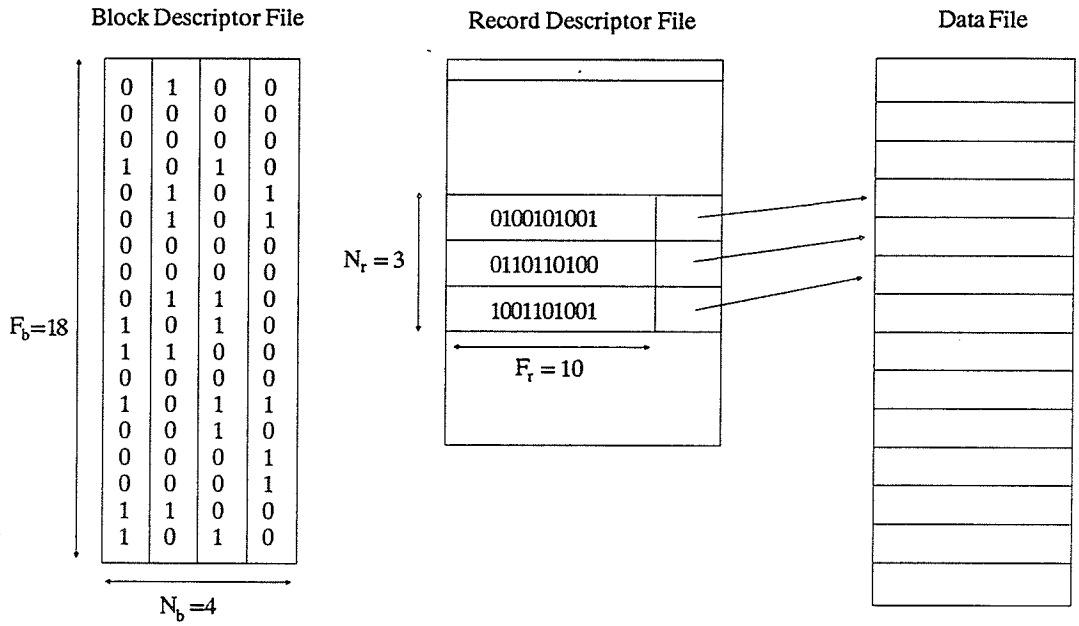


Figure 10
Layout of Multi-level Signature Files
(Example from [Sac87])

Multi-word queries, however, present difficulties. This is because a block signature can contain a bit pattern formed from a combination of words from more than one logical block, as:

a) words x and y are specified in the query, but exist in two different logical blocks.

b) words x and y form a block signature bit pattern matching a block signature, when in fact words x and y possibly do not even appear in the text.

This type of false drop is termed an "unsuccessful block match" by Sacks-Davis. A way of limiting the impact of unsuccessful block matches is by storing extra bits with the block signatures for pairs of words (and thus also support searching for word phrases). This can even be extended (like the SC method) to support the searching for parts of words.

Sacks-Davis [Sac87] has shown that, for a relatively small number (2000) of matching documents the multi-level method out-performs a one-level bit-slice method (described in the next chapter); though as the number of relevant documents increases (to greater than 5000) the one-level bit-slice method becomes more competitive.

A more elaborate scheme to prevent unsuccessful block matches is also proposed in [Sac87], and is based on analysis of the document set's vocabulary (which, unfortunately, must be done in advance).

Most of the problems stem from commonly-occurring words found in many documents (obviously somewhat document set specific). Sacks-Davis proposes that three sets of common words be identified (C_1, C_2 and C_3). The number of words in each set can be stated as $1 < C_1 < C_2 < C_3 <$ vocabulary of the document set. Each set is made up of words that have a rank in

a given range. The process of determining what ranges to use to establish each set is discussed shortly.

C_1 words are typically words in a stop-list. C_2 words usually number in the several hundred range (and can thus be explicitly stored); C_3 words number in the several thousand. C_3 words can be determined using a Bloom Filter, as shown below.

Vocabulary Analysis as Part of the ML Method.

To increase the effectiveness of the search (i.e. to increase precision), [Sac87] suggests constructing a Bloom Filter to identify and ignore common words. To determine the sets (C_1, C_2 and C_3) of common words, several passes are made upon a representative sample of the document set.

First Pass: A descriptor of length **F** with **m** bits set is formed for each unique word in a document. **F** integers keep track of the number of times a bit is set by any word.

Second Pass: A table of words and their frequencies are stored (typically 5000 or so words are tracked). For each word, the minimum of the counts for each of the **m** bits set by the word is also stored. This minimum count gives a rough estimation of the rank of the word. Words are added to the table as follows:

a) the table is searched, and if the word exists, its frequency is incremented.

b) if not found, we add the word to the table.

c) if the table is full, the count for the new word is compared with the lowest count in the table. If the new word count is greater, the new word replaces the existing word in the table.

From the frequencies of the terms in the table, the sets of words C_1, C_2 and C_3 can be determined. The descriptors for the C_3 words (created in the first pass) are then superimposed to form a Bloom Filter [Blo70]. When inserting documents into the system, C_1 and C_2 common words are ignored via the use of stop-lists; C_3 words are ignored if their word signature matches the filter, thus making explicit comparisons of common C_3 words unnecessary. A sample of the method is shown in [Sac87] where the value of **F** was 10,700 and **m** was 4. A problem with using the Bloom Filter to ignore C_3 words is that, due to collisions, useful terms may match the signature of a common term and thus be ignored. An approach offerred in [Sac87] to solve this problem is to use *multiple* filters, each with a different set of parameters of **F** and **m**. Only if a term passed through (matched) all filters would it then be considered a common term.

Note that this vocabulary analysis, and the use of the Bloom Filter, could be used for any text retrieval method. The Bloom Filter delivers, to some degree, the desirability of only indexing truly useful terms (though Sacks-Davis' method only covers frequently occurring terms). Note that the above procedure does not implement Salton's theory [Sal83] of comparing relative term frequencies across the document set. However, combining such filtering of index terms with other signature techniques should reduce the index size, improve precision, and offers a subject of future study.

3. The Doubly-Compressed Bit Slices (DCBS) Method.

Bit-slice methods have been studied in the past [Rob79,Sac87] but Faloutsos and Chan [Fal87c] have re-studied these methods for their applicability to optical disk technology.

Before we discuss the DCBS method in detail, it will be helpful to study a progression of bit-slice techniques. [Fal87c] has modified the basic bit-slice signature organization to incorporate the use of compression, creating new file structures. As these methods increase in complexity, they will lead to the design of the DCBS file structures. In order, we will discuss:

a) in Section 3.1, Bit-Sliced Signature Files (BSSF), the bit-slice adaptation of the SC method described in Section 2.7;
b) in Section 3.2, Compressed Bit Slices (CBS), an attempt to improve performance by reducing the size of the signature file;
c) in Section 3.3, Doubly-Compressed Bit Slices (DCBS), a technique to further reduce the size of a CBS organization but retaining the same expected number of false drops.

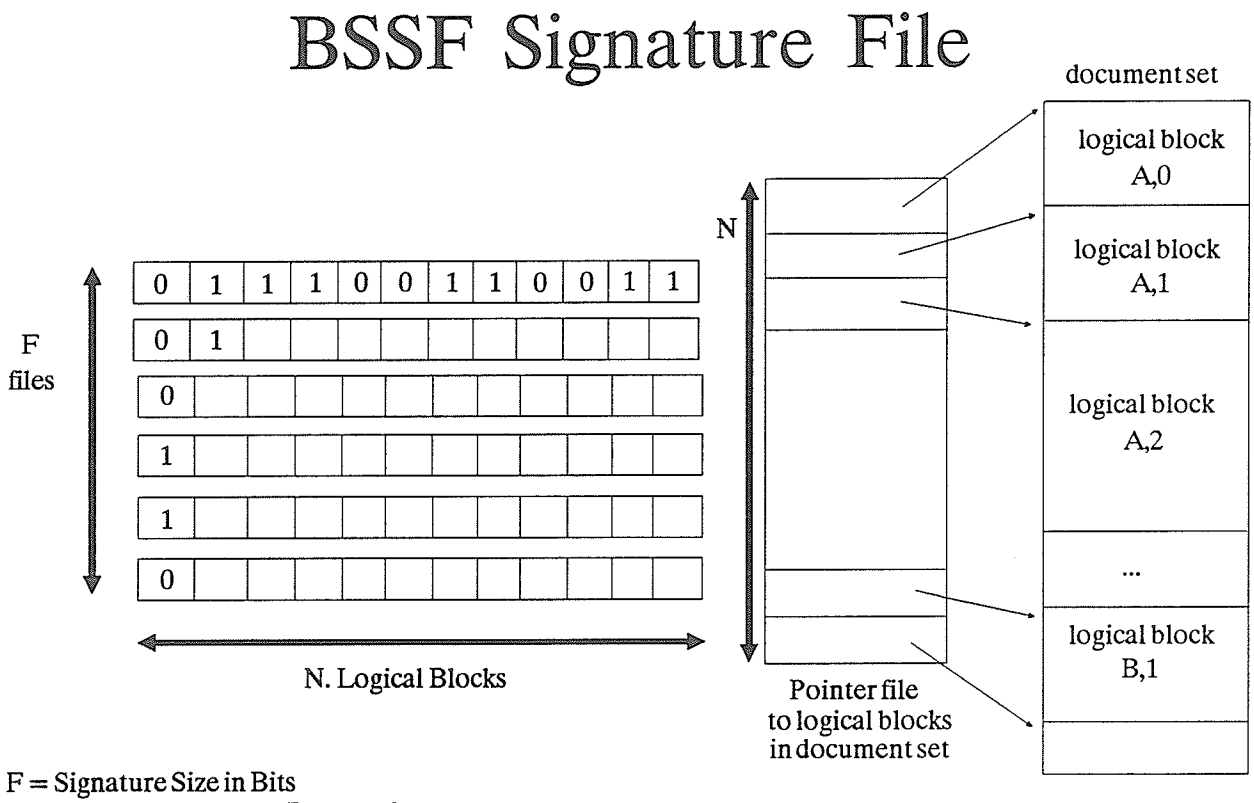
After the discussion of the DCBS file structures, Section 3.3.1 will discuss the method from a mathematical viewpoint. In particular, four performance measurements are analyzed: false drop probability, space overhead, and costs of both successful and unsuccessful retrieval (using strictly disk accesses as the only cost variable). This is followed in Section 3.3.2 by a brief description of a further modification to DCBS named NFD (for No False Drops) which approximates inversion (but using a hash table) and shows the relationship between inversion and these signature (hashing) methods quite clearly.

3.1 Bit-Sliced Signature Files (BSSF).

The purpose behind the BSSF method is to reduce the number of I/O operations required when performing a partial-match query on the SC method (see above). With the BSSF method, signatures are created as with the Superimposed Coding (SC) method (the mathematics underlying the two methods are identical). As with the SC method, **m** is the number of bits set per non-common word, and **F** denotes the signature size. However, as discussed in Section 2.6, with a one-level bit-slice technique the conceptual the "signature file matrix" is stored column-wise (one file per column) on the physical device; searching involves the retrieval of **m** bit vectors, instead of all of the **F** bit vectors. These **m** bit vectors are then AND'ed together to determine which logical blocks qualify as possible true drops.

To handle optical disks and allow for insertions, the **m** bit vectors are stored as **m** independent files. Thus inserting a record into the signature file requires only the addition of one bit at the end of each file, thus satisfying current optical disk limitations (WORM) (no rewriting of records). Unfortunately, the insertion of a new signature requires an update to each of the **F** bit-files.

A diagram depicting the BSSF method is shown below:



F = Signature Size in Bits
N = Number of Logical Blocks of text
Note how the document set is divided into logical, contiguous blocks of text; A, B represent documents.

Figure 11
An example of the BSSF File Structure
(adapted from [Fal87c])

3.2 The Compressed Bit Slices (CBS) Method.

This method [Fal87c] adapts the Variable Bit-Block Compression method (VBC) described above to a bit-slice file organization. Here, the technique of bit-slicing is used to reduce the amount of I/O required for searching.

The CBS method is meant to improve the retrieval performance of the BSSF method by enforcing the rule that $\mathbf{m} = 1$, so that for single-word queries only one bit-slice (column of the signature matrix) must be retrieved (as opposed to **m**). Also, BSSF suffers from (relatively) poor performance in inserting a new signature, as every bit-slice must be updated.

Conceptually, the CBS method involves the treatment of a sparse bit vector (the document's signature), stored bit-wise, as a sparse hash table containing the addresses (locations) of documents that (supposedly) contain the desired word. Since **m** is forced to 1 (so that the signature can be realistically compressed), then it follows that **F** must be increased, to maintain the same false drop probability. In [Fal87c] the author uses the letter **S**, as opposed to **F**, to represent the sparse record signature. In [Fal87c] the letter **S** was felt to be more representative of the (sparse) hashing technique employed. That convention is followed in this thesis, but the reader should realize that for the CBS method, **S** and **F** are synonymous. As will be seen, this is not the case for DCBS.

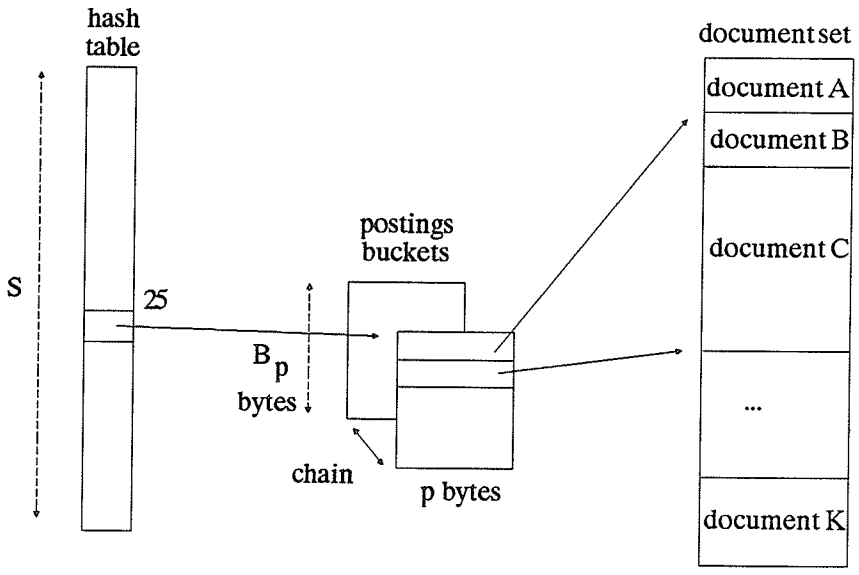
Thus, instead of storing a sparse bit-string of signatures (the 1's representing the appearance of a word in a document), a pointer to the document is stored in a "postings bucket", one pointer for each "on" bit position in the bit-slice. Since the number of pointers (1's) is unknown, the size of the "postings file" (as shown below) is also variable, and is hence stored as a linked list, with

each postings bucket made up of B_p bytes (a design parameter). Note that since a mapping is used to set a single bit in the signature, mapping collisions may occur, which will result in false drops upon retrieval. Also, if a word occurs more than once in a document, only the first occurrence occupies an entry in the postings bucket.

To handle the search for one word, the technique is as follows: the search word is hashed, the one bit turned on is found in the (sparse) hash table. The hash table entry points to the beginning of the list of postings buckets; each pointer contained within points to a (possibly relevant) document. To handle multiple-word queries, the lists of document pointers in the postings buckets must be merged; only documents that appear in each list would satisfy a conjunctive query.

In the following diagram note that, contrary to inversion, actual words do not exist in either the hash table, or the postings file.

CBS Signature File (Compressed Bit-Strings)



- S denotes the size of the hash table
- Note that the document set is stored as one contiguous stream of bytes on the physical media.

Figure 12
Illustration of the CBS Method
(taken from [Fal87c])

3.3 The Doubly-Compressed Bit Slices (DCBS) Method.

The DCBS method [Fal87c] was chosen as the file organization for the PMQ application. The DCBS method is very similar to the CBS method; basically it attempts to improve the CBS method by adding another level of indirection between the hash table of S (sparse) entries, and the "postings buckets" which contain the pointers to individual documents. The main advantage of DCBS over CBS is that of improved search retrieval speed (fewer I/O operations) with similar requirements in media overhead.

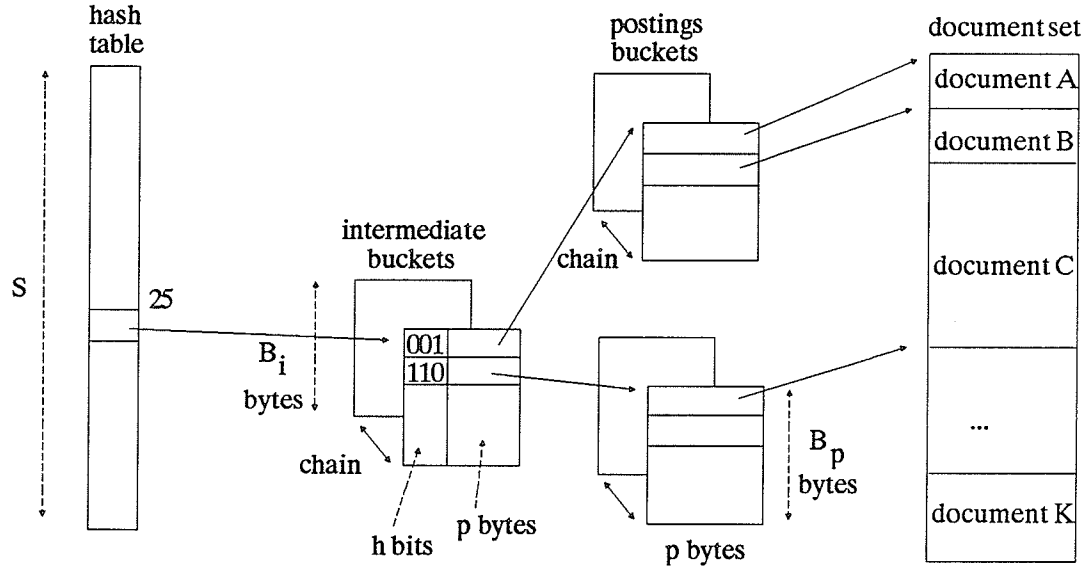
The DCBS file organization consists of three files: a sparse hash table (denoted by S), a file of "intermediate buckets" that differentiates between synonyms in S, and another file of "postings buckets" (like CBS) named the "postings file" to point to the document the word appears in. The sizes of these buckets (B_i and B_p respectively) are design parameters.

In DCBS, two hash values are used for every indexed term. The file of intermediate buckets differentiates among terms whose initial hash values (using hashing function h_1 for the hash table) have collided. A second hash function, h_2, is used to distinguish between these synonyms; this second hash value is what "keys" the intermediate bucket file. To determine which postings bucket should be used, the query word is hashed again (using h_2). Knuth[20] [Knu73] gives a detailed description of methods to compute h_1 and h_2, termed double hashing.

Note that the structure of the DCBS method is very similar to that of the CBS method (see Figure 12). The postings buckets would be exactly the same if the maximum hash value in the CBS method was $S2^h$ and the hashing function was made up of the concatenation of h_1 and h_2.

[20] [Knu73], pp. 521-526.

DCBS Signature File (Doubly-Compressed Bit-Strings)



- S again denotes the size of the hash table; $S2^h$ represents the signature size.
- Note again how the document set is stored as a contiguous byte stream.

Figure 13
Illustration of the DCBS Method
(taken from [Fal87c])

3.3.1 Mathematical Analysis of DCBS.

In this section we examine four measurements that describe the performance characteristics of the DCBS method:

a) false drop probability

b) space overhead

c) number of disk accesses for successful (and unsuccessful) retrieval

d) number of disk accesses for document insertion.

The formulae used in estimating expected performance are all from [Fal87c]. The following tables give an explanation of the variables used:

Symbol	Definition
L	average length of a document in bytes.
A	average number of documents a word appears in.
g	variance of word occurrence frequencies (A).
N	total number of documents.
$\mathbf{N_w}$	number of words in the document set.
$\mathbf{N_t}$	number of 5-letter word stems (tokens) in the document set.
D'	number of distinct words per document.
D	number of distinct non-common words per document.
V	vocabulary, or number of distinct words in the document set.
$\mathbf{V_{stem}}$	vocabulary of distinct word stems.
P	size of a disk page in bytes (e.g. 512 bytes)
b	number of bits per byte (typically 8).

Table 3
Input Parameters to DCBS.

Symbol	Definition
F	signature size.
S	size of hash table.
m	number of bits set by the encoding of a term (1).
B_p	size of postings bucket in bytes.
B_i	size of the intermediate bucket in bytes.
p	size of a pointer in bytes (4).
w	variable to denote a single word occurrence.

Table 4
Design Parameters of DCBS.

Symbol	Definition
O_v	space overhead.
c_1	chain length of intermediate file.
c_2	chain length of postings file.
F_d	false drop probability.
$d_{R,s}$	disk accesses on successful search.
$d_{R,i,s}$	disk accesses on successful search (index only).
$d_{R,u}$	disk accesses on unsuccessful search.
$d_{R,i,u}$	disk accesses on unsuccessful search (index only).
d_I	disk accesses on document insertion.

Table 5
Performance Measures.

Assumptions.

[Fal84] states that several mathematical assumptions can be made for most search environments which will dramatically reduce the complexity of the following formulae. These assumptions are:

Assumption 1: Large number of possible signatures (the hash table size is much greater than the average number of distinct, non-common words per document): $S >> D$.

Assumption 2: Large vocabulary: $V >> D$.

Assumption 3: Large document set: $N >> 1$.

Assumption 4: The number of documents greatly exceeds the average number of documents a word appears in, or $N >> A$. [Fal84] states that studies have shown that the 69 most common words appear 50% of the time; the 732 most common words appear 75% of the time. Given a large enough document set, this assumption appears valid. This also gives:

Assumption 5: A realistic assumption for the distribution of word occurrence frequencies is geometric (most words appear once, fewer twice, etc.) which gives the variance equation $g = A(A-1)$.

Assumption 6: Randomly selected words occur independently of each other in the document set.

False Drop Probability.

The formulas in Faloutsos [Fal87c] to estimate false drop probabilities were designed only for single-word queries. We continue with that assumption in this thesis. Simply put, the false drop probability F_d is equal to the probability that the signature of a non-qualifying document matches the query signature. This is the same as the formula:

$$F_d = \frac{\text{false drops}}{N - \text{actual drops}} \qquad \text{(IV)}$$

In all the bit-slice methods, superimposed coding is still used to give the document signature. However, unlike SC, the signatures are stored in a bit-slice fashion. Compression of the signatures is possible because **m** (the number of bits set in **F** per word) is set at 1. In DCBS, the second transformation h_2 only helps to separate synonyms into separate chains of buckets to reduce search time.

In effect, this means that signatures are created as in the WS method [Fal84,Fal87c]. Assuming independence of the words in each document, and a relatively large document set (which implies a relatively large vocabulary (Zipf's law)) [Fal84] showed that the false drop probability for WS for both successful and unsuccessful search is:

$$F_d = 1 - \left[1 - \frac{1}{S}\right]^D$$

(V)

This formula approximates **D/S** given the stated assumptions 1-6 above. This means that the false drop probability for unsuccessful search [Fal84]:

a) does not depend on the vocabulary size of the entire document set;

b) does not depend on the size of the database;

c) does not depend on word occurrence frequencies;

d) is not affected by word interdependencies.

For DCBS, the signature size **F** the multiplication of the hash table size (**S**) and the value 2 raised to the power $\mathbf{h_2}$. The false drop probability then becomes:

$$F_d = 1 - \left[1 - \frac{1}{S2^h}\right]^D \doteq \frac{D}{S2^h}$$

(VI)

Space Overhead.

The total space occupied by the DCBS file organization is the size of the sparse hash table, the intermediate bucket file, and the postings file.

Calculating the size of the hash table **S** is merely **S** * **p** (number of hash values time the size of a pointer). To estimate the space overhead of the intermediate and postings files, we need to determine the "average" chain length of buckets for each file. To do this, we note that each word in the document set will appear in **A** documents on average (**A** = **ND/V**). Note for single-word queries A is also equal to the average number of actually qualifying documents (true drops) for any search. First, we will estimate the size of the postings file, and then work on the intermediate file.

The chain length of the postings file for each word (**w**) is the number of times the word appears in distinct documents in the document set, multiplied by the pointer size **p**, and allowing for

pointers between buckets:

$$c_2(w) = \left\lceil wA \frac{p}{B_p - p} \right\rceil$$

(VII)

The average chain length for the postings file is then the summation over all the words in the document set (the vocabulary) of $c_2(w)$ above, multiplied by the probability that there are collisions:

$$\overline{c_2} = \sum_{w=0}^{V} \text{prob}(V, S2^h, w)\, c_2(w)$$

(VIII)

and prob($V,S2^h,w$) is the probability that a set of words w out of V possible words hash to the same value out of the range of possible hash values ($S2^h$). This probability can be calculated using the Binomial Theorem:

$$\text{prob}(V, S2^h, w) = \binom{V}{w} \left(\frac{1}{S2^h}\right)^w \left(1 - \frac{1}{S2^h}\right)^{V-w}$$

(IX)

if (we assume) the words are hashed uniformly (all hash values are equally likely). Thus the space overhead for the postings file is:

$$O_{v2} = S2^{h}\,\overline{c_2} \text{ buckets, or } S2^{h}\,\overline{c_2}\,\frac{B_p}{P} \text{ pages}$$

(X)

The space overhead for the intermediate file is calculated in a similar manner. **c1(w)** is the amount of storage needed for each synonym, which is **h/b** bytes for the $\mathbf{h_2}$ value, and **p** bytes for a pointer to the postings file. Again allowing for pointers between buckets, the formula is:

$$c_1(w) = \left\lceil w\,\frac{p + h/b}{B_i - p} \right\rceil$$

(XI)

and overhead for the intermediate file is then:

$$O_{v1} = S\overline{c_1} \text{ buckets, or } S\overline{c_1}\,\frac{B_i}{P} \text{ pages}$$

(XII)

As mentioned above, the size of the hash table is the pointer size (**p**) times the number of entries (S). So the formula for the theoretical overhead of the entire file organization is:

$$O_v = \left\lceil \frac{Sp}{P} \right\rceil + O_{v1} + O_{v2}$$

(XIII)

Unsuccessful Retrieval.

For the case of an unsuccessful search, we must retrieve the correct hash table entry (we assume 1 I/O operation to do this), and then follow the intermediate buckets (and the postings chain of buckets corresponding to the second hashing function giving $\mathbf{h_2}$), until we determine that all the drops are false). For the chain lengths, we use the averages $\mathbf{c_1}$ and $\mathbf{c_2}$ calculated above, so that:

$$d_{R,i,u} = 1 + \overline{c_1} + \overline{c_2}$$

(XIV)

Successful Retrieval.

With successful retrieval, we need to determine the average chain lengths, given that the chain contains the hash value for the desired word. If the search word has **w'** synonyms (**w'** + 1 words hash to the same value), then the length of the intermediate bucket chain is:

$$c_{S,1}(w') = \left\lceil (w' + 1) \frac{p + h/b}{B_i - p} \right\rceil$$

(XV)

and the probability prob(V-1,S,w') that a word has w' synonyms is again:

$$\text{prob}(V\text{-}1,S,w') = \binom{V-1}{w'} \left(\frac{1}{S}\right)^{w'} \left(1-\frac{1}{S}\right)^{V-w'}$$

(XVI)

Thus the corresponding chain lengths are:

$$\overline{c_{S,1}} = \sum_{w'=0}^{V-1} \text{prob}(V\text{-}1,S,w') \left\lceil (w' + 1) \frac{p + h/b}{B_i - p} \right\rceil$$

(XVII)

and

$$\overline{c_{S,2}} = \sum_{w'=0}^{V-1} \text{prob}(V\text{-}1,S2^h,w') \left\lceil (w' + 1)A \frac{p}{B_p - p} \right\rceil$$

(XVIII)

and the total, then, is:

$$d_{R,i,s} = 1 + \frac{1 + \overline{c}_{S,1}}{2} + \overline{c}_{S,2}$$

(XIX)

Note that in the middle term we assume that we need only (again on average) search half of the chain of buckets, as the search is successful.

Insertion.

Insertion is the most expensive operation that can be performed using DCBS. Basically inserting a word into the signature file involves performing a search for each word in the document. Four cases are possible:

1. If the search is successful (a postings bucket for the correct combination of $\mathbf{h}_1$ and $\mathbf{h}_2$ hash values points to the document being inserted) then no updates need be done; the word is already indexed.
2. No postings bucket for this document-word combination exists. A postings bucket entry must be added to the postings chain.
3. No intermediate bucket entry exists for the $\mathbf{h}_2$ value of the word. Here, an intermediate bucket entry must be inserted, and a new postings bucket created that contains this single entry (to the document being indexed).

4. The hash table value for this word is NULL. Then an intermediate bucket and postings bucket must be created, and chained together. The hash table is updated to point to the new intermediate bucket.

Faloutsos [Fal87c] gives a formula for estimating the average number of I/O operations per document inserted. However, the formula assumes that the vocabulary of the document set remains constant (the new document adds no words to the vocabulary). This implies that the document set must be sufficiently large.

If no words are added to the vocabulary, then only cases 1 and 2 above are possible. This is the same as executing a successful search for each word in a document, and for those words where a postings entry is not found we must update the last page of the postings chain:

$$d_I = D\left(d_{R,i,s} + 1\right)$$

(XX)

The formula assumes a document contains **D** words on the average. We will compare the results of this simplistic formula to actual model office results in Chapter 5.

3.3.2 The "No False Drops" (NFD) Modification.

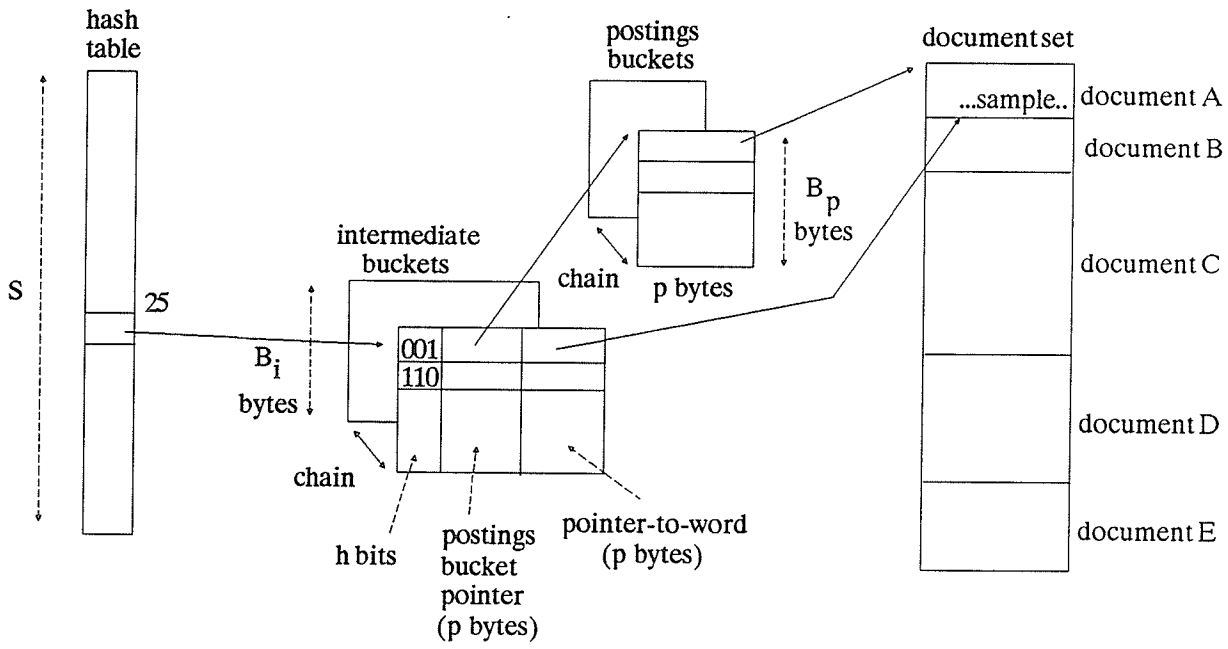
The "No False Drops" (NFD) method is a modification of DCBS, but was not implemented as part of the PMQ application for technical reasons, as will be explained. The technique avoids false drops completely, at the expense of substantially increased space overhead. This method is like the DCBS method, in that two levels of buckets are used; one (the set of intermediate buckets) to differentiate between hash-value synonyms with $\mathbf{h_1}$, and the second being the postings buckets which point to the individual documents (see Figure 14).

The motivation behind this method is to nullify any occurrence of synonyms with the h_2 transformation. Intuitively, storing the actual word with the intermediate bucket entries would be sufficient to reduce the number of synonyms to 0. However, instead Faloutsos proposes storing a pointer to the word ("word ptr" in Figure 14) in (any) document; this pointer value is, by its nature, already unique and satisfies the requirements. Space is saved as well, since the length of a pointer variable is (typically) four bytes, whereas the average length of a (non-filter) word is approximately eight bytes [Fal87c].

What should be mentioned here is that although the intermediate buckets contain a pointer to the word to save space, this technique alone is probably unsatisfactory. The problem is that it is the word stem that should appear here; otherwise, as previously discussed in Section 2.2, the user must know (absolutely) the particular spelling, tense, context, etc. of the query word to be used. The BSSF, CBS, and DCBS methods could easily support the use of word stems, since only signatures are used in those three methods, and the text used to form the signatures in the first place could be the word stems themselves. The user could even be given the capability to search for an actual word spelling in this case; the (extra) cost being the addition of false drops for those words that do not exactly match.

One additional constraint that did not permit the use of NFD for the PMQ application was that in the NFD method the document set is considered to be one contiguous file. In MS-DOS or UNIX file systems, such is not the case (each document is typically in its own file) and thus the word pointers would of necessity be larger than **p** bytes (somehow the file structure must identify the document the word appears in).

NFD Signature File
(No False Drops)



- Again, $S2^h$ represents the signature size.
- Note that pointer-to-word values can be unique if the document set is physically stored as a stream of contiguous bytes.

Figure 14
Illustration of the NFD method.
(taken from [Fal87c])

4. Partial Match Query Application.

This chapter introduces the Partial Match Query (PMQ) application. Here we describe the features, components, and interfaces of the overall software architecture of PMQ, as well as the environment in which the application operates.[21]

In the first section of this chapter we present an overview of PMQ and describe its notable features. The main functions that PMQ provides are:

a) full text indexing of Enable and ASCII-text documents,

b) retrieval of Enable or ASCII documents by content,

c) an integrated document browse capability.

In the second section we discuss the logical structure of a typical document in the system, as perceived by the workstation user. Section 3 describes the mechanics of the user interface component, that is the architecture of the menuing system. The user interface component is divided into several functional interfaces (query specification and results display, indexing, browsing, and reorganization). In Section 4, a description of the document indexing interface is given, detailing the interaction with the user and providing a "flavor" to the reader as to how PMQ is used. In Section 5 we discuss the retrieval (query specification and results) interface of the system, which assists the user in interactively specifying a query and pinpointing relevant documents. In Section 6 we discuss the file structure of the index, modelled after the DCBS method given by [Fal87c]. In that section we describe how the implementation aspects of PMQ

[21] By environment we refer specifically to the computing environment in which the software executes. See Chapter 1 for a discussion of the model office environment.

allow the system to deliver the functions described in the previous sections. We also discuss various implementation trade-offs that were made during development of the application. Finally, in Section 7 we present a list of possible future enhancements.

All of the software components and interfaces (access method, user interface component, indexing interface, query specification interface, document browsing interface, and index reorganization facilities) described in this chapter are fully operational and are being used by schools in the Pine Creek School Division.

4.1 Introduction.

In the model office environment, documents are typically created by the secretary of the School on behalf of any member of the staff. The documents are drafted using Enable word processing software resident on the workstation, and are archived on the fixed magnetic disk in one of the operating system (MS-DOS) directories. These documents are usually meant for use within the School where they are created. In some cases, documents are sent electronically to other Schools or Division offices (each of the PC's is equipped with a modem). However, each workstation is stand-alone in the sense that a distributed document management (distributed database) function is not supported.

PMQ is a stand-alone MS-DOS application that assists in the management of word processing documents created using Enable software (although ASCII text files are also supported). Conceptually, documents consist of both attributes and free-format text. Using PMQ, the user of the workstation can index already drafted documents and specify queries to find relevant, existing documents that have been previously indexed.

PMQ is not integrated with Enable, as Enable is a licensed, proprietary software product. PMQ executes on any Intel-based personal computer that is IBM-XT or IBM-AT compatible running MS-DOS 2.0 and above. The index files are stored on the fixed magnetic disk of the PC, as are the documents themselves. PMQ is specifically designed to support monochrome displays without the use of a pointing device (mouse). Currently, the version of PMQ in use utilizes memory-mapped screen I/O. If desired, a BIOS version could be used instead (only a re-compile of the source is required). The user interface functions are all driven by menu selection; overlapping windows are used for display.

At this point we describe an overview of the features of PMQ. How these features are supported by the access method and software components is discussed in subsequent sections. The system features include:

1. The parameters of the index file structure are completely variable, and can be uniquely specified for each installation. These parameters can be set to match the document set characteristics for the particular workstation.
2. The index file structure is based on the DCBS signature method as described in [Fal87c]. The DCBS method was chosen because its characteristics include excellent retrieval performance with low space overhead.
3. MS-DOS files containing Enable documents may be browsed in ASCII format to assist the user in indexing the correct documents, and determining the relevance level of documents returned by a query.
4. The user may specify conjunctive (AND) queries on any combination of multi-valued attributes (author, document title, and restricted vocabulary indexing terms[22]) and full text.

[22] We will use the term "keyword" for brevity in subsequent sections.

5. Terms in context (word adjacency) searching is supported (for both attributes and full text) by the use of post-coordination techniques and full text scanning. Word truncation (each word is reduced to its first five characters) is used so that the actual spelling of any word (in the context of any document) need not be known beforehand.
6. A stop-list of 124 words is used to filter common words from the indexing process, reducing the size of the index.
7. PMQ supports the display of documents' MS-DOS filenames contained within an MS-DOS directory. This list may be sorted by MS-DOS filename, or file extension.
8. Index and query statistics may be displayed at the discretion of the user.
9. The user interface component supports menu selection via the use of cursor keys, or individual keystrokes (speed selection). Online help is available for each menu and its selections.
10. PMQ is written in the C programming language. Its use of layered screen display routines and keyboard mapping facilitates porting to other environments in the future.

4.2 Logical Structure of Documents in PMQ.

PMQ supports retrieval of textual documents using two types of data, text and attributes. However, Enable does not directly support the specification of document attributes; therefore this function is provided by PMQ. Abstractly, the tuple representing a document (consisting of attributes and text) can be treated as a unit. This section describes the model of document tuples in PMQ and how this data can be indexed and searched.

A document can be composed of zero or more multi-valued attributes and free-format text. Attributes in this logical model are named and have a data type associated with them, as in a relational database. The attributes supported by PMQ are:

a) document author;

b) document title;

c) document format;

d) document keywords;

e) date and time of last update;

f) date and time indexed.

In almost all cases these attributes are redundant data; most often the information contained in an attribute can be found in the text itself, or in the native file system. For convenience, these attributes are stored redundantly in a more consistent format. The purpose behind attributes is to store information that can be used for content addressability. Attributes can be more useful than full text (see Section 2.2) in reducing the scope of a given search.

Attribute information is specified or extracted when a document is indexed. For example, the author of a document can be specified by the user when the document is indexed; author information is stored as the multi-valued attribute "Author". Attributes cannot be altered once the document has been indexed. Should the need arise to change an attribute value, the document may be re-indexed, at which time the new attribute values can be specified. Thus, the information in the PMQ index represents only a "snapshot" of the document set. When a document is altered the user must ensure that the PMQ index is synchronized.

PMQ was developed so that documents created with other word processing software (other than Enable) could be supported in the future. This means that automatic extraction of attribute values is difficult, as documents of different types may contain like attributes in a different format, a combination of data fields, or not at all. For this reason, the user is prompted for most attribute values. A notable exception to this is the use of each document's date and time of last update. Each document's last update date-time value is retrieved from the MS-DOS operating system, and compared to the value kept in the PMQ index structure. If the value differs, then the document has been updated since it was last indexed, and the user is prompted to re-index the document. Similarly, if the date-time values are equal, then there is no need to re-index, and an appropriate warning message is given to the user.

The free-format text portion of the document tuple may include a title, author, abstract, chapter and section headings, etc. PMQ treats all text within the document as possible index terms; no differentiation is made between document components. Searching using "terms in context" is supported for full text, and for attributes.

Note that PMQ does not support indexing or searching by "last update date and time", or "index date and time". A search of this type involves the use of range queries [Fal89,Ore82,Dat86]. A partial match query results in a true drop if an exact match if found; range queries can produce a true drop using inequality comparisons ($<, >, <=, >=$). Range queries are desirable for searching by date or time, when an exact match query may be difficult to compose. These techniques have yet to be investigated for bit string organizations [Cha89b]. Also, PMQ does not index attribute values in any special way, though searching using attributes can reduce the scope of a search. Although it is recognized that query distributions are usually skewed (80/20) towards the use of attributes, query distribution characteristics were unknown at the time of

development. Like range queries, techniques to encode more information for attributes [Fal87b] have only been defined for bit string (not bit-slice) methods [Cha89b].

4.3 User Interface Component.

In this section we describe the architecture of the user interface component of PMQ. The user interface provides an interactive, responsive medium enabling the user to index and retrieve documents quickly.

The software component that implements the user interface is itself comprised of subcomponents called managers. These managers are software routines that provide screen-handling and keyboard code translation services to the functional user interfaces. The managers are:

a) physical screen manager;

b) window manager;

c) virtual screen manager;

d) keyboard manager;

e) menu manager;

f) help manager.

The first four managers are from Rochkind [Roc88] and are used by PMQ routines. These screen managers, and the keyboard manager, are discussed in detail in [Roc88]. They will be described here only briefly. The latter two managers were specifically developed for PMQ and will be discussed in more detail.

Physical Screen Manager.

The physical screen manager is the lowest-level layer of the display managers, and is responsible for controlling the lowest-level interface to the controlling screen hardware. Thus, a specific set of physical screen manager modules must be linked into the application for each supported workstation (or workstation display hardware interface). With the layered approach, such hardware-dependent application programming interfaces (API's) (an example is memory-mapped screen I/O for Intel microprocessors) are "hidden" from the higher layers. For each destination environment, the application code remains unchanged.

The Physical Screen Manager supports only character-based monochrome displays, but does support the display of normal-intensity, reverse-video, underlined, and intense-video characters.

Functions provided by the Physical Screen Manager include PSwrite (write characters to the screen), PSfill (fill a portion of the screen with a given character), PSslide (scroll data in any direction on the screen), PSsetcur (show or hide cursor), and PSsynch (synchronize memory contents with the hardware display).

Window Manager.

The window manager routines are responsible for the display of windows (rectangles) on the display. The characteristics of a window are quite similar to that of the physical screen. The significant difference is that a window comprises only a subset of the physical screen character array and may have an optional border defining its dimensions it a visible manner.

Unlike the physical screen manager, the window manager allows the higher level routines to create new windows, dispose of old ones, change window size or location, hide existing windows, and overlap the windows on the screen. However, responsibility for redrawing portions of the screen previously obscured by another window remains with a higher interface layer.

Window Manager routines include: Wnew (make a window and display it on the screen), Wdispose (remove a window), Wzoom (zoom a window to full physical screen dimensions), Wwrite (similar to PSwrite, but for a window), Wfill (like PSfill), Wslide (like PSslide), Wgetphys (retrieve window location relative to physical screen) and Wsetfrm (set the characteristics (framed or unframed, titled or untitled, frame attribute (normal, reverse-video)) of a window's frame).

Virtual Screen Manager.

A "virtual screen" is a matrix of characters in memory, possibly even larger than the dimensions of the physical screen. A virtual screen is mapped to the physical layer by the window manager; thus the location and dimensions of the window on the display are completely independent from the dimensions of the virtual screen. The virtual screen manager also performs the redrawing function for overlapped windows.

Examples of functions provided by the Virtual Screen Manager are VSfill (fill virtual screen with a character), VSpan (pan a window relative to a virtual screen), VSnew (create virtual screen), VSdispose (destroy virtual screen), VSwrite (copy data to virtual screen), VSgetwloc (get window location relative to virtual screen) and VSslide (scroll virtual screen).

Keyboard Manager.

A major problem in developing an application that can be ported to different hardware environments are the different workstation keyboards that may be used. Even with "standard" IBM-PC's, several different keyboards may be purchased. Most keys (such as the numbers 0-9 and the alphabet) are standard; however, special key codes and rarely-used ASCII characters may not appear on some keyboards.

Software that processes special keyboard sequences for control purposes (combinations like F1-F10, or CNTL-ALT-ENTER) must also work with keyboards that cannot generate those particular character codes if the application is to be truly portable.

The Keyboard Manager from [Roc88] handles all possible keyboards by translating character codes entered from the keyboard into a virtual key code. Only these virtual key codes are processed in the application. The transformation is processed by a deterministic finite-state automaton. A table which defines the transformation sequences is compiled at each application initialization. The table is easily modified so that a keyboard code that does not exist can instead be generated by another sequence of keystrokes; thus the application program need not be modified.

The Keyboard Manager consists of the functions Kbegin (compile the transformation table) and Kget (get next virtual key code by executing the automaton).

Menu Manager.

The Menu Manager is composed of several routines that provide menu generation, processing, and error handling services to PMQ functional interfaces.

Each application functional interface that requires a menu builds a menu structure that contains, among other items, the names of the k menu options, help information, window location coordinates, and the addresses of the k functions required to process each distinct menu option. This structure is passed to the Menu Manager which:

a) builds and displays the window on the screen;

b) ensures that the current window has a double-line border, and all other windows have single-line borders;

c) processes keyboard input (virtual key codes) so that the user may:

i) invoke a routine in support of a menu option (speed selection, ENTER key, or LEFT CURSOR key);

ii) quit the current menu (ESC key);

iii) receive help for the menu (F2 key);

iv) receive help for the menu option (F1 key);

v) return to the main menu (F10 key).

Note that any PMQ function called by the Menu Manager may, in turn, recursively call the Menu Manager to create a leaf menu in the menu tree. As the cursor moves from menu option to menu option, a short help message is displayed on the last line of the physical screen corresponding to that option.

Functions provided by the Menu Manager to functional interfaces are MUassignpick (build a structure corresponding to a menu option), MUconfirm (create a "confirmation" window with two menu options, "yes" or "no"), MUmanager (display and process keyboard input for this menu), MUdisplaybox (display a character string in a window), MUdisplayboxack (display a string and wait for keyboard input until destroying it) and MUpromptline (prompt the user to input a string).

Help Manager.

The Help Manager provides menu and menu option level help windows to the workstation user upon request. The Help Manager is called by the Menu Manager whenever a help virtual key code is encountered (keys F1 and F2 for menu options and menus, respectively).

The help text is stored externally from the PMQ application in a sequential file. Help entries are named so that they can be referred to by the functional interfaces. When the Help Manager is invoked, the required help entry is searched for, by name. If found, the help text is displayed in a new window whose coordinates are proportional to the size of the help message.

These six Managers provide API services to function interface routines that make up the bulk of the PMQ application. We now discuss, in the next two sections, two of these functional interfaces: indexing, and retrieval.

4.4 Indexing Functional Interface.

The overall approach of PMQ to indexing documents is to give the workstation user flexibility in how the document set will be managed, but limit the application's complexity as much as

possible. A workstation user can define his own indexing strategy for managing his document set (he may, for example, not wish to use document keyword attributes for searching). Also, it is the choice of the user whether or not a document is actually indexed (this is a contrary approach to products like Lotus' Magellan, which will index everything on a workstation's fixed disk).

When a workstation user desires to index a particular document, he will invoke the PMQ application, and will be prompted for the full MS-DOS pathname in which the document resides. PMQ will display the list of documents within that MS-DOS directory (see Figure 15 below). PMQ correlates the MS-DOS directory list with the MS-DOS filenames of any documents previously indexed, and displays the date and time indexed for those documents. With the list, a menu is shown that enables the workstation user to browse through the list, display any document in the list, or sort the list to suit his needs.

When the document to be indexed has been found, the document is highlighted in the list, and the user is prompted for the attributes to be associated with this document (see Figures 13 and 14). These attributes may be specified in any order. The most important attribute is document type, as PMQ has different full text scanning software for ASCII files and Enable word processing files, due to their different internal formats.

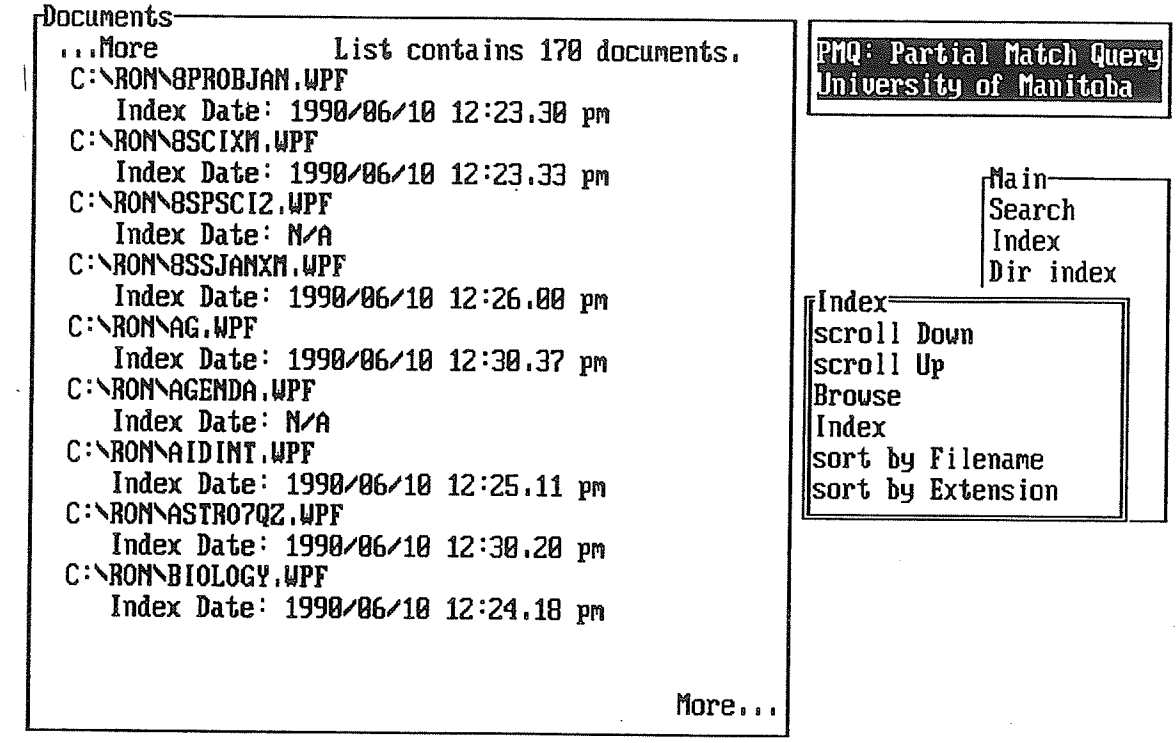


Figure 15
Indexing Menu with Document List.

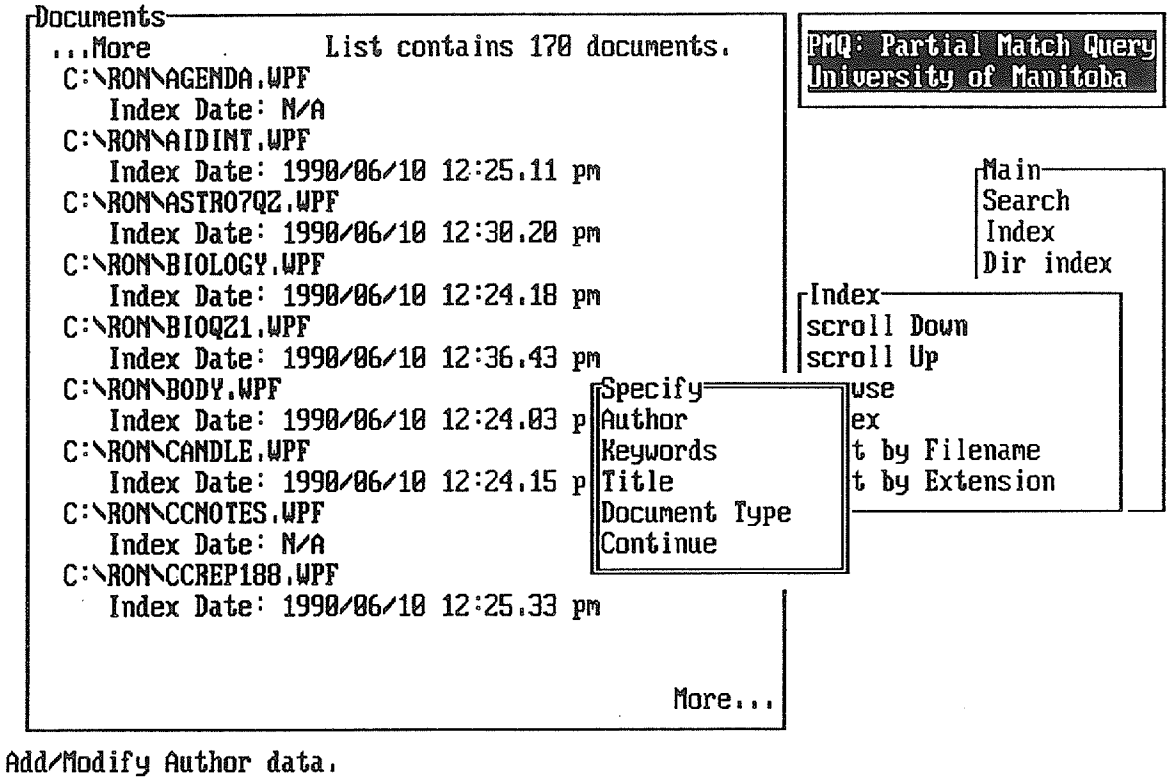


Figure 16
Attribute Specification Menu.

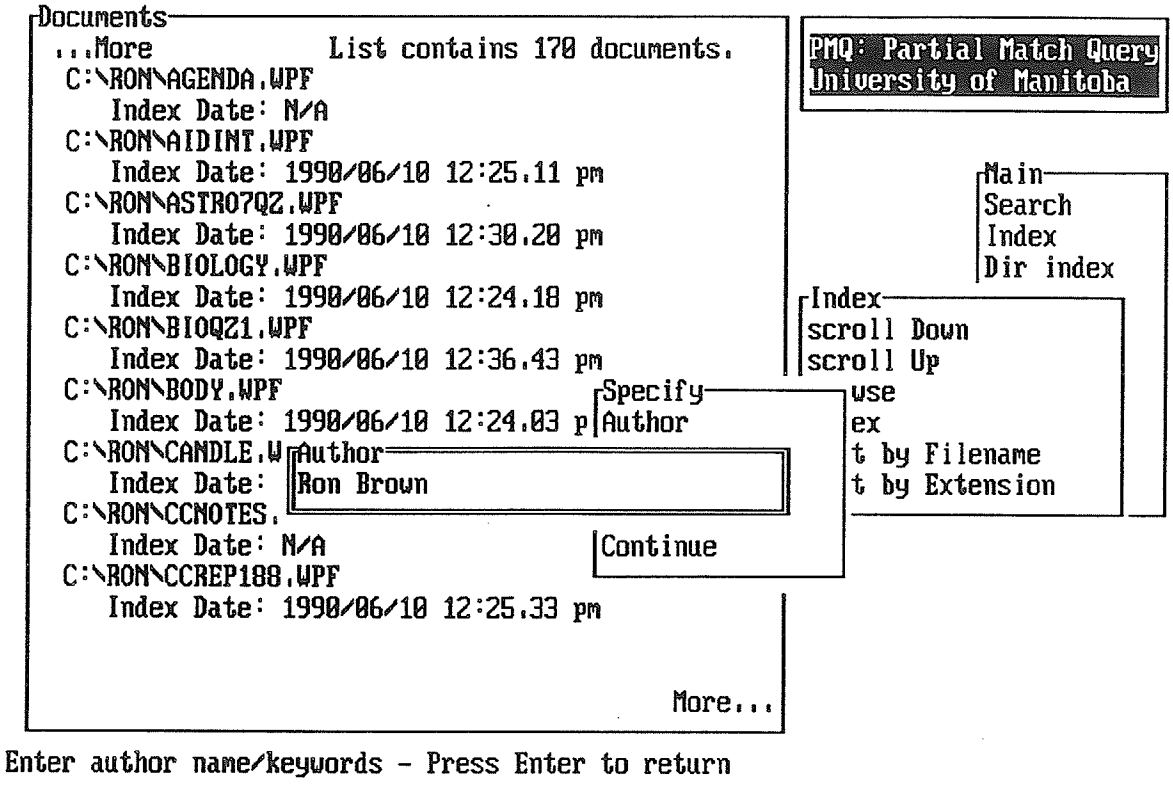


Figure 17
Specifying the Author Attribute for a Document.

When all suitable attribute information has been entered, the next step is to select the "Continue" menu option, which will start the indexing process. A window is displayed describing the progress of the indexing procedure; when complete, indexing statistics can be displayed (Figure 18).

If a document that has already been indexed is indexed again, the workstation user is prompted to confirm the operation. If so, the original index entry is logically deleted from the index, and the indexing function continues.

When the indexing process has completed successfully, the index date and time attribute is displayed with the document name in the list. The workstation user may then select another document within the same list, or return to the main menu and select a different main menu option.

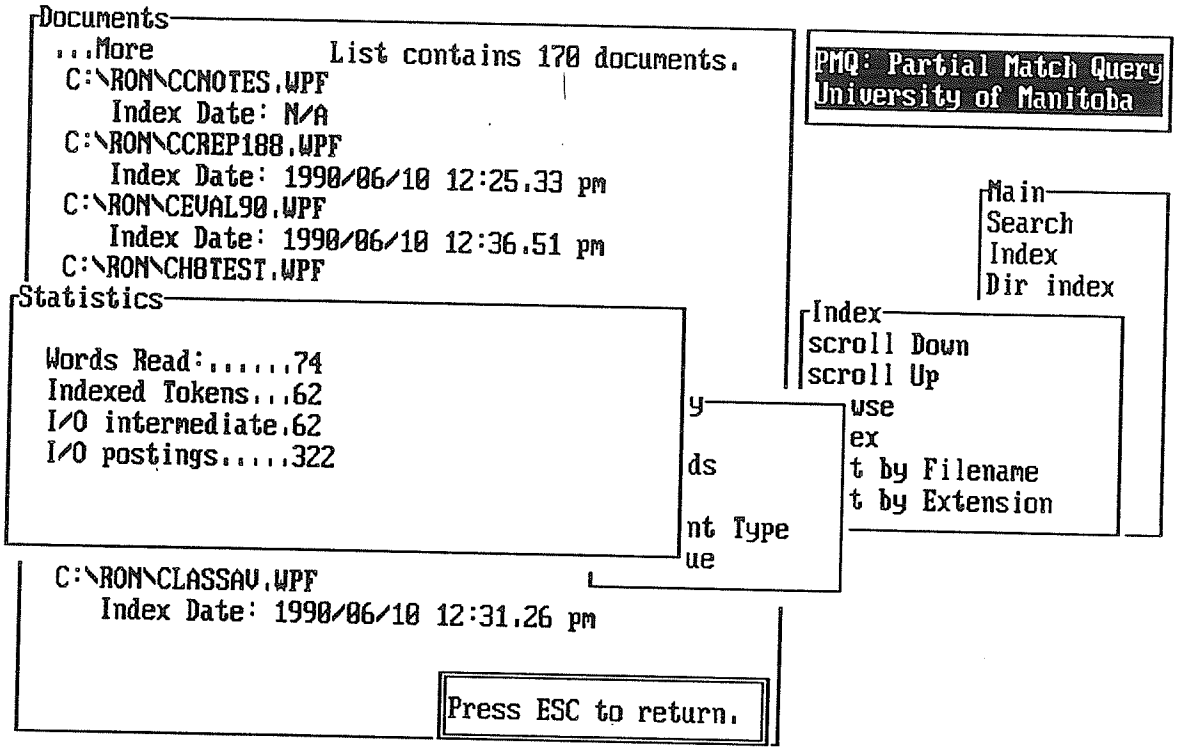


Figure 18
Display of Indexing Statistics.

4.5 Retrieval Functional Interface.

In this section we describe the PMQ functional interface that supports queries against the document set. First, we discuss the various search criteria supported and how these relate to the workstation user's indexing strategy. Following, we describe the menu dialogue that takes place during execution of a query, and give a brief overview of the Browse functional interface that enables the user to determine document relevance.

Search Criteria.

A document search begins with the workstation user selecting the criteria for the search, which may consist of any combination of full text and selected attributes (author, keyword, and document title). Queries are conjunctive only (no disjunctive queries are supported).

A separate search string is used for each attribute, and full text. Initially, these strings are NULL (if a string is NULL then these search criteria are ignored), but the strings are saved from search to search to limit unnecessary re-typing. Within each string, PMQ allows the use of "wild-card" characters ('*') and the construction of term groups for searching by terms in context.

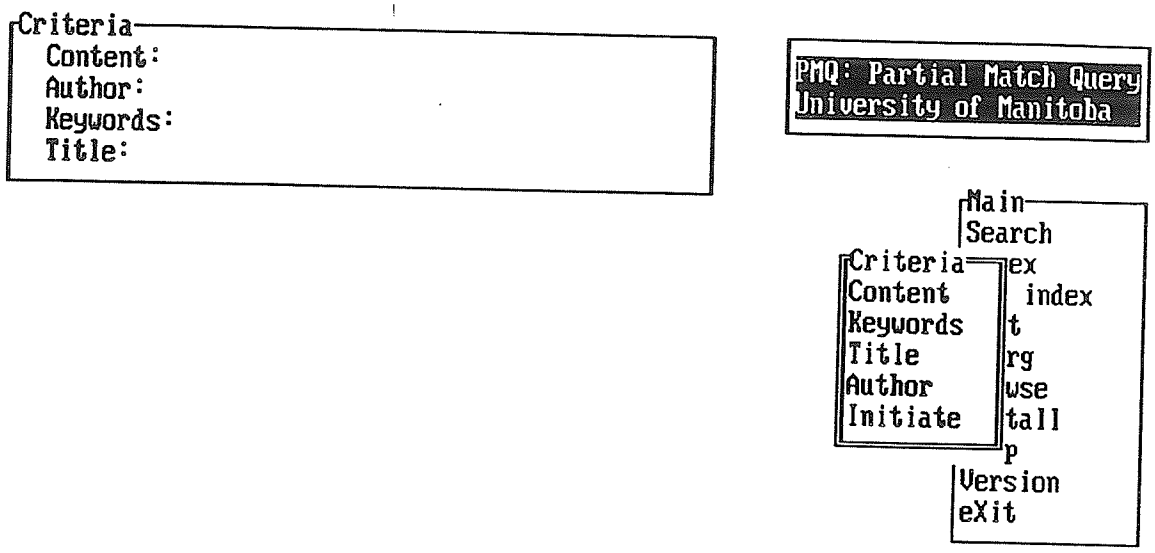
"Wild-card" characters are used when the particular spelling or suffix of a word in the document may not be known (wild-cards may only be specified at the end of any search term). Thus, the use of the search term "exami*" will find all occurrences of the term "examination" in the document set, but also "examine" and "examiner". A wild-card may be used on any term in a query string consisting of at least five characters, and may also be used for terms within term groups.

PMQ supports searching by terms in context. To specify a term group, the workstation user delimits the phrase with '<' and '>' characters (see Figure 19). PMQ uses post-coordination of the individual terms in the term group (and full text scanning of the documents) to determine if the term group actually exists in any particular document.

Query Dialogue.

The search process begins with the specification of search criteria, as outlined above (Figures 19 and 20). Search criteria are displayed in the window named "Criteria" as they are entered. Note in Figure 20 the use of both term groups and wild-card characters in the "Search by Content" string.

Once the criteria have been entered, the search process is started with the selection of the "Initiate" menu option. PMQ's search process is a two-step affair. Firstly, a query signature is created from each term in the search criteria and is compared to the term signatures in the index. If no signatures in the index match the query, then the query fails and an appropriate message is displayed. If at least one document's terms match the query signature, then the second phase of the search is begun. This second phase uses full text scanning to ensure that the document is a true drop. Once the second phase is complete, statistics are optionally displayed (Figure 21). Figure 22 shows the list of those documents that represent true drops. Like all other document lists in PMQ, this list may be sorted as desired, scrolled, document attributes displayed (Figure 23), and documents themselves displayed in ASCII format (Figure 24).



Enter string to search by document content.

Figure 19
Search Criteria Menu.

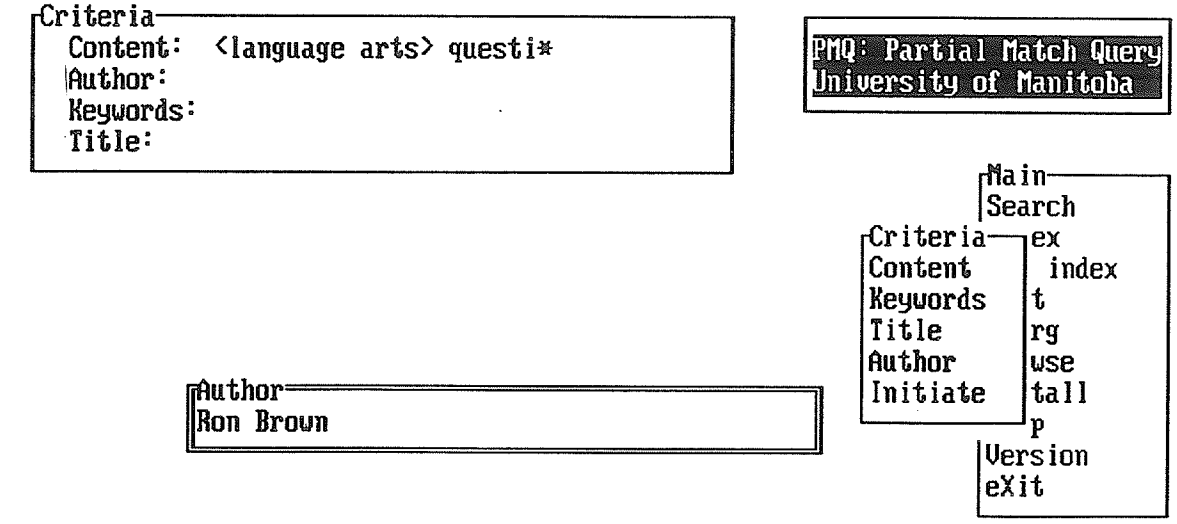


Figure 20
Specification of Search Criteria.

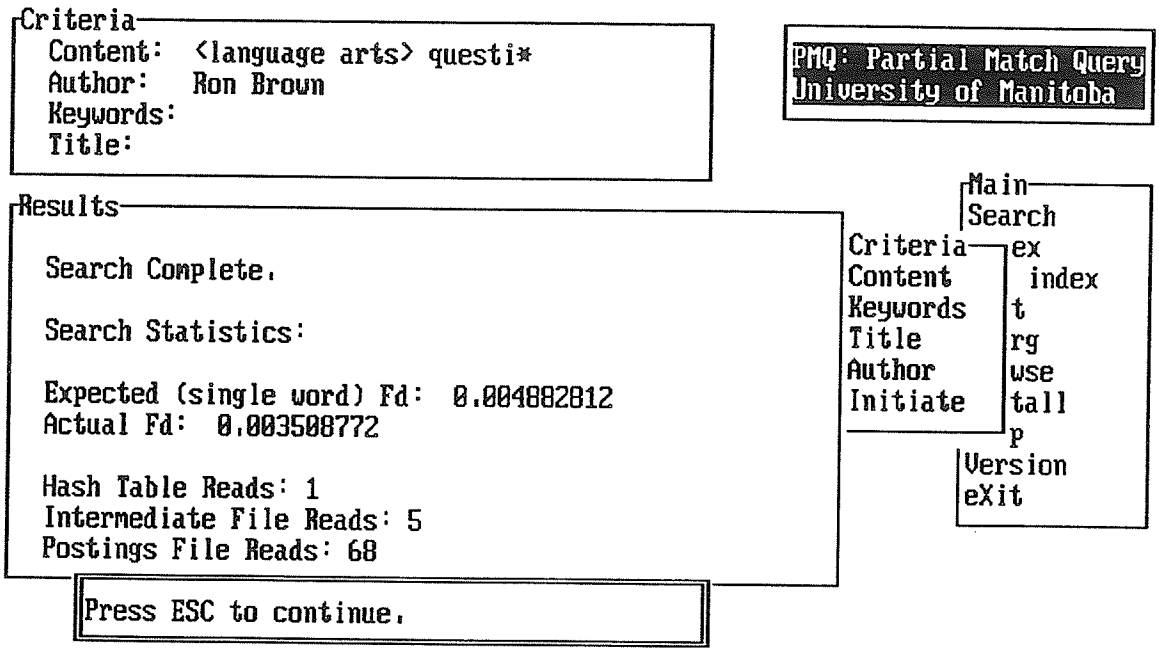


Figure 21
Display of Query Statistics.

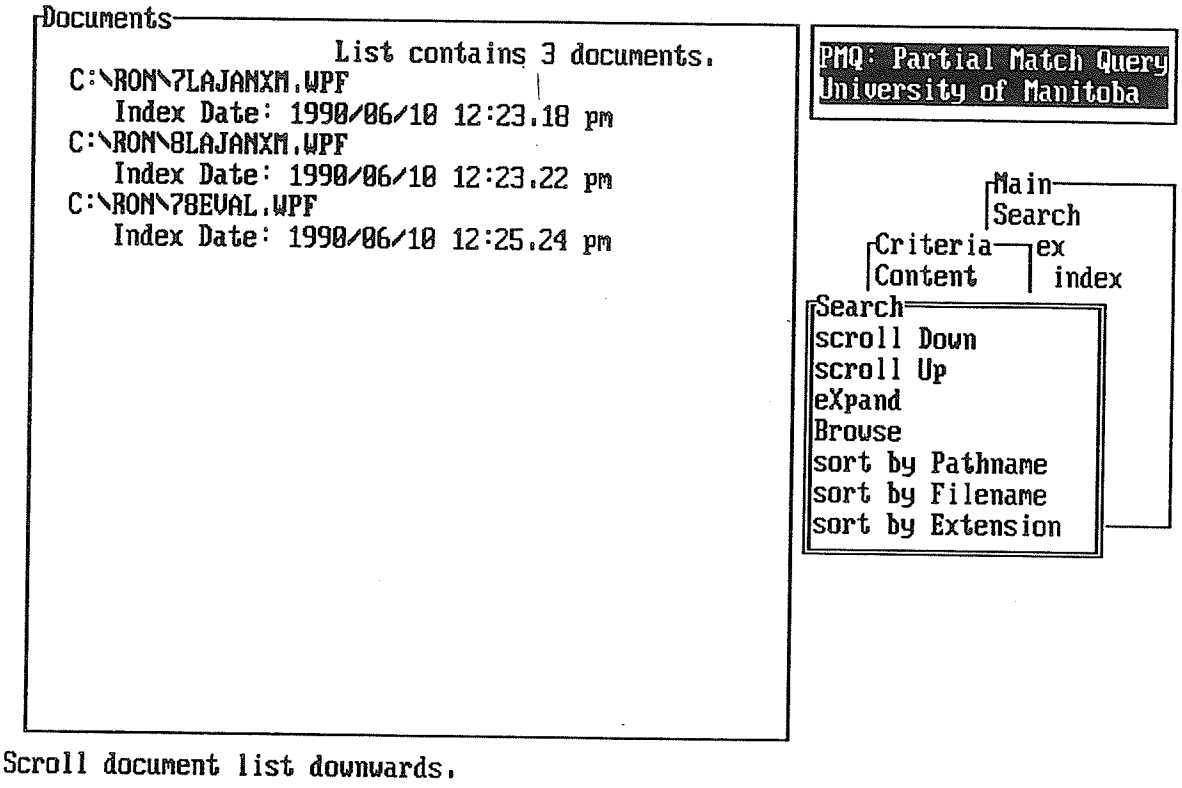


Figure 22
True Drops that Match the Search Criteria.

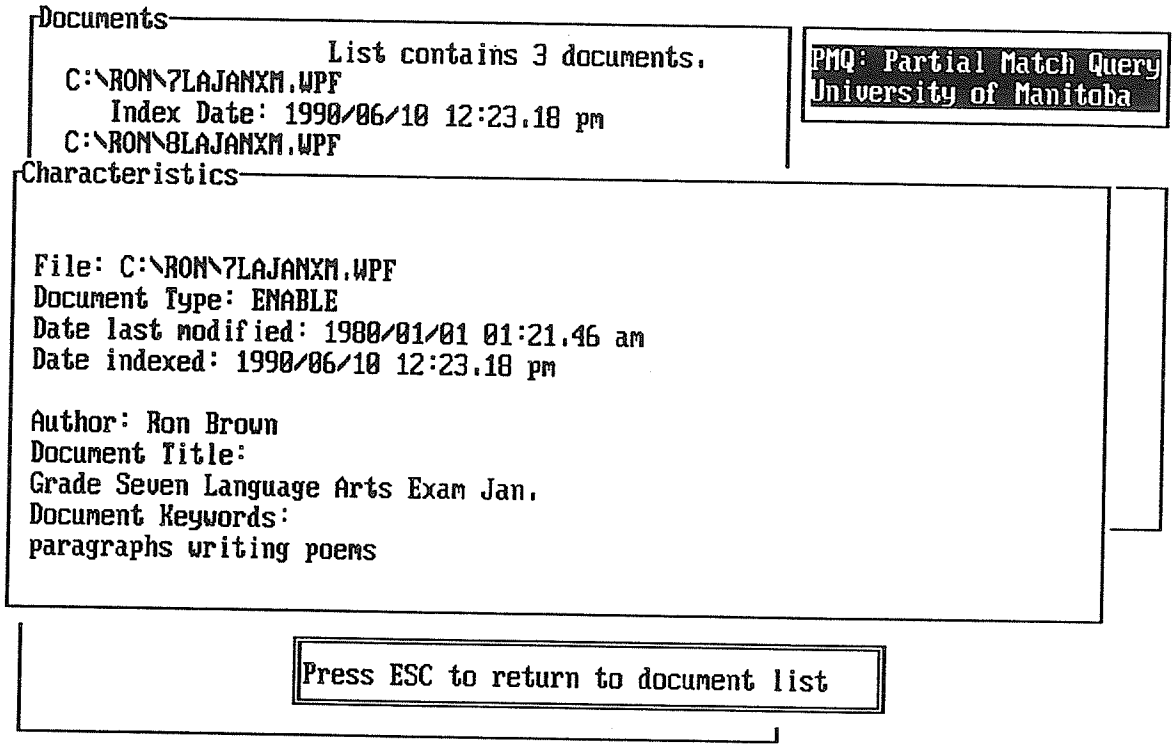


Figure 23
Display of Attribute Information.

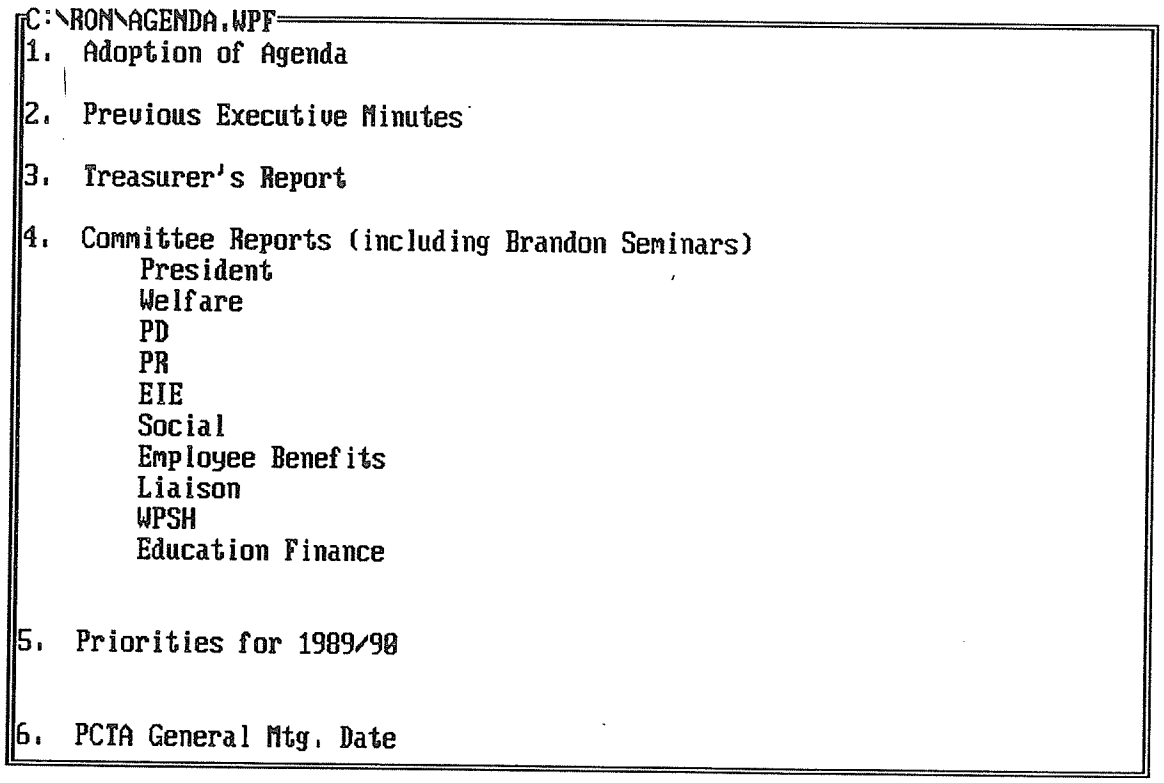


Figure 24
Browsing a Document in ASCII Format.

4.6 PMQ Implementation of DCBS.

In this section we discuss the implementation of the DCBS technique in our application. Basically the file organization is similar to that of Faloutsos (Figure 13) with minor alterations to handle the target computing environment. PMQ's DCBS file organization is diagrammed in Figure 25.

The section is organized as follows. First, we give a detailed description of the file structure, and relate it back to Faloutsos' structure. At this time we discuss implementation trade-offs that were made, given the model office and computing environments. Second, we describe the indexing algorithm in detail. Third, we describe the search algorithm, detailing the use of full text scanning to determine true drops and how the Retrieval Functional Interface is implemented.

4.6.1 File Structure.

Refer to Figure 25 for a diagram of the PMQ implementation of DCBS. The index file structure consists of four separate MS-DOS files:

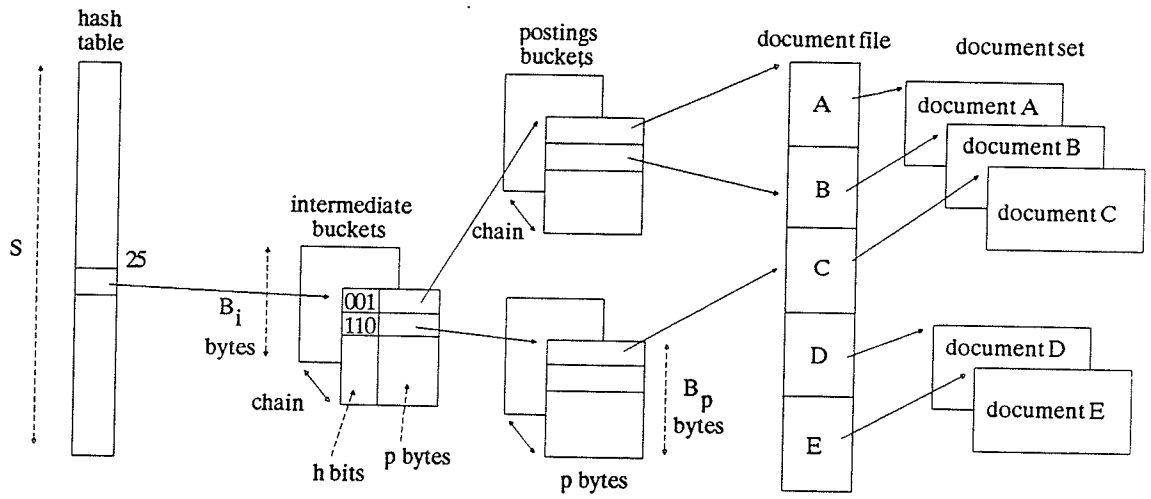
a) the HASH.PMQ file which stores the hash table;

b) the intermediate buckets which are stored in a MS-DOS file named LEVEL1.PMQ;

c) the postings buckets are kept in LEVEL2.PMQ;

d) the document filenames and attributes are stored in the DOCUMENT.PMQ file.

PMQ Implementation of DCBS (Doubly-Compressed Bit-Strings)



- S again denotes the size of the hash table; $S2^h$ represents the signature size.
- Note that documents in the document set are stored as separate DOS files.
- The "document file" contains the complete DOS filename of the document, as well as additional information such as the document's attributes, and a logical delete flag.
- p is fixed at 4 bytes; B_i, h, S, B_p are design parameters.

Figure 25
PMQ Implementation of DCBS.

Hash Table.

The HASH.PMQ file stores the hash table, which consists of four-byte Relative Byte Address (RBA) pointers to the intermediate buckets. The size of the hash table is variable up to a maximum of 4000 entries (an arbitrary maximum supported by the application code), and is specified in the PMQ configuration file (which is read at initialization).

To create the hash values from the document terms, a technique similar to that discussed in [Lar84] is used. Only the first five characters of a term are used to create the first hash value; if the term length is less than five then the full word is used. PMQ only indexes words that consist of at least two characters. A word is delimited by any character that is not a letter or a number.

The individual character values are transformed to a single integer by exclusive-OR'ing (XOR) the first character with the fourth and the second with the fifth. The middle character is unchanged. This 24-bit integer is used as input to a random number generator (taken from [Par88]) which will generate the hash table value (denoted $\mathbf{h}_1$), and the second hash value ($\mathbf{h}_2$) for the intermediate bucket. The random number generator is the "standard" linear congruential generator using $\mathbf{a} = 16807$ and modulator $\mathbf{m} = 2^{31}$-1. Declarations for the hash table, and other arbitrary limits, are given below in C notation:

```
typedef     long        RBA;   /* long integer for RBA's for disk blocks */
#define     MAXHASH     4000   /* maximum number of hash table entries */
#define     MAXWORD     25     /* maximum word length */
#define     P2MAX       127    /* max number of P2 entries */
#define     P1MAX       84     /* max number of P1 entries */

struct hashtbl {                           /* hash table file */
        RBA  pb1_ptr [MAXHASH];            /* S is maximum of 4000 */
        };
```

The P2MAX and P1MAX limits above ensure that the intermediate and postings buckets may not exceed the size of a DOS disk sector (512 bytes). As such they are maximums and should not be construed as optimal values.

Intermediate Buckets.

The intermediate buckets reside in a file named LEVEL1.PMQ. The buckets contain two-part entries for each indexed term that has a distinct $\mathbf{h}_1\mathbf{h}_2$ combination ($\mathbf{h}_2$ values must be distinct in any given chain of intermediate buckets). Intermediate bucket entries consist of a second hash value (domain may vary from 2^0 to 2^{16}, another arbitrary maximum) and an RBA pointer to a list of postings buckets. The domain of $\mathbf{h}_2$ is set in the configuration file. The number of entries per bucket is variable, again determined by a value in the configuration file. The intermediate buckets are chained using RBA pointers to other intermediate buckets to handle a varying number of entries that hash to the same hash table address.

```
struct pbucket1 {                       /* level 1 posting bucket  */
        RBA pb1_next;                   /* chain to next level-1 bucket */
        struct Entry {                  /* intermediate bucket entry */
                int  hash2;             /* second-level hash values */
                RBA    pb2_ptr;         /* level-2 posting bucket pointer */
                } entry[P1MAX];
        char filler[4];                 /* filler to fill 512 byte sector  */
        } ;
```

Postings Buckets.

The postings buckets are kept in LEVEL2.PMQ, and contain RBA pointers to the document information file. Each bucket entry represents a document that conatins at least one occurrence of a term that has hashed to a distinct $\mathbf{h}_1\mathbf{h}_2$ combination identified in the intermediate bucket. Postings buckets are quite small (to waste as little space as possible), typically less than 50 bytes. These buckets are also chained in a singly-linked list fashion (using RBA pointers) to handle

overflow. Postings buckets account for the greatest amount of storage overhead in the DCBS method, as will be shown in Chapter 5. Here is the C language declaration for postings buckets:

```
struct pbucket2 {                   /* level 2 posting bucket */
        RBA pb2_next;               /* pointer to next level2 bucket */
        RBA doc_ptr [P2MAX];        /* pointers to document file */
        } ;
```

Document File.

Document filenames and attributes are stored in the DOCUMENT.PMQ file. Each record in this file represents a document that has been indexed. Postings buckets refer to RBA's within the DOCUMENT.PMQ file. The DOCUMENT file is a sequential file, and functions as a reference for postings bucket entries. Each entry in the DOCUMENT file contains a logical pointer (i.e. the complete MS-DOS filename) to the indexed document. Each of the files identified in the DOCUMENT file, then, make up the document set. The C declaration for DOCUMENT file entries is:

```
struct document {
        char status;                /* document data in "titles" file */
        char disk[MAXDRIVE];        /* document status - D = deleted */
        char dir[MAXPATH];          /* disk letter, with colon */
        char filename[MAXFILE];     /* file pathname */
        char ext[MAXEXT];           /* filename */
        char author[31];            /* file's 3-letter extension */
        char title[71];             /* author information */
        char keywords[71];          /* additional title data (optional) */
        char type[9];               /* additional keyword data (optional) */
        char file_date[24];         /* document type */
        char index_date[24];        /* document's date/time when last modified */
        } ;                         /* date/time document was indexed */
```

For the remainder of this Chapter, we will use the term DCBS to describe the PMQ implementation of the method, unless otherwise noted. A comparison of theoretical performance results described in [Fal87c] with those from the model office is made in Chapter 5.

Implementation Tradeoffs.

The major difference between the DCBS method in [Fal87c] and the PMQ implementation of DCBS is the use of the additional DOCUMENT file. This file is required for two reasons:

a) [Fal87c] supports only full text retrieval for document content. The file structure does not directly support the storage of attribute data.
b) the target computing environment of PMQ does not involve the use of optical disk technology.

These reasons are discussed below.

Chapter 2 addressed the problems associated with full text only search methods; though there is no agreement about what additional information is required for optimal retrieval, it is agreed that full text alone is not sufficient [Ten89]. Therefore, PMQ supports the indexing of attribute information. This information had to be stored in the file structure as the Enable file format was insufficient for the purpose, and attribute information would be difficult to retrieve from ASCII text files.

The use of only fixed magnetic disk, instead of optical disk, presented another problem. In [Fal87c] the document set is a continuous byte stream; documents are stored in a contiguous fashion. Since the target environment in [Fal87c] uses Write-Once Read Many (WORM) optical disk technology [Fuj84], deletions of documents are impossible.

The target environment of PMQ is MS-DOS; here word processing documents are stored as distinct files, which may be deleted and renamed at will, and can be moved about on the same disk, or from one fixed disk to another. PMQ, then, had to support the renaming of documents and could not rely on the physical placement of documents on the media. Document deletion is supported by logically deleting entries in the DOCUMENT file by marking them with a flag byte.

[Fal87c] also does not address any document management requirements; that is, the capability of the workstation user to determine which documents in the system have been indexed (because Enable is not integrated with PMQ, this is an important fact for the workstation user to know). PMQ addresses this by displaying the index date and time for each document present in any list of documents displayed to the user.

A further difference due to the media technology employed concerns performance. With WORM devices, seek times for the disk head are approximately 230 milllseconds, an order of magnitude greater than that for typical fixed magnetic disks in IBM-compatible microprocessors [Fal87c]. Faloutsos recommends using the size of a disk page for the size of intermediate buckets to reduce the number of seeks. However, this approach is optimal only for large document sets. With a small document set too much of the page is wasted as the load factor is too low (this is discussed in more detail in Chapter 5).

4.6.2 Indexing Algorithm.

Indexing a document using the DCBS method involves indexing each distinct five-letter word stem in the document, ignoring words of only one character and the filter list of 124 words.

When a document is indexed, we can only estimate the number of distinct five-letter terms from the document length, using Zipf's law. As document sizes may be quite variable, PMQ does not create a memory-resident list of distinct word stems, but relies on the DCBS data structure to determine if any synonyms (where words i and j have hashed to the same $\mathbf{h}_1\mathbf{h}_2$ combination, i and j not necessarily distinct) exist.

For the algorithms that follow:

$\mathbf{I}_n$	- represents the last intermediate bucket in the LEVEL1 file;
$\mathbf{P}_n$	- represents the last postings bucket in the LEVEL2 file;
$\mathbf{D}_j$	- the j^{th} candidate document determined after a signature search.
$\mathbf{RBA}_d$	- represents the RBA of the document entry in the DOCUMENT file;
$\mathbf{h}_1$, $\mathbf{h}_2$	- represent the first and second hash values from encoding a five-letter string;
$\mathbf{term}_k$	- the k^{th} term of a document;
$\mathbf{term}_q$	- the q^{th} term of a query.

The document insertion algorithm is as follows:

Algorithm Insert:

I1. Prompt the user for attribute values associated with this document.

I2. Insert document record into DOCUMENT file; record $\mathbf{RBA}_d$.

I3. For each word $\mathbf{term}_k$ in the document, convert to uppercase; if a stop-word, or less than 2 characters, read the next $\mathbf{term}_{k+1}$; if end of document, goto I10.

I4. Truncate $\mathbf{term}_k$ to at most five characters; hash($\mathbf{term}_k$) yielding $\mathbf{h}_1$ and $\mathbf{h}_2$.

I5. If hash table[$\mathbf{h}_1$] is not NULL, read $\mathbf{I}_{h1}$ and goto I6; else create intermediate bucket $\mathbf{I}_{n+1}$ in storage; set all entries in $\mathbf{I}_{n+1}$ to NULL; set $\mathbf{I}_{h1}$ to $\mathbf{I}_{n+1}$; goto I7.

I6. Search $\mathbf{I}_{h1}$ for entry with $\mathbf{h}_2$ using linear search; follow chain of intermediate buckets if required. If matching $\mathbf{h}_2$ found, goto I8; else try to add new entry to $\mathbf{I}_{h1}$ (last bucket in chain). If not enough room, determine RBA of $\mathbf{I}_{n+1}$; chain $\mathbf{I}_{h1}$ to $\mathbf{I}_{n+1}$; rewrite $\mathbf{I}_{h1}$ to disk. Create new $\mathbf{I}_{n+1}$ in storage; set entry(0) to tuple ($\mathbf{h}_2$,NULL).

I7. Create new postings bucket $\mathbf{P}_{n+1}$; set entry(0) to $\mathbf{RBA}_d$. Write postings bucket to postings file. Update $\mathbf{I}_{h1}$ entry($\mathbf{h}_2$,NULL) to entry($\mathbf{h}_2$,$\mathbf{P}_{n+1}$); write $\mathbf{I}_{h1}$ to file, goto I3.

I8. Search entries in postings buckets chain $\mathbf{P}_{h1h2}$ for $\mathbf{RBA}_d$. If found, goto I3 (synonym); else attempt to insert $\mathbf{RBA}_d$ in next available entry in last postings bucket in chain. If entry is available, update entry with $\mathbf{RBA}_d$, write $\mathbf{P}_{h1h2}$; goto I3. If not available, goto I9.

I9. Create new postings bucket $\mathbf{P}_{n+1}$; set entry(0) to $\mathbf{RBA}_d$; write $\mathbf{P}_{n+1}$ to file. Chain $\mathbf{P}_{h1h2}$ to $\mathbf{P}_{n+1}$; rewrite $\mathbf{P}_{h1h2}$ to file. Goto I3.

I10. Repeat steps I3 to I9 for each indexed attribute. When complete, return.

4.6.3 Search Algorithm.

Searching for a term in the document set involves a search through the levels of the DCBS directory (hash table, intermediate buckets, and postings buckets). The postings buckets identify the records in the DOCUMENT file that represent particular documents. Note that each query in PMQ is conjunctive; all query terms must exist in a document in order to consider that document as a true drop.

Since hashing is used to determine the signature of any term, there is a non-zero probability that a hashing collision may occur between two distinct terms, which can result in false drops. To prevent the display of false drops to the user, PMQ uses full text scanning of each candidate document to ensure that the document actually contains the query terms. The workstation user may use PMQ's document browse capability in order to pinpoint relevant documents once the list of true drops has been displayed.

The search process in PMQ is, then, separated into two distinct parts: the index search, and full text scanning. The DCBS index search algorithm is as follows:

Algorithm Index Search:

S1. Prompt user for query terms; stop-words and one-letter words are invalid. Build tree of query terms, each leaf consisting of a single word or term group. Translate each query term $\mathbf{term}_q$ to uppercase.

S2. For each $\mathbf{term}_q$ in the query tree, perform steps S3 to S6.

S3. Truncate $\mathbf{term}_q$ to at most five characters. Hash $\mathbf{term}_q$ giving values $\mathbf{h}_1$ and $\mathbf{h}_2$.

S4. If hash table[h1] is NULL, return "no results"; else read intermediate bucket I_{h1} (RBA of I_{h1} is in hash table).

S5. Search chain of intermediate buckets, starting with I_{h1}, for an entry with the value ($\mathbf{h}_2$,*) (* = don't care). If none found, return "no results".

S6. If an entry ($\mathbf{h}_2$,*) is found, merge list of postings bucket entries (from all postings buckets in that chain) with other lists previously retrieved. The lists are composed of $\mathbf{RBA}_d$'s into the DOCUMENT file. The final list is the candidate list of all documents satisfying the conjunctive query.

S7. Continue with full text scanning algorithm; goto F1.

After this first search pass, we have a list of candidate drops; we now must delete any false drops in the list. To ensure the scope of full text searching is reasonable, we compare the number of candidate drops with the limit specified by the user. If the limit is exceeded, the user has the option to carry on, or cancel the query at this point and try again.

The full text scanning algorithm that follows handles matching document attributes and full text to query terms explicitly, without the use of signatures. This is done to determine if any false drops exist in the list of candidate documents retrieved by the index search algorithm. In fact, attributes are done first (as matching attributes is much faster) to reduce the scope (and elapsed time) of the search before full text must be scanned. However, the same steps apply to attributes as to full text:

Algorithm Full Text Scanning:

F1. Set all flags in the leaf nodes of the query tree to NOT FOUND.

F2. For each candidate document D_j read in sequence each $\mathbf{term}_k$ from the document. At the end of each document, goto F4. Ignore stop-list words and one-character words. Translate each $\mathbf{term}_k$ to uppercase. After the last candidate document is read, goto F5.

F3. Compare each $\mathbf{term}_k$ to each leaf in the query term tree. If a single word leaf, Compare($\mathbf{term}_k$,$\mathbf{term}_q$). If a match, set leaf node flag to FOUND. If a term group ($\mathbf{term}_0$, $\mathbf{term}_1$,... $\mathbf{term}_n$), Compare($\mathbf{term}_k$,$\mathbf{term}_q$[i]) where **i** is the next term in the group to be compared. If a match, set **i** to the next term in the group; if all terms in the group are found (**i** = **n**), set the leaf node flag to FOUND. If not a match, reset **i** to 0.

F4. If all flags in the query term tree are FOUND, the document is a true drop; else it is a false drop. Mark the candidate document as such and return to F2.

F5. Delete the false drops from the list of candidate documents; display list of true drops.

Algorithm Compare:

This algorithm compares a term in the document ($\mathbf{term}_k$) to a term in the query ($\mathbf{term}_q$). Each $\mathbf{term}_q$ can include a wildcard character ("*"). The algorithm is as follows:

C1. If $\mathbf{term}_q$ contains a wildcard character, goto C3.

C2. Set compare length **l** to size($\mathbf{term}_q$); goto C4.

C3. Set compare length **l** to size($\mathbf{term}_q$) -1.

C4. If $\mathbf{term}_q = \mathbf{term}_k$ for length **l**, then return(match); else return(no match).

4.7 Possible Enhancements to PMQ.

The PMQ application is in use today in schools of the Pine Creek School Division. Although initial response from its users has been excellent, there is room for substantial improvement to make PMQ into a more commericial product and less of an academic prototype. Research topics are listed in Chapter 5; enhancements that use known techniques include:

a) Incorporate a buffer management scheme into PMQ, possibly using a least-recently used (LRU) algorithm in an attempt to minimize I/O operations to the actual disk. Buffer management algorithms other than LRU may be more effective; this is discussed in Chapter 5. Such a scheme may improve indexing elapsed times when a disk cache is not available (not present on any workstation in the model office).

b) Modify PMQ to handle French ASCII character codes. This implies handling French PC keyboards. Support searching for French, English, or multilingual documents at the user's option. Add an attribute saved with each document indicating the document's language.

c) Provide user-controlled case-sensitive full text scanning (currently full text scanning is case-insensitive).

d) Modify the DOCUMENT file to use a different file organization. As developed, the DOCUMENT file is sequential. This becomes a problem when attempting to determine if a document has previously been indexed; the entire file must be

scanned. A hashed organization could be considered to provide the necessary direct access. However, since the intent of [Fal87c] was to support WORM optical disks, the new file organization should support WORM restrictions.

e) Upgrade PMQ to support a pointing device, graphics support, and color displays. The user interface design would require substantial changes to take advantage of this technology, but this would improve user interaction and ease of use.

f) Incorporate vocabulary analysis into PMQ. An approach similar to [Sac87] could be used, relying on Bloom Filters to filter poor quality indexing terms. Such a filter would be built for each installation.

g) Provide better integration with Enable software. One way to do this is by developing a memory-resident (known in MS-DOS as a Terminate-and-Stay-Resident program (TSR)) application to capture document names and attributes. The TSR application could be invoked using a special sequence of keystrokes from within Enable. After an Enable session, the information captured by the TSR program could then be used by PMQ to index the documents in a "batch" mode.

h) Support other word processing file formats, such as Microsoft Word, Ashton-Tate's Multimate, etc.

i) Add a document type (memo, letter, examination) attribute to the DOCUMENT file, and prompt for other attributes accordingly.

j) During the indexing process, if the document's size is smaller than a threshold then a significant amount of processing could be accomplished within memory. If the entire vocabulary of the document was determined before insertion to the signature file, no wasted I/O would be performed to the file for second and subsequent occurrences of any particular word (in DCBS only distinct words are inserted into the signature file). This would substantially improve indexing performance. In-core vocabulary analysis would be limited only by the amount of available memory under MS-DOS.

5. Performance Analysis and Conclusions.

In this chapter we compare experimental results with the use of DCBS in our model office implementation with the expected results from the equations given in Section 3.3.1.

In Section 1, we describe the characteristics of the document set in the model office, and compare those characteristics to those from [Fal87c]. In Section 2, we discuss our experimental results, and compare them to the expected results from the equations in Section 3.3.1. Some of the equations are altered slightly to conform to the PMQ implementation of the DCBS method. In Section 3, we present our conclusions, and offer topics for additional research. Section 4 provides a short summary of this thesis.

5.1 Model Office Document Set Characteristics.

When the PMQ application was installed in the model office at the Langruth Elementary School, approximately 400 documents existed on the workstation. The document browse facilities of PMQ were used to delete any unnecessary documents from the disk before the indexing process took place. Initially French documents (in the English character set) were also inserted into the signature file. However, as these documents added significantly to the document set vocabulary (9.4%), they are excluded from the results presented here (a complete, new DCBS file structure was defined). In total, 255 documents were indexed, and the document

set characteristics are (refer to Table 3 for definitions of the variables):

N = 255
V = 10283
V_{stem} = 7081
N_w = 107380
N_t = 63173
L = 2901.6
NL = 739919 (1445 512-byte pages)
D = 138.21
ND = 35244
A = ND/V = 4.97726

Vocabulary Analysis.

Because the performance of the method is directly related to the vocabulary of the document set, it is critical that a satisfactory estimate of V be determined before the design parameters for the DCBS method are set. This was one of the significant problems in the implementation of PMQ. Initial experiments using estimates from Faloutsos [Fal87c] were completely unsatisfactory (one example achieved 100% space overhead).

When time permitted, a vocabulary analysis of the model office document set was performed to establish that V = 10283 and V_{stem} = 7081. In particular it is the value of V_{stem} that interests us, as PMQ only indexes word stems, not full words. We also compared the actual vocabulary measurements to the estimate given in [Fal87c and Fal84], which is based on Zipf's law:

$$V\ln(2V) = N_w$$

where the number of words is estimated as:

$$N_w = \frac{NL}{l_w + 1}$$

where NL is the size of the database in bytes, the value lw is the average size of a word (typcially 5), and 1 represents the word delimiter. Figure 26 shows the results. Three conclusions can be drawn from this analysis:

a) Faloutsos' estimate of Zipf's law is extremely good, at least for our model office document set (and probably for other text databases of similar size);

b) As shown in Figure 27, Faloutsos' estimate shows that vocabulary continues to increase as documents are added, even if the document set size is considerable. This leads to a contradiction of the assumptions made for equation XX (document insertion). Equation XX is only valid if the new document does not add to the vocabulary.

c) As shown in Figure 28, Faloutsos' experiments were based on an experimental document set with a very unrealistic vocabulary (appoximately equal to that of our database, but the database in [Fal87c] is approximately four times the size of ours (2.8 megabytes vs. 0.74 megabytes). However, as will be seen below, the results (space overhead, false drop probability, etc.) from [Fal87c] are comparable to ours. This seems to confirm that the performance of the method is indeed related to document set vocabulary, and other characteristics of the document set seem to matter very little.

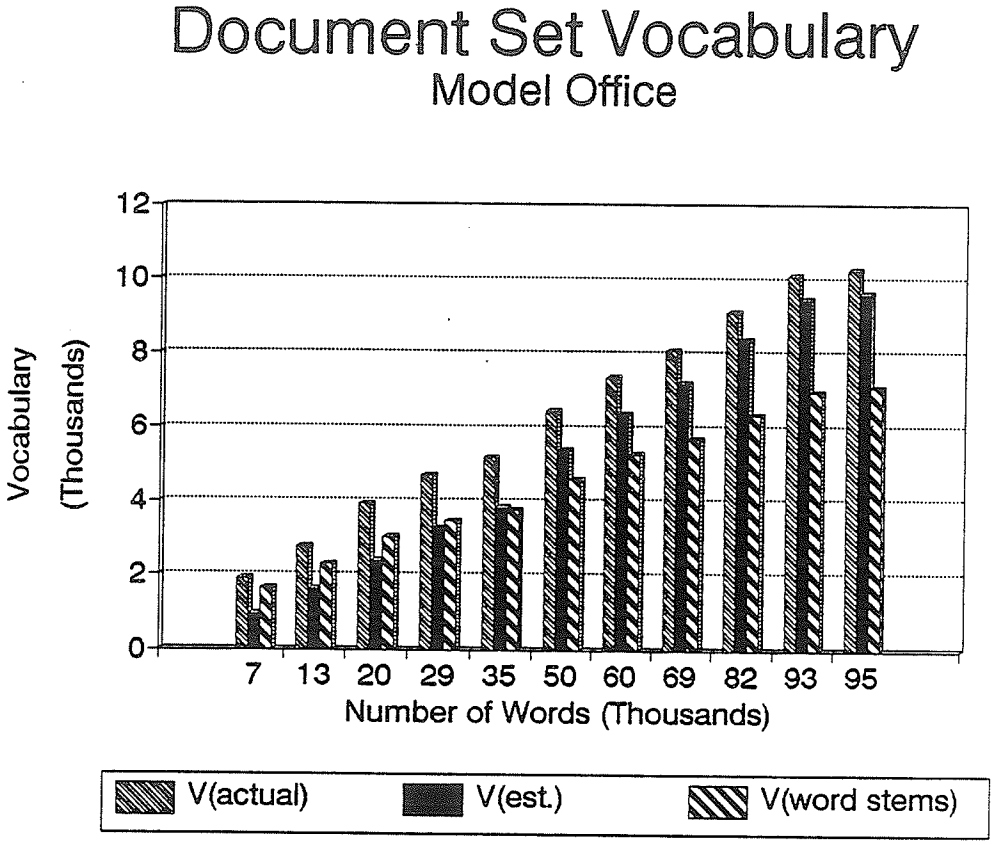


Figure 26
Document Set Vocabulary from the Model Office.

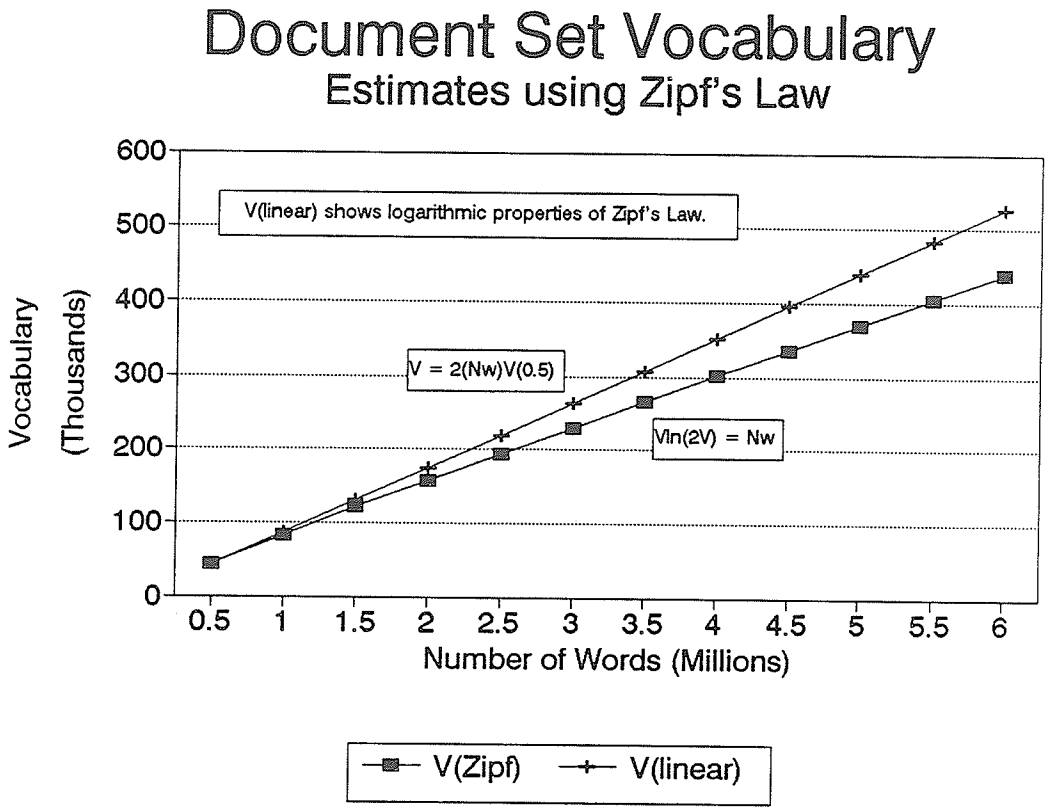


Figure 27
Expected Results of the Estimation of Zipf's Law

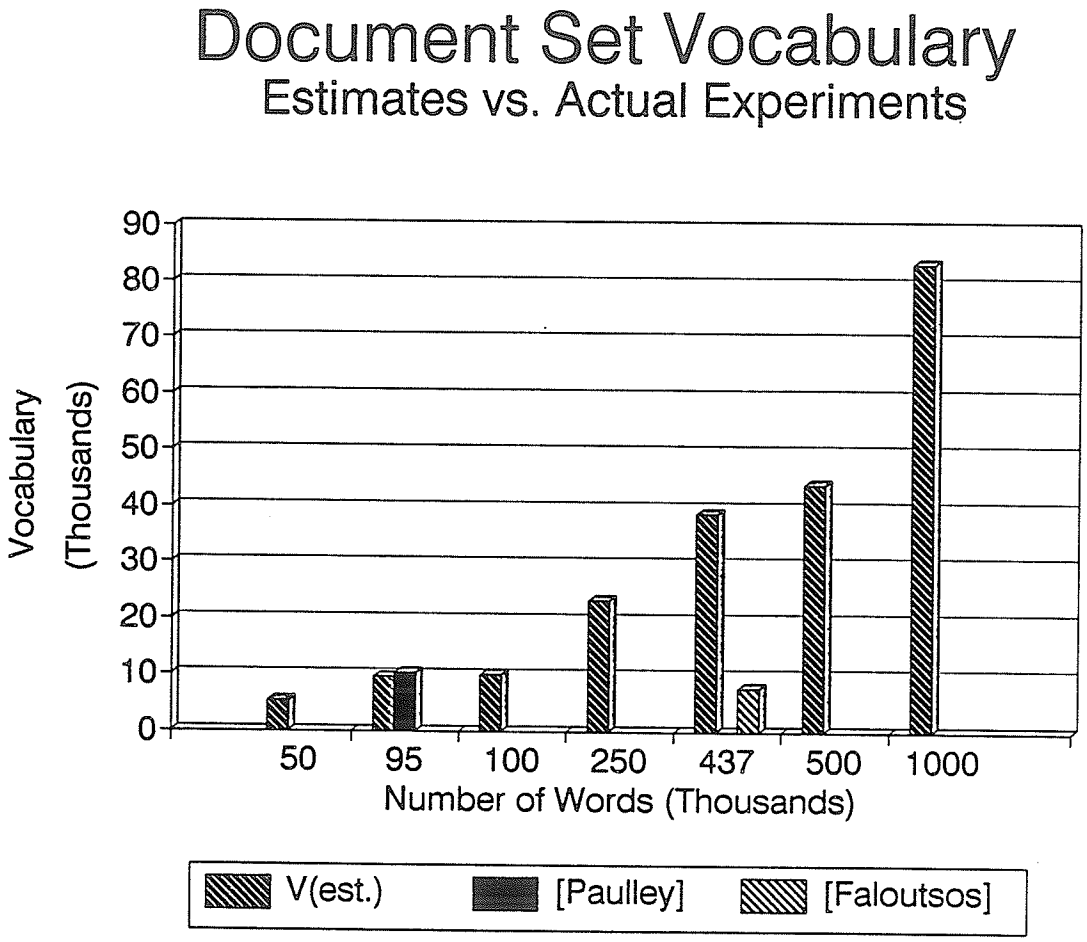


Figure 28
Comparison of Actual and Expected Vocabulary

5.2 Experimental Results.

The final choice of parameters used for our model office document set were:

B_i (intermediate bucket size) = 76 bytes (12 entries per bucket)
B_p (postings bucket size) = 20 bytes (4 entries per bucket)
S (size of hash table) = 600
p (pointer size) = 4 bytes
P (size of a disk page) = 512 bytes
b (byte size) = 8 bits
h (size of hash code in intermediate bucket) = 4 bits

These parameters did not lead to optimal space overhead (the lowest space overhead percentage achieved in the experiments was 33%), but led to a reasonable compromise between space overhead and false drop probability. Other choices of parameters may lead to better results.

One significant difference in the equations for space overhead (and also for retrieval and insertion) is the value of h. For simplicity, PMQ writes intermediate bucket entries using the *maximum* value of h (16 bits - a short integer). However, for *hashing* purposes only 4 bits were used. Thus the higher value (16) is used for space and retrieval performance. The smaller value (4) is used for calculating false drop probability.

The most significant parameter to be chosen was B_p. B_p affects search and insertion performance, as well as space overhead, but does not affect false drop probability (which is determined by the choices of S and h). [Fal87c] suggests setting $B_p = k*p+p$, where k is a small integer. Our experiments indicate that B_p should depend on A (the average number of documents a word appears in), but must be chosen carefully due to the (very) skewed nature of vocabulary usage (as will be discussed shortly).

We now discuss the individual measurements whose equations were presented in Section 3.3.1. For all the equations requiring the calculation of prob(V,S,w) an approximation to the binomial using the Poisson distribution was used. The formula for calculating probabilities using the Poisson distribution is:

$$\text{Prob}(y) = \frac{\lambda^{y} e^{-\lambda}}{y!}$$

with the value of lambda equal to the average probability:

$$\lambda = n\Pi$$

and Pi equal to:

$$\Pi = \frac{1}{S} \text{ or } \frac{1}{S2^{h}}$$

depending on the particular formula.

Space Overhead.

With the parameters chosen as above, the size of the DCBS file structure is (in bytes): hash table, 2400; intermediate buckets, 45828; and postings buckets, 220880. This gives a space overhead of 36% (note we do not include the Documents file in this measurement, so that our results can be compared more directly to [Fal87c]). Note also that the use of attributes will add intermediate and postings bucket entries to the signature file. However, attributes are not supported in [Fal87c]. A graph comparing expected and actual space overhead is shown in Figure 29.

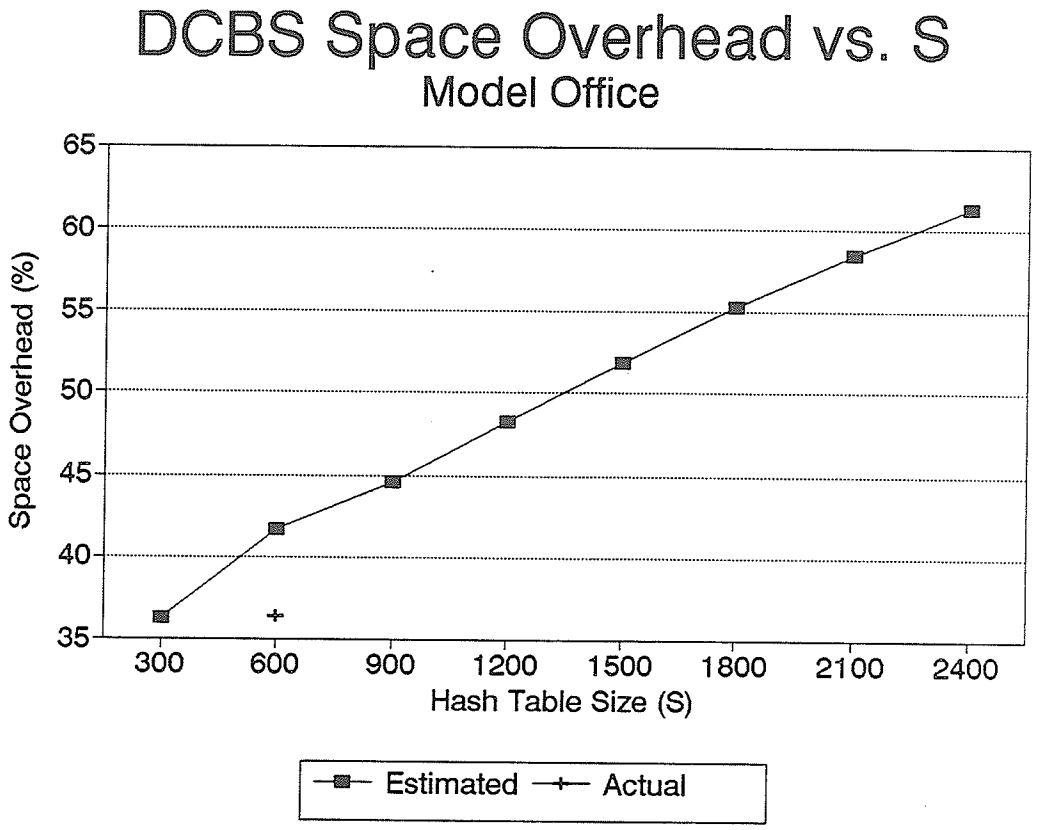


Figure 29
Comparison of Estimated and Actual Space Overhead.

The size of the postings file is nearly optimal. The 220880 bytes is made up of 11044 buckets, with room for 44176 document pointers. The postings file acutally contained 35244 pointers (ND), a load factor of 80%. It should be noted that Faloutsos in [Fal87c] uses only 3-byte pointers and only presents theoretical space overhead figures in the range 17%-22%. With all other variables constant, our experiments would result in 26% overhead if 3-byte pointers were used.

For the intermediate buckets, Faloutsos suggests the use of full disk page for each bucket (512 bytes). This is much too large a value for the size of our database, though this value will certainly reduce the disk accesses required in searching. Faloutsos does not use a full disk page (1024 bytes in his experiments) in his analysis; his choice follows the equation:

$$B_i = \frac{V}{S}(p + h/b) + p$$

which provided a reasonable overhead with our model office database. However, other intermediate bucket sizes we tried (34 bytes) seemed to work just as well. We postulate that the small size of the document set permits such a variance.

Unsuccessful Search.

To test unsuccessful search, a total of 120 single-term queries were made against the document set; the query words were chosen at random from the Metropolitan Toronto street index. Query words were truncated to 5 characters, and wild-card characters were appended if the street name was longer than 5 characters (in keeping with indexing only word stems). The results are shown in Figure 30; it appears that the estimates and the actual value (3.3917 disk accesses) agree very well.

Successful Search.

For successful search, a total of 1000 words were selected at random from the document set. Each of the words became a single-word query; words were truncated appropriately, as described for unsuccessful search. Both successful and unsuccessful search results are shown in Figure 30.

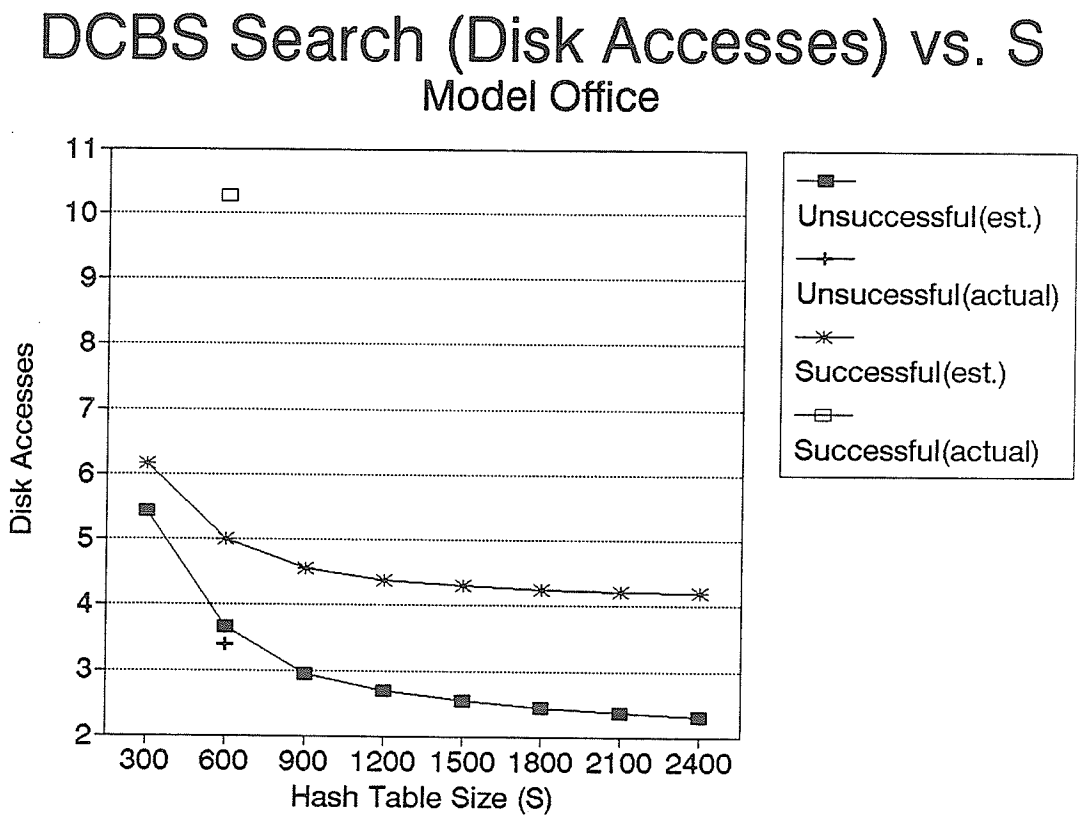


Figure 30
Comparison of Estimated and Actual Search Results.

We note immediately from the graph that a significant discrepancy exists between the expected number of disk accesses, and the actual figures encountered. The reason for this is the skewed nature of the vocabulary. On average, the number of documents a word appears in (A) should be 4.97 (ND/V). However, the average number of true drops returned in the 1000 queries was 25.342; the average number of false drops was 6.192. It appears that our small document set renders Assumption 4 (N >> A) untrue. It would be interesting to determine if Faloutsos' experiments [Fal87c] would suffer in the same manner if the successful query words were chosen from the document set, instead of the UNIX dictionary. We offer the suggestion that future papers use non-common words appearing in the document set for successful search experiments, as these values should be much more realistic. It seems reasonable that users would enter queries based on terms that they either are certain, or at least believe, exist in the document set.

False Drop Probability.

Recall from Section 3.3.1 that our expected false drop probability for DCBS (equation VI) was:

$$F_d = 1 - \left[1 - \frac{1}{S2^h}\right]^D \doteq \frac{D}{S2^h}$$

which, for our experiments, is 138.21/9600, or 0.0144. The false drop probability incurred in the unsuccessful search experiment was 0.0174; for successful search, 0.0269. This gives an average F_d of 0.0221. A plot of actual versus estimated false drop probability is shown in Figure 31.

We note that the actual value is close to the expected one. Reasons for the slight difference are:

a) the equation above is actually for the case of unsuccessful search [Fal84]; our unsuccessful search result of 0.0174 is even closer to the expected value.

b) as stated above, it would appear that Assumption 4 does not hold for our small document set.

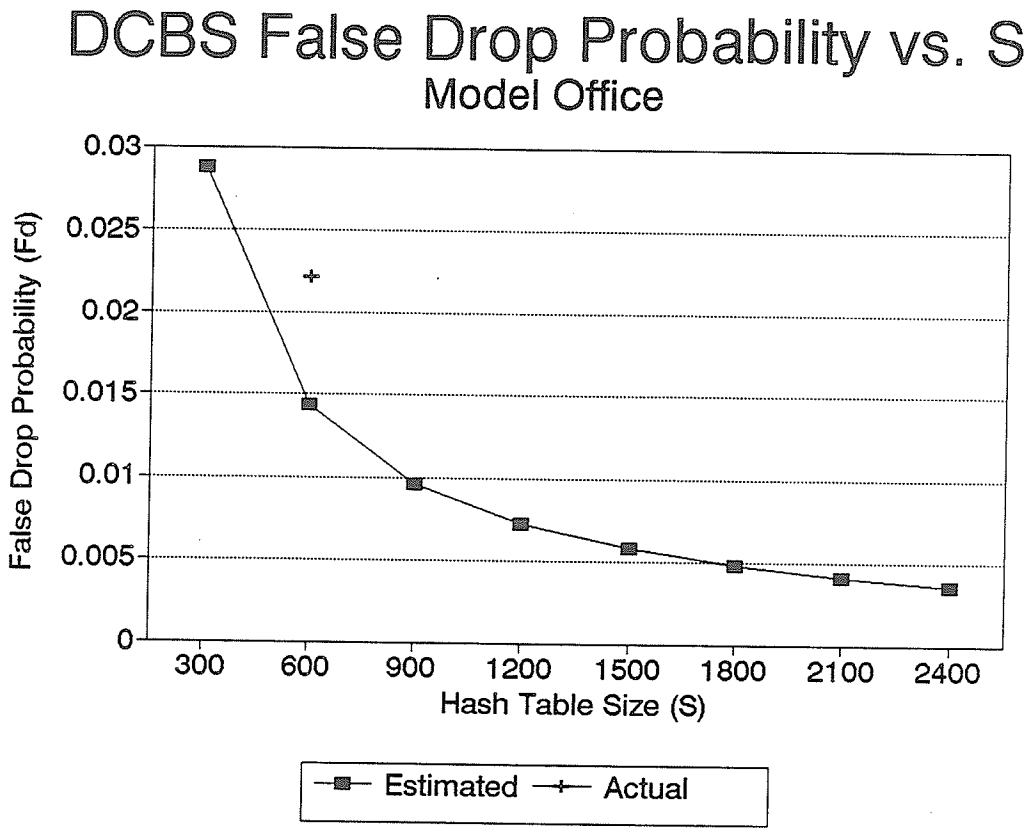


Figure 31
Comparison of Estimated and Actual False Drop Probability.

Insertion.

As discussed above under the heading "Vocabulary Analysis", the estimate of disk accesses to insert a new document (equation XX) assumes that no words are added to the document vocabulary. Section 3.3.1 discusses the four possible scenarios when inserting any given word of a document. As can be seen in Figure 32, our experiments indicate that equation XX gives an estimate far too low. To be fair, equation XX also assumes that a new document is being added to an existing document set. We did not perform such an experiment (no "new" documents were available) but we postulate that the underlying assumption of equation XX is still false, as the estimated vocabluary provided by Faloutsos' approximation of Zipf's law will reach a limit very slowly. Such an assumption may prove true only with very large text databases.

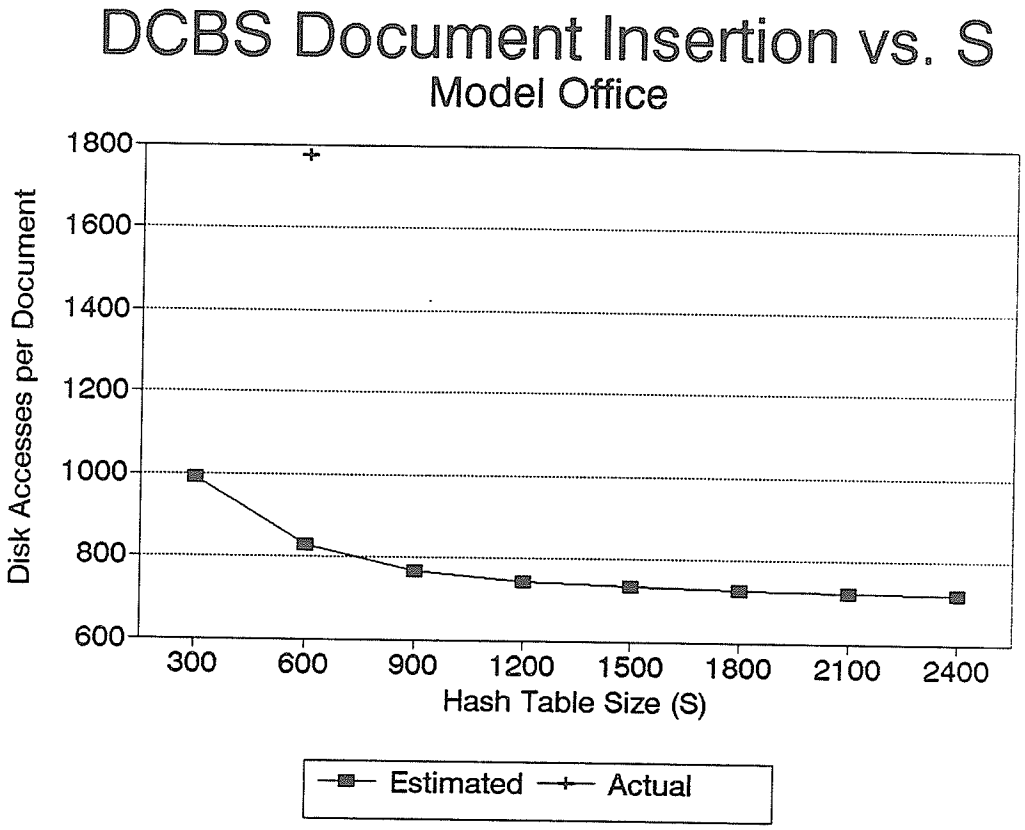


Figure 32
Comparison of Estimated and Actual Disk Accesses for Document Insertion.

5.3 Conclusions and Future Research Directions.

Though the document set of the model office is small, our results indicate that our experience compares favourably with expected values from [Fal87c]. Two areas of disappointment have been mentioned: the estimate for successful search is very optimistic, as is the estimate for the number of disk accesses for document insertion.

From the perspective of the Pine Creek School Division, the application is a success. A facility has been provided that solves some of the problems of document management on independent workstations, enabling personnel in the School to find relevant information more quickly. The application requires little space overhead, can use different parameters for each installation, and requires no fees or royalties.

Actual performance of the system, in terms of elapsed time for a search, appears adequate. In executing 1000 successful search queries, each query was completed in approximately 15 to 20 seconds[23]. The majority of the time was spent in full text scanning of candidate drops. For unsuccessful search queries, elapsed times of three to five seconds were encountered.

Future Research Directions.

Doubly-Compressed Bit Slices is a fairly new implementation of signature files. It has been extensively studied in [Fal87c] and now in this thesis, yet a significant amount of work remains in refining the method. Before it can be used commercially, we must learn a great deal more about

[23] The equipment used for the tests differed from the model office. The hardware consisted of a PC-AT 80286 clone with a 20 megabyte hard disk. The hard disk was a Seagate ST-251 with an access time of (a relatively slow) 65 milliseconds.

the characteristics of DCBS so that it can be "tuned" to the environment in which it is installed. Some possible areas of research include:

a) deriving better formulae for successful search and document insertion, so that they reflect (typically) skewed word occurence distributions.

b) defining a methodology for determining bucket sizes, and the size of the hash table. In our experiments, the complete document set was indexed nearly a dozen times before our final set of parameters was settled. In a commercial environment such tuning by "trial and error" would not be permitted.

c) defining equations to deal with multiple-word queries.

d) testing new buffering schemes, so that read accesses to the index need not require a physical I/O operation. This is especially important in the insertion process.

e) tests of Faloutsos' estimation of Zipf's Law [Fal84,Fal87c] for large document sets (in the millions of words).

f) analysis of what happens to the signature file when, over time, more documents are added and the vocabulary grows to exceed expected values. This is important, as in any office environment more and more documents are created. The effect of such a dynamic document set should be studied; indeed, our model office represents an excellent opportunity for a study of this nature.

g) encoding schemes to handle range queries and different encoding methods for attributes need to be developed for bit-slice signature methods.

5.4 Summary.

In the first chapter of this thesis we presented an overview of the characteristics of office documents, and described a model office from the Langruth Elementary School in Langruth, Manitoba. The problem was to provide a cheap, effective means of searching for Enable word processing files that reside on the fixed disk of standalone Commodore PC's. An MS-DOS application named PMQ was developed to do this, and uses a new signature method called Doubly-Compressed Bit Slices (DCBS).

Chapter 2 presented a survey of access methods for text. Here, a more detailed description of document characteristics was given, along with relevant concepts of document retrieval from the discipline of Library Science. Particular characteristics of textual documents in an office environment was presented. In Section 2.5 various algorithms for indexing documents were discussed, including full text scanning, inversion using B-trees, and multi-attribute hashing. This was followed by an overview of the literature concerning the use of signatures, and a description of bit-string methods (WS, SC, RL, BC and VBC).

In Chapter 3 we gave a detailed description of bit-slice methods (BSSF, CBS, DCBS and NFD). A very detailed analysis of the expected performance of DCBS is given in Section 3.3.1.

Chapter 4 discussed the PMQ application: its intended technical environment, its architecture and its user interface. The design was driven by the technical environment; lack of a graphics adapter for the PC monitor, lack of a pointing device, and the desire for the application to be as inexpensive as possible. A windowing system, written in the C programming language, was acquired (without licenses or royalties) to aid the development of the application.

Finally, in Chapter 5 we presented our actual results of the indexing of 255 documents in our model office. Our conclusions are:

a) that the application is quite successful; it solves the problem it was expected to solve.

b) the performance of the system is close to expected results; good retrieval performance with low space overhead.

c) more work needs to be done in order to make the implemented method more suitable as a commercial product.

For text retrieval in general, a great deal of research is currently underway, and much, much more needs to be done. The recent implementation of BLOB (Basic Large OBject) datatypes in Relational Database Management Systems (RDBMS) now permit text to be kept in a standard RDBMS along with more traditional data structures. However, to the author's knowledge no techniques have yet been developed to allow users to search such textual data effectively (almost all commercial RDBMS products only permit the use of B-tree indicies, and at the time of writing do not even support the use of a B-tree index on BLOB columns). Also, the standard data manipulation language (DML) for RDBMS is Structured Query Language (SQL). To the author's knowledge, all existing implementations of SQL do not support any effective means of specifying a search on text, except for the extremely crude LIKE operator. Better query syntax and more powerful operators are needed to enable users to search textual databases more effectively [Ten89]. Researchers at the University of Waterloo, involved with the indexing of the Oxford English Dictionary, are addressing some of these problems. As relational databases continue to be improved, and integrated, multi-media systems come of age, effective searching techniques such as DCBS will be required. Much work remains in the future to establish these methods.

Appendix A.

Abbreviations of the signature methods described in this thesis:

WS	Word Signature method.
SC	Superimposed Coding method.
RL	Run-length Encoding method.
BC	Bit-block Compression method.
VBC	Variable Bit-block Compression method.
ML	Multi-level Signature method.
SSF	Sequential Signature Files (alias to SC)
BSSF	Bit-Sliced Signature File method.
CBS	Compressed Bit Slices method.
DCBS	Doubly-Compressed Bit Slices method.
NFD	No False Drops method.

Appendix B.

The following is a reproduction of the non-disclosure agreement between Langruth Elementary School and Enable Software, Inc. covering the release of information describing the internal MS-DOS file format of Enable word processing files.

ENABLE
SOFTWARE

February 14, 1990

Mr. Ron Brown
Langruth Elementary School
Box 148
Langruth, Manitoba ROH ONO
445-2001

Dear Mr. Brown:

Thank you for returning the signed nondisclosure agreement. We are returning a copy to you for your records.

I have included a disk that contains the Enable version 2.0 word processing file format. I hope that you find this information helpful.

If you have any additional questions, please contact our Technical Support department.

Sincerely,

ENABLE SOFTWARE

Kristin C. Parrish

Kristin C. Parrish
Supervisor, Technical Support

enc. (2)

orthway Ten Executive Park • Ballston Lake, NY 12019 • 518-877-8600 • Fax 518-877-8225

NON-DISCLOSURE AGREEMENT

Langruth Elementary School, a Corporation organized and existing under the laws of Manitoba, (hereinafter called "Vendor"), and ENABLE SOFTWARE, INC., a corporation organized and existing under the laws of the State of Delaware (hereinafter called "ESI"), hereby confirm the following understanding between the parties:

The purpose of this Agreement is to protect the Confidential Information of the parties which is disclosed to each other during discussions concerning, but not limited to any trade secrets of ESI and ESI software products. Trade secrets include all aspects of the source code, the design and structure of the programs and their interaction, any unique programming techniques employed therein, any information relating to data formats and structures used in programs, and any other information which is proprietary to ESI and designated by ESI as proprietary and confidential.

"CONFIDENTIAL INFORMATION" is defined as information originated by or peculiarly within the knowledge of the disclosing party or its suppliers which is in writing and marked by disclosing party as confidential or, for such information which is orally disclosed, as may happen during meetings of Vendor and ESI.

The receiving party hereby agrees to maintain such Confidential Information of the transmitting party in confidence, to protect such Confidential Information with the highest degree of care, not to disclose such Confidential Information to others and not to use such Confidential Information except for the purposes of this Agreement. Vendor agrees to use reasonable care in the selection and assignment of personnel to work with such proprietary information.

The vendor hereby agrees to maintain any information conveyed by ESI to the vendor pertaining to the vendor's products applicability to the customer's requirements as confidential, whether such information relates to current products, future products, or modifications to current or future products.

Notwithstanding the foregoing, the receiving party shall have no obligation of confidentiality with respect to:

a. Information which was in its possession or knowledge prior to its receipt from the transmitting party;

b. Information which is now or at any time becomes a matter of public knowledge without disclosure by the receiving party;

c. Information which becomes lawfully known to a third party and is disclosed by such third party to the recipient;

d. Information which is independently developed by the recipient's personnel who have not had access to the Confidential Information of

the disclosing party;

e. Information which is hereafter furnished to others by the transmitting party without restriction on disclosure;

f. Information which may be obtained from examination of a product after it has been delivered in the marketplace.

Nothing herein shall restrict the right of the receiving party to disclose Confidential Information which is disclosed pursuant to judicial order or other lawful governmental action, but only to the extent so ordered.

The obligations set forth above with respect to such Confidential Information shall terminate at the end of five (5) years from the date of receipt of such information or sooner upon mutual agreement of the parties.

All rights and remedies arising out of a disclosure of any Confidential Information after the obligation to be maintained in confidence terminates as set forth above shall be limited to those rights and remedies existing under the patent and copyright laws of the United States.

No amendment or modification of this Agreement shall be valid or binding upon the parties unless made in writing and signed on behalf of each of such parties by their respective duly authorized officers or representatives.

ACCEPTED IN BEHALF OF

Langruth Elementary School
(VENDOR)

Box 148
Address

Langruth, MB R0H 0N0

R. Brown
Signature

R. Brown
Typed Name

Principal
Title

January 25, 1990
Date

ACCEPTED IN BEHALF OF

ENABLE SOFTWARE, INC.
Northway Ten Executive Park
Ballston Lake, NY 12019

Kay E. MacLaury
Signature

Kay E. MacLaury

Director of Technical Services

13 February 1990
Date

References.

[Ben85] Bentley, Jon - A Spelling Checker. Communications of the ACM, Volume 28, Number 5, May 1985.

[Bla85] Blair, David C. and Maron, M.E. - An Evaluation of Retrieval Effectiveness for Full-Text Document-Retrieval System. Communications of the ACM, Volume 28, Number 3, March 1985.

[Blo70] Bloom, Burton H. - Space/Time Tradeoffs in Hash Coding with Allowable Errors. Communications of the ACM, Volume 13, No. 7, July, 1970.

[Boy77] Boyer, Robert S. and Moore, J. Strother - A Fast String Searching Algorithm. Communications of the ACM, Volume 20, Number 10, October 1977.

[Car79] Cardenas, Alfonso F. - Data Base Management Systems. Allyn and Bacon, 1979.

[Cha89a] Chang, Walter W. and Schek, Hans J. - A Signature Access Method for the Starburst Database System. Proceedings of the Fifteenth International Conference on Very Large Databases, Amsterdam, 1989.

[Cha89b] Chang, Joe W., Lee, Joon H., Lee, Yoon J. - Multikey Access Methods Based on Term Discrimination and Signature Clustering. Proceedings of ACMSIGIR 1989, 12th Annual International Conference on Research and Development in Information Retrieval.

[Chr84] Christodoulakis, Stavros and Faloutsos, Christos - Design Considerations for a Message File Server. IEEE Transactions on Software Engineering, Vol. SE-10, No. 2, March 1984.

[Chr86a] Christodoulakis, Stavros and Faloutsos, Christos - Design and Performance Considerations for an Optical Disk-Based, Multimedia Object Server. IEEE Computer, December 1986.

[Chr86b] Christodoulakis, S., Theodoridou, F., Ho, F., Papa, M., and Pathria, A. - Multimedia Document Presentation, Information Extraction, and Document Formation in MINOS: A Model and a System. ACM Transactions on Office Information Systems, Vol. 4, No. 4, October 1986.

[Dat86] Date, C. J. - An Introduction to Database Systems, Volume 1, Fourth Edition. Addison-Wesley, 1986.

[Du89] Du, David H. C., Ghanta, S., Maly, Kurt J., and Sharrock, Suzanne M. - An Efficient File Structure for Document Retrieval in the Automated Office Environment. IEEE Transactions on Knowledge and Data Engineering, Vol. 1, No. 2, June 1989.

[Fal84] Faloutsos, Christos - Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation. ACM Transactions on Office Information Systems, Vol. 2, No. 4, October 1984.

[Fal85a] Faloutsos, Christos - Access Methods for Text. ACM Computing Surveys, Volume 17, No. 1, March 1985.

[Fal85b] Faloutsos, Christos - Signature Files: Design and performance comparison of some signature extraction methods. Proceedings of the ACM SIGMOD, pp. 63-82, May 1985.

[Fal87a] Faloutsos, Christos and Christodoulakis, Stavros - Description and Performance Analysis of Signature File Methods for Office Filing. ACM Transactions on Office Information Systems, Vol. 5, No. 3, 1987.

[Fal87b] Faloutsos, Christos - Signature Files: An Integrated Access Method for Text and Attributes, Suitable for Optical Disk Storage. Computer Science Technical Report Series, University of Maryland, June, 1987.

[Fal87c] Faloutsos, C. and Chan, R. - Fast Text Access Methods for Optical Disks: Designs and Performance Comparison. Proceedings of 14th International Conference on VLDB - Los Angeles, California, 1988, pp. 280 - 293.

[Fal87d] Faloutsos, Christos and Christodoulakis, Stavros - Optimal Signature Extraction and Information Loss. ACM Transactions on Database Systems, Vol. 12, No. 3, September 1987.

[Fal88] Faloutsos, Christos - Gray Codes for Partial Match Retrieval and Range Queries. IEEE Transactions on Software Engineering, Volume 14, No. 10, October 1988.

[Fal89] Faloutsos, Christos and Roseman, Shari - Fractals for Secondary Key Retrieval. Computer Science Technical Report Series, University of Maryland, May, 1989.

[Fil69] Files, J.R. and Huskey, H. D. - "An Information Retrieval System Based on Superimposed Coding". Proceedings of the AFIPS Fall Joint Computer Conference, 1969, pp. 423-432.

[Fre85] French, James C. - IDAM File Organizations. UMI Research Press, 1985.

[Fuj84] Fujitani, Larry - Laser Optical Disk: The Coming Revolution in On-Line Storage. Communications of the ACM, Volume 27, Number 6, June, 1984.

[Gal75] Gallager, Robert G. and Van Voorhis, David C. - Optimal Source Codes for Geometrically Distributed Integer Alphabets. IEEE Transactions on Information Theory, vol. IT-21, pp. 399-401, March 1975.

[Gol66] Golomb, Solomon W. - Run Length Encodings. IEEE Transactions on Information Theory, vol. IT-12, pp. 399-401, July 1966.

[Knu73] Knuth, Donald E. - The Art of Computer Programming, Volume 3 - Sorting and Searching. Addison-Wesley, 1973.

[Lar83] Larson, Per-Ake - "A method for speeding up text retrieval". Proceedings of the ACM SIGMOD, May 1983.

[Lar84] Larson, Per-Ake and Kajla, Ajay - File Organization: Implementation of a Method Guaranteeing Retrieval In One Access. Communications of the ACM, Volume 27, Number 7, July, 1984.

[Les68] Lesk, M. E. and Salton, G. - Computer Evaluation of Index and Text Processing. Journal of the ACM, Volume 15, No. 1, January 1968.

[Mem89] Memex Information Systems, Ltd - Textract Navigate and Search: Technical and User Documentation Release 3.1, Part No. 2703-103110/C. Memex Information Systems, 1989.

[Mal83] Maloney, James J. (ed.) - Online Searching Technique and Management. American Library Association, 1983.

[McI82] McIlroy, Douglas M. - Development of a Spelling List. IEEE Transactions on Communications, Vol. COM-30, No. 1, January 1982.

[Ore82] Orenstein, J. A. and Merrett, T. H. - A Class of Data Strucutes for Associative Searching. McGill University Department of Computer Science Technical Report 82-10, June 1982.

[Oto83] Otoo, E. J. and Merrett, T. H. - A Storage Scheme for Extendible Arrays. Computing 31, 1983.

[Par88] Park, Stephen K. and Miller, Keith W. - Random Number Generators: Good Ones are Hard to Find. Communications of the ACM, Vol. 31, No. 10, October 1988.

[Pfa80] Pfaltz, John L., Berman, William J. and Cagley, Edgar M. - "Partial Match Retrieval using Indexed Descriptor Files". Communications of the ACM, Volume 23, No. 9, pp. 522-528, Sept. 1980.

[Ram83] Ramamohanarao, K., Lloyd, John W., and Thom, James A. - Partial Match Retrieval Using Hashing and Descriptors. ACM Transactions on Database Systems, Volume 8, No. 4, December 1983.

[Ram84] Ramamohanarao, K. and Sacks-Davis, R. - Recursive Linear Hashing. ACM Transactions on Database Systems, Volume 9, No. 3, September 1984.

[Ram85] Ramamohanarao, K. and Sacks-Davis, R. - Partial Match Retrieval Using Recursive Linear Hashing. BIT, Volume 25, 1985.

[Ram89] Ramakrishna, M. V. - Practical Performance of Bloom Filters and Parallel Free-Text Searching. Communications of the ACM, Vol. 32, No. 10, October 1989.

[Riv76] Rivest, Ronald L. - Partial Match Retrieval Algorithms. SIAM Journal of Computing, Vol. 5, No. 1, March 1976.

[Rob79] Roberts, C. S. - Partial Match Retrieval via the Method of Superimposed Codes. Proceedings of the IEEE, Vol. 67, No. 12, December 1979.

[Roc88] Rochkind, Marc J. - "Advanced C Programming for Displays". Prentice Hall Software Series, 1988.

[Ros74] Rosenberg, Arnold L. - Allocating Storage for Extendible Arrays. Journal of the ACM, Volume 21, No. 4, pp. 652-670, October 1974.

[Ros75] Rosenberg, Arnold L. - Managing Storage for Extendible Arrays. SIAM Journal of Computing, Volume 4, No. 3, September 1975.

[Rot74] Rothnie, James B. Jr., and Lozano, Tomas - Attribute Based File Organization in a Paged Memory Environment. Communications of the ACM, Vol. 17, No. 2, February 1974.

[Sac82] Sacks-Davis, R. and Ramamohanarao, K. - A Two Level Superimposed Coding Scheme for Partial Match Retrieval. Technical Report 82/11, Department of Computer Science, University of Melbourne, Melbourne, Austrialia, 1982.

[Sac87] Sacks-Davis, R., Kent, A., and Ramamohanarao, K. - Multikey Access Methods Based on Superimposed Coding Techniques. ACM Transactions on Database Systems, Vol. 12, No. 4, December 1987

[Sal71] Salton, Rocchio, Lesk, et al. - The SMART Retrieval System - Experiments in Automatic Document Processing Gerard Salton, Editor. Prentice-Hall, 1971.

[Sal75] Salton, Gerard - Dynamic Information and Library Processing. Prentice-Hall, 1975.

[Sal83] Salton, Gerard and McGill, Michael J. - Introduction to Modern Information Retrieval. McGraw-Hill,1983.

[She89] Shepherd, M. A., Phillips, W. J., and Chu, C.-K. - A Fixed-Size Bloom Filter for Searching Textual Documents. The Computer Journal, Vol. 32, No. 3, 1989.

[Ten89] Tenopir, Carol - Issues in Online Database Searching. Libraries Unlimited, Inc. 1989.

[Tre84] Tremblay, J.P. and Sorenson, P. G. - An Introduction to Data Structures with Applications, Second Edition. McGraw-Hill, 1984.

[Tha90] Tharp, Alan L. and Kotamarti, Usha - Accelerating Text Searching Through Signature Trees. Journal of the American Society for Information Science, Volume 41, No. 2, March 1990.

[Tsi83] Tsichritzis, Dennis and Christodoulakis, Stavros - Message Files. ACM Transactions on Office Information Systems, Vol. 1, No. 1, Jan. 1983.

[Vig88] Vigil, Peter J. - Online Retrieval: Analysis and Strategy. John Wiley and Sons, 1988.

[Wie83] Wiederhold, Gio - Database Design. McGraw-Hill, 1983.