Efficient Data Structures for Information Retrieval

by

Amjad M. Daoud

Dissertation submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science and Applications

APPROVED:

Prof. Edward Fox, Chairman

Prof. Lenwood Heath	Prof. Dennis Kafura
Prof. Clifford Shaffer	Prof. Ezra Brown

August, 1993

Blacksburg, Virginia

Efficient Data Structures for Information Retrieval

by

Amjad M. Daoud

Committee Chairman: Prof. Edward Fox

Department of Computer Science

(ABSTRACT)

This dissertation deals with the application of efficient data structures and hashing algorithms to the problems of textual information storage and retrieval. We have developed static and dynamic techniques for handling large dictionaries, inverted lists, and optimizations applied to ranking algorithms. We have carried out an experiment called REVTOLC that demonstrated the efficiency and applicability of our algorithms and data structures. Also, the REVTOLC experiment revealed the effectiveness and ease of use of advanced information retrieval methods, namely extended Boolean (p-norm), vector, and vector with probabilistic feedback methods. We have developed efficient static and dynamic data structures and linear algorithms to find a class of minimal perfect hash functions for the efficient implementation of dictionaries, inverted lists, and stop lists. Further, we have developed a linear algorithm that produces order preserving minimal perfect hash functions. These data structures and algorithms enable much faster indexing of textual data and faster retrieval of best match documents using advanced information retrieval methods. Finally, we summarize our research findings and some open problems that are worth further investigation.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. Edward A. Fox for his constant guidance and effort in encouraging the completion of this dissertation. I would like to thank Dr. Heath for giving me the opportunity to work on new and innovative ideas on perfect hashing. His contributions convinced me that theoretical problems have significant practical applications. I would also like to thank the Computer Science Department staff, Linda Wilson, Newman Library staff, the Computing Center, Statistical Consulting Center, and Dr. G.I. Holtzman for their constant help during the course of this work. I would also like to thank the Center for Learning Resources, the Computing Center at Virginia Tech and OCLC, Inc. for funding the largest IR catalog experiment ever run (REVTOLC). I take the opportunity to thank my parents Mr. and Mrs. Mustafa for giving me the opportunity to pursue higher education. Finally, I would like to express my sincere appreciation and gratitude to my wife Lydia for her patience and love that made it possible for me to complete this dissertation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

The problem of storing, managing, and accessing information is a classic problem in human society. Modern libraries depend on online catalogs and information retrieval systems to handle the explosion of information. It is estimated [80] that the Library of Congress archives roughly 25 terabytes in its collection. This is undoubtedly most valued if it is accessed and searched effectively and efficiently. To browse through this volume using normal human techniques would be nearly impossible. It is also estimated [80] that it would take more than a decade to index and catalog such a vast collection as the Library of Congress using current computer technology.

So, the problem of enhancing conventional search techniques to handle the vast amount of online information and the various sources that contribute to this information explosion have taken on an even greater significance.

Generally, conventional search techniques include one or more of the following:

1. traversing menus,

2. exact matching of titles or title prefixes, or

3. submitting Boolean queries.

Most commercial information retrieval systems adopt the Boolean retrieval strategy. Yet, it is well known [72] that formulating an information need or question into a suitable Boolean query is difficult. The other two methods mentioned above have limited applicability and power.

Less popular are a number of advanced approaches that explore the statistical characteristics of various collections. Although many researchers have developed effective retrieval techniques (i.e., extended Boolean, vector), and have demonstrated their utility in the SMART experimental system, few of these advances have found their way to commercial systems.

1.1 The SMART Retrieval System

The SMART information retrieval package consists of a set of programs that make up a fully automatic document retrieval system [67]. The SMART retrieval system allows for the indexing and searching of online document collections. As information grows at an explosive rate, research must provide more effective methods of storing and accessing this information. The SMART system is an effective test-bed for investigating these new methods. The SMART system was originally implemented around the vector space model [73]. In this model, each term is associated with a dimension in a multi-dimensional vector space.

Over the years, SMART [73, 67] has been used to investigate a number of advanced retrieval techniques in experimental situations. Some of the early studies indicate that SMART's fully automatic indexing and retrieval performance compared well with the conventional retrieval systems where trained indexers and searchers used a special controlled vocabulary [66]. The SMART system has undergone several changes and was rewritten in the C language to work in the UNIX environment [28]. It was later extended to work efficiently for online users [9].

In SMART, the general model of information retrieval consists of accessing a set of online documents. These documents vary from electronic messages to library catalog entries to full text journal articles. Each document could contain several types of data such as date of publication, author name, a supplied list of keywords, or even a list of other documents that are cited. In addition, other types of information may be contained, such as dates, times, or numbers. When a user submits a query to the system, each of these classes of information could be useful. In order to distinguish between classes, for each type of document concept, the notion of classification type or ctype was developed [29]. This was made possible by extending the vector space model. The SMART system is also capable of building feedback queries automatically [72]. The SMART system includes a set of evaluation and low level data access functions that render it useful for experimental studies. The data access functions allow looking at stored information as sequences of tuples, and support efficient access to individual tuples.

The SMART package has been extended at Virginia Tech with a user interface that is based on the CURSES package [2]. The CURSES package provides a simple, yet quite portable interface. The new user interface is part of an effort to test and develop a new online retrieval system to access catalog information from Virginia Tech's Newman Library.

1.2 Research Contributions

In this research, we have studied the effectiveness and applicability of advanced retrieval methods. In the process, we have developed and used tools whose utility we believe extends beyond this dissertation. We have developed efficient perfect hashing algorithms that produce minimal perfect hash functions and order preserving minimal perfect hash functions that have optimal sizes. We have applied these algorithms to large collections for the first time (i.e., over 3.8 million keys). Moreover, these algorithms and data structures were used to organize indexes and provide faster access, and to provide effective and efficient access to very large collections, dictionaries, and online library catalogs, using an extended version of the SMART system. We describe an experiment called REVTOLC that was run to test and improve search methods for online catalogs. The REVTOLC experiment revealed the effectiveness and ease of use of the advanced information retrieval methods, namely extended Boolean (p-norm) [71], vector [73], and vector with probabilistic feedback methods [68].

1.2.1 Main Contributions

The main contributions of this research, in roughly decreasing order of importance, are as follows:

1. We have developed an efficient minimal perfect hash function generator that produces optimal MPHFs . The expected time required by the algorithm is $\Omega(n^{1+\eta})$ to find an MPHF of size $n/(\eta \ln 2)$ bits, where η is the efficiency of representing an MPHF [41]. The algorithm handles large key collections, as well as small ones, efficiently. Also the algorithm requires minimal main memory, so it is quite practical for very large key sets. The algorithm computed a minimal perfect hash function for the OCLC key collection of size 3.8 million keys for the first time.

2. We have developed a novel order preserving minimal perfect hashing algorithm for handling sorted key collections, large dictionaries, and inverted lists.

3. We have developed a dynamic perfect hashing file structure that is incremental and has high storage utilization due to the use of compact minimal perfect hashing for handling buckets. The new data structure outperforms other tree based structures, is well balanced, and achieves expected constant time for insertion, and worst case constant time bounds for retrieval and deletion. We illustrated its efficiency with well known large key sets such as the UNIX dictionary (24K keys) and the OCLC key set (3.8 million keys).

4. We have developed fast best match search algorithms that enhance the query response time in vector based information retrieval systems. These algorithms return the relevant document set computing an upperbound on closeness; eliminating the need for an exact computation in many instances and eliminating useless disk accesses.

5. An underlying theme of this research, as the title of this dissertation suggests, is a comparison of the advanced retrieval methods. Based on the various results obtained in this dissertation, we conclude that:

- The REVTOLC experiment proved that advanced retrieval techniques, namely extended Boolean (p-norm), probabilistic feedback and vector retrieval are more effective than Boolean searching.
- The vector methods retrieved more documents than the p-norm and Boolean method.
- The vector methods found more relevant documents than the p-norm and Boolean method.
- Vector with Auto Feedback was found to be the easiest method to use. Boolean was found to be the most difficult. The p-norm was found to be less difficult than the Boolean method.
- Vector with Auto Feedback was chosen as an effective aid for casual search. Vector methods were considered as more effective for comprehensive searching than p-norm and Boolean methods. The Boolean method was most disliked for casual search.

1.3 Dissertation Overview

In Chapter 2, we survey related literature on information retrieval methods. In Chapter 3, we provide detailed design and discussion of the REVTOLC experiment. In Chapter 4, we present our optimal minimal perfect hashing algorithms. In Chapter 5, we introduce novel order preserving perfect hashing algorithms. In Chapter 6, we extend our minimal perfect hashing and order preserving prefect hashing scheme to handle insertion and deletions. We also describe dynamic data structures based on these algorithms. In Chapter 7, we summarize our research findings. Finally, in Chapter 8, we discuss some open problems that are worth further investigation.

Chapter 2
Related Research

2.1 Introduction

Information retrieval is concerned with the representation, storage, organization, and accessing of information items. Usually, there are no restrictions on the type of items handled in information retrieval and, in fact, most ordinary systems have items consisting of narrative information. This information needs to be analyzed and classified in order to determine the information content and to assess the role each item may play in satisfying the information need of the users. This information then needs to be retrieved in an efficient manner.

Belkin and Croft [4] have given an extensive overview of some of the conventional and advanced retrieval techniques available. These techniques mainly address the issue of comparing a representation of a query with representations of texts for the purpose of identifying, retrieving, and ranking texts in a text collection that might be relevant to a given query. Some of the general research in the field of information retrieval and the associated retrieval techniques available have been covered quite well in [74]. Bookstein [6] has recently reviewed the use of some of the more advanced techniques based on probability and fuzzy set theories and their related retrieval methods. Although many advances have been made in finding newer and better retrieval techniques, only a few "conventional" techniques are actually used in large scale operational information retrieval systems. These techniques are mainly the Boolean and string searching varieties. Belkin and Croft [4] have rightly put this situation as, "These techniques have become established more through practice than theory." Some of the newer techniques (i.e., probabilistic retrieval, clustering, etc.), with strong theoretical basis, are well developed, but have been used mostly for an experimental setting. Belkin and Croft [4], in their review of retrieval techniques, tried to explain why the experimental techniques (which very often performed better on standard measures) are not used in the operational environment. They claim that advanced retrieval techniques then had not been validated on large collections.

For the last two decades, researchers have explored the statistical characteristics of various collections of messages, bibliographic citations, and other types of documents. They developed automatic indexing techniques [74], prepared lexical-relational thesauri [31], devised efficient storage structures and algorithms, and proposed and evaluated retrieval approaches. Yet as mentioned above, few of these advanced retrieval techniques have found their way into commercial information retrieval systems [32].

There are several excellent books about information retrieval that discuss problems related to information retrieval in general and advanced information retrieval methods in particular. [74] and [84] provide a thorough survey of retrieval approaches, with many algorithms, file organizations, and data structures. Excellent surveys have appeared on information retrieval models and related subjects [4]. Some of the recent developments are efficient ranking algorithms [33], advanced automatic feedback [69], improved techniques for processing queries [29, 28], fast inversion routines [32], and optimized inverted file searches [10].

2.2 File Organization

In a typical information retrieval system, many files are collections of records, each consisting of several keys or other attributes. Usually, these files are large and must be stored on secondary storage devices such as a magnetic disk unit, magnetic tape, or CD-ROM. Some of the measures that can be used to evaluate file organization include:

- storage required for the file structure,
- time required to read an arbitrary record,

- time required to read the next record,
- time required to insert a record,
- time required to modify data item values in a record,
- time required to read all records sequentially, and
- time required to reorganize a file.

Some of the common file organizations include sequential, indexed sequential, B-tree, search trees, and list structures. Knuth [44] has surveyed the basics of techniques for organizing a file and its index. The B-tree storage scheme is discussed at length by Comer [19].

Another scheme, called hashing, has been known for the past two decades for its fast read access. Efficiency of this scheme mainly depends on the time complexity of computing the hash function and the loading factor involved. There have been several efforts [12] to develop algorithms to produce perfect hash functions. The SMART information retrieval system, which has already been discussed, makes extensive use of hashing for storing information. There have been some interesting variations in this scheme such as order preserving hash functions, and multi-key hashing schemes [24].

In summary, we can say that there are a number of techniques available for file organization; although some techniques are without a doubt superior to the others for a particular situation, selecting a technique is rather an application-dependent process.

2.3 Hashing

Hashing is considered to be the software equivalent of associative memories [43]. Within the context of external files, hashing schemes are applied for the fast access of information in very large files through the use of content addressing. The address of a record is determined by a simple arithmetic transformation of the content of its key.

Kahonen [43] and Knuth [44] discuss hashing mainly within the context of static tables. These "conventional" methods are efficient if one can make an accurate initial estimate for the size of the file. Even if the specified size varies with time, the performance does not degrade if the file is stable. A stable file [44] refers to a structure where both insertions and deletions occur with equal probability, keeping the size of the file almost unchanged.

The idea behind key-to-address transformation methods for external files is to store and retrieve records of information according to keys that uniquely identify the information. Algorithms for hashed files have two components: 1) the hashing function, and 2) the collision resolution method. A survey of hashing functions and collision resolution techniques can be found in [22], [50], [44]. Carter et al. [11] characterize classes of functions which are suitable to be used with hashing techniques.

2.3.1 Hashing Functions

The hashing function $h(k)$ is a mapping of the key space S onto the address space Z_m where $h(k) : S \rightarrow Z_m$. Hashed files rely on the uniformity of the underlying key-to-address transformation to achieve good performance. The most popular hashing function is the division method defined as: $h(k) = k \pmod{m}$, where m is suggested to be a prime number for good performance of the method [44]. Other methods are the midsquare hashing method, folding hashing, radix transformation, algebraic coding, and multiplicative methods [44]. Also, distribution dependent hashing techniques have been proposed [82]. These methods approximate the cumulative probability distribution of the keys over the key space $F_{Z(k)}$. Some of them are the digit analysis method, piece-wise linear function, multiple frequency distribution, and the Fourier series function [22]. The performance of the most widely used hashing functions is reported in [23, 54]. Deutsher et. al. [82] characterize distribution dependent hashing functions and compare them with the division method. These methods are found to be efficient if static files are considered. Also, [22] introduces the notion of order preserving hashing based on distribution dependent functions.

2.3.2 Collision Resolution Methods

When dealing with hashing schemes the underlying key-to-address transformation function will map different key values to the same storage address location. This is referred to as a *collision.* Collisions are handled with overflow management schemes or collision resolution methods. Secondary storage systems are divided into storage blocks referred to as *buckets.* Usually, these address locations have the capacity to hold more than one record. In this case, collisions occur when the number of records that hash to a bucket exceeds its capacity. Typical collision resolution methods are open addressing, rehashing, separate chaining, coalesced chaining, and repeated hashing [44]. Several techniques use a prime (or home) area for overflow storage.

With open addressing, if the key of a record is mapped to a bucket y, which has already reached its capacity, then the home storage buckets are probed following a predetermined sequence until a free location is found. When the sequence follows consecutive buckets starting with bucket y then the method is referred to as *linear probing* or progressive overflow.

Rehashing also uses a home area to store overflow records. It is proposed as an alternative to open addressing to alleviate the problems of primary and secondary clustering as discussed by Knuth [44]. With rehashing, the probe sequence is determined by a second hashing function, which is independent of the initial function. Knuth suggests hashing functions of the form

$$h_1(k) = k \pmod{m}$$

and

$$h_2(k) = 1 + (k \pmod{m-2})$$

with k as the key and m the file size in number of storage buckets. In [22] rehashing is found to be impractical for external files.

With separate chaining, the overflowing records are stored in a linked list in an overflow storage area, which is independent of the home storage area of the file. If the overflow chains are managed with buckets of capacity greater than one, the technique is referred to as overflow bucket chaining.

Coalesced chaining mixes the two concepts of open addressing and chaining. The overflow chain is stored in the home storage area. This method was analyzed by Vitter [85].

The Repeated hashing relies on multiple hash areas. When a record overflows from one area, it is rehashed onto a smaller storage area. This idea can be applied recursively resulting in multiple hashing areas. Repeated hashing method was analyzed by Larson [45], where he concluded that its usefulness is doubtful.

The performance of hashed files is usually measured in terms of an average cost for different parameters, namely successful and unsuccessful searches, record insertion and deletion. Generally, the cost is expressed in the number of probes related to the number of disk accesses to secondary storage devices.

Yao [88] follows Ullman's conjecture [83] to prove that uniform hashing is optimal. While this is true for static hashing methods, it does not hold for dynamic schemes that require overflow handling procedures [58, 55]. Continuously changing environments result in files whose sizes cannot be accurately estimated. This led [46] to the design of file structures based on dynamic hashing methods. Although with no clear boundaries, these methods are classified with two distinct classes: dynamic hashing structures with and without a directory.

2.3.3 Dynamic Hashing Structures with Directory

Dynamic hashing structures with directory methods are represented by the extendible hashing method [25] and Larson's dynamic hashing [46]. Another method of direct interest is trie hashing [51]. The principle behind these techniques is that the file expansion is dynamically performed by splitting, upon a collision, and overflowing a bucket onto two new buckets using a split hash function.

The search cost of these methods is equal to the number of probes to the directory, and one additional probe to the leaf pages pointed to by the directory. If sufficiently small, the directory fits in main memory and does not require an additional disk access. With very large files, this assumption cannot be made, and one typically needs two disk accesses to retrieve the required record. For insertion, the number of disk accesses required is very close to two unless directory expansion or leaf splitting is required. The storage space utilization of these methods is found to vary periodically around an average value [25]. This value is asymptotically equal to 0.69 and is equivalent to the value for B-trees and their derivatives [19]

Extendible hashing was introduced and analyzed by Fagin et al. [25], and has been exhaustively analyzed by Mendelson [57], Yao [89], and Flajolet [27]. Their analysis concentrated on the asymptotic behavior for the different extendible hashing and trie file characteristics. Mendelson's analysis characterizes the probability distribution of the directory size and depth of extendible hashing and gives a design methodology for practical systems. Flajolet's analysis covers the class of dynamic hashing schemes with directories (digital search trees and their derivatives). His work, together with Yao's analysis, results in bounds for the different parameters of extendible hashing.

Some extensions to extendible hashing are found in [52], where exponential hashing is used to cancel out the periodic behavior of the storage utilization of extendible hashing. Also, overflow management is applied so that the directory size is kept within an upper bound determined by the available main memory area.

Dynamic hashing was first proposed by Larson [45] and its performance was analyzed in detail by Regnier [62]. Extensions to linear dynamic hashing were proposed as well: Scholl introduced new file structures derived from linear dynamic hashing [76]. His two schemes increase the storage utilization of linear dynamic hashing and rely on overflow management techniques. Taminen [81] proposed a data model for order preserving extendible hashing and analyzed binary trie structures. Regnier [63] presented an analysis of grid file algorithms as multidimensional generalizations of extendible hashing and linear dynamic hashing.

An important extension proposed for dynamic hashing schemes with directories is the concept of *elastic buckets*. This concept was first introduced by Lomet [53] within the context of indexed file organizations and further applied to B+ trees by Baeza-Yates et. al.

[3]. The elastic buckets concept is of particular interest to this research and is presented in detail.

The usual method for file expansion in extendible hashing and linear dynamic hashing is to redistribute the records that hash to a bucket onto two buckets whenever an overflow condition occurs. This is performed by rehashing the keys of the records using an additional hashing depth of one bit. For example, assume that the hashing function transforms keys in 8 bit code words. Consider a bucket pointed to by directory entry 3 of an extendible hashing file which is addressed by the rightmost two bits. All keys that hash to this bucket have a code word representation "xxxxxx11"; with "x" representing a don't care value. Assume that due to multiple insertions this bucket overflows. A new bucket is created and is pointed to by entry number 7 in the directory. Then, the records are redistributed onto the two buckets using the leftmost 3 bits. This process is referred to as bucket splitting and doubles the storage space allocated to a key range.

The idea with elastic buckets is, for an overflow condition, to reassign the records that hash to a bucket of capacity $b_{\min} = b_0$ to a bucket of capacity $b_1 > b_0$ records. When a home bucket with a capacity of b_i records overflows, the records are written onto a bucket with a capacity of $(b_i + \delta b)$; where δb is an incremental bucket capacity. This process is continued until a predetermined maximum of bucket expansion is reached (say $b_{\max}$). Elastic buckets expand the storage space allocated to a key range in small increments; this is referred to as a technique of partial expansions [53]. This technique results in file structures that grow incrementally delaying the splitting of the home buckets. Applied to dynamic hashing with directories, elastic buckets improve the usage of the directory space as well as its storage utilization factor.

Lomet studies elastic buckets applied to B-trees and points out that the technique can be applied to any indexed file organization. As an example, consider an extendible hashing file with elastic buckets with $b_{\min} = 20$, $b_{\max} = 35$ and $\delta b = 5$ records. One begins with a bucket capacity of 20 and expands into 3 partial expansions to a bucket capacity of 35 records. When this capacity overflows (i.e., at the 36th insertion), the 4th partial expansion takes place and the bucket splits onto two buckets with the capacity of 20 records. Note that approximately 18 records will be allocated to each bucket.

In general, if r is the number of partial expansions to a full expansion, then $b_{\text{min}} = r * \delta b$ and $b_{\text{max}} = (2*r-1)*\delta b$. With a minimum bucket capacity of 20 records and a $\delta b = 5$ records (i.e., 4 partial expansions): the first partial expansion changes the utilization factor to 84% (21/25), the second partial expansion changes the utilization factor to 86.7% (26/30), the third partial expansion changes the utilization factor to 88.5% (31/35). Finally, the split onto two buckets of 20 records each would lead to a change in the utilization factor for the buckets to 90% (36/40). It is clear that the utilization factor of such a scheme is assured to be above the 84% mark.

2.3.4 Directoriless Hashing Structures

Dynamic hashing methods that do not require a directory are mainly represented by Litwin's linear hashing [51]. In general, to implement linear hashing, starting from a file of N buckets, one needs a sequence of hashing functions $(h_0(k), h_1(k), ..., h_i(k), h_{i+1}(k), ...)$ with the following properties:

a) $0 \leq h_0(k) \leq N - 1$

b)

$$h_{i+1}(k) = \begin{cases} h_i(k) & \text{if } h_i(k) \leq \text{sp} \\ h_i(k) + N * 2^i & \text{otherwise} \end{cases}$$

The simple remainder hashing function is one which has the above property. To achieve an even load, the two cases for $h_{i+1}(k)$ should occur with equal probabilities. To keep track of the state of the file, two variables are needed: L, called the generation number, which counts the number of times the file size has doubled; and *sp*, the split pointer, which points to the next page to split. Figure 2.1 illustrates the process of splitting a bucket in linear hashing.

Expanding the file by one page requires the local reorganization of two pages: the one being split and the new page appended to the end of the file. The criterion to trigger an

Figure 2.1: Splitting a Bucket in Linear Hashing Scheme

expansion was based upon the occurrence of a collision. This mechanism is referred to as uncontrolled splitting [48]. Another mechanism is to split when a threshold for the storage utilization is exceeded; then the file is split by one page. This is called storage utilization controlled splitting. Another mechanism is one where the primary buckets load factor is controlled and is referred to as load factor controlled expansion [46].

It is noted that the split sequence follows a sequential pattern from the first to the N^{th} bucket. This means that the split does not generally take place on the bucket that undergoes a collision, which is typical of directoriless dynamic hashing methods. A collision resolution method is proposed to resolve the split by assigning overflow chains [51]. If the data is uniformly distributed, the performance of the file structure is not degraded by the overflow chains. Larson reports an average retrieval cost of 1.17 and insertion of 3.57 disk accesses for a home page size of 20, overflow page size of 5 records and storage utilization set to 85% [48].

A major extension of linear hashing is the method of partial expansions using elastic buckets. With partial expansions, the size of the file is doubled over a full expansion cycle composed of a number of partial expansion cycles. Each partial expansion cycle relocates expanded buckets into larger buckets, but keeps records physically contiguous. Splitting the expanding bucket occurs during the last partial expansion cycle of a full expansion cycle. Keeping records physically contiguous is very useful when used for sequential access. A brief overview of different linear hashing schemes is provided in [48]. Litwin's original linear hashing relies on bucket chaining as the collision resolution method. An important version of linear hashing is spiral hashing [55, 58]. Spiral hashing uses exponential hashing to cancel out the periodic behavior of linear hashing over an expansion cycle.

Recent work implemented file structures based on linear hashing that achieved one access retrieval [47, 46]. These structures use a single file version of linear hashing and a small amount of internal storage in a table. Typically, one byte of internal storage is needed for each secondary storage bucket. The table stores descriptors and separators that are used to uniquely determine the location of any record in one disk access.

Chapter 3
The REVTOLC Experiment

3.1 Introduction

Recent advances in computer hardware and dramatic improvements in CPU speed and disk storage have enabled us to carry out experiments regarding the most *effective* methods for the storage and retrieval of online catalog records. A controlled experiment was conceived to accurately determine the effectiveness of the advanced techniques applied.

Boolean searching has been commonly used in commercial information retrieval systems. It was felt in [18] that Boolean searching was the most effective method that was available. However, mostly with small test collections, other studies have indicated that some of these conventional techniques are not very effective [5].

The REVTOLC (Retrieval Experiment Virginia Tech Online Library Catalog) study represents on-going attempts to answer many of these questions with a very large collection of catalog records. The REVTOLC experiment was able to demonstrate:

1. the feasibility of applying advanced retrieval methods to provide effective access to large online public access catalogs; and

2. the effectiveness and ease of use of advanced retrieval methods using a realistically sized catalog collection and enough participants so that statistical assessments will be valid and so that interactions between user characteristics and methods can be measured.

In order to analyze collected information accurately, background information on the users in this experiment was collected through online questionnaires. Measurements were made regarding the time required for searching, documents marked as relevant by users and librarians, user interaction during the relevance feedback iterations, and user evaluations and preferences.

3.2 Advanced Information Retrieval Methods

There has been very little reported research on the application of advanced information retrieval methods to large online catalogs [32] (i.e., more than 100,000 records). Most of the studies, comparing advanced retrieval methods, have used relatively small test collections. The REVTOLC pilot study involved about 300,000 entries and a relatively small sample of volunteers drawn from all sections of the freshman English course offered during the Winter quarter 1987. A total of 53 subjects successfully completed the required sessions. Only online questionnaires were analyzed and reported [32]. It was not well understood why the extended Boolean (p-norm) and relevance feedback methods were not well received, but that may have been due to a complex user interface and insufficient time for training the subjects. Next, we review the Boolean and three advanced information retrieval models used in the REVTOLC experiment.

3.3 Boolean Model

In the standard Boolean retrieval model, the search request is formulated by using terms reflecting the user's information needs. Whereas the document content is often expressed by sets of terms, the queries are represented by Boolean expressions, consisting of search terms interrelated by the Boolean operators *and*, *or*, and *not*. Boolean systems are easy to implement using inverted files, where the Boolean operators *and* and *or* correspond to list intersection and list union operations. The documents retrieved for a given query are those that contain index-terms in the combination specified by the query. The Boolean model is popular in operational environments because it is easy to implement and is very efficient in terms of the time required to process a query. Boolean retrieval systems are also capable of giving high performance in terms of recall and precision if the query is well formulated.

The Boolean approach has been criticized for being inflexible and unfriendly [69], since it often is hard to formulate effective Boolean queries.

Other drawbacks of the Boolean model [69] are :

1. The size of the output obtained in response to a user query is difficult to control, since it depends on the occurrence frequencies of the query terms. Many documents may be retrieved, or none at all.

2. Boolean queries are strict. For example, in response to an *or* query (A *or* B *or* ... *or* Z), a document containing only one query term is considered to be as important as a document containing all query terms. For an *and* query (A *and* B *and* ... *and* Z), a document containing all query terms but one is considered as useless as a document containing none of the query terms.

3. The Boolean model gives counterintuitive results for certain types of queries. For example, consider a query of the form *A and B and C and D and E.* A document indexed by all but one of the above terms will not be retrieved in response to this query. Intuitively it appears that the user would be interested in such a document, and that it should be retrieved. Similarly, for a query of the form *A or B or C or D or E*, a document indexed by any of these terms is considered just as important as a document indexed by some or all of them. This limitation of the Boolean model is due to its strict interpretation of the Boolean operators. Hence, we need to *soften* the Boolean operators, and account for the uncertainties that are present in choosing them and the terms in queries and documents. We can do this by making the *and* query behave a bit like the *or* query and the *or* query behave somewhat like the *and* query.

4. The standard Boolean model has no provision for ranking documents. However, systems combining features of both the Boolean and the vector models have been built to allow for ranking the result of a Boolean query, eg., the SIRE system [60]. Ranking

the documents in the order of decreasing similarity often allows the user to see the most relevant document first. Also, the user would be able to sequentially scan the documents, and stop at a certain point if he/she finds that many of the documents are no longer relevant to the query.

5. During the indexing process for the Boolean model, it is necessary to decide whether a particular document is either relevant or non-relevant with respect to a given index-term. In the Boolean model there is no provision for capturing the uncertainty that is present in making indexing decisions. Assigning weights to index terms adds information during the indexing process.

6. The Boolean model has no provision for assigning importance factors or weights to query terms. Yet, searchers often can rate or rank index-terms in queries based on how indicative they are of desired content. Thus, it would be useful to allow weights to be assigned to (some) query terms, to indicate that the presence or absence of a particular query term is more important than that of another term.

7. The dependence on Boolean logic requires that the users be trained in construction of queries, because it is not intuitively obvious, and differs from common natural language usage.

3.4 Vector Model

The vector model [73] assigns a different dimension in a multi-dimensional space to each term in a collection. In contrast to the Boolean Model, the vector model allows term weighting and provides ranked output. Documents and queries are points in the space, so their vectors can be correlated. A similarity measure can be computed between a query vector and document vectors, and the retrieved documents can be presented to the user in decreasing order of the query-document similarity. When a user looks at documents "near" a query, and finds some that are relevant, a feedback process can be invoked to automatically move the query closer to the relevant documents.

3.5 Vector with Automatic Relevance Feedback

In a vector processing system, an approximation to an optimal query vector may be generated by adding to an initial query formulation terms extracted from previously retrieved documents identified as relevant to the query, and analogously by subtracting from an initial query formulation terms extracted from previously retrieved documents identified as nonrelevant documents [69]. Assuming that the relevance factors have been computed for all documents, the items may then be ranked and retrieved in decreasing order according to their presumed importance to the user. Feedback queries typically perform much better than original queries.

3.6 Extended Boolean Model (p-norm)

Besides allowing document-weights for index-terms, the *extended* Boolean model [69] [70], also allows query terms to have weights. In the p-norm model, a document D with weights d_{A_1}, d_{A_2}, ..., d_{A_n} with respect to index-terms A_1, A_2, ..., A_n is considered to be a point with co-ordinates $(d_{A_1}, d_{A_2}, \ldots, d_{A_n})$ in an n-dimensional space. These document weights are generally obtained using term frequency and inverse document frequency statistics, with proper normalization. Also, the similarity computation between a Boolean query statement and a set of terms representing a document is based on L_p vector norm computations, and is controlled by a variable parameter $p, 1 \leq p \leq \infty$. When p is large, the classical Boolean operators maintain their strict interpretations. As p gets smaller, the interpretation of Boolean operators softens. When p reaches 1, the distinction of Boolean operators *or* and *and* is lost completely. Sophisticated users may wish to assign the p values and query weights. Extensive experimentation [49] demonstrated that system assigned p-values (i.e., $p = 2$ throughout a query) can give good results, and that uniform query weighting of 1 or based on inverse document frequency will lead to effective retrieval.

Consider a set of terms $A_1, A_2, \ldots, A_n$, and let d_{A_i} represent the weight of term A_i in some document $\mathbf{D} = (d_{A_1}, d_{A_2}, \ldots, d_{A_n}), 0 \leq d_{A_i} \leq 1$. An *or* query Q_{or} is represented as

$$Q_{or}(p) = [(A_1, a_1) or^p (A_2, a_2) or^p \ldots or^p (A_n, a_n)]$$

where a_i specifies the weights of query term $A_i, 0 \leq a_i$ and $1 \leq p \leq \infty$. Similarly, an *and* query Q_{and} is

$$Q_{and}(p) = [(A_1, a_1) and^p (A_2, a_2) and^p \ldots and^p (A_n, a_n)]$$

The similarity between the document D and the queries $Q_{or}(p)$ and $Q_{and}(p)$ is computed as

$$sim[D, Q_{or(p)}] = \left(\frac{a_1^p {d_{A_1}}^p + a_2^p {d_{A_2}}^p + \ldots + a_n^p {d_{A_n}}^p}{a_1^p + a_2^p + \ldots + a_n^p} \right)^{\frac{1}{p}}$$

$$sim[D, Q_{and(p)}] = 1 - \left(\frac{a_1^p (1 - d_{A_1})^p + a_2^p (1 - d_{A_2})^p) + \ldots + a_n^p (1 - d_{A_n})^p}{a_1^p + a_2^p + \ldots + a_n^p} \right)^{\frac{1}{p}}$$

Notice that when the p value is greater than one, the computational expense can be high. This is because of the need for expensive exponentiation computations. More complex extended Boolean queries can be constructed by substituting a subquery for the basic terms in the forms above. Retrieval experiments [69] have shown these queries to yield effective results.

3.7 Optimization of Inverted Vector Searches

In this section, we present efficient algorithms to speed up finding best matches in information retrieval systems. In response to a user query, best match searching requires finding those documents that are most similar to the user query, and then ranking them according to the similarity measure used.

There are three major points in designing an efficient information retrieval system:

- importance weights assigned to documents and query terms,
- similarity measure used, and
- stopping rules applied to decrease the number of query document comparisons that must be computed in order to retrieve relevant documents.

These points are important when large document collections (i.e., over a million documents) are to be stored on slow magnetic or optical media.

Let D be the document collection consisting of n documents and T the system dictionary consisting of m index terms used for content identification.

Define an indexing function $I(D_i) \subseteq T$ that given the text of a document D_i, returns a subset of weighted index terms

$$I(D_i) = \{(t_{ij}, w_{ij}) | 1 \leq j \leq m; t_j \in T\}$$

where t_{ij} represents the assignment of term t_j to the document D_i and w_{ij} is a real number in the interval $[0, 1]$ which reflects the importance of the term t_j for identifying the document D_i.

The same indexing function can be applied to the text of a query to get a comparable query representative

$$I(Q) = \{(q_j, w_{qj}) | 1 \leq j \leq m; q_j \in T\}$$

An inverted list associated with the term t_j is the set of documents indexed by such a term

$$D_{t_j} = \{D_i | t_j \in I(D_i)\}$$

Given a query Q, the set P_Q of documents which *possibly* satisfy the query is given by

$$P(Q) = \bigcup_{q_j \in T} D_{q_j}$$

The set P_Q represents the union of the inverted lists associated with each of the query terms, (i.e., the set of documents which share at least one term in common with the query).

The set $R(Q, h)$ of the $r = |R(Q, h)|$ documents which *best* satisfy the query is given by

$$R(Q, h) = \{D_i | S(D_i, Q) \geq h; D_i \in P(Q)\}$$

where S is a similarity function which returns a real number such that a high value implies a high degree of resemblance and h is a threshold value. A ranked output can be obtained by arranging the retrieved items in decreasing order of query-document similarity as measured by the S-values.

3.7.1 Best Match Retrieval

A straightforward procedure to obtain best match documents is to match the query against each of the documents in the collection, compute similarities, sort the similarities into descending order and take the r highest rank documents. Obviously, it would require $O(n)$ computations, which is impractical for large collections. More practical approaches include:

Cluster Approach

In this approach, the collection is preprocessed and partitioned into clusters, each cluster containing similar documents. The first stage of the retrieval process is to find those similar clusters that are most significantly correlated with the given query and then the query is matched against each document contained in all identified clusters.

Inverted List Approach

If the inverted file is available, the number of documents that must be considered is reduced. Most of the commonly used similarity functions that have been used in IR systems involve the terms in common between the query and the document. Hence,

$$S(D_i, Q) \neq 0$$

iff the query and the document vectors have at least one common term. Thus, we have to process only the set $D_i \in P_Q$ of documents which appear at least once in the postings corresponding to the query terms, while all other documents can be discarded.

Several researchers have described ranking algorithms based upon an inverted file organization. Smeaton and VanRijsbergen [78] have described an algorithm which evaluates only a subset of likely relevant documents. However, their approach requires accessing a large portion of the document file. A first algorithm that reaches the goal of eliminating many of the accesses to the document file was proposed by Noreault [60]. The basic idea is to process the query lists, but when a document is encountered for the first time in the list, no attempt is made to match the corresponding document vector, thus avoiding the access to the document file. Instead a counter is allocated to such a document and set to one. When a document appears once again in a subsequent list, its counter is incremented by one. The end result is to have in each counter the number of matching terms between the document and the query. If the terms have attached weights, the counters will be incremented by the product of the query and document weights. Thus, these counters will contain the inner product between the document and the query terms. The advantage of this approach is that we need have no access to the document file if a suitable similarity function is used. In the inverted file entries, we have to store the references to the documents indexed by the term and the corresponding weights. The details of the algorithm is presented in Figure 3.1.

The presented algorithm computes the document similarity for each $D_i \in P(Q)$ – that is for each document which has a non-zero value for the similarity function. However, there are a considerable number of documents having a very small similarity with the query. This implies that many useless calculations of many low value similarities for the documents will not contribute to finding the closest set. Buckley and Lewit [10] propose an algorithm for

```
        for each QuestionTerm $q_j$ do
(1)         Read InvertedList; composed of pairs $(D_i, w_{t_{ij}})$
(2)         for each document $D_i$ in the list do
(3)             if NewDocument then AllocateCounter $C(D_i) = 0$;
(4)             $C(D_i) = C(D_i) + (w_{q_j} * w_{t_{ij}})$
(5)     Sort $C(D_i)$ in decreasing order;
(6)     Present the top $r$ documents;
```

Figure 3.1: The Basic Algorithm for Best Match Retrieval

finding the closest set computing partial similarities, taking into account only useful inverted lists. More on this approach is given in Section 3.7.3, after introducing the weighting scheme that must be considered.

3.7.2 Weighting Schemes

The weighting scheme used in the REVTOLC experiment is the augmented normalized term frequency, which is based on the occurrence frequency of each term in the document text. The weight $w_{t_{ij}}$ which reflects the presumed importance of term t_j for qualifying the content of document D_i is defined as

$$w_{t_{ij}} = (0.5 + 0.5F_{ij})/F_{max_i}$$

where F_{ij} represents the occurrence frequency of the term t_j in the document D_j normalized by F_{max_i}, the maximum occurrence frequency among the terms associated with D_i. The effect of such normalization is that longer documents do not produce higher term weights than shorter ones.

The same indexing and weighting process is performed on the text of the query; producing a query representation consisting of a set of pairs (q_j, w_{q_j}) with w_{q_j} denoting the degree of importance of the term q_j. In the REVTOLC experiment, the weight attached to each query term is determined also by its IDF (Inverse Document Frequency). For each term j, it is computed as $IDF_j = \frac{\log n}{n_j}$ where n is the number of documents in the collection and n_j is the number of documents in which the term j appears: $n_j = |D_{t_j}|$.

Using this approach, query terms are assigned weights inversely proportional to their frequency of occurrence in the collection. Thus, infrequently occurring terms are assigned larger weights than terms which occur in many documents.

During the matching process, the effect of this approach is to combine the local occurrence of a term within a document, which reflect its *importance*, with the total occurrence of the same term within the collection, which reflects its *discrimination power*.

A retrieval operation involves the identification of documents most similar to the submitted query. A range of matching functions is available in order to measure the query-document closeness. They are normally computed as a function of the number and the weights of attributes in common between the document and query terms. Thus, a similarity of a document D_i, with respect to the query Q; that consists of the query terms $(q_1, \ldots, q_n)$ is:

$$S(D_i, Q) = \sum_{j=1}^{k} w_{q_j} w_{t_{ij}}$$

3.7.3 The Algorithm

In this section, we present a ranking algorithm in which partial similarities are maintained *only* for documents which are presumed to reach the closest set. The search procedure attempts to stop as soon as the *minimum* number of inverted query term lists have been examined enough to guarantee the retrieval of the best documents. The algorithm is detailed in Figure 3.2. RelSet is the set of relevant documents and is maintained in decreasing order of partial similarity. BestDocOut is the document being replaced by a presumably more relevant document.

In (1), the query terms are sorted in order of decreasing weight. As the term weight is determined by its IDF value, the effect is to have those terms which apply to few documents at the top. Consequently, we process the shorter inverted lists first, leaving at the bottom the lists which include a larger number of documents. Then, we start to process the query lists in this order. Let us assume that we have processed l query lists out of k and we have a current closest set R including the documents $D_1, \ldots, D_r$ in decreasing order of similarity. D_r always represents the current last document in the relevant set R.

We have to process now the $(l+1)^{th}$ query list. For each document D_i referenced in the list, an upperbound for the similarity value can be computed assuming D_i should happen to match all of the remaining query terms. If the computed upperbound for D_i is less than the similarity value associated with D_r, it means the document D_i will never reach the relevance set so that it can be removed from further consideration.

Moreover, we terminate the algorithm if the *best* document *out* of the relevance set R is not expected to give a similarity better than the *last* document in the set R, that is D_r. Now, let us look at the evaluation of an upperbound $U(D_i)$ for the total query-document similarity, given a query Q consisting of $(q_1, \ldots, q_k)$ terms and given the document D_i.

After processing the first l query lists, let s_i be the current score for the document D_i. Then, the total similarity for D_i can be bounded according to the following relation:

$$S(Q, D_i) \leq s_i + \sum_{j=l+1}^{k} w_{q_j} * w_{t_{ij}}$$

The summation

$$\sum_{j=l+1}^{k} w_{q_j} * w_{t_{ij}}$$

corresponds to the maximum remaining similarity, assuming the worst case that all the remaining un-inspected terms (k-l) are in common between the document and the query.

Since the weighting scheme in effect computes the document term weight as the intra-document normalized frequency, the document term weight $w_{t_{ij}}$ is bounded by 1.0. It follows that

$$S(Q, D_i) \leq s_i + \sum_{j=l+1}^{k} w_{q_j} * w_{t_{ij}} \leq s_i + \sum_{j=l+1}^{k} w_{q_j}$$

```
(1)    sort QuestionTerms in decreasing order of weight
       **repeat for** each QuestionTerm $q_j$
       (* RelSet is maintained in decreasing order of partial similarity *)
           Read InvertedList; composed of pairs($D_i, w_{t_{ij}}$)
(2)        **for** each document $D_i$ in the list **do**
(3)        **if** $|RelSet| \leq r$ **then**  // RelSet is not full
               $C(D_i) = C(D_i) + w_{q_j} * w_{t_{ij}}$
               enter $D_i$ into the RelSet;
(4)        **else**
               **if** $U(D_i) > C(D_r)$ **and**
               $(C(D_i) = C(D_i) + (w_{q_j} * w_{t_{ij}}) > C(D_r)$ **then**
                   add $D_i$ to RelSet
(5)                Compute $U$(BestDocOut);
(6)    **until** LastQuestionTerm or $U$(BestDocOut) $\leq C(D_r)$
(7)    Present the RelSet sorted in decreasing order;
```

Figure 3.2: The Enhanced Algorithm for Best Match Retrieval

Hence, an upperbound for $S(Q, D_i)$ can be defined as

$$U(D_i) = s_i + \sum_{j=l+1}^{k} w_{q_j}$$

The result is that we can compute $U(D_i)$ taking into account only the weights of the remaining query terms, which are already known. The disadvantage is that $U(D_i)$ is not tight. Nevertheless, we used this algorithm in our experiment to achieve faster retrieval.

3.8 The Method

The hypothesis that was tested was: advanced retrieval techniques such as extended Boolean (p-norm), probabilistic feedback and vector retrieval are more effective and easier to use than some conventional techniques such as Boolean searching. The other variables considered were:

- question — Would results vary based on the question considered?

- time — Would subjects achieve *better* results if allowed more time?
- participant — Would results vary between the participants?
- order of searching — Since we decided that each participant should search for records in response to two queries for each of two retrieval methods (i.e., 4 queries per participant), would the order of the two methods influence the results? Or the order of the two queries inside a method?
- evaluation metric — Would results differ based on the evaluation metric (i.e., average precision, E measure)?

In order to test this hypothesis, and to determine the effects on other variables, a number of subjects were chosen from the Virginia Tech student body and asked to participate in a retrieval experiment. Their interaction and use of the system was carefully recorded and the information they provided was carefully collected for analysis.

3.9 The Experiment

The first task was to provide all the supporting technology that is required for such a large experiment. A huge effort was launched to find *better* techniques for handling the large number of keys that were commonly found such as names, terms, and keywords.

Since the number of keys in library catalogs was estimated to be of the order of hundreds of thousands from initial experimentation with pilot studies, better techniques for accessing dictionaries were developed. In order to improve efficiency of accessing large dictionaries and inverted lists, we can use minimal perfect hash functions to provide access to these static data structures in an optimal fashion. Minimal perfect hash functions locate a record in a single disk access, are efficient to evaluate, and require minimal storage overhead.

A number of steps were taken in order to enhance the technology for the experiment.

1. The SMART information retrieval system [28] that was developed at Cornell was selected as the basis for the test bed.

2. A disk-based method was developed to build inverted files. The method makes near optimal use of available primary memory and requires $O(n)$ time given a collection of compressed document vectors with total length n [29].

3. The SMART system was then ported to the Sequent Balance computer running Dynix, a version of the UNIX operating system. Since the Sequent is a multi-processor system, some of the indexing, inversion, and computations were studied with a view to parallelization.

4. The algorithms previously developed [30] for computing minimal perfect hash function were dependency graph based algorithms. Therefore, simpler and cheaper algorithm was developed. The expected time required by the algorithm is $\Omega(n^{1+\eta})$ to find an MPHF of size $n/(\eta \ln 2)$ bits, where η is the efficiency of representing an MPHF [41].

5. The search, retrieval, and ranking algorithms in SMART were considerably improved for these four methods.

6. A form-based interface was developed [77] for SMART that would run on VT100 character terminals.

7. Tutorial and logging software was developed (see Appendix B). This provided the subjects with a gentle introduction to searching. The logging software collected information both explicitly through online questionnaires and implicitly from the keystrokes that users entered during the search session.

The REVTOLC experiment involved 216 students as subjects, 35 librarians who provided relevance judgments, four retrieval methods being compared, eighteen questions collected from library users, and roughly half a million catalog records provided by the VPI&SU library in 1986. Several publications have reported on our fast methods for file inversion, and perfect hashing techniques (ordered and unordered) suitable for millions of keywords [34, 35].

3.10 Experimental Design

The REVTOLC experiment involved an extended version of the SMART [28] system. The experiment and retrieval methods were explained to the subjects through extensive online tutorials, help menus, and written explanations. A collection of REVTOLC queries, forms, and tutorials are included in B.1, B.3, and B.4.

We developed a powerful experimental design so that conclusive results can be derived. The following ingredients were necessary:

- access to data for a large number of catalog records — in our case about 500,000 records from the Newman Library VTLS system were included.
- a software system that could be used to search through that data — the SMART system (originally developed at Cornell) was modified for this purpose, with a great deal of interface and library record manipulation software.
- a computer system that could support a number of simultaneous users searching against the catalog database — a Sequent Symmetry in the Department of Computer Science was used, with its 10 processors, along with related network connections, terminals, and other networked computers to allow TELNET sessions.
- participants who would try out the system on the records — over 220 students at Virginia Tech were recruited.
- questions asked by students in the library — many were collected using a specially designed form (see Appendix B). From these, 18 were selected and searched by the participants and also later by expert librarians.

The questions selected are shown in Table 3.1.

We chose four advanced retrieval methods to compare:

- Boolean retrieval

No.	Question
1	Active phase cancellation, noise attenuation systems
2	Thermal desorption spectroscopy
3	Neural networks
4	Sayarner Truth, black abolishionist
5	Foot kicking procedures used with infants
6	A. Stindberg and his influence on the writings of O'Neill
7	Remote sensing
8	Ceramic matrix composites
9	Human relationships in Colonial America
10	Endangered species and fisheries ecology
11	Franchising in USA and international markets
12	Consumer studies and behavior
13	Primitive and organic farming
14	Electrolytic solution temperature dependence on Electromotive Force
15	Early weaning of pigs
16	Communication taxonomies and relational communication
17	A book by Ayn Rand
18	Exploratory behaviors with conjugate reinforcement using foot kicking

Table 3.1: Selected Questions

Group	Method 1	Method 2
1	Vector with Auto Feedback	Boolean
2	Vector with Auto Feedback	Vector
3	Vector with Auto Feedback	P-norm
4	Boolean	Vector with Auto Feedback
5	Boolean	Vector
6	Vector	Vector with Auto Feedback
7	Vector	Boolean
8	Vector	P-norm
9	P-norm	Vector with Auto Feedback
10	P-norm	Boolean
11	P-norm	Vector
12	Boolean	P-norm

Table 3.2: The REVTOLC Experiment Assignment of Methods to Groups

- Extended Boolean using the p-norm model
- Vector retrieval
- Vector retrieval with probabilistic feedback

Each of the 216 subjects worked with two methods. To eliminate the effects of the order of working with the two methods, we varied the order as well, and so had 4*3 = 12 test groups as shown in Table 3.2. Subjects were assigned randomly without duplication to these groups. Every group had 18 subjects.

Besides handling the order of searching with the two methods, we have attempted to ensure the generality of our results by roughly balancing the subjects with respect to gender, academic level (from freshman to senior to graduate student), and college (randomly selecting courses offered during the Spring and first Summer sessions, 1990). After each query is processed, users are encouraged to look at a detail screen for any retrieved document, and are then forced to reply regarding whether this is indeed relevant: "Y" for yes, "N" for no, or "D" for don't know.

Sessions typically lasted for 60-90 minutes. Each subject was paid $5 dollars for participating in the experiment. Figure 3.3 illustrates the steps of a typical REVTOLC session.

Figure 3.4 shows the average training time for each method.

In Figure 3.5, we see one possible Boolean search formulation for one of these questions. Note that this form-based scheme simplifies the task for novices, yet gives a great deal of flexibility. Terms on each line are ANDed, but lines for each "field" can be either ANDed or ORed, and the various fields also can be either ANDed or ORed (see explanation about PF2 and PF3 in upper right of the figure).

The results of this Boolean search are given in Figure 3.6, where four books were retrieved, judged relevant by this subject. Figures 3.7 and 3.8 show two portions of the detailed information for the fourth entry listed in Figure 3.6.

Figure 3.3: A Typical REVTOLC Session

Figure 3.4: Average Training for the Four Retrieval Methods

Figure 3.9 is for another question and illustrates form-based entry for vector queries. In this scheme, each term entered is assigned a weight, and pattern matching between the query and the documents leads to selection and ranking for the ten best matches. The results are shown in Figure 3.10, captured after relevance judgments were added by the author. Users may elect to have the computer system automatically construct a new query based on relevance feedback; the results are shown in Figure 3.11. Training time varied, depending on method. Boolean took longest, with descending times for p-norm, vector with auto feedback, and vector methods.

Figure 3.5: Screen with Boolean Question

Figure 3.6: Screen Listing Boolean Question Results and Relevance Judgments

Figure 3.7: Screen Showing First Page for Selection 4 from Boolean Question

Figure 3.8: Screen Showing Second Page for Selection 4 from Boolean Question

3.11 Detailed Results from Questionnaire Analysis

For each of the following questionaire questions, results were compared as a function of the four retrieval methods:

1. How many computer courses have you taken before college or in college?
2. How often do you use a computer?
3. Do you like to use a computer?
4. How many times have you used VTLS?
5. How many times have you used another retrieval system?
6. What is your typing skill?
7. What is your gender?
8. How would you describe your patience?
9. What is your overall grade point average?
10. What is your academic level?
11. What is the category that best describes your academic area?
12. How do you describe the question you are trying to find documents to satisfy?
13. How do find the time delay when the computer is finding documents?
14. Has this computer search helped you to find items you were looking for?
15. Among the citations provided by this search, what percentage seemed relevant to the question you entered?
16. How easy you would best describes your search?
17. In general, how do you judge using this method to search for library material?

Figure 3.9: Screen with Vector Question

Figure 3.10: Screen Listing Vector Question Results and Relevance Judgments

Figure 3.11: Screen Listing Vector Question Results (starting with number 11) and Feedback Results (entries 1-10)

18. Were the results of the searches what you expected?

19. To what degree do you think this method is an effective aid to casual searching for a few library references?

20. To what degree do you think this method is an effective aid to comprehensive searching for as complete a set as possible of relevant library references?

21. Was being able to manually edit your earlier query helpful?

22. How do describe the display of the details of documents retrieved?

23. Was the information and assistance given to you by the method easy to understand and adequate?

24. Is it easy to learn to use this method?

25. In general, Is this method easy to use?

26. Is searching with this method easier than searching with VTLS?

27. In general, did the method do a good job helping you find useful items?

28. Do you have anything else you want to say or comment about the method or this study?

29. Is the editor for entering your question easy to use?

30. Did the feedback facility help you find additional useful items?

31. The computer has tried to rank output so best results are first – Does this ranking of output help quickly identify relevant documents?

An analysis was done using GLM (Generalized Linear Model) with the SAS package. Using a comparison error rate of 0.01, no significant differences among the methods were found regarding questions 1-12, 28, 30, 31. Other questions showed significant differences

Questionaire	Means for 4 Retrieval Methods				Value Range
Question	1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)	and Description
13	2.84	3.00	2.44	2.53	1:long - 4:short
14	2.28	2.20	2.75	2.35	1:more - 4:nothing
15	2.63	2.83	2.42	2.76	1:0-20 - 5:80-100%
16	1.78	1.67	2.17	1.90	1:easy - 4:difficult
17	2.25	2.29	1.72	2.09	1:too much - 3:less
18	2.68	2.36	2.92	2.64	1:st.agree - 5:st.disag.
19	2.31	2.03	3.09	2.63	1:very eff. - 5:not eff.
20	2.27	2.25	3.02	2.53	1:very eff. - 5:not eff.
21	1.64	1.51	1.76	1.76	1:st.agree - 5:st.disag.
22	2.17	2.03	2.52	2.29	1:st.agree - 5:st.disag.
23	1.11	1.07	1.29	1.22	1:yes - 2:no
24	1.90	1.77	2.56	2.46	1:st.agree - 5:st.disag.
25	1.93	1.74	2.72	2.41	1:st.agree - 5:st.disag.
26	2.65	2.54	3.06	2.89	1:st.agree - 5:st.disag.
27	2.25	2.05	2.83	2.37	1:st.agree - 5:st.disag.
29	1.86	1.96	2.68	2.28	1:st.agree - 5:st.disag.

Table 3.3: Means for Questionaire Questions for the Four Retrieval Methods

in responses, depending on which method was used. For those questions, the means are shown in Table 3.3.

The tests indicated many significant differences. For grouping purposes we refer to Vector and Vector with Feedback as "vector-type" and refer to Boolean and P-norm as "Boolean-type". For question:

13 — Regarding computer time delay in response, there was a significant difference between vector-type methods (Vector, Vector with Feedback) which people found shorter than Boolean-type methods (Boolean, P-norm). See Table 3.4.

14 — Regarding the number of useful items found, there was a significant difference between (Vector, Vector with Feedback, P-norm) methods (which people found yielded more) and the Boolean method. See Table 3.5.

15 — Regarding the percentage of the documents found that were relevant, there was

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0 : LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	2.84	1	-	0.0141	0.0001	0.0001
2	3.00	2	0.0141	-	0.0001	0.0001
3	2.44	3	0.0001	0.0001	-	0.1612
4	2.53	4	0.0001	0.0001	0.1612	-

Table 3.4: Means for Questionaire Question 13 for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0 : LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	2.28	1	-	0.2958	0.0001	0.4444
2	2.20	2	0.2958	-	0.0001	0.0700
3	2.75	3	0.0001	0.0001	-	0.0001
4	2.35	4	0.4444	0.0700	0.0001	-

Table 3.5: Means for Questionaire Question 14 for the Four Retrieval Methods

a significant difference between P-norm (yielding a higher percentage) and Boolean. Similarly, Vector with Feedback yielded a higher percentage than Boolean, see Table 3.6.

16 — Regarding ease of search, there was a significant difference between (Vector, Vector with Feedback, P-norm) methods (which people found easier) and the Boolean method. Vector with Feedback was preferred to Boolean, see Table 3.7.

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0 : LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	2.63	1	-	0.0846	0.0530	0.2738
2	2.83	2	0.0846	-	0.0002	0.5390
3	2.42	3	0.0530	0.0002	-	0.0024
4	2.76	4	0.2738	0.5390	0.0024	-

Table 3.6: Means for Questionaire Question 15 for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0 : LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	1.78	1	-	0.1061	0.0001	0.0660
2	1.67	2	0.1061	-	0.0001	0.0005
3	2.17	3	0.0001	0.0001	-	0.0001
4	1.90	4	0.0660	0.0005	0.0001	-

Table 3.7: Means for Questionaire Question 16 for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0 : LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	2.25	1	-	0.3404	0.0001	0.0007
2	2.29	2	0.3404	-	0.0001	0.0001
3	1.72	3	0.0001	0.0001	-	0.0001
4	2.09	4	0.0007	0.0001	0.0001	-

Table 3.8: Means for Questionaire Question 17 for the Four Retrieval Methods

17 — Regarding time for a search, there was a significant difference between vector-type methods (which people found took less) and Boolean-type methods. See Table 3.8.

18 — Regarding if the results of the search were what was expected, there was a significant difference between (Vector, P-norm) methods (agree), Vector with Feedback, and Boolean. See Table 3.9.

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0 : LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	2.68	1	-	0.0001	0.0010	0.6114
2	2.36	2	0.0001	-	0.0001	0.0001
3	2.92	3	0.0010	0.0001	-	0.0002
4	2.64	4	0.6114	0.0001	0.0002	-

Table 3.9: Means for Questionaire Question 18 for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|\ H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	2.31	1	-	0.0024	0.0001	0.0004
2	2.03	2	0.0024	-	0.0001	0.0001
3	3.09	3	0.0001	0.0001	-	0.0001
4	2.63	4	0.0004	0.0001	0.0001	-

Table 3.10: Means for Questionaire Question 19 for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|\ H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	2.27	1	-	0.7818	0.0001	0.0031
2	2.25	2	0.7818	-	0.0001	0.0011
3	3.02	3	0.0001	0.0001	-	0.0001
4	2.53	4	0.0031	0.0011	0.0001	-

Table 3.11: Means for Questionaire Question 20 for the Four Retrieval Methods

19 — Regarding effectiveness of helping casual searching, there was a significant difference between (Vector, vector with Feedback, P-norm) methods (which people found effective) and Boolean — and also between Vector with Feedback (effective) and (Vector, P-norm), and Vector (effective) and Boolean. See Table 3.10.

20 — Regarding effectiveness of helping comprehensive searching, the means are in order (from most effective to least): Vector, Vector with Feedback, P-norm, Boolean with significant differences Vector vs. Boolean type, Vector with Feedback vs. Boolean-type, P-norm vs. Boolean. See Table 3.11.

21 — Regarding if it was helpful to manually edit earlier queries, there was a significant difference between methods Vector with Feedback (agree) and Boolean-type. See Table 3.12.

22 — Regarding if the display of details was clear and adequate, there was a significant difference between methods (Vector, Vector with Feedback, P-norm) and Boolean,

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	1.64	1	-	0.0246	0.0222	0.0226
2	1.51	2	0.0246	-	0.0001	0.0001
3	1.76	3	0.0222	0.0001	-	0.9607
4	1.76	4	0.0226	0.0001	0.9607	-

Table 3.12: Means for Questionaire Question 21 for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	2.17	1	-	0.0830	0.0001	0.1222
2	2.03	2	0.0830	-	0.0001	0.0010
3	2.52	3	0.0001	0.0001	-	0.0029
4	2.29	4	0.1222	0.0010	0.0029	-

Table 3.13: Means for Questionaire Question 22 for the Four Retrieval Methods

and between Vector with Feedback and P-norm. See Table 3.13.

23 — Regarding if the information was easy to understand and adequate, there was a significant difference between vector-type methods and Boolean-type methods. See Table 3.14.

24 — Regarding if the method was easy to learn, there was a significant difference between

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	1.11	1	-	0.1746	0.0001	0.0001
2	1.07	2	0.1746	-	0.0001	0.0001
3	1.29	3	0.0001	0.0001	-	0.0125
4	1.22	4	0.0001	0.0001	0.0125	-

Table 3.14: Means for Questionaire Question 23 for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|\ H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	1.90	1	-	0.0810	0.0001	0.0001
2	1.77	2	0.0810	-	0.0001	0.0001
3	2.56	3	0.0001	0.0001	.	0.1803
4	2.46	4	0.0001	0.0001	0.1803	-

Table 3.15: Means for Questionaire Question 24 for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|\ H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	1.93	1	-	0.0146	0.0001	0.0001
2	1.74	2	0.0146	-	0.0001	0.0001
3	2.72	3	0.0001	0.0001	-	0.0001
4	2.41	4	0.0001	0.0001	0.0001	-

Table 3.16: Means for Questionaire Question 25 for the Four Retrieval Methods

vector-type methods (agree) and Boolean-type methods. See Table 3.15.

25 — Regarding if the method was easy to use, there was a significant difference between vector-type methods (agree) and Boolean-type methods — also between methods P-norm (agree) and Boolean. See Table 3.16.

26 — Regarding if the method was easier than VTLS (the locally used library catalog system), there was a significant difference between vector-type methods (agree) and Boolean — also between methods Vector with Feedback (agree) and P-norm. See Table 3.17.

27 — Regarding if the method did a good job, there was a significant difference between vector-type methods (agree) and Boolean — also between methods P-norm (agree) and Boolean, and Vector with Feedback and P-norm. See Table 3.18.

29 — Regarding if the editor was easy to use, there was a significant difference between

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	2.65	1	-	0.2958	0.0001	0.0294
2	2.54	2	0.2958	-	0.0001	0.0012
3	3.06	3	0.0001	0.0001	-	0.1152
4	2.89	4	0.0294	0.0012	0.1152	-

Table 3.17: Means for Questionaire Question 26 for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	2.25	1	-	0.0094	0.0001	0.1224
2	2.05	2	0.0094	-	0.0001	0.0001
3	2.83	3	0.0001	0.0001	-	0.0001
4	2.37	4	0.1224	0.0001	0.0001	-

Table 3.18: Means for Questionaire Question 27 for the Four Retrieval Methods

vector-type methods (which people found faster) and Boolean-type methods, and P-norm vs. Boolean.

Clearly, method 3 (Boolean) is not well liked, and method 2 (vector with relevance feedback) is generally preferred.

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Vector)	2 (Fdbk)	3 (Boolean)	4 (P-Norm)
1	1.86	1	-	0.3565	0.0001	0.0001
2	1.96	2	0.3565	-	0.0001	0.0011
3	2.68	3	0.0001	0.0001	-	0.0001
4	2.28	4	0.0001	0.0011	0.0001	-

Table 3.19: Means for Questionaire Question 29 for the Four Retrieval Methods

3.12 Effectiveness Results

In this section, we present a complete evaluation of the relative effectiveness of the four retrieval methods tested in the REVTOLC experiment.

The standard formulation for recall and precision [72, 67] were used; if A is the set of relevant documents for a given query, and B is the set of retrieved documents, then

$$\text{Recall} = \frac{|A \cap B|}{|A|}$$
$$\text{Precision} = \frac{|A \cap B|}{|B|}$$

where "$|x|$" is the number of documents in set x.

We have computed the recall and precision of each search based on the previously identified relevant documents. Relevance judgments were done by librarians at Virginia Tech Library. For each query, all documents retrieved by any of the subjects were inspected by at least two librarians and judged on a scale of 4: 1 – not relevant, 2 – possibly relevant, 3 – likely relevant, 4 – definitely relevant. The judgments were averaged and fed into the SMART evaluation package at two cutoff points: 3 (likely relevant) and 4 (definitely relevant).

Figure 3.12 and Figure 3.13 show the overall means for the effectiveness of the four retrieval methods for different scales (likely relevant, definitely relevant) of relevance judgments. Since the results are similar, in further analysis the binary categorization "relevant" was defined by average judgment of at least 3. The graphs show that vector methods achieved higher precision than the Boolean method at different recall levels.

Table 3.20 lists average precision for each question and the four retrieval methods. Best performing methods are highlighted. We have noticed that methods performance is dependent on the question and have tried to study the relationship between different retrieval metrics and question length. We define question length as the number of words in the question that are not stop words. Stop words are high frequency words such as "the" and "by".

Figure 3.12: Average Precision vs. Recall for the Four Retrieval Methods- Likely Relevant

Figure 3.13: Average Precision vs. Recall for the Four Retrieval Methods- Definitely Relevant

Question	Vector with Auto Feedback	Vector	Boolean	P-norm
1	0.3838	**0.4931**	0.1664	0.3161
2	**0.7741**	0.3174	0.3119	0.4341
3	**0.6549**	0.5505	0.0077	0.1076
4	0.0717	0.2850	0.3517	**0.5459**
5	0.4045	0.2049	0.2427	**0.5770**
6	0.5757	0.5071	**0.6979**	0.3767
7	0.4351	0.4089	0.2212	**0.5539**
8	**0.7080**	0.4662	0.1772	0.6120
9	0.1804	**0.2591**	0.0304	0.1767
10	**0.5122**	0.1286	0.1957	0.3642
11	**0.8854**	0.5187	0.3641	0.7447
12	0.0612	**0.9788**	0.0970	0.4703
13	0.7819	0.7732	0.2555	**0.9593**
14	**0.6335**	0.2595	0.0411	0.5553
15	0.0681	0.0320	**0.0881**	0.0086
16	0.2430	**0.3989**	0.1004	0.0560
17	**0.9322**	0.8271	0.7748	0.4762
18	0.1819	0.1979	0.1892	**0.2252**

Table 3.20: Average Precision Values for the Four Retrieval Methods

In Tables 3.21, 3.22, 3.23, 3.24, and 3.25, we show the number of retrieved documents, and the number of relevant retrieved documents, then we compute the precision and recall for each method and for different question lengths. We notice that the Vector with Auto Feedback method retrieved more relevant items than other methods. Also, we notice that Boolean queries yield higher precision but lower recall, and retrieve fewer documents.

In Table 3.26, we show recall values for the four retrieval methods. The exact recall is

Method	Retrieved	Rel-ret	Precision	Recall
Vector with Auto Feedback	**4643**	**783**	0.1686	**0.6032**
Vector	3478	578	0.1662	0.4453
Boolean	1873	387	**0.2066**	0.2982
P-norm	3814	632	0.1657	0.4869

Table 3.21: Total Retrieved, Total Relevant-Retrieved, Precision, and Recall for the Four Retrieval Methods, out of 1298 Relevant

Method	Retrieved	Rel-ret	Precision	Recall
Vector with Auto Feedback	**27.69**	12.42	0.448	0.077
Vector	27.51	**12.66**	0.460	**0.079**
Boolean	9.91	6.34	**0.640**	0.039
P-norm	22.27	6.78	0.304	0.042

Table 3.22: Average Retrieved, Average Relevant-Retrieved, Precision, and Recall for the Four Retrieval Methods for Queries of Length 2, out of 161 Relevant

Method	Retrieved	Rel-ret	Precision	Recall
Vector with Auto Feedback	**32.83**	**13.14**	0.400	**0.031**
Vector	26.38	11.88	0.450	0.028
Boolean	10.84	8.27	**0.763**	0.019
P-norm	27.09	13.08	0.483	0.030

Table 3.23: Average Retrieved, Average Relevant-Retrieved, Precision, and Recall for the Four Retrieval Methods for Queries of Length 3, out of 430 Relevant

Method	Retrieved	Rel-ret	Precision	Recall
Vector with Auto Feedback	**43.06**	**9.19**	0.213	**0.018**
Vector	28.15	6.67	0.237	0.012
Boolean	16.81	4.34	**0.258**	0.013
P-norm	28.61	6.34	0.222	0.016

Table 3.24: Average Retrieved, Average Relevant-Retrieved, Precision, and Recall for the Four Retrieval Methods for Queries of Length 4, out of 505 Relevant

Method	Retrieved	Rel-ret	Precision	Recall
Vector with Auto Feedback	**64.35**	**9.92**	0.154	**0.049**
Vector	30.91	5.83	0.189	0.029
Boolean	21.15	6.35	**0.300**	0.031
P-norm	33.44	8.29	0.248	0.041

Table 3.25: Average Retrieved, Average Relevant-Retrieved, Precision, and Recall for the Four Retrieval Methods for Queries of Length 6, out of 202 Relevant

Method	Exact	at 5 docs	at 10 docs	at 15 docs	at 30 docs
Vector with Auto Feedback	**0.6281**	0.0028	0.0098	0.0153	0.0298
Vector	0.5145	**0.0081**	**0.0169**	**0.0275**	0.0569
Boolean	0.3631	0.0035	0.0126	0.0271	**0.0672**
P-norm	0.5320	0.0055	0.0110	0.0194	0.0346

Table 3.26: Recall Values for the Four Retrieval Methods

Method	Exact	at 5 docs	at 10 docs	at 15 docs	at 30 docs
Vector with Auto Feedback	0.3292	0.4000	0.4389	0.4185	0.4130
Vector	**0.3314**	**0.4889**	**0.4500**	**0.4778**	**0.4648**
Boolean	0.3199	0.1889	0.2500	0.2741	0.2685
P-norm	0.3244	0.3444	0.35	0.3963	0.4296

Table 3.27: Precision Values for the Four Retrieval Methods

the recall for exactly the retrieved document set, averaged over all queries (num_rel_docs with rank less than or equal to num_wanted / num_rel_docs). Also, we show recall values after 5, 10, 15, and 30 documents have been retrieved. Generally, we notice that the Vector method has the highest recall values.

In Table 3.27, we show precision values for the four retrieval methods. The exact precision is the precision for exactly the retrieved document set (i.e., after num_wanted documents were retrieved). Also, we show precision values after 5, 10, 15, and 30 documents have been retrieved. We notice that the Vector method has the highest precision.

In addition, van Rijsbergen's [84] E measure was calculated for each method. This measure is a weighted combination of recall and precision that ranges between 0 and 1, lower values of E indicated *better* performance.

$$E = 1 - \frac{1}{\alpha[\text{precision}^{-1}] + (1-\alpha)[\text{recall}^{-1}]}$$

where

$$\alpha = \frac{1}{\beta^2 + 1}$$

and recall and precision are calculated for a given number of documents at the top end of

$E_{\text{docs},\beta}$	Vector with Auto Feedback	Vector	Boolean	P-norm
$E_{5,0.5}$	0.9584	**0.9325**	0.9646	0.9591
$E_{5,1.0}$	0.9821	**0.9702**	0.9839	0.9822
$E_{5,2.0}$	0.9886	**0.9809**	0.9896	0.9886
$E_{10,0.5}$	0.9137	**0.8855**	0.9176	0.9237
$E_{10,1.0}$	0.9603	**0.9448**	0.9584	0.964
$E_{10,2.0}$	0.9742	**0.9636**	0.9721	0.9764
$E_{15,0.5}$	0.8862	**0.8348**	0.8756	0.8807
$E_{15,1.0}$	0.9447	**0.9144**	0.9299	0.9398
$E_{15,2.0}$	0.9634	**0.9421**	0.951	0.9596
$E_{30,0.5}$	0.8196	**0.7524**	0.8174	0.7977
$E_{30,1.0}$	0.8999	**0.8496**	0.8725	0.8836
$E_{30,2.0}$	0.9305	**0.8912**	0.9006	0.9177

Table 3.28: Average E Values for the Four Retrieval Methods

a ranked output list. The E measure evaluates the result of a search when the retrieved documents are ranked, but the particular order of documents within the cutoff limit does not affect the E value. This is in contrast to recall-precision figures which tend to be sensitive to the order of the documents. The advantage of the E measure is that it provides a single measure of effectiveness that can be used in significance tests. Lower E values are better.

Results were calculated for four cutoff levels, for 5, 10, 15 and 30 documents, and three β parameter values of the E measure were used to indicate relative effectiveness in terms of recall and precision. The E values reported in Table 3.28 are the averages for all 18 queries. We notice that the E measure favors the Vector method.

3.12.1 SAS Tests

Relevant Retrieved Test

The Relevant Retrieved test revealed a statistically significant difference between the four retrieval methods at the $p = 0.01$ level. We tested the hypothesis that there is no difference in the number of relevant documents retrieved by different methods. Table 3.29

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Fdbk)	2 (Vector)	3 (Boolean)	4 (P-Norm)
1	10.98	1	-	0.0076	0.0001	0.0010
2	8.94	2	0.0076	-	0.0001	0.5298
3	5.56	3	0.0001	0.0001	-	0.0002
4	8.46	4	0.0010	0.5298	0.0002	-

Table 3.29: Means for RelevantRetrieved for the Four Retrieval Methods

Question Length	(Fdbk)	Vector	Boolean	P-Norm
2	12.42	**12.66**	6.34	6.78
3	**13.14**	11.88	8.27	13.08
4	**9.19**	6.68	4.34	6.34
6	**9.92**	5.83	6.36	8.29

Table 3.30: LSMEAN Results of the Relevant Retrieved Metric for Different Query Lengths

shows the LSMEAN results of this metric for the four retrieval methods. There is a significant difference between the Boolean method and the others, the two vector methods, and Feedback vs. P-norm. We observe that the Vector with Auto Feedback method retrieved more relevant documents than other methods. The Boolean method retrieved the least number of relevant documents.

Also there is evidence that the Relevant Retrieved metric is dependent on the query length. The LSMEAN results are tabulated in Table 3.30. Query length 3 seems generally best.

NumRetrieved (Number of Documents Retrieved) Test

The NumRetrieved test revealed a statistically significant difference between the four retrieval methods at the $p = 0.01$ level. We tested the hypothesis that there is no difference in the number of retrieved documents by different methods. Table 3.31 shows the LSMEAN results of this metric for the four retrieval methods. There is a significant difference between the Vector with Auto Feedback method and the others, since it retrieved more documents

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0 : LSMEAN(i) = LSMEAN(j)$				
			1 (Fdbk)	2 (Vector)	3 (Boolean)	4 (P-Norm)
1	42.80	1	-	0.0009	0.0001	0.0005
2	26.67	2	0.0009	-	0.0288	0.8485
3	16.06	3	0.0001	0.0288	-	0.0467
4	25.75	4	0.0005	0.8485	0.0467	-

Table 3.31: Means for NumRetrieved for the Four Retrieval Methods

Question Length	(Fdbk)	Vector	Boolean	P-Norm
2	**27.47**	25.73	9.85	21.51
3	**32.64**	27.96	13.02	27.36
4	**42.74**	27.93	15.71	28.42
6	**63.95**	31.37	21.69	28.42

Table 3.32: LSMEAN Results of the NumRetrieved Metric for Different Query Lengths

than other methods. The Boolean method retrieved the least number of documents.

Also there is evidence that the Relevant Retrieved metric is dependent on the query length. The LSMEAN results are tabulated in Table 3.30. As expected, the Vector with Auto Feedback method retrieved more documents than other methods. The Boolean retrieved the least. In all cases, longer queries retrieved more documents.

Average Precision Test

Table 3.33 shows the LSMEAN results of the 11-point average precision test for the four retrieval methods. There is a significant difference between the Vector with Auto Feedback, the Vector and P-norm methods, and the Boolean method at the 0.01 level.

Exact Recall Test

Table 3.34 shows the LSMEAN results of the exact recall test for the four retrieval methods. The exact recall is the recall for exactly the retrieved document set, averaged over all queries (num_rel_docs with rank less than or equal to 10 / num_rel_docs). There

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Fdbk)	2 (Vector)	3 (Boolean)	4 (P-Norm)
1	0.145	1	-	0.8252	0.0035	0.4505
2	0.149	2	0.8252	-	0.0017	0.3287
3	0.094	3	0.0035	0.0017	-	0.0303
4	0.132	4	0.4505	0.3287	0.0303	-

Table 3.33: Means for Average Precision for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Fdbk)	2 (Vector)	3 (Boolean)	4 (P-Norm)
1	0.208	1	-	0.1860	0.0001	0.0794
2	0.181	2	0.1860	-	0.0032	0.6614
3	0.120	3	0.0001	0.0032	-	0.0124
4	0.172	4	0.0794	0.6614	0.0124	-

Table 3.34: Means for Exact Recall for the Four Retrieval Methods

is a significant difference between the Vector with Auto Feedback, the Vector method, and the Boolean method at the 0.01 level.

Exact Precision Test

Table 3.35 shows the LSMEAN results of the exact precision test for the four retrieval methods. The exact precision is the precision for exactly the retrieved document set (i.e., after num_wanted documents were retrieved). There is no significant difference between the four methods at the 0.01 level.

The $E_{\text{docs},\beta}$ Measure Tests

Tables 3.36, 3.37, 3.38, 3.39, 3.40, and 3.41 show the LSMEAN results of the the $E_{\text{docs},\beta}$ measure tests for the four retrieval methods. There are significant differences between various methods at the 0.01 level: between 1 and 3 (except in Table 3.39), between 1 and 2 (for Tables 3.36, 3.37, 3.38, and 3.40), between 2 and 3 (for Table 3.38), and between 3

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0 : LSMEAN(i) = LSMEAN(j)$				
			1 (Fdbk)	2 (Vector)	3 (Boolean)	4 (P-Norm)
1	0.372	1	-	0.5085	0.1335	0.7922
2	0.392	2	0.5085	-	0.3985	0.3553
3	0.417	3	0.1335	0.3985	-	0.0782
4	0.364	4	0.7922	0.3553	0.0782	-

Table 3.35: Means for Exact Precision for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0 : LSMEAN(i) = LSMEAN(j)$				
			1 (Fdbk)	2 (Vector)	3 (Boolean)	4 (P-Norm)
1	0.793	1	-	0.0019	0.0007	0.0209
2	0.715	2	0.0019	-	0.7667	0.4296
3	0.708	3	0.0007	0.7667	-	0.2790
4	0.735	4	0.0209	0.4296	0.2790	-

Table 3.36: Means for $E_{5,0.5}$ for the Four Retrieval Methods

and 4 (for Tables 3.37 and 3.38).

3.13 Timing Results

Table 3.42 shows the average overall time subjects spent on each query (averaging over subjects and methods). The overall time includes training time, search time, checking the documents for relevance, and in case of the vector with feedback method, it also include

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0 : LSMEAN(i) = LSMEAN(j)$				
			1 (Fdbk)	2 (Vector)	3 (Boolean)	4 (P-Norm)
1	0.864	1	-	0.0024	0.0001	0.0226
2	0.803	2	0.0024	-	0.0288	0.4526
3	0.760	3	0.0001	0.0288	-	0.0035
4	0.818	4	0.0226	0.4526	0.0035	-

Table 3.37: Means for $E_{5,1.0}$ for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Fdbk)	2 (Vector)	3 (Boolean)	4 (P-Norm)
1	0.893	1	-	0.0091	0.0001	0.0436
2	0.847	2	0.0091	-	0.0001	0.5575
3	0.778	3	0.0001	0.0001	-	0.0001
4	0.857	4	0.0436	0.5575	0.0001	-

Table 3.38: Means for $E_{5,2.0}$ for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Fdbk)	2 (Vector)	3 (Boolean)	4 (P-Norm)
1	0.720	1	-	0.0143	0.2159	0.0824
2	0.649	2	0.0143	-	0.2265	0.4786
3	0.684	3	0.2159	0.2265	-	0.6174
4	0.669	4	0.0824	0.4786	0.6174	-

Table 3.39: Means for $E_{10,0.5}$ for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Fdbk)	2 (Vector)	3 (Boolean)	4 (P-Norm)
1	0.776	1	-	0.0069	0.0037	0.0582
2	0.702	2	0.0069	-	0.8351	0.4202
3	0.696	3	0.0037	0.8351	-	0.3122
4	0.724	4	0.0582	0.4202	0.3122	-

Table 3.40: Means for $E_{10,1.0}$ for the Four Retrieval Methods

GLM Procedure Least Squares Means						
Method Number	LSMEAN	$Pr > \|T\|$ $H0: LSMEAN(i) = LSMEAN(j)$				
			1 (Fdbk)	2 (Vector)	3 (Boolean)	4 (P-Norm)
1	0.794	1	-	0.0108	0.0001	0.1181
2	0.726	2	0.0108	-	0.1695	0.3252
3	0.689	3	0.0001	0.1695	-	0.0189
4	0.752	4	0.1181	0.3252	0.0189	-

Table 3.41: Means for $E_{10,2.0}$ for the Four Retrieval Methods

Question	Length	Average Overall Time
1	6	2166
2	3	1994
3	2	2920
4	4	2033
5	4	1787
6	4	2491
7	2	2611
8	3	1150
9	4	1158
10	4	2840
11	4	2472
12	3	1902
13	3	2230
14	6	1904
15	3	2257
16	4	2013
17	3	2835
18	6	2655

Table 3.42: Average Overall Time (Editing, Searching, Results Inspecting) in Seconds for Each Question

feedback iterations. There is no correlation between average overall time and queries. Also, there is no correlation between average overall time and subjects. Table 3.43 shows the search time for each query for each method and the average search time.

3.14 Conclusions

The statistical analysis of the REVTOLC experiment logs revealed the following important results:

- The REVTOLC experiment proved that advanced retrieval techniques, namely extended Boolean (p-norm), probabilistic feedback and vector retrieval are more effective than Boolean searching.

Question	Vector	Vector with Fdbk	Boolean	P-norm	Average
1	4.3	4.8	3.7	17.7	7.6
2	4.6	3.7	6.5	7.2	5.5
3	2.3	4.2	10.3	7.0	6.0
4	3.2	3.5	4.7	12.6	9.2
5	3.5	9.6	45.8	5.2	10.0
6	2.6	5.3	2.7	4.5	3.8
7	3.7	4.0	4.7	10.2	5.7
8	3.4	4.5	28.1	7.3	10.8
9	4.7	3.7	36.0	12.9	14.3
10	3.6	4.2	16.4	13.9	9.5
11	3.8	4.3	11.0	49.5	17.2
12	5.8	4.0	58.7	4.0	18.1
13	3.3	3.5	18.6	10.4	9.0
14	4.2	5.7	34.5	17.2	15.4
15	3.4	4.0	36.0	8.1	12.9
16	3.2	5.3	35.6	18.0	15.5
17	3.7	3.7	3.0	5.2	3.9
18	4.2	5.0	59.7	6.1	18.7

Table 3.43: Search Time for Each Question

- The vector methods retrieved more documents than the p-norm and Boolean method (see Table 3.12.1).

- The vector methods found more relevant documents than the p-norm and Boolean method (see Table 3.12.1).

- Long queries retrieved fewer relevant documents than short queries. As expected, longer queries achieved higher recall (see Table 3.32).

The questionaire analysis of the REVTOLC experiment revealed the following results:

- Vector with Auto Feedback was found to be the easiest method to use. Boolean was found to be the most difficult. The p-norm was found to be less difficult than the Boolean method (see Table 3.16).

- Vector with Auto Feedback was found to be the easiest method to use. Boolean was found to be the most difficult. The p-norm was found to be less difficult than the Boolean method (see Table 3.7).

- Vector techniques in our system had faster response time to retrieve documents than the Boolean and p-norm methods. There was no significant difference between the Boolean and p-norm methods (see Table 3.8).

- Results of searches were as subjects expected in vector methods, more so than in the Boolean method. The number of retrieved documents for the Boolean method was reasonable (see Table 3.9).

- Vector with Auto Feedback was chosen as an effective aid for casual search. The Boolean method was most disliked for casual search (see Table 3.10).

- The REVTOLC system was easier to use than VTLS. The query editor used in the experiment was easy to use. The documentation was satisfactory and the tutorials were a sufficient introduction to the methods searched (see Table 3.17).

- Vector methods were judged to be more effective for comprehensive searching than p-norm and Boolean methods (see Table 3.11).

A wide range of performance was observed in the experiment. This was partly due to wide variations in students' attitude regarding the experiment. There were some students who became interested in the system's capabilities and stayed over two hours carrying out repeated searches beyond the assigned tasks. Some others tried to finish questions assigned to them as quickly as possible.

Most of the users were not experienced searchers, but got successful results within 5 minutes because there was no search logic to learn. This ease of use was combined with the natural ability of a ranking retrieval system to handle partial matches.

The scores for the final questionnaire, which evaluated the system, indicated that the system was well liked by all groups, and that the various features offered by the system were also well received. Overall, the use of advanced retrieval methods in REVTOLC on large online bibliographic collection can be regarded as a success.

Chapter 4

Minimal Perfect Hash Functions

4.1 Introduction

A classic problem in computer science is storing information so that it can be searched and retrieved efficiently. In this section, we review various schemes for generating minimal perfect hash functions and compare their performance. Next, we present algorithms that build compact hash functions suitable for external storage, that are optimal in terms of space on external storage, and achieve the goal of guaranteed single access. The expected time required by the algorithm is $\Omega(n^{1+\eta})$ to find an MPHF of size $n/(\eta \ln 2)$ bits, where η is the efficiency of representing an MPHF [41]. Keys are assumed to have some *finite maximum length.*

4.2 Definitions and Terminology

A hash table T consists of a finite number of addressable locations, indexed by $Z_m = \{0 \ldots m-1\}$. Assume we have n unique keys in a key set $S = \{k_1, \ldots, k_n\}$ selected from U, a finite universe of keys that are strings having some *finite maximum length.* If $h(k) = i$, where $k \in S$ and $i \in Z_m$, then we say that k is hashed to address i. If function h is a one-to-one onto mapping, h is called a *minimal perfect hash function* (MPHF).

The searching problem is to answer quickly membership queries of the form "Is $x \in S$?", and if yes, return the location of x in T. The location of x in T can be the record number in case of fixed length records, or the offset from a reference location (i.e., beginning of the file) in case of variable length records. If we construct h such that $h(k) = i_k, \forall k \in S$ and i_k is a predetermined address (i.e., offset, order, etc.) for k, then h is called an *order*

U	=	universe of keys
N	=	cardinality of U
k	=	key for data record
S	=	subset of U, set of keys in use
n	=	cardinality of S
T	=	hash table, with slots numbered $0, \ldots, (m-1)$
m	=	number of slots in T
h	=	function to map key k into hash table T
$\|h\|$	=	space to store hash function
h_0, h_1, h_2	=	three separate random functions easily computable over keys
g	=	function mapping $0, \ldots, (r-1)$ into $0, \ldots, (m-1)$
words/key	=	$\|h\|/n$ in computer words (i.e. 32 bit word length)

Table 4.1: Terminology Used Throughout MPHF Section

preserving minimal perfect hash function (OPMPHF). As shown in Table 4.1, we adopt the notation used in [30].

4.2.1 Evaluation Strategy

We define four important metrics in judging the merit of a perfect hashing algorithm:

1. **Generation Time Efficiency**: Given a static set of keys S, how much time does it take to find an MPHF for S?

2. **Space Efficiency**: How much space does the method require to specify an MPHF?

3. **Reliability**: Is the method guaranteed to produce MPHFs?

4. **Calculation Time Efficiency**: Does the method generate perfect hash functions that can be calculated efficiently?

4.2.2 MPHF Size : Lower Bound

Mehlhorn [56] shows that for most $S \subseteq U$, $|S| = n$, $|U| = N$, the shortest perfect hash function for S has size $\Omega(n + \log_2 \log_2 N)$ bits. First, he shows that any fixed function

$h : [0 \ldots N-1] \to [0 \ldots m-1]$ is perfect for at most $(N/m)^n \binom{m}{n}$ different subsets $S \subseteq [0 \ldots N-1], |S| = n$. Since there are $\binom{N}{n}$ different subsets of $[0 \ldots N-1]$ of size n, the size of the perfect class of hash functions H is

$$|H| \geq \binom{N}{n} / \left((N/m)^n \binom{m}{n} \right)$$

Clearly, $\log_2 |H|$ bits are necessary to represent an arbitrary PHF uniquely. Assuming N grows faster than n^2 [30], we can use the asymptotic estimate $\binom{N}{n} \sim N^n/n!$, and taking $m = n$ (minimal perfect hashing case), we have the asymptotic lower bound $n^n/n!$. Taking the base 2 logarithm, and applying Stirling's approximation for $n!$, we have an approximate asymptotic lower bound of

$$\begin{aligned} \log_2 e^n/\sqrt{2\pi n} &= n \log_2 e - 0.5 \log_2 2\pi n \\ &\approx 1.4427n \qquad\qquad \text{bits} \end{aligned}$$

Our algorithm in Section 4.5 achieves MPHF size of $\approx 2n$ bits.

4.3 Related Research

Twenty years ago, it was suggested that good minimal perfect hash functions are hard to find: Knuth [44] observed that only one in 10 million functions is a perfect hash function for mapping 31 of the most frequently used English words into 41 addresses. Since that time, many researchers have explored this problem. Their algorithms are discussed below, and compared in Table 4.2.

Sprugnoli [79] proposed algorithmic methods to produce perfect hash functions of the form:

$$\begin{aligned} \text{Quotient Reduction Method} &: \; H_q(k_i) = \lfloor (k_i + A)/B \rfloor \\ \text{Remainder Reduction Method} &: \; H_r(k_i) = \lfloor (A + k_i B) \rfloor \, mod \, C)/D \end{aligned}$$

where k_i is the i^{th} key in the set of N keys and A, B, C, D are constants.

Jaeschke [42] devised Reciprocal Hashing to generate perfect hash functions of the form:

$$H_j(k_i) = \lfloor (A/Bk_i + C) \rfloor \, mod \, D$$

where k_i is the i^{th} key in the set of N keys and A, B, C, D are constants.

Sprugnoli's and Jaeschke's methods perform exhaustive search and require exponential expected execution time. Thus, they only produce minimal perfect hash functions for limited sets of keys. Key sets larger than 15 keys must be partitioned into smaller segments for each of which a perfect hash function is computed. Although the existence of minimal perfect hash functions is guaranteed, the set of keys S must be of distinct positive integers.

Cichelli [17] devised an algorithm for constructing minimal perfect hash functions of the form:

hash value = hash key length + associated value of the key's first letter + associated value of the key's last letter.

Cichelli's algorithm is simple, but expensive. It performs exhaustive search with backtracking to find the letter value assignments for the construction of a minimal perfect hash function. Its time complexity is exponential and it is not practical for sets of more than 45 keys. Cichelli's method is also limited since two keys with the same first and last letters and the same length are not permitted. Thus, it is suitable for handling *some* static letter oriented key sets.

Brain and Tharp [7] preserved the simplicity of Cichelli's perfect hashing algorithm, but enhanced its speed to allow larger word sets to be used.

Chang [14, 13] proposed minimal perfect hash functions of the form:

$$h(k) = C \bmod p(k)$$

where C is a parameter and p is a function that assigns a distinct prime to each key. By the Chinese Remainder Theorem, C can be found algorithmically; C is likely to be a very large number that requires $n \log_2 n$ bits. Chang gives no general method for finding $p(k)$. Thus, the method is of theoretical interest only.

Chen, Chang, and Jan [15] presented a near minimal perfect hashing scheme for letter-oriented sets of key words. Their scheme uses Ziegler's row displacement compression technique for producing the parameters of hashing functions. They give no practical algorithm for finding a unique n-tuple for an arbitrary list of word sets.

Winters [87] proposed a universally applicable algorithm for generating minimal perfect hashing functions in polynomial time (worst case). An adjunct algorithm for reducing parameter magnitudes in the generated hash functions is given.

Sager [65, 64] proposed minimal perfect hash functions of the form:

$$h(k) = (h_0(k) + g(h_1(k)) + g(h_2(k)) \pmod n$$

where h_0, h_1, h_2 are three pseudo-random functions. The mincycle algorithm is used for finding MPHFs with an expected time complexity that is polynomial in n and has been used successfully only on sets of cardinality up to around 512.

Fox, Heath, and Chen [30] modify Sager's method and obtain an algorithm which requires a space of $O(n \log n)$ bits. They reported an expected running time of $O(n \log n)$. They seek minimal perfect hash functions of the form:

$$h(k) = (h_0(k) + g(h_1(k)) + g(h_2(k)) \pmod n$$

where h_0, h_1, h_2 are three pseudo-random functions.

Fredman, Komlós, and Szemerédi [38, 37] presented a data structure for storing a set of n integers from the universe of M integers, which uses space $O(n \log_2 n)$ bits and accommodates membership queries in constant time.

Cormack, Horspool, and Kaiserswerth [20] proposed a practical method that permits retrieval from a table in constant time that requires $O(n)$ space and can be constructed in $O(n)$ expected time. They extended the scheme of Fredman et al. [38] to handle insertions and deletions.

Aho and Lee [1] used MPHFs functions to store a dynamic sparse table that can process a sequence of insert, delete, and lookup instructions. Lookup and deletion is done in constant

Author	Method	Year	Time Efficiency	Space Efficiency	Guaranteed To Work?	Calculation Efficiency
Sprugnoli	[79]	1978	$O(2^n)$	$O(n \log n)$	no	$O(1)$
Jaeschke	[42]	1980	$O(2^n)$	$O(n \log n)$	yes	$O(1)$
Cichelli	[17]	1980	$O(2^n)$	$O(n \log n)$	no	$O(1)$
Brain	[7]	1989	$O(2^n)$	$O(n \log n)$	no	$O(1)$
Chang	[14]	1986	$O(n^2 \log n)$	$O(n \log n)$	no	$O(n \log n)$
Sager	[65]	1985	$O(n^4)$	$O(n \log n)$	yes	$O(1)$
Fredman	[38]	1984	$O(n)$	$O(n \log n)$	yes	$O(1)$
Cormack	[20]	1985	$O(n)$	$O(n \log n)$	yes	$O(1)$
Fox	[30]	1989	$O(n \log n)$	$O(n \log n)$	yes	$O(1)$
Daoud	[21]	1990	$O(n^{1+\eta})$	$O(\frac{n}{\eta \ln 2})$	yes	$O(1)$
Schmidt	[75]	1990	$O(n)$	$O(n + \log\log m)$	yes	$O(1)$

Table 4.2: Comparison of Different Perfect Hashing Algorithms

worst-case time. Insertion is done in constant expected time. The space used by their data structure is $O(n \log_2 n)$ where n is the number of elements.

Brain and Tharp [8] [7] proposed a simple algorithm for packing sparse 2-D arrays into minimal 1-D arrays in $O(n^2)$ time. Retrieving an element from the packed 1-D array requires $O(1)$ time. The algorithm is applied to create minimal perfect hashing functions for large word lists. The scheme was used to create minimal perfect hashing functions for word sets of up to 5000 words.

Pearson [61] proposed a hashing function tailored to variable-length text strings. He claims that using only a few simple instructions, this algorithm efficiently maps variable-length text strings onto small integers.

Schmidt and Siegel [75] presented tight bounds on the spatial complexity of perfect hash functions. They described a variation of an explicit $O(1)$ time single probe perfect hash function that can be specified in $O(n + \log\log m)$ bits.

In Table 4.2, we compare the different perfect hashing algorithms.

4.4 The Early Algorithm

In this section, we present an efficient algorithm for finding MPHFs for very large key sets. The class of functions searched is

$$h(k) = \begin{cases} (h_0(k)g(h_1(k)) + h_2(k)g^2(h_1(k)))\ mod\ n & \text{if mark}(h_1(k)) = 1 \\ g(h_1(k)) & \text{if mark}(h_1(k) = 0 \end{cases}$$

where

$$\begin{aligned} g: &\quad \{0,\ldots,r-1\} &\rightarrow&\quad \{0,\ldots,n-1\} \\ \text{mark}: &\quad \{0,\ldots,r-1\} &\rightarrow&\quad \{0,1\} \\ h_0: &\quad U &\rightarrow&\quad \{0,\ldots,n-1\} \\ h_1: &\quad U &\rightarrow&\quad \{0,\ldots,r-1\} \\ h_2: &\quad U &\rightarrow&\quad \{0,\ldots,n-1\} \end{aligned}$$

$r = \lceil cn/log_2n \rceil$, and c is a constant, typically < 4.

Here g is a function whose values are determined during the Searching step, and *mark* is a function whose values are determined during the Ordering step. Since both g and *mark* require $cn(1 + 1/log_2n)$ bits to represent an MPHF, our algorithm constructs an MPHF of size $O(\frac{n}{\eta \ln 2})$ bits.

4.4.1 The Mapping Step

The Mapping step takes a set of n keys and produces the three auxiliary hash functions h_0, h_1, and h_2. These three functions map each key k into a unique triple

$$(h_0(k), h_1(k), h_2(k)).$$

First, three tables ($table_0$, $table_1$, $table_2$) of random numbers are constructed, one for each of the functions h_0, h_1, and h_2. Each table contains one random number for each possible character at each position i in the key. Let a key be the character string $k = k_1k_2\ldots k_y$, $y = \text{length}(k)$. Then, the triple is computed in $O(y)$ time using the following formulas:

```
(1)    build random tables for $h_0$, $h_1$, and $h_2$
(2)    for each i $\in$ [0...r − 1] do
           A[i].firstkey = 0
           A[i].card = 0
(3)    for each i $\in$ [1...n] do
           keys[i].$h_0$ = $h_0(k_i)$
           keys[i].$h_1$ = $h_1(k_i)$
           keys[i].$h_2$ = $h_2(k_i)$
           keys[i].nextkey = 0
(4)    check that all keys have distinct ($h_0, h_1, h_2$) triples.
(5)    if triples not distinct then
           repeat from step (1).
```

Figure 4.1: The Mapping Step

$$
\begin{aligned}
h_0(k) &= \left(\sum_{i=1}^{y} table_{0i}(k_i)\right) mod\, n \\
h_1(k) &= \left(\sum_{i=1}^{y} table_{1i}(k_i)\right) mod\, r \\
h_2(k) &= \left(\sum_{i=1}^{y} table_{2i}(k_i)\right) mod\, n
\end{aligned}
$$

Figure 4.1 details the Mapping step. Let $k_1, k_2, \ldots, k_n$ be the set of keys. The h_0, h_1, and h_2 triples are computed from the three tables of random numbers. If triples are not distinct for all keys, new random tables are generated, defining new h_0, h_1, and h_2 functions. The probability that random tables must be generated more than once is very small [30]. Therefore, the expected time for the Mapping step is $O(n)$. If the length of individual keys is not constant, then the expected time for the Mapping step is proportional to the sum of all key lengths.

4.4.2 The Ordering Step

Ordering involves two passes through the triplets $(h_0(k), h_1(k), h_2(k))$. In the first pass, h_1 values are used to build a one dimensional array A of bins (maintained as linked lists) labeled $0, \ldots, r-1$. The set of keys stored in bin A_i, the i^{th} record of A, $0 \leq i \leq r-1$ is

$$K(A_i) = \{k | h_1(k) = i\}.$$

The A array is

```
A: array [0..r-1] of record
    firstkey: integer;
    card: integer;
    g: integer;
    mark: Boolean;
    end
```

firstkey is the header for a singly-linked list of keys in $K(A_i)$, i.e., those with $h_1(k) = i$. card is the cardinality of $K(A_i)$. g is the g value for A_i, which is assigned later in the Searching step. mark is set if $|K(A_i)| > 1$ and is reset if $|K(A_i)| \leq 1$. More formally

$$mark(i) = \begin{cases} 1 & \text{if } |K(A_i)| > 1 \\ 0 & \text{if } |K(A_i)| \leq 1 \end{cases}$$

The keys array is

```
keys: array [1..n] of record
    $h_0$, $h_1$, $h_2$: integer;
    nextkey: integer;
    end
```

where nextkey points to the next key in the linked list whose head is given by firstkey in the A array. In the second pass for Ordering, a number of stacks equal to the maximum cardinality, MaxCard, of the A_i are initialized. Each stack is assigned to store entries of a

```
        for each $i \in [1, \ldots, n]$ do
(1)         add keys[i] to $A[key[i].h_1]$ linked list
            increment A[key[i].$h_1$].card
(2)         **if** MaxCard < A[$key[i].h_1$]].card **then**
                MaxCard = A[key[i].$h_1$])].card
(3)     **for** each i $\in$ [0...r − 1] **do**
            **if** A[i].card $\leq$ 1 **then** A[i].mark = 0 **else** A[i].mark= 1
(4)     initialize a number of stacks equal to MaxCard
(5)     **for** each i $\in$ [0...r − 1] **do**
            push i in stacks[A[i].card]
```

Figure 4.2: The Ordering Step

certain size. For example, Stack[1] is assigned to store entries of the A_i array of cardinality equal to 1, and so on. The A_i array is scanned and all elements are pushed into their corresponding stacks Stack[A_i.card]. For very large key sets, these stacks are maintained on external storage.

Figure 4.2 details the Ordering step. In (1), keys according to their h_1 values are added to the appropriate bin; (2) updates MaxCard; (3) determines the `mark` bit according to $|K(A_i)|$; (4) initializes `stacks` used later in the Searching step; (5) loads these stacks with indices to A_i according to $|A_i|$. Clearly the worst-case time for the Ordering step is $O(n)$.

4.4.3 The Searching Step

The set of keys $k \in K(A_i)$ constitute a *pattern*. The Searching step determines an offset $g(i)$ that fits this pattern in empty slots of the hash table *simultaneously*. The Searching step pops entries of A_i from stacks and assigns hash values to all keys in $K(A_i)$, in descending order of A_i.card. Thus, stacks for larger patterns are processed before stacks with smaller patterns. Processing large patterns early, when the hash table is mostly empty, ensures that the fitting is done quickly. Since $h(k) = h_0(k)g(i) + h_2(k)g^2(i), i = h_1(k)$, assigning hash values to all keys in $K(A_i)$ is done by assigning a value to $g(i)$. This $h(k)$ is a variation of quadratic hashing and has the advantage of eliminating both elementary and secondary clustering in the hash table. When all patterns of size greater than 1 are processed, assigning the rest of the patterns of size 1 in A is done by setting g to the addresses of the remaining empty slots in the hash table and using the hash function $h(k) = g(i)$. The $mark(h_1(k))$ function is used to distinguish which hashing function has been used to hash k.

Figure 4.3 gives the algorithm for the Searching step. At the beginning of the Searching step [30], a set of 20 small primes is chosen (or fewer if n is quite small) that do not divide n. Each time (3) is executed, one of the primes q is chosen at random to be s_1 and is used as an increment to obtain the remaining $s_j, j \geq 2$. Thus, the random probe sequence is

$$0, q, 2q, 3q, \ldots, (n-1)q$$

In (2), we start fitting patterns according to their cardinality in descending order. Entries are popped from the stack of the current pattern size being processed. In (4), locations in the hash table T are calculated and checked if assigned previously (5). If the probes are successful (6), we assign the keys to the hash table locations. If any of these probes were assigned, we pick another s_j value and repeat the process. If we fail to fit a pattern within n trials, the Searching step flags an error. In case of an error in the Searching step, the hash table is reinitialized and the Searching step is repeated with a different random probe sequence. In (8), we assign patterns of size 1.

4.4.4 Analysis

It is important to show that the Mapping step succeeds in assigning a unique triple to each key. Assume the triples $(h_0(k), h_1(k), h_2(k))$ are random. Let $t = nr^2$ be the size of the universe of triples. The probability that n triples chosen uniformly at random from t triples are distinct is [26]

$$p(n,t) = \frac{t(t-1)\cdots(t-n+1)}{t^n} = \frac{(t)_n}{t^n}$$

We use the asymptotic estimate of $\frac{(t)_n}{t^n}$

```
(1) for i ∈ [0...n − 1] do
        hash_table[i].assigned = false
(2) for i = MaxCard down to 2 do
        while stacks[i] not empty do
(3)                 establish a random probe sequence $s_0, s_1, \ldots, s_{n-1}$ for [0...n − 1]
                    j = 0
                    collision = false
                    pop $h_1$ from stacks[i] and assign it to v
                    for k = 1 to i do
                    pop an entry from stacks[i] and assign it to Keys[k]
(4)                     $h(k) = keys[k].h_0 * s_j + keys[k].h_2 * {s_j}^2 \pmod{n}$
(5)                     if hash_table[h(k)].assigned then
                            collision = true
(6)                 if not collision then
                        A[v].g = $s_j$
                        for each k ∈ K($A_i$) do
                            hash_table[h(k)].assigned = true
                            hash_table[h(k)].key = k
                        else
                        j = j + 1
(7)                     if j > n − 1 then
                            fail
            while collision
        endwhile
(8) while stacks[1] is not empty do
            pop $h_1$ from stacks[1] and assign it to v
            pop an entry from stacks[1] and assign it to Keys[k]
            locate the first unassigned hash_table entry. Let $x$ be its index
            A[v].g = $x$
            hash_table[x].assigned = true
            hash_table[x].key = k
    endwhile
```

Figure 4.3: The Searching Step

n	$p(n,t)$
2	0.8825
4	0.8825
8	0.9105
16	0.9394
32	0.9617
64	0.9768
128	0.8964
256	0.9922
512	0.9956
1024	0.9976
2048	0.9986
131072	0.9999
262144	0.9999
524288	0.9999
1048576	0.9999
1200502	0.9999
3875766	0.9999

Table 4.3: The probability of Generating Unique Triplets: $p(n,t)$ versus n, $c = 2$

$$\begin{aligned}\frac{(t)_n}{t^n} &\sim \exp\left\{-\frac{n^2}{2t} - \frac{n^3}{6t^2}\right\} \\ &\sim \exp\left(-\frac{\log_2 n}{2cn}\right)\end{aligned}$$

When $c > 1, n \to \infty$

$$p(n,t) \sim 1$$

As evident from Table 4.3, the probability of generating unique triplets for large key sets is ~ 1.

The Searching step involves fitting all the bins of A in descending order. Processing a bin involves fitting a pattern of the same size as the size of the bin and requires a number of trials. For each bin, the number of trials is bounded by n in the worst case. Each trial consists of a random probe by $h(k)$ for all keys in the bin. For a given bin, a trial is successful if all keys are assigned distinct locations in the hash table T simultaneously. A minimal perfect hash function is generated when all bins are processed successfully. Thus, the cost of the Searching step is the sum of the cost of all trials. Let Y_j be the random variable that equals the number of empty cells in T before fitting the j^{th} pattern in T. Z_j is a random variable that equals the number of trials required to fit the j^{th} pattern successfully in T. The cost of each individual trial C_j is proportional to the size of the pattern involved. For the j^{th} pattern being processed, let us denote the pattern size as s_j. Thus $C_j = cs_j$ where c is a constant. The cost of processing a bin of size s is $\sum_{r=1}^{s} cjZ_j$.

The probability that a particular key is assigned to the i^{th} entry of A_i is $p = 1/r$. Let X be the random variable that equals $|K(A_i)|$. Then the X distribution is binomial. As $n \to \infty$, the Poisson approximation applies:

$$\Pr(X = s) \approx \frac{e^{-\lambda}\lambda^s}{s!}$$

where

$$\lambda = n/r = \frac{n}{cn/\log_2 n} = (\log_2 n)/c$$

The expected number of bins of size s is

$$E(X = s) \quad \approx \quad \left(\frac{cn}{\log_2 n}\right)\left(e^{\frac{-\log_2 n}{c}}\right)\left(\frac{\log_2 n}{c}\right)^s / s!$$

The expected number of bins of size 1 is $\approx ne^{-log_2 n/c}$

We now consider the probability of fitting a pattern of size j into a hash table T of size n. Assume that f slots of T are occupied and consider fitting a pattern of size j. According to [30] the probability that the pattern fits is

$$\Pr(\text{fit}) = 1 - e^{-\mu}$$

where

$$\mu = \left(\left(1 - \frac{f}{n}\right)^{j-1} (n - f)\right)$$

Since we fit patterns of size 1 at the end, and we fit all patterns of size $j > 1$ in descending order

$$\begin{aligned} n - f &> ne^{-log_2 n/c} \\ 1 - f/n &> e^{-log_2 n/c} \\ \mu &> ne^{\frac{-j \log_2 n}{c}} \end{aligned}$$

Note that choosing a suitable value of c and as $n \to \infty$ then $\mu \to \infty$. Also, the probability of fitting patterns of size $j > 1$ goes to 1. Thus, the search can be approximated as the task of placing $n - ne^{-log_2 n/c}$ words in a hash table of size n using $h_0(k)$ as the primary hash function and quadratic probing to resolve collisions. Assuming no secondary clustering, the expected number of probes to place a single word in the table containing i words is [44]:

$$C_i \approx (1 - \alpha)^{-1}$$

where $\alpha = i/m$ is the load factor of the hash table, and i varies between 0 and $n - ne^{-log_2 n/c}$. The total number of probes to place all patterns of size greater than one can be approximated by:

$$\sum_i C_i \leq (n - ne^{-log_2 n/c})[1 - \alpha_{max}]^{-1}$$

As is evident from experimental results, choosing $c \geq 2$ produces an MPHF quickly.

4.4.5 Experimental Results

We have carried out extensive experiments with large key sets taken from the Online Computer Library Center (OCLC). The time for those runs is reported in Table 4.4. Timing results for some very large runs using external storage are reported in Table 4.5.

n	Alg. [30] Totals (words/key =0.6)	Alg. [21] Totals (words/key =0.6)	bits/key Alg. [21]				
			bits/key	Map	Order	Search	Total
32	0.10	2.18	2.28	2.15	0.02	0.02	2.19
1024	1.33	2.95	3.19	2.58	0.05	1.13	3.76
131072	189.28	98.93	3.24	58.18	7.43	14569.12	14634.73
262144	374.09	194.43	3.61	117.63	15.82	4606.75	4740.20
524288	808.34	383.57	3.60	228.10	34.43	14129.87	14831.73

Note: for all runs,
Machine = Sequent;
Time (CPU) is in seconds.

Table 4.4: Timing Results for Both Algorithms Using Internal Memory

n	bits/key	Mapping	Ordering	Searching	Total	Machine
524288	3.59	447.47	597.25	11476.13	12520.85	Sequent
1200502	3.60	1060.62	1432.48	48118.00	50611.10	Sequent
3875766	4.58	2114.15	2955.60	28243.25	33313.00	NeXT 68030

Note: for all runs except for n=3,875,766,
Machine = Sequent;
for n=3,875,766,
Machine=NeXT
Time (CPU) is in seconds.

Table 4.5: Timing Results for the Early Algorithm Using External Storage, for Runs Near Lowest Bits/Key Achieved

Figure 4.4 illustrates the linear time complexity, giving total time (generally dominated by Searching) for various set sizes. Figure 4.5 illustrates that few bits/key are required, regardless of set size, but that time to find a MPHF increases rapidly as the theoretical lower bound on space is approached.

4.4.6 Example

In this section, we show an example of finding a MPHF for the 20 key set listed in Table 4.6 along with their unique triples $(h_0(k), h_1(k), h_2(k))$. Table 4.7 shows the corresponding **A** array and *mark* bits for $A_i, 0 < i < r-1$. The bits/key $c = 1.9$. The class of functions searched is

$$h(k) = \begin{cases} \left(h_0(k)g(h_1(k)) + h_2(k)g^2(h_1(k))\right)(mod\ P)\,(mod\ n) & \text{if mark}(h_1(k)) = 1 \\ g(h_1(k)) & \text{if mark}(h_1(k) = 0 \end{cases}$$

Table 4.8 shows the g assignment. Table 4.9 shows final hash addresses for all keys in the example key set. The function is indeed minimal and perfect.

4.5 A More Space Efficient Algorithm

This algorithm for finding MPHFs for very large key sets is a modification of the earlier algorithm. This Enhanced Algorithm is designed to use minimal RAM when finding minimal perfect hash functions. It doesn't use any expensive random tables or mark bits. The algorithm takes an additional input parameter, B, that determines the number of loads that will be used in processing the key set. If the number of loads is larger than one, then the enhanced algorithm uses external storage for transient data structures. The algorithm makes several passes over these loads to accumulate the required statistics such as maximum cardinality and sizes of patterns.

The class of functions searched is:

$$h(k) = (h_0(k) + h_1(k) + 7 * (h_2(k) - 1) * g(h_1(k)))\,(mod\ n)\,(mod\ P)$$

Figure 4.4: Total Time vs. Key Set Size for Basic Algorithm

Figure 4.5: Search Time vs. Bits/Key for Basic Algorithm, using Secondary Storage

Key	$h0$	$h1$	$h2$
x-rays	2	3	13
Euclidean	2	1	0
ethyl ether	4	0	8
Clouet	2	1	7
Bulwer-Lytton	14	0	13
dentifrice	11	3	1
Lagomorpha	12	2	10
Chungking	5	4	1
quibbles	16	4	17
Han Cities	17	4	15
treacherous	12	0	1
calc-	4	6	2
deposited	11	1	6
rotundus	14	6	17
antennae	9	4	6
sodium lamp	0	2	4
oculomotor nerve	5	5	14
tussle	8	1	14
imprecise	0	6	2
meridiem	15	2	1

Table 4.6: Example Key Set and Their (h_0, h_1, h_2) Triples, $n = 20$, $r = 7$, bits/key = 1.9

i	K(A_i)	$\|K(A_i)\|$	mark(i)
0	ethyl ether,Bulwer-Lytton,treacherous	3	1
1	Euclidean,Clouet,deposited,tussle	4	1
2	Lagomorpha,sodium lamp,meridiem	3	1
3	x-rays,dentifrice	2	1
4	Chungking,quibbles,Han Cities,antennae	4	1
5	oculomotor nerve	1	0
6	calc-,rotundus,imprecise	3	1

Table 4.7: Example Key Set Mapped to A Array, n=20, *ratio*=0.35, r=7, c=1.9, P=23

i	0	1	2	3	4	5	6
g	13	11	17	14	4	7	2

Table 4.8: g Values Assignment of Example Key Set

Key k	$h0$	$h1$	$h2$	$h(k)$
x-rays	2	3	13	$(2 * 14 + 13 * (14)^2) \pmod{23} \pmod{20} = 0$
Euclidean	2	1	0	$(2 * 11 + 0 * (11)^2) \pmod{23} \pmod{20} = 2$
ethyl ether	4	0	8	$(4 * 13 + 8 * (13)^2) \pmod{23} \pmod{20} = 1$
Clouet	2	1	7	$(2 * 11 + 7 * (11)^2) \pmod{23} \pmod{20} = 18$
Bulwer-Lytton	14	0	13	$(14 * 13 + 13 * (13)^2) \pmod{23} \pmod{20} = 10$
dentifrice	11	3	1	$(11 * 14 + 1 * (14)^2) \pmod{23} \pmod{20} = 5$
Lagomorpha	12	2	10	$(12 * 17 + 10 * (17)^2) \pmod{23} \pmod{20} = 12$
Chungking	5	4	1	$(5 * 4 + 1 * (4)^2) \pmod{23} \pmod{20} = 13$
quibbles	16	4	17	$(16 * 4 + 17 * (4)^2) \pmod{23} \pmod{20} = 14$
Han Cities	17	4	15	$(17 * 4 + 15 * (4)^2) \pmod{23} \pmod{20} = 9$
treacherous	12	0	1	$(12 * 13 + 1 * (13)^2) \pmod{23} \pmod{20} = 3$
calc-	4	6	2	$(4 * 2 + 2 * (2)^2) \pmod{23} \pmod{20} = 16$
deposited	11	1	6	$(11 * 11 + 6 * (11)^2) \pmod{23} \pmod{20} = 19$
rotundus	14	6	17	$(14 * 2 + 17 * (2)^2) \pmod{23} \pmod{20} = 4$
antennae	9	4	6	$(9 * 4 + 6 * (4)^2) \pmod{23} \pmod{20} = 17$
sodium lamp	0	2	4	$(0 * 17 + 4 * (17)^2) \pmod{23} \pmod{20} = 6$
oculomotor nerve	5	5	14	$g[5] = 7$
tussle	8	1	14	$(8 * 11 + 14 * (11)^2) \pmod{23} \pmod{20} = 11$
imprecise	0	6	2	$(0 * 2 + 2 * (2)^2) \pmod{23} \pmod{20} = 8$
meridiem	15	2	1	$(15 * 17 + 1 * (17)^2) \pmod{23} \pmod{20} = 15$

Table 4.9: Example Key Set and the Final Hash Addresses

where

$$\begin{aligned} g: &\quad \{0,\ldots,r-1\} &\rightarrow&\quad \{0,\ldots,n-1\} \\ h_0: &\quad U &\rightarrow&\quad \{0,\ldots,n-1\} \\ h_1: &\quad U &\rightarrow&\quad \{0,\ldots,r-1\} \\ h_2: &\quad U &\rightarrow&\quad \{0,\ldots,n-1\} \end{aligned}$$

$r = \lceil cn/log_2 n \rceil$, P is a prime, and c is typically < 2.

Here g is a function whose values are determined during the Searching step. The expected time required by the algorithm is $\Omega(n^{1+\eta})$ to find an MPHF of size $n/(\eta \ln 2)$ bits, where η is the efficiency of representing an MPHF [41].

Key to address transformation functions used in this algorithm for $h_0(k)$, $h_1(k)$, and $h_2(k)$ are efficient and can be combined in a driver function that calls a separate module that contains all parameters specific to a given key set. This algorithm has been used in our dynamic perfect hashing scheme presented in Chapter 6, because it handles small key sets efficiently. Also, as an example, we show the MPHFs generated for the C and C++ reserved words in Appendix A.

4.5.1 The Mapping Step

The Mapping step takes a set of n keys and produces the three auxiliary hash functions h_0, h_1, and h_2.

The three functions map each key k into a unique triple $(h_0(k), h_1(k), h_2(k))$. These functions are detailed in Figure 4.6, Figure 4.7, and Figure 4.8. These functions are easy to compute and have $O(1)$ time complexity.

The algorithm uses the h_1 values to divide the key set into loads and stores them separately in different files on external storage. The number of loads B is chosen when running the algorithm and is usually less than 20. We map keys to loads according to

$$h_1(k) * B/r$$

```
#define PRIME1 37
#define PRIME2 1048583

long h_0(key, len)
          register char *key;
          register long len;
{
          register long h;

          h = 0;
          /* Convert string to long */
          while (len--)
                    h = h * PRIME1 ^ (*key++ - ' ');
          h %= PRIME2;
          return (h);
}
```

Figure 4.6: $h_0(k)$ C code

Thus, the set of keys stored in load L_i, $0 \leq i \leq B-1$ is

$$K(L_i) = \{k | h_1(k) * B/r = i\}$$

The mapping step is detailed in Figure 4.9.

Clearly, the expected time for the Mapping step is $O(n)$.

4.5.2 The Ordering Step

Ordering involves two passes through all loads, one load at a time. In the first pass, h_1 values are used to build a one dimensional array A of bins (maintained as linked lists) labeled $0, \ldots, r-1$. The set of keys stored in bin A_i, the i^{th} record of A, $0 \leq i \leq r-1$ is

$$K(A_i) = \{k | h_1(k) = i\}$$

After processing all loads, the cardinality of each $K(A_i)$ is accumulated, and the maximum cardinality is determined.

```
/* This code assumes a 32-bit machine */
#define dcharhash(h, c) ((h) = 0x63c63cd9*(h) + 0x9c39c33d + (c))

long h_2(key, len)
        register char *key;
        long len;
{
        register char *e, c;
        register long h;

        e = key + len;
        for (h = 0; key != e;) {
                c = *key++;
                if (!c && key > e)
                        break;
                dcharhash(h, c);
        }
        return (h);
}
```

Figure 4.7: $h_1(k)$ C code

```
long h_2(key, len)
        register char *key;
        register long len;
{
        register long h, loop;

#define HASH4a h = (h << 5) - h + *key++;
#define HASH4b h = (h << 5) + h + *key++;
#define HASH4 HASH4b

        h = 0;
        if (len > 0) {
                loop = (len + 8 - 1) >> 3;

                switch (len & (8 - 1)) {
                case 0:
                        do { /* All fall throughs */
                                HASH4;
                case 7:
                                HASH4;
                case 6:
                                HASH4;
                case 5:
                                HASH4;
                case 4:
                                HASH4;
                case 3:
                                HASH4;
                case 2:
                                HASH4;
                case 1:
                                HASH4;
                        } while (--loop);
                }

        }
        return (h);
}
```

Figure 4.8: $h_2(k)$ C code

```
(1)     for each i $\in$ [0...r - 1] do
            A[i].firstkey = 0
            A[i].card = 0
(2)     for each i $\in$ [1...n] do
            keys[i].$h_0$ = $h_0(k_i)$
            keys[i].$h_1$ = $h_1(k_i)$
            keys[i].$h_2$ = $h_2(k_i)$
            keys[i].nextkey = 0
(3)     check that all keys have distinct ($h_0$, $h_1$, $h_2$) triples.
(4)     if triples not distinct then
                choose different prime P
            repeat from step (1).
```

Figure 4.9: The Enhanced Algorithm Mapping Step

In the second pass, a number of files equal to the maximum cardinality, MaxCard, are initialized. Each file is used to store entries of bins of certain size. For example, File[1] is assigned to store entries of the A_i array of cardinality equal to 1, and so on. All loads are scanned and all elements are stored into their corresponding File[A_i.card]. Clearly the expected time for the Ordering step is $O(n)$.

4.5.3 The Searching Step

As in the earlier algorithm, the set of keys $k \in K(A_i)$ constitute a *pattern*. The Searching step determines an offset $g(i)$ that fits this pattern in empty slots of the hash table *simultaneously*. The Searching step reads entries of A_i from files and assigns hash values to all keys in $K(A_i)$, in descending order of A_i.card. Thus, files for larger patterns are processed before files with smaller patterns.

bits/key	ratio	Mapping	Ordering	Searching	Total
		(seconds)	(seconds)	(seconds)	(seconds)
4.0	1.0	0.33	0.00	0.30	0.63
3.0	0.75	0.30	0.02	0.00	0.32
2.9	0.725	0.32	0.00	0.00	0.32
2.8	0.7	0.32	0.00	0.00	0.32
2.7	0.675	0.35	0.00	0.00	0.35
2.6	0.65	0.32	0.00	0.02	0.33
2.5	0.625	0.30	0.00	0.02	0.32
2.4	0.6	0.32	0.00	0.00	0.32
2.3	0.575	0.32	0.00	0.00	0.32
2.2	0.55	0.32	0.00	0.28	0.60
2.1	0.525	0.33	0.00	0.32	0.65
2.0	0.5	0.32	0.00	0.30	0.62
1.9	0.475	0.32	0.00	0.32	0.63
1.8	0.45	0.32	0.00	0.30	0.62
1.7	0.425	0.35	0.00	0.00	0.35
1.6	0.4	0.33	0.00	0.00	0.33
1.5	0.375	0.32	0.00	0.30	0.32

Table 4.10: MPHF Timing Results for Enhanced Algorithm, Key Set Size n=16

4.5.4 Experimental Results

Our experimental results show that our algorithm is capable of producing minimal perfect hash functions that are close to theoretical bound on space and runs efficiently using external storage for large key sets. Timing results for generating minimal perfect hash functions for various key set sizes are reported in Tables 4.10, 4.11, 4.12, 4.13, 4.18, 4.19, 4.20, and 4.21. All runs were carried out on a DECstation 5000/25. Figures 4.10, and 4.11 show our total MPHF generation time versus key set size for various key sets.

bits/key	ratio	Mapping	Ordering	Searching	Total
		(seconds)	(seconds)	(seconds)	(seconds)
5.0	1.0	0.32	0.00	0.02	0.33
4.0	0.8	0.30	0.02	0.32	0.63
3.0	0.6	0.33	0.00	0.32	0.65
2.9	0.58	0.32	0.00	0.32	0.63
2.8	0.56	0.32	0.00	0.00	0.32
2.7	0.52	0.30	0.00	0.00	0.30
2.6	0.52	0.32	0.00	0.03	0.35
2.5	0.50	0.32	0.00	0.03	0.35
2.4	0.48	0.32	0.00	0.32	0.63
2.3	0.46	0.32	0.00	0.32	0.63
2.2	0.44	0.32	0.00	0.32	0.63
2.1	0.42	0.32	0.00	0.02	0.33
2.0	0.4	0.30	0.00	0.33	0.63
1.9	0.38	0.32	0.00	0.30	0.62
1.8	0.36	0.33	0.00	0.30	0.63
1.7	0.34	0.32	0.00	0.32	0.63
1.6	0.32	0.32	0.00	0.32	0.63
1.5	0.30	0.30	0.02	0.32	0.63

Table 4.11: MPHF Timing Results for Enhanced Algorithm, Key Set Size n=32

4.5.5 Example

In this section, we show an example of finding a MPHF using the Enhanced Algorithm for the 20 key set listed in Table 4.14 along with their unique triples $h_0(k), h_1(k), h_2(k)$. Since the size of the key set is small, the number of loads $B = 1$.

bits/key	ratio	Mapping	Ordering	Searching	Total
		(seconds)	(seconds)	(seconds)	(seconds)
6.0	1.0	0.33	0.00	0.00	0.33
5.0	0.83	0.32	0.00	0.32	0.63
4.0	0.67	0.33	0.00	0.00	0.33
3.0	0.5	0.32	0.00	0.10	0.42
2.9	0.48	0.32	0.00	0.00	0.32
2.8	0.467	0.33	0.00	0.03	0.37
2.7	0.45	0.32	0.00	0.10	0.42
2.6	0.433	0.33	0.00	0.47	0.80
2.5	0.417	0.32	0.00	0.23	0.55
2.4	0.4	0.32	0.00	0.12	0.43
2.3	0.383	0.32	0.00	0.05	0.37
2.2	0.367	0.32	0.00	0.32	0.63
2.1	0.35	0.32	0.00	0.32	0.63
2.0	0.33	0.32	0.00	0.32	0.64
1.9	0.31	0.35	0.00	0.60	0.95
1.8	0.30	0.30	0.00	0.60	0.90
1.7	0.283	0.32	0.00	0.48	0.80
1.6	0.267	0.32	0.00	0.32	0.64
1.5	0.25	0.32	0.00	0.38	0.70

Table 4.12: MPHF Timing Results for Enhanced Algorithm, Key Set Size n=64

bits/key	ratio	Mapping	Ordering	Searching	Total
		(seconds)	(seconds)	(seconds)	(seconds)
7.0	1.0	0.35	0.00	0.02	0.37
6.0	0.857	0.37	0.00	0.02	0.38
5.0	0.714	0.32	0.00	0.02	0.33
4.0	0.57	0.32	0.00	0.02	0.33
3.0	0.428	0.32	0.00	0.38	0.70
2.9	0.414	0.32	0.00	0.02	0.33
2.8	0.4	0.33	0.00	0.35	0.68
2.7	0.385	0.32	0.00	0.67	0.98
2.6	0.371	0.32	0.00	0.32	0.63
2.5	0.357	0.33	0.00	0.28	0.62
2.4	0.34	0.32	0.00	0.32	0.64
2.3	0.328	0.32	0.00	1.12	1.43
2.2	0.314	0.32	0.00	0.32	0.64
2.1	0.3	0.30	0.00	0.33	0.65
2.0	0.286	0.32	0.00	0.35	0.67
1.8	0.257	0.33	0.00	0.47	0.80
1.6	0.228	0.32	0.00	0.80	1.12

Table 4.13: MPHF Timing Results for Enhanced Algorithm, Key Set Size n=128

Key	$h0$	$h1$	$h2$
x-rays	18	4	8
Euclidean	3	2	2
ethyl ether	1	5	10
Clouet	2	3	8
Bulwer-Lytton	3	6	8
dentifrice	5	3	19
Lagomorpha	4	2	14
Chungking	3	5	10
quibbles	7	3	11
Han Cities	2	3	4
treacherous	8	6	13
calc-	3	4	12
deposited	18	5	13
rotundus	4	6	8
antennae	14	1	2
sodium lamp	2	6	7
oculomotor nerve	1	5	1
tussle	10	6	4
imprecise	6	5	9
meridiem	8	3	8

Table 4.14: Example Key Set and Their (h_0, h_1, h_2) Triples, $n = 20$, $r = 7$, $c = 1.6$

i	K(A_i)	$\|K(A_i)\|$
0		0
1	antennae	4
2	Euclidean,Lagomorpha	2
3	meridiem,Han Cities,quibbles,Clouet,dentifrice	5
4	calc-,x-rays	4
5	Chungking,deposited,ethyl ether,oculomotor nerve,imprecise	5
6	tussle,sodium lamp,Bulwer-Lytton,treacherous,rotundus	5

Table 4.15: Example Key Set Mapped to A Array, n=20, r=7, c=1.6

i	0	1	2	3	4	5	6
g	0	18	14	1	7	6	2

Table 4.16: g Values Assignment of Example Key Set

Table 4.15 shows the corresponding **A** array for $A_i, 0 < i < r-1$. The bits/key $c = 1.6$. Table 4.16 shows the g assignment. Table 4.17 shows final hash addresses for all keys in the example key set. The function is indeed minimal and perfect.

Key k	$h0$	$h1$	$h2$	$h(k)$
x-rays	18	4	8	16
Euclidean	3	2	2	3
ethyl ether	1	5	10	6
Clouet	2	3	8	14
Bulwer-Lytton	3	6	8	4
dentifrice	5	3	19	2
Lagomorpha	4	2	14	7
Chungking	3	5	10	8
quibbles	7	3	11	17
Han Cities	2	3	4	9
treacherous	8	6	13	10
calc-	3	4	12	13
deposited	18	5	13	11
rotundus	4	6	8	5
antennae	14	1	2	18
sodium lamp	2	6	7	12
oculomotor nerve	1	5	1	19
tussle	8	1	14	1
imprecise	0	6	2	15
meridiem	8	3	8	0

Table 4.17: Example Key Set and the Final Hash Addresses

bits/key	ratio	Mapping	Ordering	Searching	Total
		(seconds)	(seconds)	(seconds)	(seconds)
10.0	1.0	0.35	0.02	0.07	0.43
9.0	0.9	0.37	0.02	0.07	0.45
8.0	0.8	0.37	0.00	0.08	0.45
7.0	0.7	0.35	0.00	6.13	6.48
6.0	0.60	0.35	0.00	10.53	10.88
5.0	0.5	0.35	0.02	4.28	4.65
4.0	0.4	0.33	0.02	25.57	25.92
3.0	0.3	0.37	0.02	42.67	43.05
2.9	0.29	0.37	0.00	0.63	1.00
2.8	0.28	0.35	0.02	2.60	2.97
2.7	0.27	0.35	0.02	0.7	1.07
2.6	0.26	0.35	0.02	50.08	50.45
2.5	0.25	0.37	0.02	25.10	25.48
2.4	0.24	0.37	0.00	1.30	1.67
2.3	0.23	0.35	0.02	12.03	12.40
2.2	0.22	0.37	0.02	6.72	7.10
2.1	0.21	0.35	0.00	20.17	20.52
2.0	0.2	0.38	0.02	41.78	42.1
1.9	0.19	0.37	0.00	13.50	13.87
1.8	0.18	0.35	0.02	235.40	235.77
1.7	0.17	0.37	0.02	109.07	109.45

Table 4.18: MPHF Timing Results for Enhanced Algorithm, Key Set Size n=1024

4.6 Conclusions

In this chapter, we have presented simple and efficient algorithms for finding efficient minimal perfect hash functions for very large key sets. These generated functions are optimal in time (perfect) and hash table space (minimal).

bits/key	ratio	Mapping	Ordering	Searching	Total
		(seconds)	(seconds)	(seconds)	(seconds)
11.0	1.0	0.42	0.02	1.32	1.75
10.0	0.9	0.40	0.02	22.08	22.50
9.0	0.82	0.38	0.03	62.30	62.72
8.0	0.727	0.40	0.02	259.45	259.87
7.0	0.636	0.40	0.02	47.18	47.60
6.0	0.545	0.38	0.03	18.95	19.37
5.0	0.455	0.38	0.03	30.68	31.10
4.0	0.364	0.38	0.03	20.53	20.95
3.0	0.273	0.40	0.02	203.23	203.65
2.9	0.264	0.40	0.03	32.08	32.52
2.8	0.255	0.40	0.02	20.20	20.62
2.7	0.245	0.40	0.02	19.22	19.63
2.6	0.236	0.40	0.02	36.05	36.47
2.5	0.227	0.40	0.02	88.37	88.78
2.4	0.218	0.40	0.02	41.62	42.03
2.3	0.209	0.40	0.02	146.13	146.55
2.2	0.207	0.42	0.02	1930.77	1931.20
2.1	0.191	0.38	0.03	1612.25	1612.67
2.0	0.182	0.40	0.02	30.68	31.10
1.9	0.173	0.38	0.03	15.12	15.53
1.8	0.164	0.40	0.03	31.85	32.28
1.7	0.155	0.40	0.02	113.57	113.98

Table 4.19: MPHF Timing Results for Enhanced Algorithm, Key Set Size n=2048

bits/key	ratio	Mapping	Ordering	Searching	Total
		(seconds)	(seconds)	(seconds)	(seconds)
12.0	0.82	1.35	0.35	3.80	5.50
11.0	0.754	1.30	0.35	2.85	4.50
9.0	0.617	1.28	0.33	2.57	4.18
8.0	0.549	1.30	0.33	2.55	4.18
7.0	0.480	1.28	0.33	3.02	4.63
6.0	0.412	1.28	0.33	3.18	4.80
5.0	0.343	1.28	0.33	3.82	5.43
4.0	0.274	1.30	0.33	6.73	8.37
3.0	0.206	1.30	0.35	28.20	29.85
2.9	0.199	1.28	0.37	39.60	41.25
2.8	0.192	1.30	0.37	55.95	57.62
2.7	0.185	1.30	0.35	62.93	64.58
2.6	0.178	1.32	0.35	90.38	100.05
2.5	0.171	1.33	0.37	146.33	148.03
2.4	0.164	1.30	0.37	248.32	249.98
2.3	0.158	1.30	0.38	320.43	322.12
2.2	0.151	1.32	0.37	734.65	736.33
2.1	0.144	1.30	0.40	1057.77	1059.47
2.0	0.137	1.32	0.42	2096.38	2098.12
1.9	0.130	1.30	0.43	3840.88	3842.62
1.8	0.123	1.32	0.42	11698.43	11700.17

Table 4.20: MPHF Timing Results for Enhanced Algorithm, UNIX Dictionary, n=24474

bits/key	ratio	Mapping	Ordering	Searching	Total
		(seconds)	(seconds)	(seconds)	(seconds)
4.0	0.183	686.50	798.15	798.15	3344.52
3.9	0.178	687.77	792.87	2375.42	3856.05
3.8	0.174	685.37	792.45	3106.42	4584.23
3.7	0.169	686.30	790.32	4111.65	5588.27
3.6	0.164	684.50	790.47	5700.20	7175.17
3.5	0.160	685.05	790.48	8195.17	9670.70
3.4	0.155	686.28	789.50	12321.67	13797.45

Table 4.21: MPHF Timing Results for Enhanced Algorithm, OCLC Key Set, n=3875766

Figure 4.10: Mapping, Ordering, and Searching Time for the UNIX Dictionary, n= 24474

Figure 4.11: Mapping, Ordering, and Searching Time for the OCLC key set, n= 3875766

Chapter 5
Order Preserving Minimal Perfect Hashing

This chapter proposes an algorithm for finding order preserving minimal perfect hash functions. Our method allows external storage of static sets in any order desired and requires $O(n \log n)$ bits for the function specification.

5.1 Introduction

We map the problem of finding an OPMPHF into a problem on a random bipartite graph, where each given key k is represented as an edge, and where randomness allows us to make use of important features of these graphs. As shown in Table 5.1, we adapt and extend the notation used in [30, 34].

Since we are building minimal perfect hash functions, we take $m = n$. In the bipartite dependency graph G, there are two parts involving a total of $2r$ vertices connected by n edges. One end of each edge associated with key k is at the vertex labeled $h_1(k)$, and the other end is at the vertex labeled $h_2(k)$. The functions $h_{direct}(k)$ and $h_{indirect}(k)$ are used to easily compute the location of the key k, given a specification of g.

This method is based on the idea that acyclic components of G can be used to record order information for all edges (keys) in these components. Moreover, they provide a free extra location to store additional information. In the algorithm given in [30], these free vertices are initialized randomly. Consider a random bipartite graph with $words/key \geq 1$, then we observe [30] that more than 13% of the vertices in G are isolated (zero-degree). If these isolated vertices were used to record order information for keys in cyclic components of G, then it is possible to generate a OPMPHF. However, we need an extra indirection bit

U	universe of keys
N	cardinality of U
k	key for data record
S	subset of U, set of keys in use
n	cardinality of S
T	hash table, with slots numbered $0, \ldots, (m-1)$
m	number of slots in T
h	function to map key k into hash table T
$\|h\|$	space to store hash function
G	dependency graph
r	parameter specifying the number of vertices in one part of G
ratio	$2r/m$, which specifies the relative size of G
h_0, h_1, h_2	three separate random functions easily computable over keys
g	function mapping $0, \ldots, (2r-1)$ into $0, \ldots, (m-1)$
$h_{\text{direct}}(k)$	$(h_0(k) + g(h_1(k)) + g(h_2(k))) \pmod n$ direct hashing function
$h_{\text{indirect}}(k)$	$g((h_0(k) + g(h_1(k)) + g(h_2(k))) \pmod{2r})$ indirect hashing function
v	vertex in G
$K(v)$	for a given v in the vertex ordering, the set of keys in that ordering level
VS	vertex sequence produced during the Ordering step
t	length of VS

Table 5.1: Terminology Used Throughout OPMPHF Section

mark to distinguish between direct and indirect cases.

5.2 A Lower Bound on the Size of OPMPHFs

We define a (N, m, n) order-preserving perfect class H of OPPHF functions as a set of functions h

$$h : [0 \dots N-1] \rightarrow [0 \dots m-1]$$

such that for any permutation P of any subset $S \subseteq N$ of size $|S| = n$, there is an $h \in H$ such that h is OPPHF for that permutation P. If $m = n$, then H is an OPMPHF class. Now if h is order preserving for a given permutation P of S, then any other permutation $P' \neq P$ of S cannot be order preserving and hashed by h [34]. As in the case of MPHF, any fixed function $h : [0 \dots N-1] \rightarrow [0 \dots m-1]$ is perfect for at most $(N/m)^n \binom{m}{n}$ permutations of different subsets $S \subseteq [0 \dots N-1], |S| = n$. There are $\binom{N}{n}$ different subsets of $[0 \dots N-1]$ of size n, and for each subset there are $n!$ permutations, hence the size of the perfect class of order preserving hash functions H is

$$|H| \geq \binom{N}{n} n! / \left((N/m)^n \binom{m}{n} \right)$$

Assuming N grows faster than n^2 [30], we can use the asymptotic estimate $\binom{N}{n} \sim N^n/n!$. Taking the base 2 logarithm, we have an asymptotic lower bound of $n \log_2 n$ bits to represent an arbitrary OPMPHF uniquely.

5.2.1 Three Methods to Find OPMPHFs

There are at least three methods to obtain an OPMPHF [34]:

Method 1: Acyclic Graphs

Method 1, the acyclic technique, involves constructing a bipartite graph G sufficiently large so that no cycles are present. If there are no cycles, we have sufficient freedom during the Searching step to select g values that will preserve any a priori key order.

Figure 5.1: Each Acyclic Component Provides a Free Entry Location to Store Order

However, the size of the generated OPMPH is much larger than OPMPHF sizes generated in the next two methods described below.

Method 2: Two Level Hashing

The second method is to use two level hashing. Here an MPHF is used in the first level and an array of pointers is used in the second. A hash value computed using the MPHF addresses the second level where the actual addresses of records are stored. The records are arranged in the desired order. This method requires $O(n)$ bits to store the MPHF specification, and for the second level $n \log n$ bits is required to store addresses of data records. Note, however, that it is much easier and faster to find small OPMPHFs using Method 3, which is discussed next.

Method 3: Using Indirection

The third method is based on the idea of using G to store the additional information required to specify a MPHF that also preserves order. For n keys, if our graph has somewhat more than n vertices (i.e., if ratio > 1, where ratio is defined as $2r/m$), then there should be enough space to specify the OPMPHF. In a random graph of this size, a significant number of vertices will have zero degree. We use indirection for handling some of the keys. This means that some keys will be mapped using indirection, in this case using the composition

$$order(k) = h(k) = g\Big(\{h_0(k) + g(h_1(k)) + g(h_2(k))\}\, mod\, 2r\Big)$$

For the remaining keys, the desired location of a key is the value of:

$$order(k) = h(k) = \{h_0(k) + g(h_1(k)) + g(h_2(k))\}\, mod\, n.$$

Note that we use the g function in two ways, one for direct keys and the other for keys that are handled through indirection.

5.3 The Algorithm

In this section, we present our algorithm for finding OMPHFs. The class of functions searched is:

$$\mathrm{h_{final}(k)} = \begin{cases} g\big(h_0(k) + g(h_1(k)) + g(h_2(k)) \pmod{2r}\big) & \text{if } (mark(h_1(k)) \text{ OR } mark(h_2(k))) \\ h_0(k) + g(h_1(k)) + g(h_2(k)) \pmod{n} & \text{otherwise} \end{cases}$$

The algorithm for selecting proper g values and setting mark (indirection) bits for vertices in G consists of the three steps: Mapping, Ordering, and Searching. Each step, along with implementation details, will be described in a separate subsection below.

5.3.1 The Mapping Step

This step is essentially identical to that discussed in [30]. The only addition is that the mark bit must be included in the vertex data structure.

The Mapping step takes a set of n keys and produces the three auxiliary hash functions h_0, h_1, and h_2. These three functions map each key k into a unique triple

$$(h_0(k), h_1(k), h_2(k))$$

The h_0, h_1, and h_2 values are used to build a bipartite graph called the *dependency graph.* Half of the vertices of the dependency graph correspond to the h_1 values and are labeled $0, \ldots, r-1$. The other half of the vertices correspond to the h_2 values and are labeled $r, \ldots, 2r-1$. There is one edge in the dependency graph for each key in the original set of keys. A key k corresponds to an edge labeled k between the vertex labeled $h_1(k)$ and the vertex labeled $h_2(k)$. Notice that there may be other edges between $h_1(k)$ and $h_2(k)$, but those edges are labeled with keys other than k. There are two data structures that constitute the dependency graph, one for the edges (keys) and one for the vertices (determined by the h_1 and h_2 values). Both are implemented as arrays. The `vertex` array is

```
vertex: array [0..2r-1] of record
    firstedge: integer;
    degree: integer;
    g: integer;
    end
```

firstedge is the header for a singly-linked list of the edges incident to the vertex. degree is the number of vertices incident on the vertex. g is the g value for the vertex, which is assigned in the Searching step. The edge array is

```
edge: array [1..n] of record
    $h_0, h_1, h_2$: integer;
    $nextedge_1$: integer;
    $nextedge_2$: integer;
    end
```

h_0, h_1, and h_2 contain the h_0, h_1, and h_2 values for the edge (key). Also, $nextedge_i$, for side i $(= 1, 2)$ of the graph (corresponding to h_1, h_2, respectively), points to the next edge in the linked list whose head is given by firstedge in the vertex array.

Figure 5.2 details the Mapping step. Let $k_1, k_2, \ldots, k_n$ be the set of keys. The h_0, h_1, and h_2 functions are selected (1) as the result of building tables of random numbers. The construction of the dependency graph in (2) and (3) is straightforward. (4) examines sets of edges having the same h_1 value to check for distinct (h_0, h_1, h_2) triples; since vertex degrees are small, (4) takes expected time that is linear in n. If triples are not distinct, new random tables are generated (5), defining new h_0, h_1, and h_2 functions. The probability that random tables must be generated more than once is very small [30]. Therefore, the expected time for the Mapping step is $O(n)$.

```
(1)    build random tables for $h_0$, $h_1$, and $h_2$
(2)    for each $v \in [0 \ldots 2r-1]$ do
           vertex[v].firstedge $= 0$
           vertex[v].degree $= 0$
(3)    for each $i \in [1 \ldots n]$ do
           edge[i].$h_0$ = $h_0(k_i)$
           edge[i].$h_1$ = $h_1(k_i)$
           edge[i].$h_2$ = $h_2(k_i)$
           edge[i].$nextedge_1 = 0$
           add edge[i] to linked list with header vertex[$h_1(k_i)$].firstedge
           increment vertex[$h_1(k_i)$].degree
           edge[i].$nextedge_2 = 0$
           add edge[i] to linked list with header vertex[$h_2(k_i)$].firstedge
           increment vertex[$h_2(k_i)$].degree
(4)    for each $v \in [0 \ldots r-1]$ do
           check that all edges in linked list vertex[v].firstedge have
               distinct $(h_0, h_1, h_2)$ triples.
(5)    if triples not distinct then
           repeat from step (1).
```

Figure 5.2: The Mapping Step

5.3.2 The Ordering Step

The Ordering step explores the dependency graph so as to partition the set of keys into a sequence of levels. The step actually produces an ordering of the vertices having degree at least one. From the vertex ordering, the sequence of levels is easily derived. If the vertex ordering is $v_1, \ldots, v_t$, then the level of keys $K(v_i)$ corresponding to a vertex v_i, $1 \leq i \leq t$, is the set of edges incident both to v_i and to a vertex earlier in the ordering. More formally

$$K(v_i) = \begin{cases} \{k_j | h_1(k_j) = v_i, h_2(k_j) = v_s, v_s < v_i\} & \text{if } 0 \leq v_i \leq r-1 \\ \{k_j | h_2(k_j) = v_i, h_1(k_j) = v_s, v_s < v_i\} & \text{if } r \leq v_i \leq 2r-1 \end{cases}$$

Our algorithm builds on two observations. First, the vertex ordering of every connected component in the graph G must start with a vertex v such that $|K(v)| = 0$. Second, edges that can be directed are always in levels with $|K(v_i)| = 1$, while edges that must be indirected occur in levels with $|K(v_i)| > 1$. Acyclic components when ordered yield a vertex ordering with all levels having a cardinality of one. Cyclic components when ordered have *some* levels with cardinality greater than one. Hence using both observations, all components in G are identified as cyclic or acyclic and assigned unique component ID's. These ID's are used later in the Searching step to identify these components. The current implementation doesn't assign component ID's to zero degree vertices (zero degree vertices are acyclic components) since they can be identified simply by checking the vertex degree field which is equal to zero. It can be easily shown [36] that every acyclic component has precisely one vertex that can be used to store order information about indirect edges. Furthermore, this vertex can be any vertex in that acyclic component. However, once this "free choice" is occupied, the rest of the acyclic subgraph can not be used to store order information of any edge (key) other than its own edges (keys). Hence, all direct edges in these acyclic subgraphs can be processed once that "free choice" vertex is used. Clearly the identification step can be finished in $O(n)$ time because we traverse the vertex sequence only once.

Since the vertex degree distribution is decidedly skewed and the graph is relatively sparse, the algorithm uses a sufficient number of stacks to identify the next maximum degree processed and the unprocessed vertices. These stacks accelerate choosing the next vertex with maximal degree. This implementation of the Ordering step requires only $O(n)$ time to finish. Figure 5.3 details the Ordering step. In step (1), STACKS and vertex ordering VS are initialized. In step (2), we choose a vertex v_1 of maximum degree. In step (3), all vertices adjacent to v_1 are pushed on STACKS according to their degree. In (4), the rest of the vertices in the current component are processed and added to the vertex ordering VS. STACKS are used to identify those vertices that have not been selected and to return an unselected vertex of maximum degree. In (5), we initialize ID to zero. In step (6), we follow the vertex ordering marking every vertex with its component ID. Also every component is marked cyclic or acyclic. As discussed earlier, the start of a component is found by checking for $|K(v)| = 0$ where v is the current vertex during traversal of the vertex ordering. For acyclic components, all vertices in the component must have $|K(v)| = 1$.

5.3.3 The Searching Step

The Searching step takes the levels produced in the Ordering step and tries to assign hash values to indirect edges according to the ordering. Assigning hash values to $K(v_i)$ amounts to assigning a value to $g(v_i)$.

When a value is to be assigned to `vertex`[i].g, there are usually several choices for `vertex`[i].g that place all the indirect keys in $K(v_i)$ into acyclic components. The direct edges in these acyclic components are hashed to their desired locations as well. After hashing all indirect edges according to the vertex ordering, the rest of the direct edges in unassigned acyclic components are processed to finish the searching. In looking for a value for `vertex`[i].g, the Searching step uses a random probe sequence to access the slots $0, \ldots, \mathtt{n} - 1$ of the g table.

Figure 5.4 gives the algorithm for the Searching step. A random probe sequence of length n is chosen in step (3). At the beginning of the Searching step, the current implementation chooses a set of 20 small primes that do not divide n. Each time (3) is executed, one of the

```
(1) initialize(STACKS)
    initialize ordering sequence VS
    $v_1$ = a vertex of maximum degree
(2) mark $v_1$ SELECTED and add to VS list
(3) for each w adjacent to $v_1$ do
        push(w, STACKS[deg(w)])
    i = 2
(4) while some vertex of nonzero degree is not SELECTED do
        while STACKS are not empty do $v_i$ = popmax(STACKS)
            mark $v_i$ SELECTED and add to VS list
            for w adjacent to $v_i$ do
                if w is not SELECTED and w is not in STACKS[deg(w)]  then
                    push(w, STACKS[deg(w)])
            i = i + 1
        endwhile
    endwhile
(5) initialize ID to zero
(6) while not the end of the VS do (* follow the VS sequence *)
        (* All vertices in a component have the same component ID *)
        mark vertex $v_i$ with ID
        if $|K(v_i)| > 1$ then
            vertex[$v_i$].mark = 1
            Component[ID ] = cyclic
        else
            vertex[$v_i$].mark = 0
            Component[ID ] = acyclic
        if $|K(v_i)| = 0$ then
            increment ID
        VS = vertex[i].succ
    endwhile
```

Figure 5.3: The Ordering Step

primes q is chosen at random to be s_1 and is used as an increment to obtain the remaining $s_j, j \geq 2$. Thus, the random probe sequence is

$$0, q, 2q, 3q, \ldots, (2r-1)q.$$

In step (4), direct edges are handled. In step (5), first we compute $h(k)$. Then in (6) we check if the indirect vertex address is cyclic or assigned. If so, we try another random probe value s_j. If the indirect vertex address is in an acyclic component and it is not assigned, then we store the order information as detailed in step (6). The Searching step `fails` if all random probe values result in collision.

5.3.4 Analysis

We bound the expected total length of cycles in the graph. Let C_{2k} denote the number of cycles of length $2k$. To build a cycle of length $2k$, we select $2k$ vertices and connect them with $2k$ edges in any order. There are $\binom{r}{k}^2$ ways to choose $2k$ vertices out of $2r$ vertices of a graph, $k!k!/2k$ ways of connecting them into a cycle, and $(2k)!$ possible ordering of the edges. The cycle can be embedded into the structure of the graph in $\binom{r}{r-2k} r^{2(r-2k)}$ ways. Hence, the number of graphs containing a cycle of length $2k$ is

$$\binom{r}{k}^2 ((k!)^2/2k)(2k)! \binom{r}{r-2k} r^{2(r-2k)}$$

Also, the expected number of cycles of length $2k$ in the graph is

$$\sum_{k=1}^{n/2} 2kE(C_{2k}) = \sum_{k=1}^{n/2} \frac{\binom{r}{k}^2((k!)^2/2k)(2k)!\binom{r}{r-2k} r^{2(r-2k)}}{n^{2k}}$$

And the expected number of cycles in the graph is

$$\sum_{k=1}^{n/2} E(C_{2k}) = \sum_{k=1}^{n/2} \frac{\binom{r}{k}^2((k!)^2/2k)(2k)!\binom{r}{r-2k} r^{2(r-2k)}}{n^{2k}}$$

Next, we count the number of tree components in G, excluding zero-degree vertex components. We have the number of different trees in a bipartite graph G':

$$R_{ij} = j^{i-1} \cdot i^{j-1}$$

```
(* edge[$k$].order is the desired location (order) of the Key $k$ *)
(1)  for i $\in$ [0 ... n $-$ 1] do
         vertex[i].assigned = false
(2)  for i = 1 to t (number of levels in $VS$) do
(3)      establish a random probe sequence $s_0, s_1, \ldots, s_{n-1}$ for [0 ... n $-$ 1]
         j = 0
         do
             collision = false
             if $v_i \in$ [0 ... r $-$ 1] then w = 1 else w=2
                 for each k $\in$ K($v_i$) do
                     if $|K(v_i)|$ =1 and vertex[edge[k].$h_{3-w}$].mark = 1 then
                         a = edge[k].$h_0$ + vertex[edge[k].$h_{3-w}$].g   (mod $n$)
                         if edge[k].order $\geq$ a then
                             vertex[$v_i$].g = edge[k].order -a
                         else vertex[$v_i$].g = n- a + edge[k].order
                     else
(4)                      h(k) = edge[k].$h_0$ + vertex[edge[k].$h_{3-w}$].g + $s_j$   (mod 2r)
(5)                      if component[vertex[h(k)]] is cyclic or assigned then
                             collision = true
(6)          if not collision then
                 for each k $\in$ K($v_i$) do
                     component[vertex[h(k)]].assigned = true
                     vertex[h(k)].g = edge[k].order
                     vertex[$v_i$].g = $s_j$
                 else j = j + 1
(7)              if j > n $-$ 1 then fail
         while collision
```

Figure 5.4: The Searching Step

ratio	$N_{acyclic}$	$N_{indirect}$	Mapping	Ordering	Searching	Total
			(seconds)	(seconds)	(seconds)	(seconds)
1.40	463	108	2.63	0.63	0.93	4.19
1.3	364	113	2.60	0.63	0.98	4.21
1.2	280	152	2.63	0.57	1.10	4.30
1.18	282	196	2.58	0.58	1.23	4.39
1.16	272	216	2.60	0.58	3.57	6.75

Table 5.2: OPMPHF Timing Results, Key Set Size n=1024

The expected number of trees of distinct edges of size from 1 to $\min(n, 2r-1)$ in a bipartite graph G with r vertices on each side is

$$\mathrm{E(TR)} = \sum_i \sum_j \frac{\binom{r}{i}\binom{r}{j} \cdot R_{ij} \cdot \binom{n}{i+j-1} \cdot (i+j-1)! \cdot \left(r^2 + i \cdot j - r \cdot (i+j)\right)^{|n-i-j+1|}}{r^{2n}}$$

where i and j should satisfy the constraints $n - i \geq 1$ and $n - j \geq 1$ when $i + j - 1 < n$, or $n - i \geq 0$ and $n - j \geq 0$ when $i + j - 1 = n$.

Additional analysis is provided in [34].

5.3.5 Experimental Results

To provide further insights into our algorithm, we provide some experimental and timing statistics for various key set sizes and ratios, collected from random dependency graphs and their corresponding orderings. These statistics include the number of acyclic components $N_{acyclic}$, and the number of indirect edges $N_{indirect}$. The timing statistics include Mapping time, Ordering time, Searching time, and total time. Tables 5.2 - 5.4 show these statistics for various key set sizes.

5.3.6 Example

We show an example of finding an OPMPHF for the 20 key set listed in Table 5.5. First, we restrict the range of g values to $\{0..19\}$. The words/key used is 1.2. The h_0, h_1,

ratio	$N_{acyclic}$	$N_{indirect}$	Mapping (seconds)	Ordering (seconds)	Searching (seconds)	Total (seconds)
1.40	4355	710	6.02	6.00	5.60	17.62
1.30	3514	1027	5.98	5.73	5.72	17.43
1.20	2859	1706	5.95	5.38	7.27	18.60
1.19	2815	1820	5.95	5.37	8.05	19.37
1.18	2685	1754	5.98	5.35	8.25	19.58
1.17	2615	1819	5.97	5.35	9.17	20.49
1.16	2574	1936	5.97	5.33	10.35	21.65
1.15	2551	2083	5.98	5.25	14.30	25.53
1.14	2393	1968	6.05	5.30	14.00	25.35
1.13	2366	2113	5.98	5.23	43.78	54.99

Table 5.3: OPMPHF Timing Results, Key Set Size n=10000

ratio	$N_{acyclic}$	$N_{indirect}$	Mapping (seconds)	Ordering (seconds)	Searching (seconds)	Total (seconds)
1.40	56649	9137	53.58	78.63	68.80	201.01
1.30	46324	14513	52.73	74.30	70.43	197.46
1.20	36857	21490	52.17	69.98	89.30	211.45
1.15	32761	26195	52.17	68.05	162.60	282.82
1.14	31921	27086	51.90	67.22	212.73	331.85
1.13	31048	27883	51.95	66.98	354.98	473.91

Table 5.4: OPMPHF Timing Results, Key Set Size n=130198

Key	$h0$	$h1$	$h2$
x-rays	17	7	17
Euclidean	12	1	14
ethyl ether	4	6	19
Clouet	15	4	23
Bulwer-Lytton	2	4	17
dentifrice	2	7	20
Lagomorpha	2	11	17
Chungking	13	8	15
quibbles	15	7	18
Han Cities	5	5	16
treacherous	14	1	23
calc-	5	6	23
deposited	14	3	23
rotundus	10	9	17
antennae	10	5	23
sodium lamp	9	7	13
oculomotor nerve	2	1	21
tussle	5	0	20
imprecise	8	9	20
meridiem	5	5	21

Table 5.5: The Key Set Used in Example, Words/Key = 1.2

and h_2 values (see Table 5.5) are used to build a bipartite graph. Half of the vertices of the dependency graph correspond to the h_1 values and are labeled $0, \ldots, r-1$. The other half of the vertices correspond to the h_2 values and are labeled $r, \ldots, 2r-1$. There is one edge in the dependency graph for each key in the original set of keys. Figure 5.5 shows the dependency graph with edges labeled with $h_0(k)$ values.

Table 5.6 shows the vertex ordering VS, $K(v_i)$, $|K(v_i)|$, and v_i.mark produced by the Ordering step.

We notice that we have a large cycle and five acyclic components (four isolated vertices, one tree). Table 5.7 shows the specification of the g function produced by the Searching Step. Table 5.8 verifies that we indeed have a OPMPHF for the example key set. Each element of the g table is $\log_2 n$ bits. Next, we allow the range of g to be $0..2^{\lceil \log_2 n \rceil} - 1 = 31$, which

Figure 5.5: The Graph with Edges labeled, for the Key Set Used in Example

requires $\lceil \log_2 n \rceil$ bits and we show its effect on the lowest ratio we achieved in producing a OPMPHF for this key set. The words/key achieved is 1.1, which is very close to the optimal case, that is when words/key is 1.0 (see Section 5.2). Table 5.9 shows the key set and the corresponding (h_0, h_1, h_2) triples for every key. Table 5.10 shows the mark bits and the specification of the g function produced by the algorithm. Table 5.11 verifies that we indeed have a OPMPHF for the example key set.

Level	Vertex	$K(v)$	$\|K(v)\|$	mark
1	v_{23}	{}	0	0
2	v_1	{14}	1	0
3	v_{14}	{12}	1	0
4	v_5	{10}	1	0
5	v_{21}	{5,2}	2	1
6	v_4	{15}	1	0
7	v_{17}	{2}	1	0
8	v_7	{17}	1	0
9	v_{20}	{2}	1	0
10	v_0	{5}	1	0
11	v_9	{8,10}	2	1
12	v_6	{5}	1	0
13	v_{19}	{4}	1	0
14	v_{18}	{15}	1	0
15	v_{16}	{5}	1	0
16	v_{11}	{2}	1	0
17	v_{13}	{9}	1	0
18	v_3	{14}	1	0
19	v_{15}	{}	0	0
20	v_8	{13}	1	0

Table 5.6: The Vertex Ordering for Example, Words/Key = 1.2

Vertex	0	1	2	3	4	5	6	7	8	9	10	11
g	11	9	19	11	1	17	19	2	18	1	18	3
Mark	0	0	0	0	0	0	0	0	0	1	0	0
Vertex	12	13	14	15	16	17	18	19	20	21	22	23
g	13	4	0	16	7	1	11	19	1	4	0	7
Mark	0	0	0	0	0	0	0	0	0	1	0	0

Table 5.7: g Values Assignment to Vertices, Words/Key = 1.2

key	h_0	h_1	h_2	v[h_1].mark	v[h_2].mark	h_f
x-rays	17	7	17	0	0	17+2+1 (mod 20) = 0
Euclidean	12	1	14	0	0	12+9+0 (mod 20) = 1
ethyl ether	4	6	19	0	0	4+19+19 (mod 20) = 2
Clouet	15	4	23	0	0	15+1+7 (mod 20) = 3
Bulwer-Lytton	2	4	17	0	0	2+1+1 (mod 20) = 4
dentifrice	2	7	20	0	0	2+2+1 (mod 20) = 5
Lagomorpha	2	11	17	0	0	2+3+1 (mod 20) = 6
Chungking	13	8	15	0	0	13+18+16 (mod 20) = 7
quibbles	15	7	18	0	0	15+ 2+11 (mod 20) = 8
Han Cities	5	5	16	0	0	5+17+7 (mod 20) = 9
treacherous	14	1	23	0	0	14+9+7 (mod 20) = 10
calc-	5	6	23	0	0	5+19+7 (mod 20) = 11
deposited	14	3	23	0	0	14+11+7(mod 20) = 12
rotundus	10	9	17	1	0	g[10+1+1 (mod 20)] = 13
antennae	10	5	23	0	0	10+17+7 (mod 20) = 14
sodium lamp	9	7	13	0	0	9+2+4 (mod 20) = 15
oculomotor nerve	2	1	21	0	1	g[2+9+4 (mod 20)] = 16
tussle	5	0	20	0	0	5+11+1 (mod 20) = 17
imprecise	8	9	20	1	0	g[8+1+1 (mod 20)] =18
meridiem	5	5	21	0	1	g[5+17+4 (mod 20)] = 19

Table 5.8: Keys and Their Final Hash Addresses, Words/Key = 1.2

Key	$h0$	$h1$	$h2$
x-rays	3	1	11
Euclidean	22	1	14
ethyl ether	16	4	20
Clouet	6	1	17
Bulwer-Lytton	2	0	16
dentifrice	8	0	18
Lagomorpha	18	7	19
Chungking	25	5	11
quibbles	13	6	21
Han Cities	14	10	17
treacherous	14	9	18
calc-	4	3	18
deposited	25	1	19
rotundus	22	8	19
antennae	15	4	11
sodium lamp	18	6	19
oculomotor nerve	13	2	12
tussle	8	2	17
imprecise	5	10	14
meridiem	19	7	13

Table 5.9: The Key Set Used in Example, Words/Key = 1.1

Vertex	0	1	2	3	4	5	6	7	8	9	10
g	14	17	29	21	19	1	27	17	21	10	15
Mark	0	0	0	0	0	0	0	0	0	0	0
Vertex	11	12	13	14	15	16	17	18	19	20	21
g	12	6	15	20	1	20	12	18	2	31	31
Mark	0	0	0	0	1	0	0	0	0	0	0

Table 5.10: g Values Assignment to Vertices, Words/Key = 1.1

key	h_0	h_1	h_2	v[h_1].mark	v[h_2].mark	h_f
x-rays	3	1	11	0	0	3+17+12 (mod 32) = 0
Euclidean	22	1	14	0	1	g[22+17+20(mod 22)] = 1
ethyl ether	16	4	20	0	0	16+19+31 (mod 32) = 2
Clouet	6	1	17	0	0	6+17+12 (mod 32) = 3
Bulwer-Lytton	2	0	16	0	0	2+14+20 (mod 32) = 4
dentifrice	18	7	19	0	0	18+17+2 (mod 32) = 5
Lagomorpha	25	5	11	0	0	25+1+12 (mod 32) = 6
Chungking	13	6	21	0	0	13+27+31 (mod 32) = 7
quibbles	8	0	18	0	0	8+ 14+18 (mod 32) = 8
Han Cities	14	10	17	0	0	14+15+12 (mod 32) = 9
treacherous	14	9	18	0	0	14+10+18 (mod 32) = 10
calc-	4	3	18	0	0	4+21+18 (mod 32) = 11
deposited	25	1	19	0	0	25+17+2(mod 32) = 12
rotundus	22	8	19	0	0	22+21+2 (mod 32) = 13
antennae	15	4	11	0	0	15+19+12 (mod 32) = 14
sodium lamp	18	6	19	0	0	18+27+2 (mod 32) = 15
oculomotor nerve	13	2	12	0	0	13+29+6 (mod 32) = 16
tussle	8	2	17	0	0	8+29+12 (mod 32) = 17
imprecise	5	10	14	0	1	g[5+15+20 (mod 22)] =18
meridiem	19	7	13	0	0	19+17+15 (mod 32) = 19

Table 5.11: Keys and Their Final Hash Addresses, Words/Key = 1.1

5.4 Conclusions

In this chapter, we have presented a simple and efficient algorithm for finding order preserving hash functions that store very large key sets according to any prior order. These functions are optimal in time (perfect) and space (minimal). These features are highly desirable when designing data structures for slow media such as CD-ROMs.

Chapter 6
Dynamic Perfect Hashing

6.1 Introduction

There has been considerable work in several areas related to dynamic hashing. The major work concerns the design of their file structures for efficient query processing. As this research is closely related to secondary storage media, we present algorithms that are suitable for disk based file structures. In this chapter, we present a dynamic file structure that combines linear hashing and minimal perfect hashing to achieve retrieval performance of one single access, and constant expected time for insertion and deletion. In Section 6.2, we present our algorithm. In Section 6.3, we compare our algorithm with other algorithms and their file structures.

6.2 The File Structure

In this section, we present an efficient file structure that has the following advantages:

- Fast random access: given a search key, only one disk access is required, which is optimal.
- Fast sequential processing or range query access: given a range for a search key, the file structure requires only one seek followed by successive block reads for each contiguously allocated storage area of the file.
- Dynamic: The file structure is incrementally expandable with low insertion and deletion overhead.

Our file structure is a two level dynamic file. The first level is designed to adapt to non-uniform distributions by proper partitioning of the key space into uniformly distributed subranges. The second level is composed of contiguous storage areas that are independently and dynamically managed using minimal perfect hash functions or order preserving minimal perfect hash functions.

This scheme is derived from linear hashing, with the additional feature that the key-sequential order can be preserved within consecutive bucket ranges for efficient sequential processing and range queries as well as random access. Records in *elastic buckets* are stored in contiguous disk areas, and are accessed with minimal perfect hash functions, whose specifications are stored in bucket headers. The size of elastic buckets can grow as needed, given that a perfect hash function is found to access elements stored in the bucket.

At the first level, a hash function $h(k)$ that distributes the universe uniformly over the interval $[0 \dots N-1]$ is used to map the keys in S into integers in the range 0 to $N-1$. A key k is stored in the bucket $B_{h(k)}$. We may choose

$$h(k) = k \pmod{N}.$$

At the second level, buckets of keys are managed using minimal perfect hash functions or order preserving hash functions. Every time a new key is inserted, an appropriate perfect hash function is computed if the bucket size is greater than a threshold. The threshold value is chosen such that the cost of finding a perfect hash function is justifiable. Experimental runs show that a threshold value of 5 is a good choice. The parameters of the perfect hash function are stored in the bucket header. Efficient algorithms for computing minimal perfect hash functions are presented in Chapter 4. Algorithms for computing order preserving minimal perfect hash functions are presented in Chapter 5.

As in linear hashing, buckets are split when the load factor is reached, to keep buckets manageable and relatively small. To facilitate locating keys quickly, we maintain two masks and use bit operations. The low mask is equal to the maximum split bucket and the high mask is equal to the next maximum split bucket. To locate a specific key, we compute $h(k)$

```
bucket($k$) = $h(k)$ & high_mask
**if** (bucket($k$) $\geq$ max_bucket ) **then**
    bucket($k$) = $h(k)$ & low_mask
**return** (bucket)
```

Figure 6.1: The Fast Address Computation Algorithm

and mask it with high_mask. If the resulting number is greater than the maximum bucket in the table, the hash value is masked with the low mask. This fast address computation algorithm is illustrated in Figure 6.1. Splits occur in a predefined order (linearly). The time at which pages are split is determined by page overflows and when the table fill factor is exceeded.

The insertion, retrieval, and deletion operations are detailed in Figure 6.2, Figure 6.3, and Figure 6.4, respectively.

6.3 Experimental Results

In this section, we compare our dynamic perfect hashing scheme (Dphf) with other data structures such as random search trees (rs_tree) [40], red black trees (rb_tree) [40], skip lists (skiplist) [40], and chaining hashing (ch_chain) [44, 40]. We have chosen two widely available libraries of efficient data structures and algorithms; LEDA [59] and LEND [16]. We also compare our DPHF scheme with dynamic perfect hashing schemes available in LEDA (LEDA_dp) and LEND (LEND_dp). LEDA is a library of efficient data types and algorithms that is available in the public domain from most anonymous ftp servers. LEDA's main strength is the inclusion of many of the most recent and efficient implementations of graph algorithms and related data structures. LEND is a library of efficient data structures that uses minimal perfect hashing algorithms to optimize performance and guarantee minimum disk accesses and near minimal space overhead. Also, LEND supports efficient hash indexing, caching, and buffering. Figure 6.5 shows the code used to run the comparisons. Our file structure was initialized to empty with zero buckets. The file grew as keys were

```
found = table.found($k$) // check if key $k$ is already in table
**if** found **return**
bucket($k$) = $h(k)$ & high_mask
**if** (bucket($k$) $\geq$ max_bucket ) **then**
    bucket($k$) = $h(k)$ & low_mask
**if** bucket($k$).empty() **then**
    bucket($k$).init()
    bucket($k$).mark = DIRECT
    bucket($k$).size = 1
    bucket($k$).add($k$) // insert item in bucket
**else**
    bucket($k$).mark = INDIRECT
    increment bucket($k$).size
    bucket($k$).insert($k$)
    bucket($k$).compute_mphf() // compute a new MPHF or OPMPH
increment key_count
**if** load_factor $\geq$ max_load_factor **then**
    bucket($k$).expand() // expand by one bucket
```

Figure 6.2: The Insertion Algorithm

```
bucket($k$) = $h(k)$ & high_mask
**if** (bucket($k$) $\geq$ max_bucket ) **then**
    bucket($k$) = $h(k)$ & low_mask
**if** (bucket($k$).mark == DIRECT) **then**
    **return** item // only one record in this bucket
**else**
    **if** bucket($k$).mphf_exist() **then** // MPHF exists
        item = bucket($k$).mphf_compute($k$)
    **else**
        item = bucket($k$).binary_search($k$)
    **if** (item.key() == $k$) **then**
        **return** item
```

Figure 6.3: The Retrieval Algorithm

```
bucket($k$) = $h(k)$ & high_mask
**if** (bucket($k$) $\geq$ max_bucket ) **then**
    bucket($k$) = $h(k)$ & low_mask
**if** bucket($k$).mark == DIRECT **then**
    delete item // only one record in this bucket
**else**
    **if** bucket($k$).mphf_exist() **then** // MPHF exists
        item = bucket($k$).mphf_compute($k$)
    **else**
        item = bucket($k$).binary_search($k$)
    **if** item.key() == $k$ **then**
        delete item
```

Figure 6.4: The Deletion Algorithm

added. We have used a load factor of 5. Minimal perfect hash functions were computed for each bucket when a threshold of 5 was reached. Also, for LEDA timing tests, we have used the default parameters that are chosen when installing LEDA.

n	Dphf	rs_tree	skiplist	rb_tree	ch_hash	LEDA_dp	LEND_dp
	(sec)	(sec)	(sec)	(sec)	(sec)	(sec)	(sec)
8	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16	0.00	0.00	0.00	0.00	0.00	0.00	0.00
32	0.00	0.00	0.00	0.00	0.00	0.00	0.00
64	0.00	0.00	0.00	0.00	0.00	0.00	0.00
128	0.00	0.00	0.00	0.02	0.00	0.00	0.00
256	0.00	0.00	0.00	0.00	0.02	0.00	0.02
512	0.00	0.00	0.02	0.02	0.00	0.03	0.03
1024	0.00	0.02	0.02	0.03	0.02	0.05	0.07
2048	0.02	0.03	0.05	0.07	0.03	0.13	0.13
4096	0.03	0.08	0.10	0.15	0.07	0.20	0.30
8192	0.08	0.18	0.25	0.35	0.15	0.48	0.60
16384	0.17	0.40	0.55	0.75	0.32	0.67	1.23
32678	0.33	1.00	1.27	1.67	0.52	1.62	2.58
65356	0.68	2.25	3.07	3.87	1.12	2.97	4.97
130712	1.38	4.83	6.85	8.88	2.58	6.88	13.08
261424	2.75	11.22	17.27	24.02	5.52	18.22	35.04
522848	5.50	28.23	45.27	48.72	12.40	55.25	125.06

Table 6.1: Insertion Timing Results Comparing Dynamic Data Structures for Various Key Set Sizes for Test Program

Tables 6.1, 6.2, 6.3, and 6.4 summarize our results. All runs were carried out on a DECstation 5000/25. Figures 6.6, 6.8, and 6.10 show that our dynamic data structure Dphf outperforms other dynamic perfect hashing schemes LEDA_dp, and LEND_dp. Also, Figures 6.7, 6.9, and 6.11 show that our dynamic perfect hashing scheme outperforms other data structures available in LEDA and LEND (rs_tree, skiplist, rb_tree, and ch_hash) . Figure 6.12 shows the linear time complexity for insertion, retrieval, and deletion operations using our dynamic perfect hashing scheme Dphf.

```
main()
{
        DB *dbp;
        KeyStructPtr key;
        AttributeStructPtr attr;
        int i;
        HASHINFO ctl;
        DBT db_item, db_key;
        float insertion_time, retrieval_time, deletion_time;
        dbp = DphfHashCreate(NULL); // create a DPHF table
        key = (KeyStructPtr) malloc(sizeof(KeyStruct)); // allocate key struct
        attr = (AttributeStructPtr) malloc(sizeof(AttributeStruct)); // allocate attribute
        key→field2 = 100.0; // fill in key fields
        attr→field2 = key→field2; // fill in attribute fields
        insertion_time = used_time1();// start insertion timing
        for (i = 0; i < count; i++) {
           key→field1 = i; // fill in key field1
           attr→field1 = i+1;// fill in attribute field1
           db_key.data = key; // assign key struct to primary key of database item
           db_key.size = sizeof(KeyStruct); // compute key size
           db_item.data = attr; // assign attribute to database item
           if (DphfHashPut(dbp, &db_key, &db_item) ≠ NULL) // insert database item
                   printf("cannot enter: key \n");
        }

        retrieval_time = used_time1();// start retrieval timing
        for (i = 0; i < count; i++) {
           key→field1 = i;
           db_key.data = key;
           db_key.size = sizeof(KeyStruct);
           DphfHashGet(dbp, &db_key, &db_item); // retrieve database item
        }
        deletion_time = used_time1(); // start deletion timing
        for (i = 0; i < count; i++) {
           key→field1 = i;
           db_key.data = key;
           db_key.size = sizeof(KeyStruct);
           DphfHashDelete(dbp, &db_key); // delete database item
        }
}
```

Figure 6.5: Insertion, Retrieval, and Deletion Test Program

n	Dphf	rs_tree	skiplist	rb_tree	ch_hash	LEDA_dp	LEND_dp
	(sec)	(sec)	(sec)	(sec)	(sec)	(sec)	(sec)
8	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16	0.00	0.00	0.00	0.00	0.00	0.00	0.00
32	0.00	0.00	0.00	0.00	0.00	0.00	0.00
64	0.00	0.00	0.00	0.00	0.00	0.02	0.00
128	0.00	0.00	0.00	0.00	0.00	0.00	0.00
256	0.00	0.00	0.00	0.00	0.00	0.00	0.00
512	0.00	0.02	0.00	0.00	0.00	0.00	0.02
1024	0.02	0.00	0.00	0.02	0.00	0.02	0.02
2048	0.03	0.02	0.02	0.02	0.00	0.02	0.02
4096	0.05	0.05	0.07	0.07	0.03	0.05	0.05
8192	0.08	0.12	0.20	0.15	0.05	0.12	0.12
16384	0.15	0.30	0.47	0.35	0.10	0.20	0.23
32678	0.32	0.70	1.05	0.80	0.23	0.42	0.50
65356	0.62	1.75	2.70	1.93	0.50	0.95	0.97
130712	1.23	3.82	5.75	4.42	1.18	2.37	2.35
261424	2.47	9.53	16.52	11.45	2.65	4.33	4.01
522848	4.90	28.98	43.80	27.40	5.70	10.05	9.24

Table 6.2: Retrieval Timing Results Comparing Dynamic Data Structures for Various Key Set Sizes for Test Program

n	Dphf	rs_tree	skiplist	rb_tree	ch_hash	LEDA_dp	LEND_dp
	(sec)	(sec)	(secs)	(sec)	(sec)	(sec)	(sec)
8	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16	0.00	0.00	0.00	0.00	0.00	0.00	0.00
32	0.00	0.00	0.00	0.00	0.00	0.00	0.00
64	0.00	0.00	0.00	0.00	0.00	0.00	0.00
128	0.00	0.00	0.00	0.00	0.00	0.02	0.00
256	0.00	0.00	0.02	0.02	0.00	0.02	0.00
512	0.00	0.00	0.00	0.02	0.00	0.02	0.00
1024	0.00	0.02	0.02	0.02	0.00	0.03	0.00
2048	0.00	0.03	0.03	0.07	0.02	0.10	0.03
4096	0.03	0.05	0.08	0.15	0.05	0.18	0.07
8192	0.07	0.13	0.18	0.33	0.10	0.43	0.13
16384	0.15	0.33	0.45	0.75	0.23	0.65	0.27
32678	0.30	0.82	1.03	1.67	0.37	1.55	0.52
65356	0.62	1.95	2.63	3.78	0.82	3.05	1.07
130712	1.22	4.32	5.87	9.63	1.73	7.42	2.20
261424	2.43	10.57	16.05	28.58	3.80	16.32	4.53
522848	4.92	31.12	42.62	57.92	9.63	34.25	10.05

Table 6.3: Deletion Timing Results Comparing Dynamic Data Structures for Various Key Set Sizes for Test Program

n	Insertion	Retrieval	Deletion
	(seconds)	(seconds)	(seconds)
1000	0.00	0.02	0.00
2000	0.02	0.02	0.02
4000	0.05	0.05	0.05
8000	0.10	0.07	0.07
10000	0.12	0.10	0.08
20000	0.27	0.23	0.25
40000	0.42	0.37	0.40
80000	0.85	0.77	0.73
100000	1.07	0.95	0.93
200000	2.12	1.87	1.88
400000	4.20	3.75	3.77
800000	8.45	7.50	7.52
1000000	10.60	9.38	9.35
2000000	21.08	18.75	18.72
3000000	31.63	28.20	28.02
4000000	42.23	37.80	37.70
5000000	52.97	47.00	46.77

Table 6.4: Insertion, Retrieval, and Deletion Timing for Various Key Set Sizes for Test Program

6.4 Example

The basic idea of combining linear hashing and minimal perfect hash functions is illustrated in Figure 6.13. We start with a file of 5 buckets 0 to 4, each with the maximum capacity to store 3 records. The *sp* split pointer points to bucket #0. L, the generation number is 0. Recall that the generation number indicates how many times the file has doubled. The home function that determines the address of a bucket is given by $b_L(k) = k \pmod{N * 2^L}$. If $b_L(k) < sp$, then we use the split function $b_{L+1}(k) = k \pmod{N * 2^{L+1}}$. As discussed earlier in Section 4.5, buckets are accessed using minimal perfect hash functions to speed up insertion, retrieval, and deletion. For this example, we assume that we compute MPHFs for all buckets irrespective of their sizes. Obviously, in a real situation, we compute MPHFs for buckets of sizes higher than a threshold value (i.e., the load factor ≤ 5). For buckets of

Figure 6.6: Dphf, LEDA_dphf, and LEND_dphf Insertion Timing for Various Key Set Sizes for Test Program

Figure 6.7: Dphf, Rs_tree, Skip_list, Rb_tree, and Ch_chain Insertion Timing for Various Key Set Sizes for Test Program

Figure 6.8: Dphf, LEDA_dphf, and LEND_dphf Retrieval Timing for Various Key Set Sizes for Test Program

Figure 6.9: Dphf, Rs_tree, Skip_list, Rb_tree, and Ch_chain Retrieval Timing for Various Key Set Sizes for Test Program

Figure 6.10: Dphf, LEDA_dphf, and LEND_dphf Deletion Timing for Various Key Set Sizes for Test Program

Figure 6.11: Dphf, Rs_tree, Skip_list, Rb_tree, and Ch_chain Deletion Timing for Various Key Set Sizes for Test Program

Figure 6.12: Dphf Insertion, Retrieval, and Deletion Timing for Various Key Set Sizes for Test Program

smaller sizes, searching a bucket requires constant time and is $O(1)$. For each bucket, we compute an MPHF $h_{\text{bucket\_number}}$ to access its elements in $O(1)$ operations.

Initially, the file is loaded with 12 records as shown in Figure 6.13. The bucket address hashing function is $b_0(k) = k \pmod 5$. Note that buckets 1 and 2 are full. A collision takes place when a new record's key, to be inserted, hashes to a bucket that is already full.

The expansion of the file is performed by extending it by appending one bucket at a time. This bucket receives some records moved from an existing bucket, which undergoes a split (i.e., expands). The next bucket to split is determined by the split pointer sp that moves sequentially, after each split, from bucket 0 to bucket 4. The file gradually grows from 5 to 10 buckets (0 to 9). When all 5 buckets have been split and the table size has doubled to 10, the split pointer sp is reset to bucket #0 and the splitting process starts over again. This time the split pointer sp travels from 0 to 9, doubling the table size to 20 and the generation number L would be 2. This expansion process can continue as long as required.

At the beginning of an expansion cycle, the split pointer is at bucket 0 (arrow in Figure 6.13). Then the split pointer is advanced to point to bucket 1. L is incremented and is equal to 1. The split is resolved by rehashing the splitting bucket with the function $b_1(k) = k \pmod{10}$. After splitting a bucket into two buckets, new minimal perfect hash functions are computed for each bucket. To illustrate the expansion process, let's insert key = 122: $b_1(122) = 2$. Since bucket #2 is full, elements of Bucket #2 are moved into a larger bucket with a capacity of 4 elements along with key 122 as shown in Figure 6.14. Also, bucket #0 splits and keys 0, 65 are rehashed with $b_1(k)$, moving k = 65 to the new bucket 5.

Figure 6.15 shows the result of inserting 109 ($b_0(109) = 4$) in Bucket #4. Figure 6.16 illustrates another split when inserting 67 ($b_0(67) = 2$) in bucket #2 that is full. Elements of Bucket #2 are moved into a larger bucket with a capacity of 5 elements along with key 67. Also bucket #1 splits, and sp is advanced to bucket #2. Inserting 244 ($b_0(244) = 4$) in bucket #4 that is already full triggers another split. Bucket #2 is split into two buckets as shown in Figure 6.17 and 6.18. Figures 6.19 and 6.20 illustrate inserting 68 ($b_0(68) = 3$) and

Figure 6.13: Initial File Status for DPHF Example

Figure 6.14: Splitting Bucket #0 in DPHF Example

Figure 6.15: Inserting Key 109 in Bucket #4 in DPHF Example

Figure 6.16: Inserting Key 67 and Splitting Bucket #1 in DPHF Example

Figure 6.17: Inserting Key 244 in Bucket #4 in DPHF Example

Figure 6.18: Splitting Bucket #2 in DPHF Example

Figure 6.19: Inserting Keys 68, 78 in Bucket #3 in DPHF Example

Figure 6.20: File Status after Splitting Buckets #3, #4 in DPHF Example

78 ($b_0(78) = 3$). Now, the file has doubled in size, L, the generation number is incremented to 1, the home function is set to $b_1(k) = k \pmod{10}$, and the split function is set to $b_2(k) = k \pmod{20}$. A new expansion cycle can begin with the split function $b_2(k) = k \pmod{20}$. The split pointer is reset to bucket 0.

6.5 Conclusions

In this chapter, we have presented a dynamic file structure that combines linear hashing and uses minimal perfect hashing to achieve high storage utilization due to the use of compact minimal perfect hashing for handling buckets. The new data structure outperforms other tree based structures, is well balanced, and achieves retrieval performance of one single access (worst case constant time), worst case constant time for deletion. Our dynamic perfect hashing scheme allowed us to implement a very efficient file structure, suitable for large information retrieval systems and features optimal random access: given a search key, only one disk access is required, and fast sequential processing: the file structure requires only one seek followed by successive block reads for each contiguously allocated storage area of the file. Moreover, the file structure is incrementally expandable with low insertion and deletion overhead.

Chapter 7

Conclusions

In this dissertation, we have demonstrated the feasibility and applicability of advanced retrieval methods, perfect hashing schemes, and efficient ranking methods to provide effective and efficient access to very large collections, dictionaries, and online library catalogs. We have described an experiment called REVTOLC that was designed to test and improve search methods for online catalogs. Also, we have assessed the effectiveness and ease of use of major advanced information retrieval methods, namely extended Boolean (p-norm), vector, and vector with probabilistic feedback methods. We have presented an efficient minimal perfect hash function generator that produces optimal MPHFs. We have presented novel order preserving techniques for handling large dictionaries, and efficient algorithms to generate them. Also, we have combined linear-hashing techniques with perfect hashing methods to design an efficient dynamic data structure. We illustrated its efficiency with large key sets such as the UNIX dictionary and the OCLC key set. We have presented fast best match search algorithms that enhance the query response time in modern information retrieval systems. These algorithms return the relevant document set computing an upper space bound on closeness; eliminating the need for an exact computation in many instances and eliminating useless disk accesses.

Chapter 8

Future Research

Finally, we suggest several problems that are worth investigating:

- It remains an open problem to find an efficient key to address transformation that can handle variable length alphanumeric keys. Throughout this dissertation, we have assumed that primary keys are bounded in length. However, if strlen(k) is greater than the number of keys n, and keys differ slightly (i.e., the last character), then more suitable key to address transformations are required.

- In Chapter four, we have presented an efficient minimal perfect hash function generator that produces very compact MPHFs for small key sets as well as large ones. It remains to be seen how can we use this generator for compacting a trie and what would be the storage savings.

- It remains an open problem to see how Melhorn's lower bound on the size of MPHFs relates to implicit data structures. Is there any general implicit key to address transformation that can be evaluated in constant time? If so, what is the load factor?

- In Chapter five, we have presented a probabilistic algorithm for finding order preserving minimal perfect hash functions for static sets. It remains an open problem to find a deterministic algorithm that runs in linear time and produces order preserving minimal perfect hash functions.

- It remains an open problem to extend the order preserving MPHFs algorithm to generate functions that handle range queries efficiently.

- It remains an open problem to choose an optimal combination of weighting schemes, similarity functions, and stopping rules for extended Boolean and vector retrieval methods.

- The REVTOLC experiment revealed the effectiveness of advanced retrieval methods for OPACs. It would be very useful to run a similar experiment for: abstracts and full texts of research articles and technical papers.

- It remains an open problem to determine if there exists a data structure that requires $O(n)$ space, has constant insertion time (worst case), and has constant retrieval and deletion time (worst case).

Appendix A
MPHFs for C and C++ Reserved Words

In this section, we show an example of finding a MPHF for the C and C++ reserved words listed in Tables A.1 and A.2. Figures A.1 and A.2 show the key set dependent part of the minimal perfect hash functions generated for both C and C++ reserved words. Figure A.3 shows the fixed part of the code. The driver code is the same for all generated minimal perfect hash functions. The details specific to a key set is hidden in the *compute* function. We show the MPHF parameters in a i*switch* statement. However, for larger functions, g is an array of integers indexed by h. This approach doesn't require messy random tables or mark bits and can handle efficiently small key sets. The function is indeed minimal and perfect in each case. We have described the details of the MPHF generator algorithms in Section 4.5.

C Reserved Words
auto, break, case, char, const, continue, default,
do, double, else, enum, extern, float, for, goto,
if, int, long, register, return, short, signed, sizeof,
static, struct, switch, typedef, union, unsigned, void, volatile, while

Table A.1: C Reserved Key Set, $n = 32$

C++ Reserved Words
asm, auto, break, case, char, class, const, continue, default, delete, do, double, else, enum, extern, float, for, friend, goto, if, inline, int, long, new, operator, overload, private, protected, public, register, return, short, signed, sizeof, static, struct, switch, this, typedef, union, unsigned, virtual, volatile, void, while

Table A.2: C++ Reserved Key Set, $n = 45$

```
long compute(h,m,n)
register long h,m,n;//
{register long g; // MPHF parameters
switch( abs(h%3)){ // use h to index g table
case 0 :g =1314049; break;
case 1 :g =182094; break;
case 2 :g =107;
}
return(abs((abs(n)+(7*abs(m)-1)*g)%37)%32); // compute final address
}
```

Figure A.1: C Reserved Words Minimal Perfect Hash Function

```
long compute(h,m,n)
register long h,m,n;
{register long g;
switch( abs(h%6)){ // use h to index g table
case 0 :g =395422; break;
case 1 :g =178811; break;
case 2 :g =2805276; break;
case 3 :g =4130; break;
case 4 :g =18350; break;
case 5 :g =106;
}
return(abs((abs(n)+(7*abs(m)-1)*g)%25147)%45); // compute final address
}
```

Figure A.2: C++ Reserved Words Minimal Perfect Hash Function

```
#define M m=(m≪5)+m+*w++ // computes $h_2(k)$
#define N n=*w1++ +65599*n // computes $h_0(k)$
short mphf(key)
char* key;
{
  register long g, h, n, loop, m, len;
  register char* w;
  register char* w1;

  w = w1 = key;
  h = n = m =0;
  loop= len = strlen(key);// computer $h_1(k)$
  while(loop--)
    h = h*37∧(*w++ -' ');
  h %= 1048583;
  w = key;
  loop = (len + 8 - 1)≫3;
  switch (len & (8-1)){
    case 0:do
           { N; M;
    case 7: N; M;
    case 6: N; M;
    case 5: N; M;
    case 4: N; M;
    case 3: N; M;
    case 2: N; M;
    case 1: N; M;
           } while(--loop);
  }
  return(compute(h,m,n)); // (h,m,n) is unique triplet
}
```

Figure A.3: The Driver Code For Minimal Perfect Hash Functions

Appendix B
REVTOLC Forms

B.1 Questions

Table B.1 shows the 18 questions selected in the REVTOLC experiment. The questions were collected from library patrons visiting the Newman Library at Virginia Tech.

No.	Question
1	Active phase cancellation, noise attenuation systems
2	Thermal desorption spectroscopy
3	Neural networks
4	Sayarner Truth, black abolishionist
5	Foot kicking procedures used with infants
6	A. Stindberg and his influence on the writings of O'Neill
7	Remote sensing
8	Ceramic matrix composites
9	Human relationships in Colonial America
10	Endangered species and fisheries ecology
11	Franchising in USA and international markets
12	Consumer studies and behavior
13	Primitive and organic farming
14	Electrolytic solution temperature dependence on Electromotive Force
15	Early weaning of pigs
16	Communication taxonomies and relational communication
17	A book by Ayn Rand
18	Exploratory behaviors with conjugate reinforcement using foot kicking

Table B.1: REVTOLC Questions

B.2 Protocol

The purpose of this experiment is to lead to more *effective* retrieval methods, especially for improving access to online public access catalogs. The objective of this study is to compare four methods (Boolean, extended Boolean, vector, and vector with feedback) to determine which are better, regarding efficiency, effectiveness, and usability. The actual experiment will involve roughly a half million records provided by staff of the University Library in 1986, that will be searchable using computers in the Department of Computer Science, and computer terminals in public laboratories (i.e., McBryde 116) and other locations.

Participants will number around 220, so that adequate statistical power will allow us to identify small differences between methods. Since there are 4 methods considered, and since it will be necessary for each subject to work with 2 methods, there are 4*(4-1) = 12 combinations of retrieval method pairs; each participant will be randomly assigned to one of these.

All of the participants will be introduced to the use of the system and will receive explanations regarding the process of the experiment at the beginning of the session. They will receive and be asked to sign a consent form which explains the process and informs them that they are free to quit the experiment at any time.

Each participant that completes the experiment will be paid $5 for their assistance, which should require roughly 60-90 minutes of time. A participant will read a short booklet of instructions, and then do all further work with the computer, which will log all key strokes, especially responses, adding timestamps. The steps with the computer are as follows.

1. complete an initial questionaire that provides demographic data

2. for each of two search methods assigned, do the following

   (1) go through a tutorial about that method

   (2) do several practice searches

(3) for each of 2-3 questions provided, endeavor to carry out as many comprehensive searches as seem appropriate to find all relevant books listed in the catalog

3. complete a final questionaire regarding the overall system performance

Each search involves filling in blanks on the terminal screen that allow specification of a query of the proper form, examining the list of titles that are identified, examining detail records on titles of interest, and indicating which titles seem relevant to the question.

The proposed experiment is considered risk free to the participants, and only requires their typing into a computer. They will provide some information regarding their background so that we can determine if some search methods are better for some classes of users. They also will be making judgments regarding how to describe an information need (question) using this software system, and regarding whether retrieved book descriptions are for relevant books — so we can determine how well various retrieval methods perform regarding quickly finding relevant items.

Participants will all be University students, and we will endeavor to obtain a representative sample of the student community. This will give a relatively homogeneous group, but with enough diversity that we may identify trends regarding educational level, major, or other characteristics.

Participants will each have an ID number that we assign, but the log that associates names with ID numbers will be destroyed after data collection for the experiment is complete. Subsequent analysis will be done for the group as a whole, and student-provided information will not be distinguishable for individual identification. Names will also be recorded, along with signatures, for all who receive payment, but ID numbers will not be used with that log.

After completion of the study and subsequent analysis, a technical report will be prepared describing this experiment, and copies will be available to the public (including students who have been participants). A paper will be submitted for publication to share our findings with the scientific community.

B.3 Consent Form

I,________________________________, consent to participate in research concerning advanced methods of information storage and retrieval and their application to online public access library catalogs. I am aware that the purpose of this research is to compare various retrieval methods and to determine how efficient, effective, and usable they are by students — so that in the future it may be possible to have better catalog systems.

I acknowledge that my participation in this experiment is completely voluntary and will take roughly 60-90 minutes. I do reserve the right to quit at any time without penalty or retribution. I do realize, however, that $5 payment will not be made to me unless and until I have completely gone through all steps involved in the experiment.

I understand that the complete experiment for a given participant involves reading a brief explanation booklet, and then interacting through a terminal with a computer system. That interaction will include an initial questionaire, a final questionaire, and shorter intermediate questionaires after searches. I understand that I will work with two different search methods so that comparisons will be possible, and that for each of the two I will go through a brief tutorial, some practice searches, and then no more than 3 searches for various questions that will be assigned to me.

I am aware that my keystrokes and responses will be logged by the computer for future analysis aimed at comparing the retrieval methods and cross tabulating performance with characteristics of searchers. I have been assured that any personal information I provide as a part of this experiment will be kept confidential. I also understand that this information will be compiled for group analysis and used for statistical purposes only, and that the results of my participation will in no wise be attributed to me as an individual.

I am convinced that participating in this study poses no risks to my well-being. I expect that my participation will contribute to research in information retrieval and may lead to more effective catalog systems.

____

Signature

____

Date

B.4 REVTOLC Explanations

REVTOLC (Retrieval Experiment with Virginia Tech OnLine Catalog)

B.4.1 FORMS

Thank you for volunteering to help with our experiment! If you are willing, before proceeding any further please sign the consent form you have been given, and hand it to the person running the experiment. Later, when you have completed the tasks assigned to you in the pages below, inform the person running the experiment of that fact, and you will be asked to sign another form indicating that you have been paid $5 for your time and cooperation.

Speaking of forms, PLEASE READ THE REST OF THIS EXPLANATION BOOKLET IN ITS ENTIRETY BEFORE TOUCHING YOUR ASSIGNED COMPUTER TERMINAL!!

B.4.2 PURPOSE OF THE EXPERIMENT

As was explained on the consent form, we are trying to compare how well various information retrieval methods help people searching for relevant entries in an online library catalog. You will work with two methods so we can see which one you like better, and then can correlate that with opinions of others participating in the experiment. For each method you will work through a tutorial and do a practice search, prior to searching for entries that satisfy each of two questions asked by other students who visited Newman library during this past year. How well you do and what comments you make about each search and each method will help us reach conclusions in our study, hopefully leading to better methods for searching through online library catalogs.

B.4.3 BACKGROUND

You may have looked for books in libraries using an old style card catalog. In recent years, the information in many of those catalogs has been put into computers so that people can more quickly find relevant books, and can easily make comprehensive studies to find as many items as possible on a subject (i.e., when planning research to be sure no one else has found the solution to a problem before). For example, at VPI&SU there is VTLS, a type of online catalog system.

Our system is somewhat like VTLS. At the end of 1986 we were given data by people in Newman Library so we could experiment with different methods of searching. That data, roughly 500,000 entries from the VPI&SU library that were cataloged by 1986, is accessible through REVTOLC. Since we are only studying about searching, and since the data we received is old, we only consider descriptions about the books and other entries – i.e., we ignore circulation records.

B.4.4 SHORT TUTORIAL ON SEARCHING

Finding entries using an online catalog usually involves several actions:

1. think a little about what you are looking for, namely your question
2. tell the computer something about that, in the form of a "query"
3. tell the computer to search using that query
4. examine the results
5. if you want more and believe more can be found, repeat steps 2-4

APPENDIX B. REVTOLC FORMS

In this experiment, you will be given questions asked by other students, recorded on forms we collected in 1989 in Newman Library. Use whatever knowledge you have to help you understand what is meant. A good way to think about this is as follows: A friend of yours wants to find everything available in the library to do a comprehensive report, and writes down for you the question he is investigating. You want your friend to get a good grade, and so you must be as persistent and clever as you can, to tell the computer what it needs, so it can do searches. You must continue doing so until you have found everything you can relevant to the question. Since this could go on forever if you are really thorough, don't spend more than about 15 minutes on any question, and try to find entries "relevant" to the question, not just things that might be interesting.

Some librarians receive special training in searching, but you can do fairly well if you remember the following important points:

1. you must give the computer the right words or names

2. once you find something relevant, you can use it to get more words and/or names

3. you probably need to do at least 2-3 different searches for each question

Getting the right words or names takes thought. The problem is easier if you remember that librarians divide up descriptions of books into separate categories or "fields," for author, title, subject, call number, etc. If you know the name of an author, tell the computer to look for that in the author field. If you know words in the title, tell the computer to look for that in the title field. If you know the first part of the call number, that should be looked for in the call number field. If you know what Library of Congress subject heading would be assigned, that should be looked for in the subject field.

If you just know a little about the subject of the book desired, things are not so easy. Probably the safest strategy is to ask the computer to search for those words, looking in both the title and subject heading fields.

Remember point (2) above also! If you find some book that is relevant, you can use the author, subject heading, title, etc. in a follow-up search. It is likely that the author might have written something else similar. Other books classified with the same subject heading are likely to be related. Also, works with similar titles are probably useful. Call numbers can also be helpful, though it may not be very helpful to put in a call number like "QA76" which is used to describe thousands of engineering books!

One last point has to do with word forms. In the title field, this computer system will treat words with or without plural endings as being the same – so if you ask for "book" in the title and the author had "books" you will still get a "match." Furthermore, in the subject field, since people often don't know exactly what is counted as a subject heading by the Library of Congress, matches are generally considered as proper regardless of the endings. Thus "learning" would match "learns" – you may find some things you don't want because of this, but at least it is less likely that you will miss something you want!

B.4.5 USING THE COMPUTER

You will be using a terminal that will act like a VT100 terminal, since those are easy to find. This terminal is connected to a computer that has the catalog data stored so you can search. The computer is not very big, so be patient if it is a little slow at times.

The computer will tell you or show you things, and you should read whatever it says carefully. In some cases you will be asked to reply with a number, in which case you enter the number and then hit the RETURN key. In most other cases, you need only hit one letter or another key to get the computer to continue. What keys you can hit is listed in the upper right part of the screen, which changes as you proceed, since certain things only make sense at certain times.

The computer lets you use the backspace key to make corrections. When you are entering your query into the computer, the special key PF2 allows you to switch between overwriting what you typed before, or inserting newly typed characters. Also note that when you have finished with typing in a query, the special key PF1 is used to tell the computer to search for books.

If you are confused or unclear about what to do, ask one of the people running the experiment for help!

B.4.6 EXAMINING RESULTS

After the computer has searched based on one of your new queries, it will return with a list of entries, numbered 1-10, showing the title of each. If any title looks like it might possibly be relevant, please follow the instructions to examine the detailed description of that entry, and then indicate if you think it is indeed relevant. Remember that you should act as if your friend's report grade may depend on your persistence!

If you search again for the same question, after each such search the computer will find 10 more items, calling those 1-10, moving the others found earlier down in rank. You can still look over things found earlier, but new items are right at the top, easy to find.

B.4.7 FILLING IN QUESTIONNAIRES

There are a number of questionnaires that will be given online. Please take your time on these, and answer honestly, picking the most accurate reply at each point. At the very end of the experiment, in the final questionnaire, there will be a chance for you to write in more detailed comments of your own, so then you will be able to go beyond multiple choice replies!

B.4.8 YOUR MAIN TASK – WORKING WITH 2 OF 4 RETRIEVAL METHODS

You are assigned to work with two retrieval methods. Here is a short explanation of the four methods being considered in REVTOLC. No one knows how these compare when used for searching online catalogs. See more details about the two you will work with when you go through the online tutorials.

Boolean: Many retrieval systems allow you to precisely characterize what you are looking for by indicating combinations and alternatives of words or names. An example of a Boolean query is:

(Author: Gerard AND Salton) OR (Title: information AND retrieval)

which will find books having an author with the two names "Gerard" and "Salton", or books that have "information" and "retrieval" in the title. No other books would be found. You would miss a book by "G. Salton" and also would miss a book where the title was "Searching for Information". Thus, you will need to carefully think about the wording and combinations to form a Boolean query. Note finally that a Boolean query may not match any books, may match less than or equal to 10 (in which case you will be shown only those), or may match more than 10 (in which case you only will be shown 10 that the computer arbitrarily selects from among the matches).

You can manually alter your query after a search by adding or deleting words in the various fields on the screen. That way you can try to be narrower in your query formulation if you found lots of items, or give a broader query if you did not find much.

Note that the tutorial gives special instructions regarding filling in your Boolean query, and that the special keys PF3 and PF4 can be used to make that process easier. By hitting those keys repeatedly you can make the computer toggle or alternate between using AND and OR to combine what is specified on different lines and what is specified in different fields.

P-norm: This method is like Boolean. You should put in a query in much the same way. But the computer acts differently. It considers all books that have at least one of the words or names you have given in the query, and then ranks them based on how well it thinks they match what you asked for. Thus, it is more forgiving and can find relevant books that do not exactly match your query. On the other hand, it might find things you don't want. It uses special mathematical computations to rank what it finds, so the ones closer to the top of the list are likely to be the best.

Vector: Vector retrieval is very different from Boolean. The idea is to put in all words or names that you are pretty sure are in the books you want, and to leave the rest to the computer. It uses mathematical methods to treat your query as a list or vector of terms, and employs pattern matching techniques to find books that seems to have the best fit to your query. You don't need to worry about ANDs and ORs. The computer will try to rank all items that have any of the words you give, putting those with the best and most matches nearer the top of the list.

To make another search for a given query, manual feedback lets you add or delete words to the query you used before, and try again.

Vector with automatic feedback: This is very much like vector retrieval. The only difference is that there is another option besides manual feedback, which is available when you want to make another search. It is called automatic feedback.

It can be used if you marked one or more books with "y" to indicate relevance to the question. In automatic feedback, the computer looks at the descriptions of all books you marked as relevant to the question, and adds words and names from them to your query. It uses special mathematical computations to try to find more books like the resulting query (that has been enhanced with these new words and names). Thus, automatic feedback is a way for the computer to relieve you of the necessity to make a new query.

You have been assigned to use [filled in with METHOD1] as your first search method, and to use [filled in with METHOD2] as your second search method.

B.4.9 FINAL COMMENTS, SUMMARY

1. Ask for help from a person running the experiment if you are confused or want assistance regarding how the computer is behaving or how to get it to help you better.

2. On each of the 4 assigned questions, don't plan on spending more than 15 minutes per question unless you really want to.

3. Note that you will now do the following on the computer:

   (1) When asked, type in your group number: [filled in with GROUP]

   (2) When asked, type in your ID number: [filled in with ID]

(3) Read more about the experiment and about using the computer

(4) Answer questions for the initial questionnaire

(5) Learn about your search method 1, through the online tutorial. Your method 1 is: [filled in with METHOD1]

(6) Do at least one practice search to get used to method 1. For that, you should look for books by Gerard Salton or other books about "information retrieval". If you want more practice, go ahead only if you are willing to spend more than the 60-90 minutes agreed upon. In that case make up a question of interest to you, and search.

(7) When you finish each search and go on to another search, reply to the questions in the search questionnaire.

(8) Still using method 1, do two new searches, and fill in the questionnaire that follows each search, for the following two questions:

(9) When you are done searching with method 1, fill in the questionnaire about that method, and proceed to method 2.

(10) Repeat steps e-h for method 2, which for you is: [filled in with METHOD2], searching for the following two questions:

(11) When you are done searching with method 2, fill in the questionnaire about that method, and then proceed to fill in the final questionnaire.

(12) Go to the person running the experiment, and when you show that you are done, sign the receipt sheet and receive your payment.

Now, go ahead and work with the computer, following instructions given above. Thank you for helping with our experiment!

REFERENCES

[1] A. V. Aho and D. Lee. Storing a dynamic sparse table. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, Toronto, Canada, pages 55–60, Oct. 1986.

[2] Ken Arnold. Screen Updating and Cursor Movement Optimization: A Library Package. ULTRIX Supplementary Documents Vol. II:Programmer.

[3] R. A. Baeza-Yates and P.-A. Larson. Analysis of B+-trees with partial expansions. Technical Report CS-87-04, Department of Computer Science, University of Waterloo, Waterloo, Canada, 1987.

[4] N. J. Belkin and W. B. Croft. Retrieval techniques. In *Annual Review of Information Science and Technology*, 22:109–145, 1987.

[5] D. C. Blair and M. E. Maron. An evaluation of retrieval effectiveness for a full-text document-retrieval system. *Communications of the Association for Computing Machinery*, 28(3):289–299, March 1985.

[6] A. Bookstein. Probability and fuzzy-set applications to information retrieval. In *Annual Review of Information Science and Technology*, 20:117–152, 1985.

[7] M. D. Brain and A. L. Tharp. Near-perfect hashing of large word sets. *Software – Practice and Experience.*, 19(10):967–978, 1989.

[8] M. D. Brain and A. L. Tharp. Perfect hashing using sparse matrix packing. *Information Systems*, 15(3):281–290, 1990.

[9] C. Buckley. Implementation of the SMART information retrieval system. Technical Report 85-686, Cornell University, Department of Computer Science, May 1985.

[10] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Research and Development in Information Retrieval, Eighth Annual International ACM SIGIR Conference*, pages 97–110, Montreal, June 1985.

[11] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[12] N. Cercone, M. Krause, and J. Boates. Minimal and almost minimal perfect hash function search with application to natural language lexicon design. *Computers and Mathematics with Applications*, 9:215–231, 1983.

[13] C. C. Chang. The study of an ordered minimal perfect hashing scheme. *Communications of the Association for Computing Machinery*, 27:384–387, 1984.

[14] C. C. Chang. Letter oriented reciprocal hashing scheme. *Information Sciences*, 38:243–255, 1986.

[15] C. Y. Chen, C. C. Chang, and J. K. Jan. On the design of a machine-independent perfect hashing scheme. *The Computer Journal*, 33(3):277-281, 1990.

[16] Qi Fan Chen. *An object-oriented database system for efficient information retrieval applications.* PhD thesis, Virginia Tech Dept. of Computer Science, March 1992.

[17] R. J. Cichelli. Minimal perfect hash functions made simple. *Communications of the Association for Computing Machinery*, 23:17–19, 1980.

[18] P. A. Cochrane and Karen Markey. Catalog use studies – since the introduction of online interactive catalogs: Impact on design for subject access. *Library and Information Science Research*, 5(4):337–363, 1983.

[19] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[20] G. V. Cormack, R. N. S. Horspool, and M. Kaiserswerth. Practical perfect hashing. *The Computer Journal*, 28:54–58, 1985.

[21] A. M. Daoud. *Efficient Data Structures for Information Retrieval.* Dissertation proposal, Department of Computer Science, Virginia Polytechnic Institute& State University, 1990.

[22] R. F. Deutsher, P. G. Sorenson, and J. P. Tremblay. Key-to-address transformation techniques. *INFOR*, 16(1):1–34, 1978.

[23] M. Dodd, V. Y. Lum, and S. T. Yuen. Key-to-address transform techniques: A fundamental performance study on large existing formatted files. *Communications of the ACM*, 14(4):228–239, 1971.

[24] R. J. Enbody and H. C. Du. Dynamic hashing schemes. *ACM Computing Surveys*, 20:85–113, 1988.

[25] R. Fagin, J. Nievergelt, and H. R. Strong. Extendible hashing-a fast access method for dynamic files. *ACM Transactions on Office Information Systems*, 2(4):267–288, 1984.

[26] W. Feller. *An Introduction to probability theory and its applications.* Vol. 1. New York: John Wiley, 1968

[27] P. Flajolet. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica*, 20:345–369, 1983.

[28] E. A. Fox. Implementing SMART for minicomputers via relational processing with abstract data types. In *Joint Proceedings of SIGSMALL Symposium on Small Systems and SIGMOD Workshop on Small Data Base Systems, ACM SIGSMALL Newsletter, Volume 2, Number 2*, Orlando, pages 119–129, October 1981.

[29] E. A. Fox. *Extending the Boolean and Vector Space Models of Information Retrieval with P-Norm Queries and Multiple Concept Types.* PhD thesis, Cornell University Dept. of Computer Science, August 1983. Available from University Microfilms Int.

[30] E. A. Fox, Lenwood Heath, and Qi Fan Chen. An $O(nlogn)$ algorithm for finding minimal perfect hash functions. Technical Report TR-89-10, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, April 1989.

[31] E. A. Fox. Lexical relations: Enhancing effectiveness of information retrieval systems. *ACM SIGIR Forum*, 15(3):5–36, Winter 1980.

[32] E. A. Fox. Experimental systems for testing the applicability of advanced retrieval methods: Lessons learned from SMART and CODER, Seminar for Graduate Library School, University of Chicago, April 1987.

[33] E. A. Fox. Improved automatic analysis, indexing, storage and retrieval of corporate documentation and network information. Technical report, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, October 1987. Final Report on C.I.T. Grant INF-85-016.

[34] E. A. Fox, Qi Fan Chen, Amjad M. Daoud, and Lenwood S. Heath. Order preserving minimal perfect hash functions and information retrieval. In *Proc. SIGIR 90, 13th Int'l Conference on R&D in Information Retrieval*, pages 279–311, Brussels, Belgium, September 1990.

[35] E. A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. Practical minimal perfect hash functions for large databases. Technical Report TR 90-41, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, August 1990. Appeared in CACM, Jan. 1992.

[36] E. A. Fox, guest editor. Special section on interactive technologies. *Communications of the Association for Computing Machinery*, 32(7), 1989. Included are the introduction by Fox and articles by: Dixon, Frenkel, Lippman and Butera, Mackay and Davenport, Ripley, Stevens, Tinker, and Yu et al.

[37] M. L. Fredman and J. Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic and Discrete Methods*, 5:61–68, 1984.

[38] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with O(1) worst case access time. *Journal of the Association for Computing Machinery*, 31:538–544, 1984.

[39] G. H. Gonnet and P. Larson. External hashing with limited internal storage. *Journal of the Association for Computing Machinery*, 35:161–184, 1988.

[40] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*, Addison Wesley Publishers, Chatham, Kent., 1991.

[41] Lenwood S. Heath and Ramana R. Juvvadi. A lower bound for MOS perfect hashing. *Submitted to the Allerton Conference on Communication, Control, and Computing*, 1993.

[42] G. Jaeschke and G. Osterburg. Technical correspondence: On Cichelli's minimal perfect hash functions method. *Communications of the Association for Computing Machinery*, 23:728–729, 1980.

[43] T. Kahonen. *Content-Addressable Memories*, 2nd Edition, Springer Series in Information Sciences. Springer-Verlag, 1987.

[44] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, 1973.

[45] P. Larson. Dynamic hashing. *BIT*, 18(2):184–201, 1978.

[46] P. Larson. Linear hashing with separators – a dynamic hashing scheme achieving one-access retrieval. *ACM Transactions on Database Systems*, 13:366–388, 1988.

[47] P. Larson and A. Kajla. File organization: implementation of a method guaranteeing retrieval in one access. *Communications of the Association for Computing Machinery*, 27:670–677, 1984.

[48] P. Larson and M. V. Ramakrishna. External perfect hashing. In *Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data*, pages 190–199, 1985.

[49] Whay C. Lee and Edward A. Fox. Experimental Comparison of Schemes for Interpreting Boolean Queries *TR 88-27, VPI&SU*, 1988.

[50] T. G. Lewis, and W. D. Maurer. Hash table methods. *Computing Surveys*, 7(1), pages 5–19, 1975.

[51] W. Litwin. Trie hashing. *ACM SIGMOD Conference on Management of Data*, Ann Arbor, Michigan, pages 19–29, 1981.

[52] D. B. Lomet. Bounded index exponential hashing. *ACM Transactions on Database Systems*, 8(1):136–165, 1983.

[53] D. B. Lomet. Partial expansions for file organizations with an index. *ACM Transactions on Database Systems*, 12(1):65–84, 1987.

[54] V. Y. Lum. General performance analysis of key-to-address transformation methods using an abstract file concept. *Communications of the ACM*, 16(10):603–612, 1973.

[55] G. Martin. Spriral storage: incrementally augmentable hash addressed storage. *Theory of Computation Rep. 27*, University of Warwick, England, 1979.

[56] K. Mehlhorn. On the program size of perfect and universal hash functions. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, Chicago, Illinois, pages 170–175, Nov. 1982.

[57] Mendelson H. Analysis of extendible hashing. *IEEE Transactions on Software Engineering*, 8(6):611–619, 1982.

[58] J. K. Mullin. Spiral storage: Efficient dynamic hashing with constant performance. *The Computer Journal*, 28(3):330–334, 1985.

[59] S. Näher. *LEDA User Manual Version 2.0*, TR A17/90, Univ. des Saarlandes, Saarbrücken, 1990.

[60] T. Noreault, M. Koll, and M. J. McGill. Automatic ranked output from Boolean searches in SIRE. *Journal of the American Society for Information Science*, 28(6):333–339, November 1977.

[61] P. K. Pearson. Fast hashing of variable-length text strings. *Communications of the Association for Computing Machinery*, 33(6):677–680, June 1990.

[62] M. Regnier. *Performance Evaluation of Dynamic Hashing.* PhD thesis, Paris-Sud University, April 1983.

[63] M. Regnier. *Analysis of Grid File Algorithms.* BIT 25:335-357, 1985.

[64] T. J. Sager. A new method for generating minimal perfect hashing functions. Technical Report CSc-84-15, Department of Computer Science, University of Missouri-Rolla, 1984.

[65] T. J. Sager. A polynomial time generator for minimal perfect hash functions. *Communications of the Association for Computing Machinery*, 28:523–532, 1985.

[66] G. Salton. A new comparison between conventional indexing (MEDLARS) and text processing (SMART). *Journal of the American Society for Information Science*, 23(2):75–84, 1972.

[67] G. Salton. The SMART system 1961-1976: Experiments in dynamic document processing. In *Encyclopedia of Library and Information Science*, pages 1–36. 1980.

[68] G. Salton, C. Buckley, and E. Fox. Boolean query formulation with relevance feedback. Technical Report TR 83-539, Cornell University Dept. of Computer Science, Ithaca, NY, January 1983.

[69] G. Salton, E. Fox, and E. Voorhees. Advanced feedback methods in information retrieval. Technical Report TR 83-570, Cornell University Dept. of Computer Science, Ithaca, NY, August 1983.

[70] G. Salton, E. A. Fox, and E. Voorhees. A comparison of two methods for Boolean query relevance feedback. Technical Report TR 83-564, Cornell University Department of Computer Science, July 1983.

[71] G. Salton, E.A. Fox, and H. Wu. Extended Boolean information retrieval. *Communications of the Association for Computing Machinery*, 26(11):1022–1036, November 1983.

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

[73] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the Association for Computing Machinery*, 18(11):613–620, November 1975.

[74] Gerard Salton. *Automatic Text Processing.* Addison-Wesley, 1989.

[75] J.P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM J. Comput.*, 19(5):775–786, 1990.

[76] M. Scholl. New file organization based on dynamic hashing. *ACM Transactions on Database Systems*, 6(1):194–211, 1981.

[77] S. Shah. User Interface and Dictionary Support *Master's Report, Virginia Polytechnic Institute and State University*, 1988.

[78] A. F. Smeaton and C. J. van Rijsbergen. Information retrieval in an office filing facility and future work in project MINSRELl. *Information Processing & Management*, 22(2):135–149, 1986.

[79] R. Sprugnoli. Perfect hashing functions: a single probe retrieving method for static sets. 20:841–850, 1978.

[80] C. Stanell, and R. Thau. Information Retrieval on the Connection Machine:1 to 8192 gigabytes. *Information Processing and Management*, 27(4), 285-310, 1991.

[81] M. Tamminen. Order Preserving Extendible Hashing and Bucket Tries. BIT 21:419-435, 1981.

[82] J. P. Tremblay, R. F. Deutsher, and P. G. Sorenson. Distribution-dependent hashing functions and their characteristics. *ACM SIGMOD Proceedings of the International Conference on Management of Data*, pages 224–236, May 14-16, 1975.

[83] J. D. Ullman. A note on the efficiency of hashing functions. *Journal of the ACM*, 19(3):569–575, 1972.

[84] C.J. Van Rijsbergen. *Information Retrieval: Second Edition.* Butterworths, London, England, 1979.

[85] J.S. Vitter. Analysis of coalesced hashing. *JACM*, 30(2):231–258, 1983.

[86] Ellen M Voorhees. The efficiency of inverted index and cluster searches. In *Proc. 1986 ACM Conf. on Research and Development in Information Retrieval*, pages 164–174, Pisa, Italy, September 1986.

[87] V. G. Winters. Minimal perfect hashing in polynomial time. *BIT*, 30:235–244, 1990.

[88] A. C. Yao. Uniform hashing is optimal. *Journal of the ACM*, 32(3):687–693, 1985.

[89] A. C. Yao. A note on the analysis of extendible hashing. *Information Processing Letters*, 11(2):84–86, 1980.

VITA

Amjad M. Daoud was born in 1962 in Amman, Jordan. He received his Bachelor's degree in Electrical Engineering from Yarmouk University, Irbid in 1983. He was awarded the Master's degree in Electrical Engineering from University of Jordan, Amman in 1986. He worked as a maintenance manager in Muhanna Engineering House, Amman from 1983 to 1985. He worked as a vice president in International Trade and Investment, Inc., Amman, from 1985 till Fall 1986, when he joined the Ph.D. program in computer science at Virginia Tech, Blacksburg. From 1991 to 1992, he worked as a senior software engineer in Western Atlas International, Houston. Currently, he is the president of Tera Knowledge Systems, Inc., Houston.