74-20,230

RIVEST, Ronald Linn, 1947-
ANALYSIS OF ASSOCIATIVE RETRIEVAL ALGORITHMS.

Stanford University, Ph.D., 1974
Computer Science

ANALYSIS OF ASSOCIATIVE RETRIEVAL ALGORITHMS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

by

Ronald Linn Rivest

March 1974

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Robert W. Floyd
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David A. Klarner

Approved for the University Committee
on Graduate Studies:

Lincoln E. Moses
Dean of Graduate Studies

PREFACE

I would like to thank those people whose support, assistance, and encouragement made the writing of this thesis an enjoyable task. I would particularly like to give thanks to David Klarner for the many hours he spent with me discussing this work ; to Donald Knuth for his careful reading of the manuscript and suggestions regarding the proofs of theorems 1, 2 and 3 ; to Ronald Graham for his ABD construction method mentioned in § 3 ; to N. G. de Bruijn for his suggestions regarding a proof of the inequality (60); to Robert W. Floyd for his support, encouragement and detailed suggestions on the style and content of this thesis ; to Malcolm Newey for his hours of assistance , and to the Stanford Artificial Intelligence Laboratory for the use of their computer, in the preparation of this manuscript ; to the National Science Foundation, IRIA-Laboria, and my parents for their financial support ; and to Gail for keeping my spirits high throughout.

QUOTATIONS

Oh where, oh where, has my little dog gone?
Oh where, oh where can he be?
With his tail cut short, and his ears cut long,
Oh where, Oh where can he be?

[Nursery rhyme]

You must look where it is not, as well as where it is.

[Gnomologia - Adages and Proverbs.
by T. Fuller (1732)]

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

CHAPTER 1
INTRODUCTION

In this thesis we examine algorithms for "associatively" searching a direct-access file to determine their optimal form and achievable efficiency. This chapter presents an abstract model of the file and query specifications, and we analyze the search algorithms within this framework. Chapter 2 discusses the historical development of the "associative search" problem, and reviews previously published search algorithms. Chapters 3 and 4 examine partial-match search algorithms, and chapter 5 studies a best-match search algorithm.

An information retrieval system must consist of at least the following parts:

(i) a collection of information, called a file. An individual unit of this collection is usually called a record. If records may be added to or deleted from the file (that is, the file may be updated), the file is said to be dynamic, otherwise it is said to be static.

(ii) a storage or recording procedure by which to represent the file (in the abstract) on some physical medium for future reference. This operation we call the encoding of the file. The encoded version of the file must of course be distinguishable from the encoded versions of other files. The medium used is entirely arbitrary: for example, punched or printed cards, ferromagnetic cores, magnetic tape or disk, holograms or knotted ropes. There are clearly many possible encoding functions, even for a given storage medium. To choose the best one for an application is called the encoding or data structure problem.

(iii) a method by which to access and read (or decode) the encoded file. The access method depends only on the storage medium used, while the encoding function determines what interpretation should be given to the accessed data. The encoded version of the file will in general consist of the encodings of its constituent records, together with the encoding of some auxiliary information. If (the encoded version of) any particular item of information can be independently accessed with (approximately) unit cost, we say the file is stored on a direct-access storage device. Card files and magnetic disks are thus direct-access, whereas magnetic tapes are not. The access cost usually consists of two independent quantities: the physical access time needed to move a reading head or some other mechanical unit into position, and the transmission time required to actually read the desired data. The transmission time is proportional to the amount of information read, while the physical access time usually depends on the relative location of the last item of information read. Devices such as core memory have zero physical access time.

(iv) a user of the system. This person is assumed to have one or more queries (information requests) for the system. The response to a query is assumed to be a subset of the file - that is, the user expects some portion of the records of the file to be retrieved and presented to him. If the user presents his queries one at a time in an interactive fashion, we say that the retrieval system is being used on-line, otherwise we say that it is being used in batch mode. In this thesis we shall only consider on-line systems.

(v) a search algorithm. This is a procedure for accessing and reading part of the encoded file in order to produce a response to a user's query. It is of course dependent, but not entirely, on the choice of storage medium and encoding function. This algorithm may be performed either by a computer or some individual who can access the file (such as a librarian).

The above broad outline of an information retrieval system needs to be fleshed out with more detail in order to make precise the problem to be studied. We now present some formal definitions required for the rest of this thesis. These details restrict the model's generality somewhat, although it remains a good approximation to a large class of practical situations.

1.1. ATTRIBUTES, RECORDS, AND FILES

A record R is defined to be an ordered k-tuple $(r_1, r_2, \ldots, r_k)$ of values (that is, each record contains exactly k keys, or attributes). We will assume that the j-th key can have at most v_j values, for some finite v_j, $2 \le v_j < \infty$, so that $0 \le r_j < v_j$ for $1 \le j \le k$ and any record R. For simplicity we shall usually assume that all the v_j's are equal to a particular value v. In addition, we will usually consider only the case v = 2, since any other record type can easily be encoded as a binary string. Binary records are thus in a certain sense the most general case. In this situation each record is a binary string (or word) of length k. Let $\mathbf{R} = \{R_1, R_2, \ldots\}$ denote the set of all valid records, so that $|\mathbf{R}| = v_1 v_2 \cdots v_k$. We also reserve the notation $\mathbf{R}_k$ for the set of all binary words of length k. A file $\mathbf{F}$ is defined to be any nonempty subset of $\mathbf{R}$. We shall consistently use the letter n to denote $|\mathbf{F}|$, the number of records in the file being considered.

These conventions are not the most general possible. For example, in the model proposed by Hsiao and Harary [Hs70], a record is defined to be an arbitrary collection of (attribute, value) pairs rather than a complete list of values for a predetermined set of attributes. A study of the complexity of associative retrieval in this more general setting, however, would certainly require many additional assumptions about the file characteristics.

1.2. QUERIES

Let $\mathbf{Q}$ denote the set of queries the information retrieval system is designed to handle. For a given file $\mathbf{F}$, the proper response to a query $Q_i \in \mathbf{Q}$ is denoted by $Q_i(\mathbf{F})$ and is assumed to be a (perhaps null) subset of the records in $\mathbf{F}$.

The following sections give a framework within which to categorize query types, and describe the particular query types to be considered in this thesis.

1.2.1. INTERSECTION QUERIES.

The most common query type is certainly the <u>intersection query</u>, which is named after the defining characteristic of its response: a record in the file $\mathbf{F}$ is to be retrieved if and only if it is also in a predetermined subset $Q_i(\mathbf{R})$ of $\mathbf{R}$, so that

$$Q_i(\mathbf{F}) =_{\text{def}} \mathbf{F} \cap Q_i(\mathbf{R}) . \qquad (1)$$

The notation here is consistent since if $\mathbf{F}=\mathbf{R}$ then (1) implies $Q_i(\mathbf{F}) = Q_i(\mathbf{R})$. The sets $Q_i(\mathbf{R})$ completely characterize the functions $Q_i(\mathbf{F})$ for any file $\mathbf{F}$ by the above intersection formula. Intersection queries enjoy the property that whether some record $R \in \mathbf{F}$ is in $Q_i(\mathbf{F})$ does not depend upon the rest of the file (that is, upon $\mathbf{F}-\{R\}$), so that no "global" dependencies are involved. The class of intersection queries contains many important subclasses which we present in a hierarchy of increasing generality:

(1) Exact match queries: Each $Q_i(\mathbf{R})$ contains just a single record of $\mathbf{R}$. An exact match query thus asks whether a specific record is present in $\mathbf{F}$.

(2) Single-key queries: $Q_i(\mathbf{R})$ contains all records having a particular value for a specified attribute. For example, consider the query defined by

$$Q_i(\mathbf{R}) = \{R \epsilon \mathbf{R} \mid r_3 = 1\}. \tag{2}$$

(3) Partial match queries: A "partial match query Q_i with t keys specified" (for some $t \le k$) is represented by a record $R \epsilon \mathbf{R}$ with k-t keys replaced by the special symbol "$*$" (meaning "unspecified"). If $Q_i = (q_{i1}, q_{i2}, \ldots, q_{ik})$ then for t values of j we have $0 \le q_{ij} < v_j$ and for the other values of j we have q_{ij}="$*$". The set $Q_i(\mathbf{R})$ is the set of all records agreeing with Q_i in the specified positions. Thus,

$$Q_i(\mathbf{R}) = \{R \epsilon \mathbf{R} \mid (\forall j, 1 \le j \le k)[(q_{ij}=*) \vee (q_{ij}=r_j)]\}. \tag{3}$$

A sample application might be a cross-word puzzle dictionary, where a typical query could require finding all words of the form "B*T**R" (that is: BATHER, BATTER, BETTER, BETTOR, BITTER, BOTHER, BUTLER, BUTTER). We shall use $\mathbf{Q}_t$ throughout to denote the set of all partial match queries with t keys specified.

(4) Range queries: These are the same as partial match queries except that a range of desired values rather than just a single value may be specified for each attribute. For example, consider the query defined by

$$Q(\mathbf{R}) = \{ R \epsilon \mathbf{R} \mid (1 \le r_1 \le 3) \wedge (1 \le r_2 \le 4) \} \tag{4}$$

(5) Best-match queries with <u>restricted</u> <u>distance</u>: These require that a distance function d be defined on $\mathbf{R}$. Query Q_i will specify a record R_{c_i} and a distance λ_i, and have

$$Q_i(\mathbf{R}) = \{ R \epsilon \mathbf{R} \mid d(R,R_{c_i}) \le \lambda_i \} . \tag{5}$$

Query Q_i requests all records within distance λ_i of the record R_{c_i} to be retrieved. The distance function $d(R,R')$ is usually defined to be the number of attribute positions for which R and R′ have different values; this is the Hamming distance metric.

(6) Boolean queries: These are defined by Boolean functions of the attributes. For example, consider the query Q defined by

$$Q(\mathbf{R}) = \{ R \epsilon \mathbf{R} \mid ((r_1 = 0) \vee (r_2 = 1)) \wedge (r_3 \neq 3)\} \tag{6}$$

The class of Boolean queries is identical to the class of intersection queries, since one can construct a Boolean function which is true only for records in some given subset $Q_i(\mathbf{R})$ of $\mathbf{R}$ (the <u>characteristic</u> <u>function</u> of $Q_i(\mathbf{R})$).

Note that each intersection query requires total recall, that is, every record in $\mathbf{F}$ meeting the specification must be retrieved. Many practical applications have limitations on the number of records to be retrieved, so as not to burden the user with too much information if he has specified a query too loosely.

1.2.2. BEST-MATCH QUERIES.

A different query type is the pure best-match query. A pure best-match query Q_i requests the retrieval of all the nearest neighbors in $\mathbf{F}$ of the record $R_i \in \mathbf{R}$ using the Hamming distance metric d over $\mathbf{R}$. Performing a pure best-match search is equivalent to decoding the input word R into one or more of the "code words" in $\mathbf{F}$, using a maximum likelihood decoding rule (see Peterson [Pe72]). Thus we have

$$Q_i(\mathbf{F}) = \{R \in \mathbf{F} \mid \neg(\exists R' \in \mathbf{F})(d(R',R_i)<d(R,R_i)) \} \qquad (7)$$

1.2.3. QUERY TYPES TO BE CONSIDERED.

In this thesis we shall only consider partial-match and best-match queries. The justification for this choice is that these query types are quite common yet have not been "solved" in the sense of having known optimal search algorithms to answer them. In addition, these query types are the ones usually considered as the paradigms of "associative" queries. The simpler intersection query types seem to already have adequate algorithms for handling them. The more general situation where it is desired to handle any intersection query can be easily shown to require searching the entire file in almost all cases, if the file is encoded in a reasonably efficient manner. (Besides, it takes an average of $|\mathbf{R}|$ bits to specify which intersection query one is interested in, so that it would generally take longer to specify the query than to read the entire file!) A practical retrieval system must therefore be based on a restricted set of query types or detailed knowledge of the query statistics.

1.3. COMPLEXITY MEASURES

The difficulty of performing a particular task on a computer is usually measured in terms of the amount of time required. We shall measure the difficulty of performing an associative search by the amount of time it takes to perform that search.

Our measure is the "on-line" measure, that is, how much time it takes to answer a single query. This is the appropriate measure for interactive retrieval systems, where it is desired to minimize the user's waiting time. Many information retrieval systems can of course handle queries more efficiently in an "off-line" manner - that is, they can accumulate a number of queries until it becomes efficient to make a pass through the entire file answering all the queries at once, perhaps after having sorted the queries. The practicality of designing a retrieval system to operate "on-line" thus depends on the relative efficiency with which a single query can be answered. That is thus the study of this thesis.

When a file is stored on a secondary storage device such as a magnetic disk unit, the time taken to search for a particular set of items can be measured in terms of (i) the number of distinct accesses, or read commands, issued, and (ii) the amount of data transmitted from secondary storage to main storage. For most of our modeling we shall consider only the number of accesses. Thus, for the generalizations of hash-coding schemes discussed in §3, we count only the number of buckets accessed to answer the query.

Several measures are explicitly <u>not</u> considered here. The amount of storage space used to represent the file is not considered, except in §3.3 to show that using extra storage space may reduce the time taken to answer the query. The time required to update a particular file structure is also not considered - this can always be kept quite small for the data structures examined.

1.4. RESULTS TO BE PRESENTED

A brief exposition of the historical development of the subject is presented in §2.

In §3 generalized hash functions are studied as a means for answering partial match queries. A lower bound on their achievable performance is proved, and the class of optimal hash functions is precisely characterized. A new class of combinatorial designs, called associative block designs, is then introduced. When interpreted as hash functions, associative block designs are found to have excellent worst-case behavior while maintaining optimum average retrieval times. We also examine a method for utilizing storage redundancy (that is, we examine the achievable efficiency gains obtainable from storing each record in more than one place).

In §4 we study tries as a means for responding to partial match queries. "Tries" (plural of "trie") are a particular kind of tree in which branching decisions are made only according to the specific record being inserted or searched for, and not according to the results of comparisons between that record and others in the tree. Their average performance turns out to be nearly the same as the optimal hash functions of §3.

The results of §3 and §4 seem to support the following.

Conjecture: There is a positive constant c such that for all positive integers n, k, and t the average time required by any algorithm to answer a single partial match query $Q \in \mathcal{Q}_t$ must be at least

$$c\, n^{(k-t)/k},$$

where the average is taken over all queries $Q \in \mathcal{Q}_t$ and all files $\mathbf{F}$ of n k-bit records which are represented efficiently on a direct-access storage device. (That is, no more than snk bits of storage are used, for some small constant s.)

In §5 we again consider hash functions, this time as a means for answering best-match rather than partial-match queries. An algorithm due to Elias is proved to be optimal.

CHAPTER 2

HISTORICAL BACKGROUND

2.1. ORIGINS IN HARDWARE DESIGN

The class of associative search problems was first discussed by people interested in building associative memory devices. According to Slade [Sl64] the first associative memory design was proposed by Dudley Buck in 1955. Many other designs soon appeared in the literature (see Slade & MacMahon [Sl57] or Kiseda [Ki61]). These memories could perform arbitrary partial match searches, as well as searching for the maximum or minimum record stored, or finding all records between specified limits (interpreting a record as a number in radix v notation.)

The hoped-for technological breakthrough allowing large associative memories to be built cheaply has not (yet) occurred, however. Small associative memories (on the order of 10 words) have found applications - most notably in "paging boxes" for virtual memory systems (see [De70]). The only large associative processor available commercially is the STARAN S, introduced by

Goodyear Aerospace Corp. in 1971 [Ru72]. This $500,000 system has 512 256-bit words of associative memory (as well as 24K of random access core memory). An associative search for a partial match query takes 150 nanoseconds per bit specified. STARAN is cost-effective only for applications demanding very high data rate processing in real time - such as air traffic controlling. Minker [Mi72] has written an excellent survey of the development of associative processors up to the appearance of STARAN.

2. 2. EXACT MATCH ALGORITHMS

New algorithms for performing searches on a conventional computer with random-access memory were also being rapidly discovered at the same time. The first problem studied (since it is an extremely important practical problem) was the problem of searching for an exact match in a file of single-key records. Binary searching of an ordered file was first proposed by Mauchly [Ma46]. The use of binary trees for searching was invented in the early 1950's according to [Kn72], with published algorithms appearing around 1960 (see for example Windley [Wi60] - there were also many others).

Tries were first described about the same time by Rene de la Briandais [de59]. These are similar to binary trees, except that the i-th key or bit of a record R is used to make the i-th branching decision, instead of using a comparision between all of R and the record associated with the current tree node (treating them as binary numbers). Tries are roughly as efficient as binary trees for exact-match searches. We shall examine trie algorithms in chapter 4 for performing partial match searches.

Hash-coding (invented by Luhn around 1953 according to Knuth [Kn72, vol. 3]) seems to provide the best solution for many applications. Given b storage locations (with $b \geq n$) in which to store the records of the file, a hash function $h: \mathbf{R} \to \{1, 2, \ldots, b\}$ is used to compute the address $h(R_i)$ of the storage location at which to store each record R_i. The function h is chosen to be a suitably "random" function of the input - the goal is to have each record of the file assigned to a distinct storage location. Unfortunately this is nearly impossible to achieve (consider generalizations of the "birthday phenomenon" as in Knuth [Kn72,§6.4]), so a method must be used to handle "collisions" (two records hashing to the same address). Perhaps the simplest solution (separate chaining) maintains b distinct lists, or buckets. A record R_i is stored in bucket B_j (where $1 \leq j \leq b$) iff $h(R_i) = j$. Each bucket can now store an arbitrary number of records, so collisions are no longer a problem. To determine if an arbitrary record $R_i \epsilon \mathbf{R}$ is in the file one need merely examine the contents of bucket

$B_{h(R_i)}$ to see if it is present there. Since the expected number of records present in each bucket is small, very little work need be done. Chaining can be implemented easily with simple linear linked list techniques (see [Kn72,§6. 4]).

2. 3. SINGLE-KEY SEARCH ALGORITHMS

The next problem to be considered was that of single-key retrieval for records having more than one key (that is, $k > 1$). This is often called the problem of "retrieval on secondary keys". L. R. Johnson [Jo61] proposed the use of k distinct hash functions h_i and k sets of buckets $B^i{}_j$ - for $1 \le i \le k$ and $1 \le j \le b$. Record R_m is stored in k buckets - bucket $B^i{}_{h_i(r_{mi})}$ for $1 \le i \le k$. This is an efficient solution, although storage and updating time will grow with k. Prywes and Gray suggested a similar solution - called Multilist - in which each attribute-value is assigned a unique bucket through the use of indices (search trees) instead of hash functions to compute bucket addresses (see [Gr59], [Pr63]). Davis and Lin [Da65] describe another variant in which list techniques are replaced by compact storage of the record addresses relevant to each bucket. The above class of methods are often called <u>inverted list</u> techniques since a separate list is maintained of all the records having a particular attribute value, thus mapping attribute to records rather than the reverse as in an ordinary file.

2.4. PARTIAL-MATCH SEARCH ALGORITHMS

Inverted list techniques, while adequate for single-key retrieval of multiple key records, do not work well for partial match queries unless t, the number of keys specified in the query, is small. This is because the response to a query is the intersection of t buckets of the inverted list system. Thus the amount of work required to perform this intersection grows with the number of keys given, while the expected number of records satisfying the query decreases! One would expect a "reasonable" algorithm to do an amount of work that decreases with $E(|Q_i(\mathbf{F})|)$, the expected size of the answer. One might even hope for an amount of work proportional to the number of records in the answer. Unfortunately, no such "linear" algorithms have been discovered that do not use exorbitant amounts of storage. The algorithms presented in chapters 3 and 4, while non-linear, easily outperform inverted list techniques. These algorithms do an amount of work that decreases approximately exponentially with t, the number of bits specified in the query. When t=0, the whole file must of course be searched, and when t=k unit work must be done. In between, log(work) decreases linearly with t.

J. A. Feldman's and P. D. Rovner's system LEAP [Fe69] allows complete generality in specifying a partial match query. LEAP handles only 3-key records, however, so that there are at most eight query types. This is not as restrictive as might seem at first, since any kind of data can in fact be expressed as a collection of "triples": (attributename, objectname, value). While arbitrary Boolean queries are easily programmed, the theoretical retrieval efficiency is equivalent to an inverted list system.

Several authors have published algorithms for the partial match problem different from the inverted list technique. One approach is to create a very large number of tiny buckets so that the response to each query can always be constructed as the union of some of the buckets, instead of an intersection. Wong and Chiang [Wo71] discuss this approach in detail. Note, however, that the requisite number of buckets is at least $|\mathbf{R}|$ if the system must handle all partial match queries (since exact match queries are a subset of the partial match queries)! Having such a large number of buckets (most of them empty if $n << |\mathbf{R}|$, as is usual) is not practical. A large number of authors (C. T. Abraham, S. P. Ghosh, D. K. Ray-Chaudhuri, G. G. Koch, David K. Chow, and R. C. Bose - see references for titles and dates) have therefore considered the case where t is not allowed to exceed some fixed small value t' (for example, $t' = 3$). It is easy to see that the number of buckets required is now at most

$$\sum_{1 \le i_1 \le \cdots \le i_{t'} \le k} (v_{i_1} v_{i_2} \cdots v_{i_{t'}})$$

$$\le \quad C(k,t')\,(\max v_i)^{t'} . \tag{8}$$

(Here $C(k,t')$ denotes the binomial coefficient "k choose t'".) This is achieved by reserving a bucket for the response to each query with the maximal number t' of keys given; the response to other queries is then the union of existing buckets. Note that each record is now stored in $C(k,t')$ buckets, however! The papers referred to show how to reduce the number of buckets used and record redundancy somewhat, by the clever use of combinatorial designs, but another approach is really needed to escape combinatorial explosion.

The first efficient solution to an associative retrieval problem is described by Richard A. Gustafson in his Ph. D. thesis [Gu69,Gu71]. Gustafson assumes that each record R_i is an unordered list $\{r_{i1},r_{i2},\ldots\}$ of at most k' attribute values (these might be keywords, where the records represent documents). Let w be chosen so that $C(w,k')$ is a reasonable number of buckets to have in the system, and let a hash function h map attribute values into the range $\{1, 2, \ldots, w\}$. Each bucket is associated with a unique w-bit word having exactly k' ones in it, and each record R is stored in the bucket associated with the word having ones in positions $h(r_1)$, $h(r_2)$, $\ldots$, $h(r_{k'})$. (If these are not all distinct positions, extra ones are added randomly until there there are exactly k' ones.) A query specifying attributes a_1, a_2, $\ldots$, a_t (with $t \le k'$) need only examine the $C(w-t,k'-t)$ buckets associated with words having ones in positions $h(a_1)$, $h(a_2)$,

..., $h(a_t)$. The amount of work thus decreases rapidly with t. Note that the response to the query is not formed merely by taking the union of the relevant buckets, since records not satisfying the query may also be stored in these buckets. We are guaranteed, however, that all the relevant records are stored in the examined buckets. In essence Gustafson reduces the number of record types by creating w attribute classes, a record being filed according to which attribute classes describe it. His method has the following desirable properties:

(a) each record is stored in only one bucket (so updating is easy), and

(b) the expected amount of work required to answer a query decreases approximately exponentially with the number of attributes specified.

His definition of a record differs from the one used here, however, so that the allowable queries in his system correspond to a proper subset of our partial match queries - those having no zeros specified. (Convert each of his records into a very long bitstring having ones in exactly k' places - each bit position corresponding to a permissible keyword in the system.)

Terry Welch, in his Ph.D. thesis [We71], studies the achievable performance of file structures which include directories. His main result is that the size of the directory is the critical component of such systems. He briefly considers directoryless files, for which he derives a lower bound on the required average time to perform a partial match search with hash-coding methods that is much smaller than the precise answer given in §3. He also presents Elias' algorithm for handling best-match queries without proof of optimality.

CHAPTER 3

HASHING ALGORITHMS FOR PARTIAL MATCH QUERIES

The problem is: given a universe $\mathbf{R}$ of possible records, and a number b of lists (buckets) desired in a filing scheme, construct a good hash function $h: \mathbf{R} \to \{1, 2, \ldots, b\}$ so that partial match queries can be answered efficiently (either on the average or in the worst case). A record $R \in \mathbf{F}$ is stored in bucket B_j iff $h(R)=j$, with collisions handled by separate chaining. In a notation analogous to that used for the responses to intersection queries, we define

$$B_j(\mathbf{R},h) = \{R \in \mathbf{R} \mid h(R)=j\}, \tag{9}$$

$$B_j(\mathbf{F},h) = B_j(\mathbf{R},h) \cap \mathbf{F}, \text{ for any } \mathbf{F} \subseteq \mathbf{R}. \tag{10}$$

When a particular hash function h is understood from context, we shall usually omit it from the argument list of B_j. The set $B_j(\mathbf{R},h)$ we call the <u>extent</u>, and $B_j(\mathbf{F},h)$ the <u>contents</u>, of bucket j. This notation is consistent since (10) is an identity when $\mathbf{F} = \mathbf{R}$. We shall often denote the extent $B_j(\mathbf{R})$ of a bucket by the notation B_j, when no confusion can arise. The sets $B_1, \ldots, B_b$ form a partition of $\mathbf{R}$ since they are disjoint sets whose union is $\mathbf{R}$. A hash function is said to be <u>balanced</u> if $|B_j| = |\mathbf{R}|/b$ for $1 \le j \le b$. To answer a query $Q_i \in \mathbf{Q}$, the contents of the buckets whose indices range over

$$h(Q_i) =_{\text{def}} \{ j \mid (B_j \cap Q_i(\mathbf{R})) \neq \text{null} \} \tag{11}$$

or equivalently,

$$h(Q_i) = \cup_{R \in Q_i(\mathbf{R})} \{ h(R) \} , \qquad (12)$$

must be examined to find the response to Q_i (that is, $Q_i(\mathbf{F})$). Here we make the natural extension of h onto the domain $\mathbf{Q}$.

Here we present the basic retrieval algorithm:

```
procedure SEARCH({B_1,B_2,...,B_b},h,Q);
comment SEARCH finds the response to query Q∈Q, given that the file F⊆R is
stored in the buckets B_1,...,B_b, using the hash function h.
begin integer i; record R;
    for each i ∈ h(Q) do
        for each R ∈ B_i(F,h) do
            if R ∈ Q_i(R) then print( R );
end SEARCH;
```

The difficulty of computing the set $h(Q)$ depends very much on the nature of the hash function h. It is conceivable that for some pseudo-random hash-functions it is more time-consuming to determine whether $j \in h(Q)$ than it is to read $B_j(\mathbf{F})$ from the secondary storage device! (For some hash functions the relation (12) is the only way to compute $h(Q)$.) Such hash functions are of course useless, since one would always skip the computation of h(Q) and read the entire file in order to answer a query. We will restrict our attention to hash functions h for which the time required to compute h(Q) is always negligible in comparison with the time required to read the required bucket contents.

We shall use the following notation for the average and worst case costs of using various hash functions to answer a partial match query with t keys specified:

$$\alpha(h,t) =_{def} (\Sigma_{Q \in \mathbf{Q}_t} |h(Q)|) / |\mathbf{Q}_t| \qquad (13)$$

$$\beta(h,t) =_{def} \max_{Q \in \mathbf{Q}_t} |h(Q)| \qquad (14)$$

These are the average and worst-case number of buckets examined by SEARCH to answer a query $Q \in \mathbf{Q}_t$ as a function of the hash function h used. If the second argument is omitted from the function α, we assume that the average is taken over all queries in $\mathbf{Q}$:

$$\alpha(h) =_{def} (\Sigma_{Q \in \mathbf{Q}} |h(Q)|) / |\mathbf{Q}|. \qquad (15)$$

We shall also use the following notation for the best possible cost of any hash function:

$$A(k,w,t,v) =_{def} \min_h \alpha(h,t) \qquad (16)$$

where h ranges over all balanced hash functions mapping $\mathbf{R} \rightarrow \{1,\ldots,b\}$. This is the minimum possible average number of buckets examined by SEARCH to answer a partial match query $Q \epsilon \mathbf{Q}_t$, over all balanced hash functions h. We assume that the file contains k-key records, and that $b = 2^w$ buckets are used. All the v_i's are assumed to be equal to the v given, with the convention that if v is omitted, $v = 2$ is assumed.

Note that the number of buckets examined in either case does not depend at all upon the particular file $\mathbf{F}$ being searched, but only upon the particular hash function h being used.

3. 1. CONSIDERATION OF THE AVERAGE SEARCH TIME

For some applications it is easy to construct an efficient hash function. For example, suppose we wish to construct a "crossword puzzle" dictionary for six-letter English words. Let $b = 2^{12}$ be the number of buckets used. Given a word (for example, "SEARCH") we can construct a 12-bit bucket address by forming the concatenation

$$h(\text{"SEARCH"})=g(\text{"S"})\ g(\text{"E"})\ g(\text{"A"})\ g(\text{"R"})\ g(\text{"C"})\ g(\text{"H"}) \tag{17}$$

of six two-bit values; here g is an auxiliary hash function mapping the alphabet into two-bit values. For a query with t letters given we have

$$\alpha(h,t) = \beta(h,t) = 2^{12-2t}. \tag{18}$$

This approach is clearly feasible as long as $b \geq 2^k$, since one or more bits of the bucket address can be associated with each attribute position. A similar technique has been proposed by M. Arisawa [Ar71] in which the i-th key determines, via an auxiliary hash function, the residue class of the bucket address modulo the i-th prime (see also [Ni72]).

3.1.1. THE OPTIMAL SHAPE OF A BUCKET FOR BINARY RECORDS.

When $k > w$, where $b = 2^w$, it is not immediately clear what should be done. Terry Welch in [We71] suggests, but does not prove, that extracting the first w keys of each record for a bucket address may be optimal. His conjecture is correct for binary records; in this section we give a proof of this fact.

We will say that two buckets B and B′ have the same "shape" if there exists a permutation of the bit positions, followed by the complementation of bits in certain positions, which transforms every record of B into a record of B′. In other words, B and B′ have the same shape if there is an automorphism of $\mathbf{R}$ which carries B into B′.

We introduce the notation $\Phi(B)$ to denote the number of queries in $\mathbf{Q}$ which examine a bucket B. More precisely,

$$\Phi(B) =_{def} |\{ Q \in \mathbf{Q} \mid Q(\mathbf{R}) \cap B \neq \text{null} \}| . \qquad (19)$$

Let $\pi(s,k)$ denote the minimum possible number of queries in $\mathbf{Q}$ which examine any bucket B with an s element extent chosen from the record space $\mathbf{R}_k$. More precisely,

$$\pi(s,k) =_{def} \min_B \ \Phi(B), \tag{20}$$

where the minimum is taken over all s-element subsets B of $\mathbf{R}_k$. Let $\pi(s,k) = \infty$ if $s > 2^k$, and let $\pi(1,0)=1$, $\pi(0,0)=0$. The characteristics of an s-element bucket B chosen from $\mathbf{R}_k$ which achieves the minimum $\Phi(B) = \pi(s,k)$ will be those of an optimal bucket. We will investigate individual s-element buckets to find what characteristics they must have in order be optimal. Then we may construct an optimal balanced hash function by selecting **b** optimal buckets which cover $\mathbf{R}_k$ (if possible), since

$$\begin{aligned}\alpha(h) &= (\Sigma_{Q_i \in \mathbf{Q}} \ |h(Q_i)| \ ) \ / \ |\mathbf{Q}| \\ &= (\text{number of pairs } B_j,Q_i \text{ such that } B_j \cap Q_i \neq \emptyset) \ / \ |\mathbf{Q}| \\ &= (\Sigma_{1 \le j \le b} \ \Phi(B_j) \ ) \ / \ |\mathbf{Q}| \\ &\ge b \ \pi(|\mathbf{R}_k|/b,k)/|\mathbf{Q}| \end{aligned} \tag{21}$$

We require that the hash function be balanced in order to avoid the degenerate solution having all the records in a single bucket (costing one bucket per search). If we also counted the cost of reading each record, by arguments of symmetry we would find a balanced hash function to be optimal, once the expected cost of reading a bucket becomes commensurate with accessing it. Furthermore, the physical constraints imposed by a particular storage device, such as magnetic core, sometimes make a balanced hash function the only reasonable model.

Theorem 1. Let $s = 2^u$ for some integer u, $1 \le u \le k$, and let B be an s-element subset of $\mathbf{R}_k$. Then $\Phi(B) = \pi(s,k)$ if and only if B is a "subcube" of $\mathbf{R}_k$; that is, if and only if B is a cartesian product

$$B = D_1 \times D_2 \times \ldots \times D_k \tag{22}$$

where each D_i is a nonempty subset of $\{0,1\}$.

Proof: Let T(s,k) denote the s-subset of $\mathbf{R}_k$ consisting of those records which have binary value less than s when interpreted as binary numbers. In other words, T(s,k) consists of the s "tiniest" k-bit numbers (for those who like mnemonics). We will first prove that T(s,k) is an optimal bucket for any s, not just s a power of two. This will imply the "if" part of our theorem, since T(s,k) is a subcube of $\mathbf{R}_k$ whenever s is a power of two. We will then examine the proof a little more closely to derive the "only if" part of the theorem.

We first need to derive $\Phi(T(s,k))$. We will do this by defining an auxiliary function $\lambda(x)$, then proving that $\Phi(T(s,k))=\lambda(x)$ if x is the record in T(s,k) with largest binary value. (The binary value of x will of course be s-1.) Define λ by the following recurrence relations:

$$\lambda(\text{null}) = 1, \tag{23}$$

$$\lambda(0x) = 2\,\lambda(x), \text{ and} \tag{24}$$

$$\lambda(1x) = 2\,\lambda(1^{|x|}) + \lambda(x). \tag{25}$$

Here we treat λ's argument as a string of 0's and 1's, and define λ in terms of shorter strings. We denote the length of x by $|x|$, so that the notation $1^{|x|}$ represents a string of $|x|$ 1's. The notation $0x$ (or $1x$) stands for the concatenation of 0 (or 1) and the string x. Let $<x>$ denote the binary value of the string x.

Lemma. $\Phi(T(s,k)) = \lambda(x)$ if x is the record in $T(s,k)$ with the largest binary value.

Proof: By induction on $|x| = k$. It is clearly true for $k = 1$, since $\lambda(0) = 2$ and $\lambda(1) = 3$ are correct. It is true for $x = 0x'$ by (24) since all of the records in $T(s,k)$ will have a 0 in first position. Thus any query which examines $T(s,k-1) = T(<x'>+1,|x'|)$ may be preceded by either a "0" or a "*" to obtain a query which examines $T(s,k)$. On the other hand, if $x = 1x'$ then $T(s,k)$ contains two different kinds of records: 2^{k-1} will begin with a zero and finish up in all possible ways, and the remaining $s - 2^{k-1}$ will begin with a 1 and finish up identically to the records in $T(s-2^{k-1},k-1) = T(<x'>+1,k-1)$. The first term of (25) thus counts all queries beginning with a "0" or a "*", while the second term counts all queries beginning with a "1". This completes the proof of the lemma.

The rightmost column of the following table gives the value of $\lambda(x)$ for some small strings x. (The rest of the table shall be used later.)

	$\lambda_t(x)$				$\lambda(x)$
x	t=0	1	2	3	
null	1				1
0	1	1			2
1	1	2			3
00	1	2	1		4
01	1	3	2		6
10	1	4	3		8
11	1	4	4		9
000	1	3	3	1	8
001	1	4	5	2	12
010	1	5	7	3	16
011	1	5	8	4	18
100	1	6	10	5	22
101	1	6	11	6	24
110	1	6	12	7	26
111	1	6	12	7	27

Figure 1. Table of values for $\lambda(x)$ and $\lambda_t(x)$

We must now show that $\lambda(x) = \pi(s,k)$ (again, assuming that x is the record of $T(s,k)$ with largest binary value). To do this, we must first prove the recurrence

$$\pi(s,k) = \min \left[\pi(\max(f_0,f_1),k-1) + \pi(f_0,k-1) + \pi(f_1,k-1)\right], \qquad (26)$$

where the minimum is taken over all pairs of nonnegative integers f_0, f_1 such that $f_0 + f_1 = s$. Suppose a bucket B containing s records has f_0 records which begin with a zero and f_1 records which begin with a 1. Then $\pi(f_0,k) + \pi(f_1,k)$ is the number of queries which examine B which begin with a digit. The number of queries which examine B which begin with a "*" is clearly at least $\pi(\max(f_0,f_1),k)$. Furthermore, it can be held to this value by requiring that the number of distinct k-1 tuples ocurring in positions 2 through k of the records of B be held to this number. Thus, if $f_0>f_1$ and $1y \in B$ (for some string y, $|y|=k-1$), we would require that $0y \in B$ as well. This proves (26).

We will need the following two lemmas.

Lemma.

$\lambda(x1) = 3\,\lambda(x)$, for any string x of 0's and 1's. (27)

Proof: If $\lambda(x) = \Phi(T(s,k))$, then $\lambda(x1) = \Phi(T(2s,k+1))$. For each query q counted in $\Phi(T(s,k))$, the queries q0, q1 and q* are counted in $\Phi(T(2s,k+1))$, since $x \in T(s,k)$ implies that x0 and x1 are both in $T(2s,k+1)$.

Let $p(x)$ denote 2^j, where j is the number of zeros in the string x.

Lemma.

$\lambda(x) - \lambda(x-1) = p(x)$, for any string x of 0's and 1's, $\langle x\rangle \neq 0$. (28)

Here $x - 1$ denotes the string y of length $|x|$ such that $\langle y\rangle = \langle x\rangle - 1$.

Proof: By induction on $|x|$. By inspection of Figure (1) it is true for $|x| \leq 3$. For larger values of $|x|$ it will be true from the inductive hypothesis and the definition of λ when x and $x - 1$ begin with the same digit. The exceptional case occurs when $x = 1\ 0^{k-1}, x - 1 = 0\ 1^{k-1}$. But here we have

$$\lambda(x) - \lambda(x-1) = (2 \cdot 3^{k-1} + 2^{k-1}) - 2 \cdot 3^{k-1} = 2^{k-1}, \tag{29}$$

since $\lambda(0^{k-1}) = 2^{k-1}$. This proves the lemma.

To prove $\lambda(x) = \pi(s,k)$, it now suffices by (26) to prove that

$$\lambda(x) \leq 2\ \lambda(y) + \lambda(z), \tag{30}$$

for any pairs of strings y, z such that $|y| = |z| = |x| - 1$, $<y> + <z> + 1 = <x>$, and $<y> > <z>$, since $\lambda(0) = \pi(1,1) = 2$ and $\lambda(1) = \pi(2,1) = 3$. The proof is by induction on $|x| = k$, although it goes from the right end of x to the left, instead of the other way around.

The proof of (30) now proceeds by a four-part case analysis, depending on the right-most digits of y and z. It will also be an inductive proof, so we assume that (30) holds for all strings x' shorter than the current x. The last three cases will have two subparts as we reduce (30) in two different ways using the lemmas. In each case at least one of the two reductions must be true.

Case 1: $y=y'1$, $z=z'1$, and $x=x'1$.

Here (30) is implied directly by the inductive hypothesis, since it is equivalent in this case to:

$$3\ \lambda(x') \leq 6\ \lambda(y') + 3\ \lambda(z'), \tag{31}$$

with $<x'> = <y'> + <z'> + 1$, and $y' \geq z'$.

Case 2: $y=y'0$, $z=z'1$, and $x=x'0$.

Here we reduce (30) to the two inequalities:

$$3\,\lambda(x'-1) + 2\,p(x') \le 6\,\lambda(y'-1) + 4\,p(y') + 3\,\lambda(z'), \text{ and} \tag{32}$$

$$3\,\lambda(x') - p(x') \le 6\,\lambda(y') - 2\,p(y') + 3\,\lambda(z'). \tag{33}$$

Since $\langle x'-1\rangle = \langle y'-1\rangle + \langle z'\rangle + 1$ and $\langle x'\rangle = \langle y'\rangle + \langle z'\rangle + 1$, we may reduce these two equalities using the inductive hypothesis to the statements $p(x') \le 2\,p(y')$ and $p(x') \ge 2\,p(y')$, at least one of which must be true.

Case 3: $y=y'1$, $z=z'0$, and $x=x'0$.

In this case we get (30) reducing to:

$$3\,\lambda(x'-1) + 2p(x') \le 6\,\lambda(y') + 3\,\lambda(z'-1) + 2\,p(z'), \text{ and} \tag{34}$$

$$3\,\lambda(x') - p(x') \le 3\,\lambda(y') + 3\,\lambda(z') - p(z'). \tag{35}$$

Using the inductive hypothesis we get that (34) and (35) are equivalent to $p(x') \le p(z')$ and $p(x') \ge p(z')$, at least one of which must be true. There is the exceptional case when $\langle z'\rangle=0$, where (34) is sufficient (if we define $\lambda(z'-1)$ to be zero), since $x'-1=0y'$ and $p(z') \ge p(x')$.

Case 4: $y=y'0$, $z=z'0$, and $x = x'1$.

In this case (30) can be reduced to the two inequalities:

$$3\,\lambda(x') \le 6\,\lambda(y') - 2\,p(y') + 3\,\lambda(z'-1) + 2\,p(z'), \text{ and} \tag{36}$$

$$3\,\lambda(x') \le 6\,\lambda(y'-1) + 4\,p(y') + 3\,\lambda(z') + p(z'). \tag{37}$$

Since $\langle x' \rangle = \langle y' \rangle + \langle z'-1 \rangle + 1 = \langle y'-1 \rangle + \langle z' \rangle + 1$, we invoke the inductive hypothesis twice to obtain $p(y') \leq p(z')$ and $p(y') \geq p(z')/4$. These can not both hold simultaneously so one of these reductions will suffice to prove (30) in this case. This argument must be amended to consider two exceptional conditions: when $\langle z' \rangle = 0$, so that $3\,\lambda(z'-1)$ is undefined in (36), and when $y'=z'$ so that the condition $y'-1 \geq z'$ does not hold for the inductive hypothesis used on (37). In the first condition we have $0y'=x'$, and that (30) is equivalent to

$$3\,\lambda(x') \leq 6\,\lambda(y') - 2p(y') + 2p(z'). \tag{38}$$

The other exception to this argument occurs when $\langle y' \rangle = \langle z' \rangle \neq 0$, so that the inductive hypothesis can not be invoked to reduce (37). Here though we have $p(y')=p(z')$, so the proof follows from (36).

This completes the proof of the "if" portion of the theorem, since for any bucket B such that $|B|=s$, we will have $\Phi(B) = \Phi(T(s,k))$ if B has the same "shape" as $T(s,k)$.

The "only if" part of the theorem shall again be proved by an induction on $|x|=k$, using the previous analysis as a foundation. What needs to be proved is that if s is a power of two then (30) will hold with equality only if $y = z$. Here we have $s = 2^u$, so that $x = 0^{k-u}\,1^u$. It is clear that equality does obtain when $y = z$. It needs then to be shown that (30) holds with equality only when $y = z$. This is equivalent to

$$3\,\lambda(0\,1^{u-1}) < 2\,\lambda(i + 0\,1^{u-1}) + \lambda(-i + 0\,1^{u-1}), \tag{39}$$

for all i, $1 \le i \le 2^{u-1}-1$. This reduces directly to

$$\lambda(1^{u-1}) < \lambda(i-1) + \lambda(1^{u-1} - i). \tag{40}$$

It is in fact easier to prove the general statement:

$$\lambda(x) < \lambda(y) + \lambda(z), \tag{41}$$

for $|x|=|y|=|z|=k$, $\langle y\rangle + \langle z\rangle + 1 = \langle x\rangle$, and $\langle y\rangle > \langle z\rangle$. We shall prove this by an induction on k. If $x=0x'$, then $y=0y'$ and $z=0z'$, so that the theorem follows directly from (24). Similarly, if $x=1x'$ and $y=1y'$ then we may turn both of these initial 1's into 0's and lose an equal amount from each side of (41). The remaining case is when $x=1x'$, $y=0y'$, and $z=0z'$. Divide y and z into two pieces each so that $\langle y_1\rangle + \langle y_2\rangle + 1 = \langle y\rangle$, $\langle z_1\rangle + \langle z_2\rangle = z$, and $|y_1| = |y_2| = |z_1| = |z_2| = k$, and furthermore such that $\langle y_1\rangle + \langle z_1\rangle + 1 = \langle 0\,1^{k}-1\rangle$, so that $\langle y_2\rangle + \langle z_2\rangle + 1 = \langle x'\rangle$ as a consequence. This can be done in such a fashion that the k-bit representations of y_1, y_2, z_1, and z_2 all begin with a 0. (There is a trivial exceptional case when $\langle y\rangle = \langle z\rangle = 0$.)

Then we can derive

$$\lambda(0\,1^{k-1}) < \lambda(y_1) + \lambda(z_1), \text{ and} \tag{42}$$

$$\lambda(x') < \lambda(y_2) + \lambda(z_2), \text{ yielding} \tag{43}$$

$$\lambda(x) < \lambda(y) + \lambda(z), \text{ immediately.} \tag{44}$$

This proves the "only if" portion of the theorem.

Our theorem implies the following corollary.

Corollary: For binary records, the hashing function which extracts w of the key-bits to use as a bucket-address, where $w = \log_2(b)$, minimizes the expected number of buckets examined over all balanced hash functions, assuming that each partial match query is equally likely.

Note that there may be other optimal hash functions with respect to the expected search time. In fact, we shall examine others in the following sections.

The preceding theorem gives a good characterization of the bucket shapes which will minimize $\Phi(B)$, the number of queries in $\mathbf{Q}$ which will examine B. We shall next prove that the same shapes are optimal when the queries are restricted to $\mathbf{Q}_t$, for some t, $0 \le t \le k$. The "only if" portion of the preceding theorem shall not again be proved, however.

Let $\Phi_t(B)$, and $\pi_t(s,k)$ denote the functions Φ and $\pi(s,k)$ restricted to counting queries in $\mathbf{Q}_t$ rather than $\mathbf{Q}$. The following theorem makes the relevant assertion.

Theorem 2. . Let $s = 2^u$ for some u, $1 \le u \le k$, and let B be an s-element subset of $\mathbf{R}_k$. Then $\Phi_t(B) = \pi_t(s,k)$ if B is a "subcube" of $\mathbf{R}_k$; that is, if B is a cartesian product

$$B = D_1 \times D_2 \times \ldots \times D_k \tag{45}$$

where each D_i is a nonempty subset of $\{0,1\}$.

Proof: This proof is almost identical to the preceding one, so only the changes necessary shall be indicated.

Let $\lambda_t(x)$ be defined by the recurrence relations (here $|x|=k$):

$$\lambda_0(x) =_{def} 1, \quad \text{for all } x. \tag{46}$$

$$\lambda_t(\text{null}) =_{def} 0, \quad \text{for } t \geq 1. \tag{47}$$

$$\lambda_t(0x) =_{def} \lambda_t(1^k) + \lambda_{t-1}(x), \text{ for } t\geq 1 \text{ and all } x. \tag{48}$$

$$\lambda_t(1x) =_{def} \lambda_t(1^k) + \lambda_{t-1}(1^k) + \lambda_{t-1}(x), \text{ for } t\geq 1 \text{ and all } x. \tag{49}$$

The values of $\lambda_t(x)$ for some small values of t and x are displayed in the table. The following lemma we state without proof, as it is essentially identical to the proof of the corresponding lemma of the preceding theorem.

Lemma. $\Phi_t(T(s,k))=\lambda_t(x)$ if x is the record in T(s,k) with largest binary value.

Again, the following identity can be proved in a manner similar to the proof of its corresponding identity in the preceding theorem.

$$\pi_t(s,k) = \min [\, \pi_t(\max(f_0,f_1),k) + \pi_{t-1}(f_0,k) + \pi_{t-1}(f_1,k)], \tag{50}$$

where the minimum is taken over all pairs of nonnegative integers f_0, f_1 such that $f_0 + f_1 = s$.

To prove that $\lambda_t(x) = \pi_t(s,k)$, where x is the largest record in T(s,k), by (50) it is only necessary to show that

$$\lambda_t(x) \leq \lambda_t(y) + \lambda_{t-1}(y) + \lambda_{t-1}(z), \tag{51}$$

where $|y|=|z|=|x|-1=k-1$, $<y> + <z> + 1 = <x>$, and $<y> > <z>$, since $\lambda_t(x)=\pi(s,k)$ for x=0 and x=1. This proof will again use induction on $|x|$, proceeding from the right end of x to the left. The following two lemmas will be used instead of the corresponding lemmas of the last theorem.

Lemma.

$$\lambda_t(x1) = 2\, \lambda_{t-1}(x) + \lambda_t(x). \tag{52}$$

Proof: For each query q counted in $\lambda_{t-1}(x)=\Phi_{t-1}(T(s,k))$, we have queries q0 and q1 counted in $\lambda_t(x1)=\Phi_t(T(2s,k+1))$. In addition, for each query q counted in $\lambda_t(x)=\Phi_t(T(s,k))$ we have the query q* also in $\lambda_t(x1)$.

Let p(x,t) denote the value $C(j,|x|-t)$, where j is the number of zeros in the string x. Then the following lemma can be easily proved by induction on k (proof omitted here):

Lemma.

$$\lambda_t(x) - \lambda_t(x-1) = p(x,t), \text{ for } 0 \le t \le k,\ x \in \mathbf{R}_k,\ <x> \ne 0. \tag{53}$$

The rest of the proof follows the same four-part case analysis as the proof of the Theorem 1. It shall be omitted here as there it is merely a variation on the preceding analysis, using λ_t for λ and p(x,t) for p(x).

This theorem can not be proved in the "only if" direction for all t, since it is not true for the cases t=0 or t=k.

Q. E. D.

3.1.2. THE OPTIMAL SHAPE OF A BUCKET FOR GENERAL RECORDS.

It turns out that binary records are in fact the most difficult case to analyze. In this section we derive the optimal bucket shape for nonbinary records. Let

$$\mathbf{R} =_{\text{def}} \times_{1 \le i \le k} \{0, \ldots, v_i - 1\} \tag{54}$$

be the record space under consideration, where $v_1 \le v_2 \le \cdots \le v_k$, and let $\pi(s,\{v1, \ldots, v_k\})$ be the minimum possible number of queries in $\mathbf{Q}$ which examine a bucket B consisting of s records chosen from $\mathbf{R}$. Corresponding to (26) we have the definition:

$$\pi(s,\{v_1, \ldots, v_k\}) = \min [\, \pi(\max_i f_i,\{v_2,\ldots,v_k\}) + \Sigma_{0 \le i < v_1} \pi(f_i,\{v_2, \ldots, v_k\})\,] \tag{55}$$

where the minimum is taken over all sets of nonnegative integers $f_0, \ldots, f_{v_1 - 1}$ such that $\Sigma_{0 \le i < v_1} f_i = s$.

Here we can perform the analysis by passing to the continuous case. The analog of (55) would then be:

$$\pi'(s,\{v_1,\ldots,v_k\}) =_{\text{def}} \inf_f [\pi'(\sup_{0 \le x \le v_1} f(x),\{v_2,\ldots,v_k\}) + \int_0^{v_1} \pi'(f(x),\{v_2,\ldots,v_k\})dx \tag{56}$$

where the infimum is taken over all nonnegative functions f(x) such that $\int_0^{v_1} f(x)\,dx = s$. If we let $\pi'(s,\text{null})=1$ for $0<s\le 1$, $\pi'(s,\text{null})=\infty$ for $s>1$, and $\pi'(s,\text{null})=0$ otherwise then (56) turns out to have the solution:

$$\begin{aligned}\pi'(s,\{v_1,\ldots,v_k\}) &= 0 \quad \text{if } s=0,\\ &= (s^{1/k}+1)^k \text{ if } 0<s\le v_1{}^k,\\ &= (v_1+1)\,\pi'(s/v_1,\{v_2,\ldots,v_k\})\\ &\qquad \text{if } v_1{}^k < s \le v_1 \cdots v_k,\\ &= \infty \quad \text{if } v_1 \cdots v_k < s. \end{aligned} \tag{57}$$

The function π' is obviously a lower bound for π. The optimal function $f(x)$ is then a step function which is equal to s/s' for $0 \le x \le s'$ and 0 otherwise, where $s' =_{\text{def}} \min(v_1, s^{1/k})$. This proves the following theorem.

Theorem 3. If $\mathbf{R} = \times_{1\le i\le k}\{0,\ldots,v_i-1\}$ then $\Phi(B)=\pi(s,\{v_1,\ldots,v_k\})$ if

$$B = \times_{1\le i\le k} D_i, \tag{58}$$

where each $D_i \subseteq \{0,\ldots,v_i-1\}$, $\prod_{1\le i\le k}|D_i| = s$ and there is an integer z, $2 \le z \le \max_i v_i$, such that for all i, $1 \le i \le k$, we have $|D_i| \le z$ and furthermore, $|D_i| < z$ implies $|D_i| = v_i$.

The theorem says then that our crossword-puzzle hashing scheme of §3.1 is in fact optimal, as long as the function g divides the alphabet into four exactly equal pieces. (This is not possible for a 26-letter alphabet, but we conjecture that four nearly equal pieces are optimal in this case.)

3.1.3. NUMBER OF BUCKETS EXAMINED.

What then is the behavior of such an optimal hashing scheme for the "classic" case of retrieving k-bit words for partial match queries with t bits given? Let $w =_{def} \log_2(b)$ (and assume this is integral), and let our optimal bucket system use (say) the first w bits of a record as the bucket address. We then have

$$A(k,w,t) = C(k,t)^{-1} \Sigma_{0\le i\le t}\, C(w,i)\, C(k-w,t-i)\, 2^{w-i}$$

$$= b\cdot C(k,t)^{-1} \Sigma_{0\le i\le t}\, C(w,i)\, C(k-w,t-i)\, 2^{-i} \qquad (59)$$

The number of buckets examined satisfies the following inequality, for all $b = 2^w$ (with $k \ge w$), and all t, $0 \le t \le k$:

$$A(k,w,t) \ge b^{1-t/k} \qquad (60)$$

This inequality is a special case of a well-known mean value theorem [Ha59:Thm 86], which says

$$\Sigma_{0\le i\le t}\, q_i\, \phi(x_i) \ge \phi(\Sigma_{0\le i\le t} q_i\, x_i), \qquad (61)$$

for any positive numbers q_i which sum to one and any continuous convex function $\phi(x)$. Here we have

$$x_i = i, \qquad (62)$$

$$q_i = C(k,t)^{-1}\, C(w,i)\, C(k-w,t-i), \text{ and} \qquad (63)$$

$$\phi(x) = 2^{-x}. \qquad (64)$$

The inequality (60) will be strict unless k = w, in which case equality holds. Figure 1 graphs (60) for k = 50 and w = 5, 25, and 50. The value A(k,w,t) is an achievable lower bound on the performance of a balanced hashing scheme for binary records. Note that performance similar to Gustafson's is obtained, i. e. each record is stored only once, but search time decreases approximately exponentially with the number of bits given in the query.

Theorem 1 adequately characterizes the optimal "shape" of a single bucket, but does not tell us what the best number of buckets is. This question can be answered by using an accurate model of the particular storage device used.

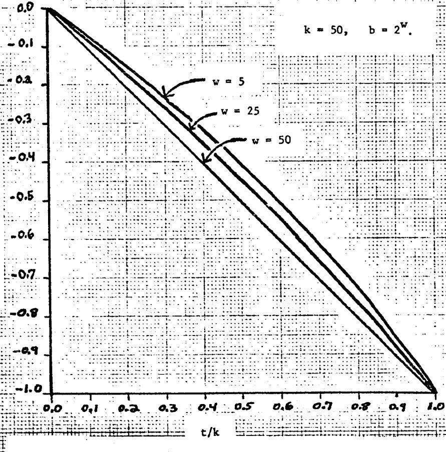

Figure 2. Graph of A(50,w,t) versus t for w=5, 25, and 50

3.1.4. A SAMPLE APPLICATION.

Let us consider a particular application in detail, in order to illustrate the preceding sections and to show how one would proceed to select the proper number of buckets for a hashing scheme.

Suppose we have a file of $n = 2^{20}$ 100-byte records, each having $k = 32$ one-bit keys, which we wish to store on an IBM 2314 disk storage device. Let us determine the optimal number, b, of equal-sized buckets for this device, assuming that all partial match queries are equally likely to occur. Let $b = 2^w$, for some w, $1 \leq w \leq 32$, and let a record be stored in the bucket whose address is the first w of the one-bit keys. If i of the first w bits are specified in a query, then only 2^{w-i} buckets need to be examined to answer the query. The time required to access these buckets is composed of three parts: head access time, rotational delay, and data transfer time. The head access time is at most the minimum of 75 milliseconds per bucket, or 25 milliseconds per cylinder required to store the entire file (1440 records can be stored per cylinder), since each seek is at most 75 ms., but the time to access an adjacent cylinder is only 25 ms. The rotational delay is 12.5 milliseconds per bucket accessed. The data transfer time will be $.32051\, n\, 2^{-i}$ milliseconds on the average (where i is defined as above). Let $c_a(i,w,n)$ be the time required to access 2^{w-i} buckets, as computed using the above information. Thus we have:

$$c_a(i,w,n) = \min(5n/288, 75\cdot 2^{w-i}) + 2^{-i}(12.5\cdot 2^w + .32051n). \qquad (65)$$

The expected time to answer a query with t bits given is then:

$$\alpha(h,t) = \Sigma_{0\le i\le w}\, C(w,i)\, C(k-w,t-i)\, C(k,t)^{-1}\, c_a(i,w,n). \qquad (66)$$

The average time to answer any partial match query is then

$$\alpha(h) = \Sigma_{0\le t\le k}\, C(k,t)\, 2^t\, 3^{-k}\, \alpha(h,t)\ . \qquad (67)$$

Figure 3 shows $\alpha(h)$ plotted against w. The optimal value of w is seen to be 13, with an average response time of 5.123 secs. This compares very favorably with the 336 secs. required to read the entire file if it is stored compactly. Figure 4 gives $\alpha(h,t)$ plotted against t, for $w = 8, 13$, and 20, and for $0\le t\le k$.

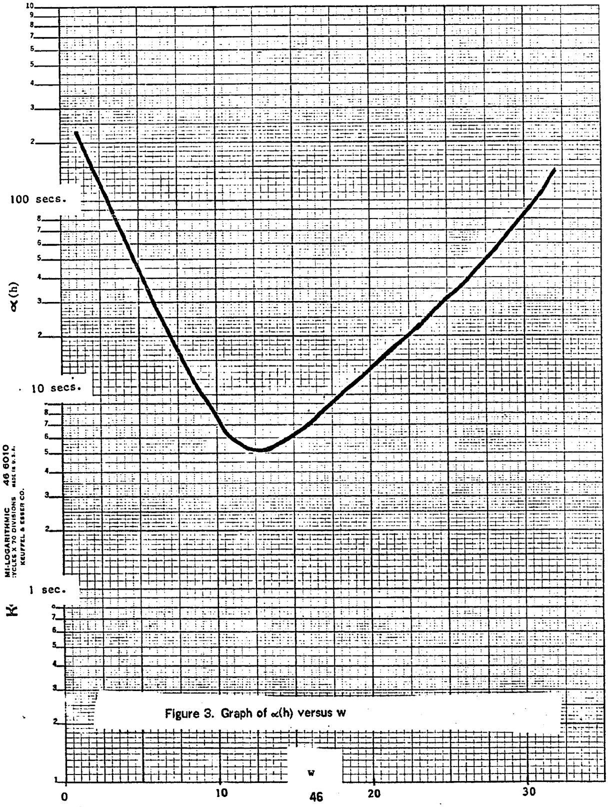

Figure 3. Graph of $\alpha(h)$ versus w

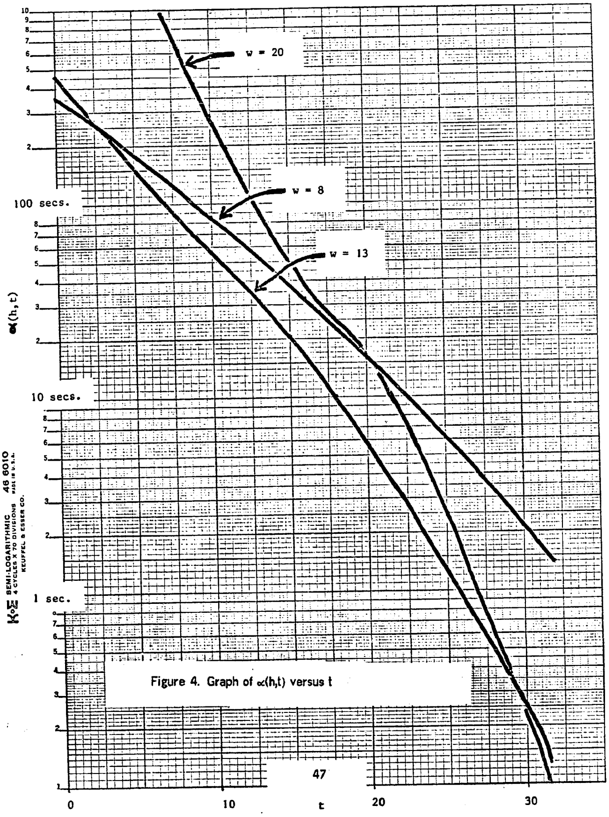

Figure 4. Graph of α(h,t) versus t

3.2. ANALYSIS OF WORST-CASE SEARCH TIME

The hashing functions of the previous section, while providing good average response time to a query with t keys given, tend to have disastrous worst-case behavior. The entire file may be searched if none of the keys given are used by the hash function to compute the bucket address. We will show how the worst-case performance can be made to approach the optimal expected time of the previous section by using either more complicated hash functions, or by using some storage redundancy.

First, let us consider the non-redundant case - that is, each record will be stored in a single bucket. Section 3.3 will consider the storage redundancy case. Our hash function

$$h: \mathbf{R} \rightarrow \{1, 2, \ldots, b\} \tag{68}$$

must now depend on <u>all</u> of the keys of a record, so that each key specified contributes approximately equally to decreasing the search time. This is simple when $k \leq \lceil \log_2(b) \rceil$, so we shall assume that $k > \lceil \log_2(b) \rceil$ from now on. We shall furthermore assume for simplicity that each record is a k-bit word (that is, $v_i = 2$ for $1 \leq i \leq k$).

There is one other assumption we shall make: that the buckets are shaped the same as in the optimal average search time case - that is, each bucket will contain $|\mathbf{R}|/b$ records which agree in $w = \log_2(b)$ bits and vary in the other $k-w$ bits. Each bucket is thus a Boolean sub-cube of dimension $k-w$ of $\mathbf{R}$. The justification for this assumption is that this minimizes the average retrieval time, which is of course a lower bound on the worst-case time. We have no proof, however, that these bucket shapes are optimal in the worst-case hash function.

The reader may be wondering if we aren't studying exactly the same hash functions as before, where we extract w bits to use as a bucket address. Indeed, these are still candidates for the best worst-case hash function, but there are others. Consider the hash function of Figure 5, with $k = 4$ and $b = 8$.

bucket address	1	2	3	4	← bit position
1	0	0	*	0	
2	1	0	0	*	
3	*	1	0	0	
4	1	*	1	0	
5	1	1	*	1	
6	0	1	1	*	
7	*	0	1	1	
8	0	*	0	1	

Figure 5. A Hash Function

Here one row is given for each bucket describing the records that can be stored there (where "*" is a "don't care" character, as before). Thus $h(0110) = 6$ and $h(1110) = 4$. It is simple to verify that each record is assigned a unique bucket by h. This function was first pointed out to me by Donald E. Knuth, and can be interpreted as a perfect matching on the Boolean 4-cube (see Figure 6).

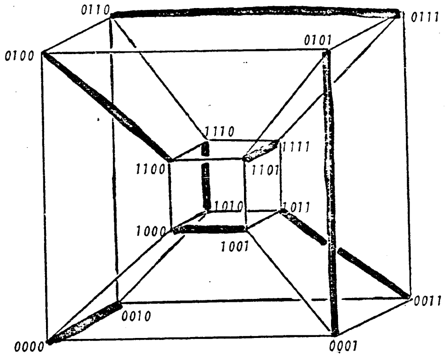

Figure 6. A perfect matching on $\mathbf{R}_4$

How well does this hash-function perform? The symmetry of this design decreases the amount of work done in the worst case. For example, any query with 2 bits specified need only examine 3 buckets (e. g. query "1*0*" requires only buckets 2, 3, and 5). Figure 7 gives the relevant statistics for each case.

t	0	1	2	3	4
$\beta(h, t)$	8	5	3	2	1
$\lceil A(4, 3, t) \rceil$	8	5	3	2	1

Figure 7. Buckets examined per query bit given

This is clearly as good as one can hope to do, since the worst-case time must be at least as big as A(4,3,t).

3.2.1. FORMAL DEFINITION OF ABD'S.

Let us call a hash function presented in tabular form as above an "associative block design with parameters k and w" (where $b = 2^w$), or an "ABD(k,w)" for short. More precisely, an ABD(k,w) will be such a hash function that is "uniform" with respect to each key.

Definition: An ABD(k,w) is a table with $b = 2^w$ rows and k columns with entries over $\{0, 1, *\}$ such that

(i) each row contains exactly w digits and $k - w$ $*$'s,

(ii) given any two rows, there exists at least one column in which the two rows contain differing digits, and

(iii) each column contains the same number $b \cdot (k-w)/k$ of $*$'s.

Condition (ii) guarantees that distinct buckets are disjoint, while condition (i) ensures that each bucket is of the same size. Each record will be associated with a unique bucket since the disjoint buckets contain a total of 2^k records. Condition (iii) restricts ABD's to hash functions having at least some uniformity with respect to how each individual bit affects the bucket address computations. This ensures that an ABD will have minimal worst-case search time for queries with one bit specified (that is, $t = 1$). Since each row of the ABD represents a bucket which is actually a subcube of $\mathbf{R}_k$ by condition (i), an ABD is guaranteed to have minimal average search time as well.

The construction of ABD's of arbitrary size is a difficult combinatorial design problem, comparable to the construction of balanced incomplete block designs (see [Co52]). In fact, an ABD will be a group-divisible incomplete block design of 2k objects (one object type for each digit type of each column) each replicated $w b / 2 k$ times in b blocks of size w, where there are k groups (the columns) with two objects in each group, and where two objects of the same group never occur together in the same block, if there is a number λ_2 such that each pair of objects of differing groups appear in exactly λ_2 blocks together (see [Bo52]). This requirement is an additional constraint, which may exclude many valid ABD's. In addition, not every group-divisible incomplete block design of the proper type will be an ABD, since the definition of a group-divisible incomplete block design does not guarantee that condition (ii) above will be met. Thus the question of the existence of ABD's of arbitrary size does not seem to be answered by any previous results of combinatorial design theory.

3. 2. 2. CHARACTERISTICS OF ABD'S.

The following lemmas give some additional details on the characteristics of ABD's.

Lemma 1. There must be an equal number of 0's and 1's in each column of an ABD.

Proof: There are an equal number of vectors in $\mathbf{R}_k$ having a 0 in a given column as there are having a 1 . Furthermore, each row with $*$ in that column contributes an equal number of each type. Finally, there are an equal number of $*$'s in each row so a digit in a column always contributes exactly 2^{k-w} vectors of that type.

Corollary. The value of bw/2k must be integral (this is the number of 0's or 1's in each column).

Lemma 2. The number of rows having u bits in common with any given record, for $0 \le u \le w$, is exactly C(w,u).

Proof: Let z_u be the number of rows having u bits in common with the given record. We must have

$$z_u = C(k,u) - \Sigma_{0 \le v < u}\, z_v\, C(k-w,u-v) \tag{69}$$

to cover all the vectors in $\mathbf{R}_k$ having exactly u bits in common with the given record. This equation is satisfied, uniquely by induction on u, by

$$z_u = C(w,u). \tag{70}$$

In particular, this lemma tells us how many rows there are having exactly u zeroes (or u ones).

3.2.3. CONSTRUCTION THEOREMS FOR ABD'S.

The following theorem, due to Ronald Graham, establishes the existence of an infinite class of simple ABD's.

Theorem 4. An ABD(2^m,2^m-1) exists for all $m \geq 2$.

Proof:

We shall use an extended notation for an ABD, using the symbol "-" in addition to the usual symbols of "0", "1", and "*". A row having s "-"'s will represent 2^s actual rows of the ABD, obtained by independently replacing each "-" of the row with a "0" or a "1".

The construction consists of two parts:

(i) The first m+1 rows have "-"'s in positions m+2 through $k = 2^m$. The i-th of these rows has a "*" in position i. The other positions are filled in with digits in such a fashion as to satisfy condition (ii) of the definition of an ABD. This is easy to do; the rotations of the string $* \; 1 \; 0^{m-1}$ will work, for example.

(ii) The remaining rows are divided into k-m-1 pairs. The rows of the i-th pair have "-"'s in positions m+2 through k except for a "*" in position m+1+i. The first m+1 positions are filled in with digits in a manner consistent with the definition of an ABD; this is simple to do.

It is easy to verify that this yields an ABD(2^m,2^m-1).

Q. E. D.

To illustrate the above construction, here is an ABD(8,7) constructed by Graham's method (that is, this is the construction for $m = 3$):

$$\begin{array}{cccccccc}
\ast & 1 & 0 & 0 & - & - & - & - \\
0 & \ast & 1 & 0 & - & - & - & - \\
0 & 0 & \ast & 1 & - & - & - & - \\
1 & 0 & 0 & \ast & - & - & - & - \\
0 & 0 & 0 & 0 & \ast & - & - & - \\
0 & 1 & 0 & 1 & \ast & - & - & - \\
0 & 1 & 1 & 1 & - & \ast & - & - \\
1 & 0 & 1 & 0 & - & \ast & - & - \\
1 & 0 & 1 & 1 & - & - & \ast & - \\
1 & 1 & 0 & 1 & - & - & \ast & - \\
1 & 1 & 1 & 0 & - & - & - & \ast \\
1 & 1 & 1 & 1 & - & - & - & \ast
\end{array}$$

Figure 8. An ABD(8,7)

This general idea, of dividing the columns into two groups and filling in each part seperately, can be carried a little further; the following figure gives an ABD(16,13), also discovered by Graham.

```
*0001**---------
*0101--**-------
*0111----**-----
1*000------**---
1*010--------**-
1*011*---------*
01*00-**--------
01*01---**------
11*01-----**----
001*0-------**--
101*0---------**
111*0**---------
0001*--**-------
0101*----**-----
0111*------**---
00000--------***
11111--------***
```

Figure 9. An ABD(16,13)

The designs of Theorem 4 are not useful hash functions, however, with the possible exception of the ABD(4,3), since the ratio k/w of key bits to bucket address bits approaches 1 as the designs get larger. What is really desired is a way to construct large designs with fewer buckets. The following theorem gives a basic upper bound on the ratio k/w achievable for a given k (that is, given a number of keys, it gives a lower bound on the number of buckets required for an ABD(k,w) to be possible).

Theorem 5.

$$k \le (w^2/2)\cdot(2^w/(2^w-1)). \tag{71}$$

Proof: Between each pair of rows of an ABD there must be at least one column in which they contain differing digits. There must be at least C(b,2) such row-row-column differences. On the other hand, there are only wb/2k 0's and 1's per column. Thus we must have

$$k\,(wb/2k)^2 \geq C(b,2) \tag{72}$$

which directly yields our theorem.

Q. E. D.

As a consequence of the above theorem and lemma 2 of §3.2.2, we can tabulate the nontrivial pairs (k,w) for which ABD(k,w)'s may exist, for small k.

k	Permissable values of w, w≠k
4	3
8	4, 5, 6, 7
10	5
12	6, 9
14	7
16	6, 7, 8, 9, 10, 11, 12, 13, 14, 15
18	6, 9, 12, 15
20	10, 15

Figure 10. Permissable values of (k,w)

One can also show, by an extension of theorem 5, that an ABD(8,4) is also impossible.

The following theorem gives a basic way of creating larger ABD's from smaller ones.

Theorem 6. (Concatenation) It is possible to construct an ABD(k+k′,w+w′) from an ABD(k,w) and an ABD(k′,w′), if k/w = k′/w′.

Proof: Form the set of all $2^{w+w'}$ rows of length k+k′ obtained by concatenating each row of the second design onto the end of each row of the first design. It is easy to see that this is an ABD(k+k′,w+w′).

Q. E. D.

Thus we can form an ABD(8,6) or an ABD(12,9) from the design of Figure 5. Figure 11 gives the ABD(8,6) so constructed.

	12345678		12345678		12345678		12345678
1	00*000*0	17	*10000*0	33	11*100*0	49	*01100*0
2	00*0100*	18	*100100*	34	11*1100*	50	*011100*
3	00*0*100	19	*100*100	35	11*1*100	51	*011*100
4	00*01*10	20	*1001*10	36	11*11*10	52	*0111*10
5	00*011*1	21	*10011*1	37	11*111*1	53	*01111*1
6	00*0011*	22	*100011*	38	11*1011*	54	*011011*
7	00*0*011	23	*100*011	39	11*1*011	55	*011*011
8	00*00*01	24	*1000*01	40	11*10*01	56	*0110*01
9	100*00*0	25	1*1000*0	41	011*00*0	57	0*0100*0
10	100*100*	26	1*10100*	42	011*100*	58	0*01100*
11	100**100	27	1*10*100	43	011**100	59	0*01*100
12	100*1*10	28	1*101*10	44	011*1*10	60	0*011*10
13	100*11*1	29	1*1011*1	45	011*11*1	61	0*0111*1
14	100*011*	30	1*10011*	46	011*011*	62	0*01011*
15	100**011	31	1*10*011	47	011**011	63	0*01*011
16	100*0*01	32	1*100*01	48	011*0*01	64	0*010*01

Figure 11. An ABD(8,6)

Theorem 6 does not allow us to increase the achievable record length/bucket address length (k/w). One might suspect that 4/3 is perhaps an upper bound for k/w. The following theorem shows that arbitrarily large ratios are possible.

Theorem 7. (Insertion) It is possible to construct an ABD(kk′,ww′) from an ABD(k,w) and an ABD(k′,w′).

Proof: We will construct the larger ABD by independently replacing each digit of the first ABD by a row from the second, and each "*" of the first ABD by a string of k′ "*"'s. If the digit being replaced is a zero, we choose a row from the top half of the second ABD (that is, from the first $2^{w'-1}$ rows) to replace it. If the digit being replaced is a one, we use a row from the bottom half of the second ABD. (Actually, any division of the second ABD into two halves may be used.) Each row of length k thus generates $2^{(w'-1)w}$ rows of length kk′, so that $2^{ww'}$ rows are generated altogether. Each such row has $w(k'-w') + (k-w)k' = kk' - ww'$ "*"'s. Each pair of rows generated will differ in at least one place, since rows used to replace differing digits differ, or if the two rows were generated from the same row of the first design, then one of the digits replaced will have been replaced with different rows from the second design, which must differ in at least one place. Finally, the number of "*"'s in each column is

$$(2^{w'-1})^{w}((k-w)2^{w}/k) + (w\,2^{w-1}/k)((k'-w')\,2^{w'}/k)(2^{w'-1})^{w-1} \qquad (73)$$

$$= 2^{ww'}(kk' - ww')/kk'\,. \qquad (74)$$

Therefore this construction yields a valid ABD(kk′,ww′).

Q. E. D.

The above theorem allows us to form ABD's with arbitrarily large ratios k/w. For example, we can now construct an ABD(16,9) or an ABD(64,27) (in general, an ABD(4^m,3^m) for $m \geq 1$) from the ABD(4,3) of Figure 5. The following figure illustrates the rows generated for an ABD(16,9).

```
00*0      →      00*000*0****00*0
                 00*000*0****100*
                 00*000*0*****100
                 00*000*0****1*10
                 00*0100*****00*0
                 . . .

100*      →      11*100*000*0****
                 11*100*0*100****
                 11*100*01*10****
                 . . .

.
.
.         .

0*01      →      00*0****00*011*1
                 00*0****00*0011*
                 00*0****00*0*011
                 . . .

ABD( 4, 3) rows  →  ABD( 16, 9) rows
```

Figure 12. Rows of an ABD(16,9)

3. 2. 4. ANALYSIS OF ABD SEARCH TIMES.

How well do these ABDs perform as hash functions for associative retrieval of binary records? Let use derive the worst-case behavior of ABDs constructed by concatenation and insertion.

Let us consider the concatenation of an ABD(k,w) with an ABD(k′,w′). Let $\beta(g,t)$ and $\beta(g',t)$ be the respective worst-case number of buckets examined in each case for a query $Q \in \mathbf{Q}_t$, and let $\beta(h,t)$ be the same function for the resultant ABD. Since g, g′ and h are fixed, we are considering the associated functions β as functions of t only. We can then easily derive

$$\beta(h,t) = \max_{u+v=t} \beta(g,u)\, \beta(g',v) \,, \tag{75}$$

for $0 \le t \le k+k'$, $u \le k$, $v \le k'$. For example, concatenating an ABD(4,3) with itself yields an ABD(8,6) with worst-case retrieval times:

t	0	1	2	3	4	5	6	7	8
$\beta(h,t)$	64	40	25	16	10	6	4	2	1
$\lceil A(8,6,t)\rceil$	64	40	25	15	9	6	4	2	1

Figure 13. Performance of an ABD(8,6)

Also shown are the values of $\lceil A(8,6,t)\rceil$, which is a lower bound for $\beta(h,t)$. The ABD(8,6) is seen to do nearly as well as possible. The exact asymptotics for the worst-case behavior of the repeated concatenation of an ABD with itself are quite simple to figure out for given values of k and w. Suppose we concatenate an ABD(k,w) with worst-case behavior $\beta(g,t)$ with itself m times, yielding an ABD(mk,mw). Consider a partial match query $Q \in \mathcal{Q}_t$. Let y_i be the number of k column blocks which have exactly i specified bits, for $0 \le i \le k$, so that

$$\Sigma_{0 \le i \le k}\, y_i = m, \tag{76}$$

and

$$\Sigma_{0 \le i \le k}\, i\, y_i = t\,. \tag{77}$$

We also have, of course, the condition that

$$y_i \ge 0 \quad \text{for } 0 \le i \le k. \tag{78}$$

The worst-case behavior $\beta(h,t)$ of the resultant ABD(mk,mw) is defined by

$$\beta(h,t) = \max\ \Pi_{0 \le i \le k}\, \beta(g,i)^{y_i} \tag{79}$$

where the maximum is taken over all sets of integers $y_0, \ldots, y_k$ satisfying (77) - (79). Let $\beta'(h,t) = \log(\beta(h,t))$ and $\beta'(g,t) = \log(\beta(g,t))$ for all t. Then (79) becomes

$$\beta'(h,t) = \max \Sigma_{0 \le i \le k}\, \beta'(g,i)\, y_i, \tag{80}$$

transforming the above into a integer programming problem in k+1 dimensional space. Since we are considering the asymptotic behavior as $m \to \infty$, the solution to the corresponding linear programming problem, in which each y_i is replaced by the corresponding fraction $x_i = y_i/m$, will give us the asymptotic behavior. The problem to be solved is thus:

$$\text{maximize } \beta'(h,t) = \Sigma_{0 \le i \le k}\, \beta'(g,i)\, x_i, \tag{81}$$

subject to:

$$\Sigma_{0 \le i \le k} x_i = 1, \tag{82}$$

$$\Sigma_{0 \le i \le k} i\, x_i = t/m, \text{ and} \tag{83}$$

$$x_i \ge 0 \text{ for } 0 \le i \le k. \tag{84}$$

We must have at least k-1 of the x_i's equal to zero in the optimal solution, since there are only k+3 constraints for this problem in k+1 dimensions. Let x_i and x_j be the two nonzero values, with $i<j$. If $\beta'(g,t)$ is a concave function we have

$$i = \lfloor t/m \rfloor = j-1 \tag{85}$$

and

$$\beta'(h,t) = \beta'(g,i)(i-t/m)/(i-j) + \beta'(g,j)(j-t/m)/(j-i). \tag{86}$$

This is the general solution. When t/m is a multiple of $1/k$, then only $x_{tk/m}$ is nonzero, and it is equal to one. This solution does not apply when $\beta'(g,t)$ is not concave. (For example, the $\beta'(g,t)$ for our ABD(4,3) is not quite concave, since $\beta(g,3)=2$ is a little too large. This convexity is the cause of the discrepancies of Fig. 13). One can show, by a combinatorial argument, that if $\beta(g,t)=A(k,w,t)$ for $0 \le t \le k$, then $\beta(h,t)=A(mk,mw,t)$ for $0 \le t \le mk$ as well. Thus concatenation of ABD's can be expected to preserve near-optimal worst-case behavior.

The behavior of an ABD constructed by insertion is more difficult to work out. It seems the worst-case here occurs when the specified bits occur together in blocks corresponding to the digits of the first ABD used in the construction (that is, the one whose digits were replaced). (I have no proof of this.) Figure 14 gives the worst-case behavior of an ABD(16,9) constructed by inserting the ABD(4,3) of Figure 5 into itself, (computed using the assumption that the worst-case behavior occurs with the queries having the specified bits occurring in blocks). Shown below the worst-case behavior $\beta(h,t)$ for the above design are the values of A(16,9,t), which are a lower bound for the number of buckets that must be examined in the worst case.

t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\beta(h,t)$	368	272	224	176	116	76	56	36	33	24	16	8	5	3	2	1
$\lceil A(16,9,t)\rceil$	368	263	186	131	91	63	43	30	20	14	9	6	4	3	2	1

Figure 14. Behavior of the ABD(16,9)

We see the the lower bound is nearly achieved, that is, the worst-case behavior of this hashing scheme approximates the average time. On the other hand, it is quite likely that even better designs for an ABD(16,9) exist - the regular fashion in which this one was constructed probably degrades its worst-case performance somewhat. An exhaustive search by computer for better designs appears to be infeasible, so that a better construction method is needed.

(I was unable to determine whether an ABD(8,5) exists or not, using one hour of computer time and a sophisticated backtracking procedure.)

3.2.5. IRREGULAR ABD'S.

The difficulty of constructing ABD's leads one to attempt simpler, less tightly constrained hash functions. Such ad hoc hash functions are easy to construct for small values of k and w. For example, consider the case k = 3, w = 2 (which does not satisfy the divisibility constraint of the corollary to Lemma 1, so that an ABD(3,2) can not exist). The following "design" yields reasonably good worst-case performance.

	1	2	3
1	0	0	*
2	1	*	0
3	*	1	1
4	0	1	0
	1	0	1

Figure 15. An "irregular" (3,2) design

Here bucket 4 contains both records 010 and 101. This hash function has worst-case behavior:

t	0	1	2	3
$\beta(h,t)$	4	3	2	1

Figure 16. Behavior of the previous design

Concatenating this function with itself will yield larger "designs" having a k/w ratio of 3/2 and having good worst-case retrieval times. Another "design" yields the k/w ratio of 2:

$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 0 & * & * \\
2 & * & 1 & * & 0 \\
3 & * & 1 & 1 & 1 \\
 & 1 & 0 & 1 & * \\
4 & * & 1 & 0 & 1 \\
 & 1 & 0 & 0 & *
\end{array}
$$

Figure 17. An "irregular" (4,2) design

The above hash function has worst-case behavior:

$$
\begin{array}{c|ccccc}
t & 0 & 1 & 2 & 3 & 4 \\
\hline
\beta(h, t) & 4 & 4 & 3 & 2 & 1
\end{array}
$$

Figure 18. Behavior of the irregular (4,2) design

3. 2. 6. CONCLUSIONS ON ABD'S.

Associative block designs will have exactly the same average retrieval time as those hash function discussed in section 3. 1, where w key-bits were extracted to use as bucket address bits, since the buckets have the same shape. But by appropriately permuting the entries in each row, we can drastically reduce the worst-case time without affecting the average retrieval time. The recursive or iterative nature of the ABD construction theorems lends itself to simple implementation. In summary, we see that the worst-case performance of hashing schemes can be nearly minimized without increasing the average retrieval time or the amount of storage used.

3.3. BENEFITS OF STORAGE REDUNDANCY

The perhaps difficult problems involved in constructing an ABD for a particular application can be circumvented if the user can afford a moderate amount of storage redundancy to achieve good worst-case behavior. By moderate I really mean moderate - the redundancy factor is not subject to combinatorial explosion as in the designs of Ghosh et al. Furthermore, both the worst-case and average behavior is even slightly improved over the designs of § 3.1 and the ABD's of § 3.2.

The technique is actually quite simple, and will be illustrated by an example. Suppose we have a file of $n = 2^{20}$ 100-bit records (that is, each record consists of 100 one-bit keys). The method of the previous section would have required the construction of an ABD(100,w), for w near 20 - a difficult task. Let us instead simply create five (= 100/20) bucket systems, and let each record be filed once in each system. Each bucket system will have 2^{20} buckets. The first system will use the first 20 bits of each record as its bucket address, the second bucket system will use the second 20 bits of the record, and so on.

Now suppose we have a query $Q \in Q_t$. At least one of the five bucket systems will have at least $\lceil t/5 \rceil$ bits specified for its bucket address - so we can use this bucket system to retrieve the desired records. The number of buckets searched is no more than $2^{20-\lceil t/5 \rceil}$ at worst.

In general, if $b = 2^w$ is the number of buckets per bucket system, and we have k-bit records to store (records with non-binary keys can of course always be encoded into binary), we will establish $m = k/w$ distinct bucket systems, divide the record into m w-bit fields, and use each field as a bucket address in one of the systems.

The worst-case behavior of this scheme follows a strict geometric inequality:

$$\beta(h,t) \le 2^{w-\lceil wt/k \rceil} \tag{87}$$

This surpasses even the best achievable <u>average</u> behavior of hash functions with no storage redundancy, although not by very much. If half of the bits are given in a query (i.e. $t = k/2$), then only $\text{sqrt}(b) = 2^{w/2}$ buckets at most need be searched. The average behavior of this scheme is difficult to compute, but it seems likely that it will approach the worst-case behavior, especially if w is large.

The above idea can be generalized further. Instead of taking each of the m subfields of the record and using it directly as an address, one can treat each subfield as a record and use an ABD(k/m,w) or some other method (such as the trie algorithm of §4) to calculate an address from each subfield. The efficiency of this composite method will of course depend on the efficiency of the the methods chosen for each sub-field.

CHAPTER 4

TRIE ALGORITHMS FOR PARTIAL MATCH QUERIES

Theorem 1, which states that an optimal bucket shape for a hash table is a subcube of $\mathbf{R}$, suggests that another data structure might be preferable to hash tables. A trie also has the property that the set of records under consideration at any point of the trie is a subcube of $\mathbf{R}$, which is recursively split into smaller subcubes at each level. Tries might thus behave like the best hash functions. They have the advantage that the data is structured all the way down to the terminal nodes (the records), in contrast with hash tables, where each bucket merely contains an unordered list. In this section we will try to estimate the average search time for a partial match query when the file $\mathbf{F}$ is maintained in random-access storage as a trie.

4.1. DEFINITION OF TRIES

"Tries" were first described by Rene de la Briandais [de59] and were elaborated on by E. Fredkin in [Fr60] (see also [Kn72,§6.3]).

Definition. A trie is a tree such that

(i) Records are its terminal (external) nodes.

(ii) Each internal node N specifies an attribute position j, such that attribute j has not been specified on any node on the path from the root of the trie to N. In a standard trie the attribute position specified is always j, where j is the level of the node N in the trie. Nonstandard tries were first introduced by G. Gwehenberger [Gw68]. Each internal node is said to be associated with all of the records of its corresponding subtrie, and

(iii) if node N specifies attribute j, then node N has v_j subtries, one for each possible value of attribute j. The records associated with node N are each placed into the subtrie of N corresponding to their value for attribute j.

Two kinds of tries will be considered. A full trie will have all records at level k+1 (where the root is at level 1). Any subtrie associated with zero records will be a special null node at some level less than k+1. A compact trie will place the terminal node corresponding to a record at the uppermost level possible. In other words, a compact trie has a terminal node whenever the corresponding node in the full trie is associated with only one record, and all of the ancestors of the node in the full trie are associated with more than one record. Figure 19 and 20 illustrate full and compact tries for the file of three-bit records $\mathbf{F} = \{ 000, 100, 101, 111 \}$.

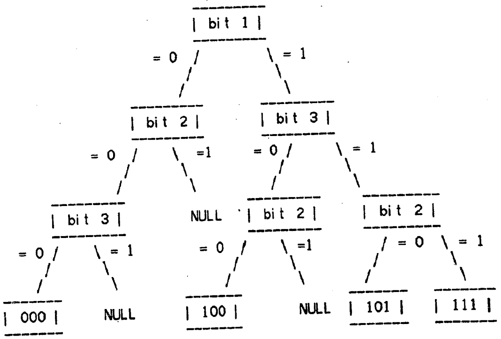

Figure 19. A full trie

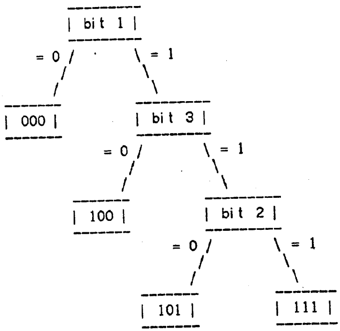

Figure 20. The corresponding compact trie

4.2. ALGORITHM FOR SEARCHING TRIES

To perform an associative search of a trie is quite simple. Given a partial match query Q the search algorithm works as follows:

Associative Search of a Trie

Step 1. Set pointer p to the root of the trie.

Step 2. If p points to a terminal node, print the associated record if it satisfies Q and return.

Step 3. (Here p points to an internal node N specifying attribute j). If attribute j is specified in the query, search the corresponding subtrie of N, otherwise search all subtries of N. (These recursive searches use this algorithm beginning at step 2).

What is the average running time of this algorithm? Let the time be the average number of nodes (both internal and external) examined by the algorithm. We shall use a slightly different assumption about the file, in order to make the mathematics easier. Instead of letting our file $\mathbf{F}$ be a randomly chosen subset of $\mathbf{R}$ of size exactly n, let us instead assume that each record $R \in \mathbf{R}$ is chosen to be in $\mathbf{F}$ independently with probability $p = n/|\mathbf{R}|$. Thus $E(|\mathbf{F}|) = n$, but $\mathbf{F}$ may also have some other size. There will be no significant bias in our results due to this change in assumption. The following notation denotes our cost measure:

$f(k,t)$ = the average number of nodes examined by the above algorithm to answer a partial match query $Q \in \mathbf{Q}_t$, where the file F consists of (approximately) n distinct records, each having k one-bit keys.

There is an interesting optimization problem resulting from the use of nonstandard tries. The problem is to select the attribute positions with which to label each internal node in such a fashion as to minimize the expected number of nodes examined for any partial match query. The interesting point here is that it will be the most unbalanced trie which will have the minimal search time, since nodes deep in the trie are seldom examined. To actually determine the optimal trie seems to be a difficult optimization problem, and we shall not remark upon it further. In fact, we shall restrict our attention to standard tries.

4.3. UPPER BOUND ON THE SEARCH TIME

The close relationship between tries and hash functions which extract bits to use as a bucket address allows an upper bound on f(k,t) to be derived very simply.

We first note that the probability $\Phi_t(N)$ that a particular node N will be examined is basically a function of the level l(N) of N in the trie:

$$\begin{aligned}\Phi_t(N) &= C(k,t)^{-1}\, \Sigma_{0\le i\le t}\, C(l(N)-1,i)\, C(k-l(N)+1,t-i)\, 2^{-i} \\ &= 2^{-l(N)+1}\, A(k,l(N)-1,t). \qquad (88)\end{aligned}$$

As noted above, since $\Phi_t(N)$ decreases so rapidly with l(N), of all the n-node tries it will be the most balanced tries which have the highest average retrieval time.

Thus we derive an upper bound for f(k,t) by only considering the most balanced tries of n nodes. Thus it is very simple to derive the bound:

$$f(k,t) \le \Sigma_{1 \le j \le \lceil \log_2(n) \rceil} A(k,j,t), \qquad (89)$$

since there are 2^{j-1} nodes on each level j of the most balanced trie except possibly the last. The dominant term in this sum will generally be the last one, corresponding to the highest level. Thus we see that tries will not generally do worse than the best hashing functions which use about n buckets. In the next section we will see that they do not perform significantly better, either.

4. 4. LOWER BOUND ON SEARCH TIME

We shall here assume that the file is stored as a full trie and not a compact trie. While a practical implementation would certainly use compact tries, we shall examine the full trie case since the mathematics is a little simpler. Compact tries are more efficient by a factor of at most

$$k/\log_2(n) = 1 - (\log_2(p)/\log_2(n)). \qquad (90)$$

This approaches 1 in the limit if we keep p fixed and let $n \to \infty$. We shall proceed with our analysis with the understanding that compact trees could be more efficient by this amount.

We now prove our basic theorem for this section, which says that the expected search time for a trie is bounded below by a exponentially decreasing function of the amount of information specified in the query.

Theorem 8.

$$f(k,t) \geq n^{(k-t)/k} = p^{(k-t)/k}\, 2^{k-t} \tag{91}$$

Proof: The basic recurrence for $f(k,t)$ is the following:

$$f(k,t) = 1 + (1-\epsilon_k)(2\, f(k-1,t)(k-t)/k + f(k-1,t-1)\, t/k\,), \text{ for } k>1. \tag{92}$$

Here we define

$$\epsilon_k =_{\text{def}} (1-p)^{2^k} \tag{93}$$

to be the probability that $\mathbf{F}$ is empty. The value $(k-t)/k$ is the probability that the bit named at the root of the trie is not specified in the query, and t/k is the probability that it is specified in the query. The 2 is in the first term because if the bit named in the root is not specified in the query, then we have to search both subtries, otherwise we only have to search one.

We will prove (91) by induction on k, using (92). The basic inductive step we need to prove is therefore the following inequality.

$$p^{(k-t)/k}\, 2^{k-t}$$
$$\leq 1 + 2^{k-t}\,(1-\epsilon_k)(((k-t)/k)\, p^{(k-t-1)/(k-1)} + t/k\, p^{(k-t)/(k-1)}) \tag{94}$$

If we prove (94), and also prove a basis for the induction, then (91) follows. Defining z to be:

$$z =_{\text{def}} p^{(k-t-1)/(k-1)}\,(\,(k-t)/k + t/k\, p^{1/(k-1)}\,) \tag{95}$$

we then get that (94) is equivalent to:

$$1 \geq 2^{k-t} \left(p^{(k-t)/k} - (1 - \epsilon_k) z \right) \qquad (96)$$

If we can show that

$$z \geq p^{(k-t)/k} \qquad (97)$$

then (96) reduces to showing that

$$1 \geq \epsilon_k 2^{k-t} p^{(k-t)/k} . \qquad (98)$$

So we will first prove (97), and then (98). Now (97) is equivalent to the following.

$$p^{(k-t)/k} \leq (k-t)/k \, p^{(k-t-1)/(k-1)} + t/k \, p^{(k-t)/(k-1)} . \qquad (99)$$

This is the same as

$$p^{t/k(k-1)} \leq (k-t)/k + t/k \, p^{1/(k-1)} . \qquad (100)$$

But this is just an instance of a well-known mean-value theorem [Thm. 37,Ha59].

We shall now prove (98). This is equivalent to

$$1 \geq \epsilon_k 2^k p \, (2 p^{1/k})^{-t}. \qquad (101)$$

Since ϵ_k is independent of t we may set $t=k$ if $2 p^{1/k} < 1$ to maximize the right hand side, reducing (101) to a trivial statement. Otherwise we set $t = 0$. Differentiating the resultant right-hand side with respect to p, we find that it reaches a maximum at

$$p_0 = 1 / (2^k + 1). \qquad (102)$$

But since $2\,p_0^{1/k} < 1$ we only need to prove (101) for $2\,p^{1/k} = 1$, in which case it is again trivial.

Therefore (91) is proved except for the basis for the induction. But for $k = 1$, (91) reduces to

$$f(1,0) \geq 2\,p, \tag{103}$$

and

$$f(1,1) \geq 1\ . \tag{104}$$

Equation (104) is certainly true, since the root node must always be examined. Equation (103) requires computing the average work for a file of 1-bit records for a query with no keys specified. There are four possible files: $\mathbf{F}=\{0,1\}$, $\mathbf{F}=\{0\}$, $\mathbf{F}=\{1\}$, and $\mathbf{F}=\{\}$, which occur with probabilities p^2, $p(1-p)$, $p(1-p)$, and $(1-p)^2$ respectively. The number of nodes examined in each of these cases is 3, 2, 2, and 1 respectively. Equation (103) thus reduces to proving the following.

$$2\,p \leq 3\,p^2 + 2\,p\,(1-p) + 2\,p\,(1-p) + (1-p)^2 = 1 + 2p \tag{105}$$

Q. E. D.

CHAPTER 5

HASHING ALGORITHMS FOR BEST-MATCH QUERIES

The task of searching a file for all best matches to a query has probably been more extensively studied than the the task of searching for all partial matches, due to the fundamental nature of identification problems when only partial and perhaps incorrect attribute data is available. Finding the best-match for a transmitted message is the crux of the decoding problem, for example. Nevertheless, only very recently has significant theoretical progress been made on this problem. As late as 1969 Marvin Minsky conjectured that

"Even for the best [algorithms], the speed-up value of large memory redundancies is very small, and for large data sets with long word lengths there are no practical alternatives to large searches that inspect large parts of the memory." [Mi69,p. 223]

We shall see that the situation is not that bad, and that best-match searches may often be made extremely rapidly, requiring the examination of only the smallest fraction of the file.

5.1. THE ALGORITHM

The method is due originally to Peter Elias, according to [We71], although Burkhard has apparently independently discovered the idea more recently [Bu73].

The algorithm is a variant of the hash-coding scheme, with slightly different hash functions. We shall therefore use the same notation as §3. We divide the space $\mathbf{R}$ of records into b regions $B_1(\mathbf{R},h)$, $B_2(\mathbf{R},h)$, $\ldots$, $B_b(\mathbf{R},h)$ as before. Given an input record Q for which we want to find the best-match, we hope to limit our examination of the file to just a few buckets. To do this we need to find an appropriate hash function.

Due to the nature of the problem, it seems likely that the buckets should be "neighborhoods" or "spheres" of $\mathbf{R}$ rather than subcubes. This conjecture is proved later on. One simple method of dividing $\mathbf{R}$ up into neighborhoods is to choose a set of "reference" records $R' = \{R'_1, R'_2, \ldots, R'_b\}$, and then to associate one bucket with each reference record. A record is placed in the bucket(s) corresponding to the nearest reference record(s) using the Hamming distance metric d. Thus,

$$B_j(\mathbf{R}) =_{\text{def}} \{ R \in \mathbf{R} \mid \neg(\exists i, 1 \le i \le b)[(i \ne j) \wedge (d(R'_i,R) < d(R'_j,R)]\}. \tag{106}$$

Note that a record may belong to more than one bucket under this scheme.

Clearly the reference records can be chosen in different ways. A large amount of research going under the name of "cluster analysis" is directed at choosing the reference records to be records of $\mathbf{F}$ near the centers of naturally occurring "clusters" in $\mathbf{F}$ (for example, see [Ja71]). This method has the advantage of being tailored to the particular file in question, but has difficulties in terms of maintaining this structure while the file is being modified and in terms of organizing the search, since it is hard to determine whether a given bucket needs to be searched (that is, whether it could possibly contain a record closer to R than the closest found so far in the search).

For the purposes of this discussion, we will assume that the file $\mathbf{F}$ is a randomly chosen subset of size n of $\mathbf{R}$. Thus it is unlikely to expect the records of $\mathbf{F}$ to be nicely clustered in any way. How should the reference records be chosen in this situation? One would suspect that they should be rather evenly distributed throughout $\mathbf{R}$.

For the case of binary records, R′ can be easily chosen if b is a power of two, so that $b = 2^w$, and there exists a perfect (k,w) error-correcting code, with minimum distance $2\lambda + 1$. Then R′ will be the set of codewords, and $B_j(\mathbf{R})$ will be the set of all k-bit words which would be interpreted under the decoding rule to be R'_j. (While it has been shown (see [Ti73]) that there are no unknown perfect codes, in those cases where a perfect code does not exist one can do nearly as well by using a quasi-perfect code [Pe72], or the best code available.)

To perform a best-match search, the following algorithm is performed. Essentially the buckets are examined in order of increasing distance of their centers from the query until all the closest records are found.

```
procedure SEARCH2({$B_1, \ldots, B_b$},h,Q,$\lambda$);

comment SEARCH2 finds all records stored in buckets $B_1, \ldots, B_b$ which
are nearest to the record (query) Q.
The value $\lambda$ is the minimal value such that every record is within distance
$\lambda$ of a reference record.;

begin set W, W′, Y; integer m, m′, i, j;
    $m \leftarrow \infty$;  $W \leftarrow$ null;  $Y \leftarrow \{1, 2, \ldots, b\}$;
    while $Y \neq$ null do
      begin
        $j \leftarrow \min\{ j \mid (j \in Y) \wedge (d(R_j', Q) = \min_{i \in Y} d(R_i', Q))\}$;
        if $B_j(F) \neq$ null then
          begin
            $m' \leftarrow \min_{R \in B_j(F)} d(R,Q)$;
            $W' \leftarrow \{ R \in B_j(F) \mid d(R,Q) = m' \}$;
            if $m' = m$ then  $W \leftarrow W \cup W'$
            else if $m' < m$ then
                          begin  $W \leftarrow W'$; $m \leftarrow m'$  end
          end;
        $Y \leftarrow \{ i \mid d(R_i,Q) \leq m + \lambda \} \cap Y - \{ j \}$
      end;
    print( W , m )
end SEARCH2;
```

5. 2. A SAMPLE APPLICATION

Let us consider a particular application. Suppose we have a file of $n=2^{15}$

23-bit words which we wish to organize for best-match searching. We can make use of the Golay perfect (23,12) code (see [Pe72, §5.2]) in our hash function. We will thus have 4096 buckets, each containing about 8 records on the average. The Golay code is capable of correcting all patterns of up to three errors, so that the minimum distance between codewords is 7.

To derive the average time needed to answer a best-match query, proceed as follows. Let p be the probability that a particular record is in $\mathbf{F}$ (here $p = 2^{15}/2^{23} = .00390625$), and let $\alpha(h)$ be the expected number of buckets examined. Then

$$\alpha(h) = \Sigma_{0 \le i \le 3} C(23, i)\ 2^{-11}\ \Sigma_{0 \le j \le 23}\ pe(j)\ nb(i,j) \tag{107}$$

where pe(j) is the probability that the nearest record to a "typical" query is at distance j. This is an average, taking as separate cases the distance i of the query from the center of its bucket. That is

$$pe(j) = (1-p)^{V(23,j-1)}\ (1 - (1-p)^{C(23,\ j)}), \tag{108}$$

where V(k,j) is the volume of a sphere with radius j in binary k-space, that is,

$$V(k,j) = \Sigma_{0 \le i \le j} C(k,i)\ . \tag{109}$$

The quantity nb(i,j) in (107) denotes the average number of buckets that need to be examined to find all words within distance j of R for a typical word $R \in \mathbf{R}$ where the distance from R to the nearest code-word is i. The values of nb(i,j) for $0 \le i \le 3$ and $0 \le j \le 23$ were determined with a computer program. Figure 21 plots $\alpha(h)$ versus p. For our application ($p = 2^{15}/2^{23}$), we see that no more than 37 buckets need be examined on the average.

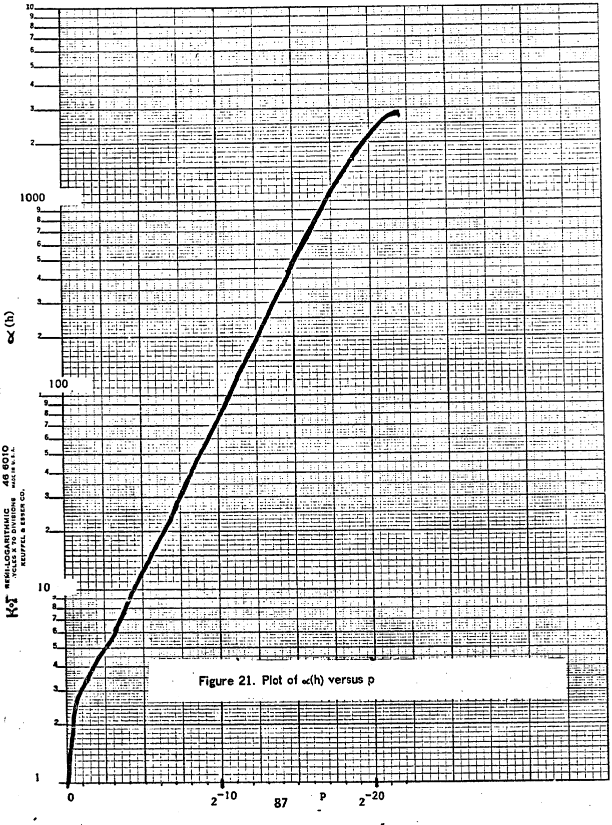

Figure 21. Plot of α(h) versus p

5.3. ANALYSIS OF ASYMPTOTIC RUNNING TIME

How well does the above algorithm perform? The expected number of records examined will be at most $p\ C(k,m+\lambda)$, where m is the distance from the query to the nearest record in $\mathbf{F}$, and λ is the common radius of the buckets. For $m+\lambda$ small in relation to k, this will be a negligible fraction of the file.

To consider the asymptotic performance, let $k \to \infty$ and $w \to \infty$ proportional to k. This corresponds to the case where p, the file density, remains fixed. Then it is well-known that there are codes such that the minimum distance of these codes will increase in proportion to k. The fraction of the file examined is at most $C(k,m+\lambda)2^{-k}$. This fraction goes to zero as k goes to infinity, since the expected value of m goes to $(1-p)$ and λ remains a fixed fraction of k.

5.4. OPTIMAL BUCKET SHAPES

The above algorithm has been previously published, as noted before. The following theorem demonstrates its optimality.

Theorem 9. For answering best-match queries from a file of binary records, $\alpha(h)$ is minimized over all balanced hash functions h having a given number b of buckets if each bucket is shaped like a "sphere" -- that is, if each bucket $B_j(\mathbf{R}_k)$ consists of a center point (record) R'_j and all records within a distance λ of R'_j.

Proof: Since we are considering balanced b-bucket hash functions, $\alpha(h)$ will be minimized if each individual bucket of size $2^k/b$ has a minimal probability of being examined, over all buckets of the same size. A bucket must be examined if it contains any records as close to the input record as the closest record found previously in the search. There are 2^k possible input queries. For a given query, there is a probability of $(1-p)^{V(k,d-1)}$ that the nearest record to the input query will be at a distance of at least d. For a given bucket B let S(B,d) be the set of records in $\mathbf{R}_k$ which are at distance d from the nearest record in $B(\mathbf{R}_k)$. The chance that B must be examined is then:

$$\Psi(B) =_{\text{def}} 2^{-k} \Sigma_{0 \le d \le k} \; |S(B,d)| \, (1-p)^{V(k,d-1)} \, . \tag{110}$$

since if the query is in S(B,d), B is only examined if the sphere of radius d-1 around the query contains no records in $\mathbf{F}$. This sum is minimized by making the values of |S(B,d)| as small as possible for small values of d, since $(1-p)^{V(k,d-1)}$ is a decreasing function of d. In fact, if we are given two buckets B and B′, then $\Psi(B)$ will be less than $\Psi(B')$ if and only if the vector $(|S(B,0)|,|S(B,1)|, \ldots, |S(B,k)|)$ is lexicographically less than the corresponding vector for B′, since

$$\Psi(B') - \Psi(B) = 2^{-k} \Sigma_{0 \le i \le k}(|S(B',i)|-|S(B,i)|)(1-p)^{V(k,i-1)}. \tag{111}$$

Assume that $|S(B,i)|=|S(B',i)|$ for $0 \le i<j$, and that $|S(B,j)|<|S(B',j)|$. Since

$$|S(B',j)|-|S(B,j)| = \Sigma_{j<i\le k} (|S(B,i)|-|S(B',i)|)\ , \tag{112}$$

we have

$$\Psi(B') - \Psi(B) = 2^{-k} \Sigma_{j\le i\le k}(|S(B',i)|-|S(B,i)|)(1-p)^{V(k,i-1)} \tag{113}$$

$$\ge 2^{-k} (|S(B',j)|-|S(B,j)|)((1-p)^{V(k,j-1)}-(1-p)^{V(k,j)}) \tag{114}$$

$$\ge 0 \tag{115}$$

For two buckets of the same size, B will be examined less frequently than B′ if |S(B,1)| is less than |S(B′,1)|. Note that |S(B,1)| is the discrete analog of the surface area of a region B, so that what we are about to show is that a sphere has minimal surface area.

Consider the mapping $\mathbf{R}_k \rightarrow \mathbf{R}_{k-1}$, obtained by dropping the first bit of each record in $\mathbf{R}_k$. The set of records in B may be divided into two subsets according to their first bit. Dropping the first bit, we get two subsets B_0, B_1 of $\mathbf{R}_{k-1}$ corresponding to the set B in $\mathbf{R}_k$. Using $|S(B_0,1)|$ and $|S(B_1,1)|$ to denote the surface area of the sets B_0 and B_1 in $\mathbf{R}_{k-1}$, we have the relationships:

$$|B| = |B_0| + |B_1|, \text{ and} \tag{116}$$

$$|S(B,1)| = |S(B_0,1)| + |S(B_1,1)| + |B_0 - (B_1 \cup S(B_1,1))| + |B_1 - (B_0 \cup S(B_0,1))| \tag{117}$$

The problem of selecting the optimal set B from $\mathbf{R}$ is thus reduced to the problem of selecting the proper sets B_0 and B_1 from $\mathbf{R}_{k-1}$.

Consider a given bucket B' (or rather, its corresponding sets B'_0 and B'_1 in $\mathbf{R}_{k-1}$). Let us "deform" this bucket into a new bucket B by making B_0 and B_1 be sets of the same size as B'_0 and B'_1 but which are spheres centered at the origin of $\mathbf{R}_{k-1}$. We will show that

$$|S(B,1)| \leq |S(B',1)|, \tag{118}$$

using an inductive proof on the dimension k; thus $|S(B_0,1)|$ and $|S(B_1,1)|$ can be assumed to be minimal over all buckets in $\mathbf{R}_{k-1}$ of the same sizes.

We may assume that $|B_0| \geq |B_1|$ without loss of generality. Thus the last term of (117) will be zero since $B_1 \subseteq B_0$. It is now clear that $|S(B,1)|$ is minimal over buckets B such that $|B_0|=|B_0'|$ and $|B_1|=|B_1'|$, since any decrease in the term $|B_0 - (B_1 \cup S(B_1,1))|$ could only come at the expense of a corresponding increase in the term $|S(B_1,1)|$. That is, either $|B_0 - (B_1 \cup S(B_1,1))|$ is zero (in which case $|S(B,1)|$ is obviously minimal) or else $B_1 \cup S(B_1,1) \subset B_0$. In the latter case there will exist several choices for B_1 which will minimise $S(B,1)$ (in fact, any B_1 will do which maintains $B_1 \cup S(B_1,1) \subset B_0$), one of which is a sphere.

Thus a sphere will have minimal surface area of any bucket of size $|B'|$, since a sequence of the above deformations using each of the k bit positions in turn will transform any bucket B' into a sphere. Although other shapes may also have minimal surface area, the sphere will also have the minimal expected chance of being examined, since $B \cup S(B,1)$ is also a sphere when B is, so that the vector $(|S(B,1)|, |S(B,2)|, \ldots, |S(B,k)|)$ is lexicographically minimal by induction on the index j of the $S(B,j)$'s .

Q. E. D.

CHAPTER 6

APPENDIX - NOTATION

The following notation is used consistently throughout:

SYMBOL	MEANING
$A(k,w,t)$	The minimal number of buckets examined to answer a query with t bits given out of k, with $b = 2^W$ buckets in the system.
b	Number of buckets used in a hash-coding scheme.
$C(m,n)$	The binomial coefficient "m choose n".
$E(x)$	The expected value of the variable x.
$\mathbf{F}$	The current file.
h	A hash function mapping $\mathbf{R} \rightarrow \{1,2,\ldots b\}$.
iff	"if and only if"
k	The number of keys in a record.
n	The number of records in the file $\mathbf{F}$.
$\mathbf{Q}$	The universe of legal queries.
Q	A query in $\mathbf{Q}$.
Q_i	A query in $\mathbf{Q}$, _or_ a function mapping subsets $\mathbf{F}$ of $\mathbf{R}$ into subsets of $\mathbf{F}$ (that is, $Q_i(\mathbf{F})$ is the _response_ to query Q_i, given $\mathbf{F}$).

$\mathbf{Q}_t$	The set of all partial match queries having exactly t keys given.
$\mathbf{R}$	The universe of legal records.
$\mathbf{R}_k$	The set of all binary words of length k.
R	A record of $\mathbf{R}$.
R_i	The i-th record of the file $\mathbf{F}$.
r_{ij}	The j-th key of record R_i.
t	The number of keys specified in a partial match query.
v_j	The number of values the j-th key of a record can have.
v	The common value of all the v_j's, if it exists.
$V(k,i)$	The number of points in binary k-space within distance i of the origin.
w	The value $\log_2(b)$.
$\|X\|$	The cardinality of the set X.
$\lceil x \rceil$	The least integer greater than or equal to x.
$\lfloor x \rfloor$	The greatest integer less than or equal to x.
$\times_i A_i$	The cartesian product of sets A_i.
$x << y$	The value of x is "very much less" than the value of y.
$\alpha(h)$	The average number of buckets examined by SEARCH when using hash function h to answer a partial match query $Q \in \mathbf{Q}$.

$\alpha(h,t)$	The average number of buckets examined by SEARCH when using hash function h to answer a partial match query $Q \in \mathbf{Q}_t$.
$\beta(h,t)$	The worst case number of buckets examined by SEARCH when using hash h to answer a partial match query $Q \in \mathbf{Q}_t$.
$\Phi(B)$	The number of queries in $\mathbf{Q}$ which examine bucket B.
$\Phi_t(B)$	The number of queries in $\mathbf{Q}_t$ which examine bucket B.

CHAPTER 7

REFERENCES

[Ab68] Abraham, C. T., S. P. Ghosh, and D. K. Ray-Chaudhuri. File Organization Schemes Based on Finite Geometries. Information and Control 12(February 1968), 143-163.

[Ar71] Arisawa, Makoto. Residue Hash Method. J. Inf. Proc. Soc. Japan 12(1971), 163-167.

[Bo52] Bose, R. C., and W. S. Connor. Combinatorial Properties of Group Divisible Incomplete Block Designs. Ann. Math. Statist. 23(1952), 367-383.

[Bo69a] Bose, R. C., C. T. Abraham, and S. P. Ghosh. File Organization of Records with Multiple-Valued Attributes for Multi-attribute Queries. Combinatorial Mathematics and its Applications. (1969) UNC Press. 277-297.

[Bo69b] Bose, R. C. and Gary G. Koch. The Design of Combinatorial Information Retrieval Systems for Files with Multiple-Valued Attributes. SIAM J. Appl. Math. 17(Nov. 1969), 1203-1214.

[Bu73] Burkhard, W. A., and R. M. Keller. Some Approaches to Best-Match File Searching. CACM 16(April 1973), 230-236.

[Ch69] Chow, David K. New Balanced-File Organization Schemes. Information and Control 15(1969), 377-396.

[Co52] Connor, W. S. Jr. On the Structure of Balanced Incomplete Block Designs. Ann. Math. Statist. 23(1952), 57-71.

[Da65] Davis, D. R. and A. D. Lin. Secondary Key Retrieval Using an IBM 7090-1301 System. CACM 8(April 1965), 243-246.

[de59] de la Briandais, Rene. Proc. Western Joint Computer Conference 15(1959), 295-298.

[De70] Denning, Peter J. Virtual Memory. Computing Surve~~f~~ys 2(Sept. 1970), 153-189

[Fe69] Feldman, Jerome A. and Paul D. Rovner. An Algol-Based Associative Language. CACM 12(August 1969), 439-449.

[Fe50] Feller, William. An Introduction to Probability Theory and its Applications. Vol. 1. Wiley & Sons(1950).

[Fi69] File Organization - Selected Papers from File 68 - An I. A. G. Conference. (Occasional Publication #3. IFIP Administrative Data Processing Group (IAG)) Swets & Zeitlinger N. V. (Amsterdam, 1969)

[Fr60] Fredkin, E. Trie Memory. CACM 3(1960), 490-500.

[Gh68] Ghosh, Sakti P. and C. T. Abraham. Application of Finite Geometry in

File Organization for Records with Multiple-Valued Attributes. IBM Journal of Research and Development 12(March 1968), 180-187.

[Gh69] Ghosh, Sakti P. Organization of Records with Unequal Multiple Valued Attributes and Combinatorial Queries of Order 2. Information Sciences 1(Oct. 1969), 363-380.

[Gh71] Ghosh, Sakti P. File Organization: Consecutive Storage of Relevant Records on a drum type storage. IBM Research Report RJ 895(July 1971), 26 pp.

[Gh72] Ghosh, Sakti P. File Organization: The Consecutive Retrieval Property. CACM 15(Sept. 1972), 802-808.

[Gr59] Gray, H. J. and N. S. Prywes. Outline for a Multi-list organized system. Annual Meeting of the ACM. Paper 41. Cambridge, Mass. (1959) 7 pp.

[Gu69] Gustafson, Richard Alexander. A Randomized Combinatorial File Structure for Storage and Retrieval Systems. Ph.D. Thesis. University of South Carolina(1969), 92 pp.

[Gu71] Gustafson, Richard A. Elements of the Randomized Combinatorial File Structure. Proc. of the Symposium on Inf. Storage and Retrieval, ACM SIGIR, Univ. of Maryland (April 1971), 163-174.

[Gw68] Gwehenberger, G. Anwendung einer binären Verweiskettenmethode beim Aufbau von Listen. Elektron. Rechenanl. 10 (1968), 223-226.

[Ha59] Hardy, G. H., J. E. Littlewood, and G. Polya. Inequalities. Cambridge University Press(1959).

[Ha63] Hanan, M. and F. P. Palermo. An Application of Coding Theory to a File Address Problem. IBM Journal of Research and Development 7(April 1963), 127-129.

[Hs70] Hsiao, David and Frank Harary. A Formal System for Information Retrieval from Files. CACM 13 (February 1970), 67-73.

[Ja71] Jardine N., and C. J. Rijsbergen. The Use of Hierarchic Clustering in Information Retrieval. Info. Stor. Retr. 7(1971), 217-240.

[Jo61] Johnson, L. R. An Indirect Chaining Method for Addressing on Secondary Keys. CACM 4(May 1961), 218-222.

[Ki61] Kiseda, J. R., H. E. Peterson, W. E. Seelback, and M. Teig. A Magnetic Associative Memory. IBM Journal of Research and Development 5(April 1961), 106-121.

[Kn71] Knott, Gary D. Hashing Functions and Hash Table Storage and Retrieval. Stanford University Computer Science Department and National Institutes of Health - Division of Computer Research and Technology. (Draft copy: 1971). 96 pp.

[Kn72] Knuth, Donald E. The Art of Computer Programming 3(Sorting and Searching). Addison-Wesley (1972).

[Ko69] Koch, Gary G. A Class of Covers for Finite Projective Geometries which are Related to the Design of Combinatorial Filing Systems. JCT 7(1969), 215-220.

[Le69] Lefkowitz, David. File Structures for On-Line Systems. Spartan Books (1969), 215 pp.

[Ma46] Mauchly, John. Theory and Techniques for the Design of Electronic Digital Computers. (ed. by G. W. Patterson) 1(1946), 9.7-9.8; 3(1946), 22.8-22.9.

[Ma68] Martin, Laurence D. A Model for File Structure Determination for Large On-Line Files. File Organization: selected papers from File 68 - an I. A. G. Conference. pp. 223-245.

[Mi69] Minsky, Marvin and Seymour Papert. Perceptrons: An Introduction to Computational Geometry. The MIT Press. (Cambridge, Mass. 1969).

[Mi71] Minker, Jack. An Overview of Associative or Content Addressable Memory Systems and a KWIC Index to the Literature. Technical Report TR-157. University of Maryland Computer Science Center. (College Park, Md: June 1971), 140 pp.

[Mi72] Minker, Jack. Associative Memories and Processors: A Description and Appraisal. University of Maryland and Auerbach Corp. Technical Report TR-195(July 1972), 70 pp.

[Ni72] Nishihara, Seiichi, Shoichiro Ishigaki, and Hiroshi Hagiwara. A Variant of Residue Hashing Method of Associative Multiple-Attribute Data. Report A-9. Data Processing Center Kyoto University. Kyoto, Japan. (February 1972), 19 pp.

[Pe72] Peterson, William W., and E. J. Weldon. Error-Correcting Codes. MIT Press (1972).

[Pr63] Prywes, Noah S. and H. J. Gray. The Organization of a Multilist-Type Associative Memory. IEEE Transactions on communication and electronics 68(Sept. 1963), 488-492.

[Pr66] Prywes, Noah S. Man-Computer Problem Solving with Multilist. Proc. IEEE 54(December 1966), 1788-1801.

[Ra68] Ray-Chaudhuri, D. K. Combinatorial Information Retrieval Systems for Files. SIAM J. Appl. Math. 16(Sept. 1968), 973-992.

[Re69] Renyi, Alfred. Lectures on the Theory of Search. Institute of Statistics Mimeo Series No. 600.7, Dept. of Statistics, University of North Carolina ar Chapel Hill. (May 1969), 80 pp.

[Ru72] Rudolph, Jack A. A Production Implementation of an Associative Array Processor - STARAN. Proc. Fall Joint Comp. Conf. (1972), 229-241.

[Sc63] Schay, G., and N. Raver. A Method for Key-to-Address Transformation. IBM Journal of Research and Development 7(April 1963), 121-126.

[Sl57] Slade, A. E., and H. O. McMahon. The Cryotron Catalog Memory System. Proc. 1957 Eastern Joint Computer Conference 10(1957), 115-120.

[Sl64] Slade, A. E. A Discussion of Associative Memories from a Device Point of View. American Documentation Institute 27th Annual Meeting, (1964).

[Ti73] Tietäväinen, Aimo. On the Non-existence of Perfect Codes over Finite Fields. SIAM J. Appl. Math. 24(Jan. 1973), 88-96.

[Wi60] Windley, P. F. Trees, Forests, and Rearranging. Comp. J. 3(1960), 84-88.

[We71] Welch, Terry A. Bounds on the Information Retrieval Efficiency of Static File Structures. Project MAC Report MAC-TR-88. MIT(June 1971) (Ph. D. Thesis) 163 pp.

[Wo71] Wong, Eugene, and T. C. Chiang. Canonical Structure in Attribute Based File Organization. CACM 14(September 1971), 593-597.