SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

PARTIAL-MATCH RETRIEVAL ALGORITHMS*

RONALD L. RIVEST†

Abstract. We examine the efficiency of hash-coding and tree-search algorithms for retrieving from a file of k-letter words all words which match a partially-specified input query word (for example, retrieving all six-letter English words of the form S**R*H where "*" is a "don't care" character).

We precisely characterize those balanced hash-coding algorithms with minimum average number of lists examined. Use of the first few letters of each word as a list index is shown to be one such optimal algorithm. A new class of combinatorial designs (called associative block designs) provides better hash functions with a greatly reduced worst-case number of lists examined, yet with optimal average behavior maintained. Another efficient variant involves storing each word in several lists.

Tree-search algorithms are shown to be approximately as efficient as hash-coding algorithms, on the average.

In general, these algorithms require time about $O(n^{(k-s)/k})$ to respond to a query word with s letters specified, given a file of n k-letter words. Previous algorithms either required time $O(s \cdot n/k)$ or else used exorbitant amounts of storage.

Key words. searching, associative retrieval, partial-match retrieval, hash-coding, tree-search, analysis of algorithms

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)]

1. Introduction.

We examine algorithms for performing partial-match searches of a direct-access file to determine their optimal forms and achievable efficiency. First we present a model for file and query specifications, within which we will analyze various search algorithms. Section 2 discusses the historical development of the "associative search" problem and reviews previously published search algorithms. Section 3 examines generalized hash-coding search algorithms, and § 4 studies a tree-search algorithm.

We begin with a general outline of an information retrieval system, and then proceed to define our problem more precisely.

An information retrieval system consists of the following parts:

(i) a collection of information, called a *file*. An individual unit of a file is called a *record*.

(ii) a storage or recording procedure by which to represent a file (in the abstract) on a physical medium for future reference. This operation we call *encoding* a file. The encoded version of a file must, of course, be distinguishable

* Received by the editors February 21, 1974, and in revised form February 19, 1975. This work was supported by the National Science Foundation under Grants GJ-331-70X and GJ-43634X.

† Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

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 the *encoding* or *data structure* problem.

(iii) a method by which to access and read (or decode) an 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 record can be independently accessed in (approximately) unit time, we say the file is stored on a direct-access storage device. (Magnetic disks are direct-access, whereas magnetic tapes are not.) The access time 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 may depend on the relative location of the last item of information read.

(iv) a user of the system, who has one or more queries (information requests) to pose to the system. *The response to a query is assumed to be a subset of the file*—that is, the user expects some of the file's records 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. 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 the 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 paper. 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 $(r_1, r_2, \cdots, r_k)$ of values chosen from some finite set Σ. That is, each record is an ordered list of k *keys*, or *attributes.* For convenience, we assume that $\Sigma = \{0, 1, \cdots, v-1\}$, so that the set Σ^k is the set of all k-letter words over the alphabet Σ, and has size v^k. We shall generally restrict our attention to the case $v = 2$, so that the set Σ^k is just the set of k-bit words. Since any record type can be encoded as a binary string, this entails no great loss in generality. Furthermore, it seems to be the case that binary records are the hardest type for which to design partial-match retrieval algorithms, since the user has the greatest flexibility in specifying queries for a given number of valid records.

A *file* F is defined to be any nonempty subset of Σ^k.

These definitions are not the most general possible; however, they do model a significant fraction of practical applications. A more general scheme, such as that proposed by Hsiao and Harary [25], would define a record to be an unordered collection of (attribute, value) pairs, rather than an ordered list of values for a predetermined set of attributes, as we have chosen here. Making realistic assumptions about the frequency of occurrence of each attribute, and related problems, seems to be difficult, and we shall not pursue these questions in this paper.

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

The following sections categorize various query types and describe the particular query types to be considered in this paper.

1.2.1. Intersection queries. The most common query type is certainly the *intersection query*, named after a characteristic of its response definition: a record in F is to be retrieved if and only if it is in a specified subset $q(\Sigma^k)$ of Σ^k, so that

$$q(F) \overset{\text{def}}{=} F \cap q(\Sigma^k).$$

(This notation is consistent since $F = \Sigma^k$ implies $q(F) = q(\Sigma^k)$.) The sets $q(\Sigma^k)$ completely characterize the functions $q(F)$ for any file F by the above intersection formula. Intersection queries enjoy the property that whether some record $r \in F$ is in $q(F)$ does not depend upon the rest of the file (that is, upon $F - \{r\}$) 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 $q(\Sigma^k)$ contains just a single record of Σ^k. An exact-match query thus asks whether a specific record is present in F.

2. *Single-key queries.* $q(\Sigma^k)$ contains all records having a particular value for a specified attribute. For example, consider the query defined by $q(\Sigma^k) = \{r \in \Sigma^k | r_3 = 1\}$.

3. *Partial-match queries.* A "partial-match query q with s keys specified" (for some $s \leqq k$) is represented by a record $\hat{r} \in R$ with $k - s$ keys replaced by the special symbol "*" (meaning "unspecified"). If $\hat{r} = (\hat{r}_1, \cdots . \hat{r}_k)$, then for $k - s$ values of j, we have $\hat{r}_j =$ "*". The set $q(\Sigma^k)$ is the set of all records agreeing with $\hat{r}$ in the specified positions. Thus

$$q(\Sigma^k) = \{r \in \Sigma^k | (\forall j, 1 \leqq j \leqq k)[(\hat{r}_j = {}^*) \vee (\hat{r}_j = r_j)]\}.$$

A sample application might be a crossword 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 Q_s throughout to denote the set of all partial-match queries with s 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

$$q(\Sigma^k) = \{r \in \Sigma^k | (1 \leqq r_1 \leqq 3) \wedge (1 \leqq r_2 \leqq 4)\}.$$

5. *Best-match queries with restricted distance.* These require that a distance function d be defined on Σ^k. Query q will specify a record $\hat{r}$ and a distance λ, and have

$$q(\Sigma^k) = \{r \in \Sigma^k | d(r, \hat{r}) \leqq \lambda\}.$$

Query q requests all records within distance λ of the record $\hat{r}$ to be retrieved. the distance function $d(r, \hat{r})$ is usually defined to be the number of attribute positions for which r and $\hat{r}$ 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

$$q(\Sigma^k) = \{r \in \Sigma^k | ((r_1 = 0) \vee (r_2 = 1)) \wedge (r_3 \neq 3)\}.$$

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 $q(\Sigma^k)$ of Σ^k (the *characteristic function* of $q(\Sigma^k)$).

Note that each intersection query requires *total recall;* that is, *every* record in 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 requests the retrieval of all the nearest neighbors in F of a record $\hat{r} \in \Sigma^k$ using the Hamming distance metric d over Σ^k. Performing a pure best-match search is equivalent to decoding the input word $\hat{r}$ into one or more of the "code words" in F, using a maximum likelihood decoding rule (see Peterson [35]). Thus we have

$$q(F) = \{r \in F | (\forall r' \in F)(d(r', \hat{r}) \geqq d(r, \hat{r}))\}.$$

1.2.3. Query types to be considered. In this paper we shall only consider partial-match queries. The justification for this choice is that this query is quite common yet has not been "solved" in the sense of having known optimal search algorithms to answer it. In addition, partial- and best-match query types are usually considered as the paradigms of "associative" queries. The simpler intersection query types 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 $|\Sigma^k|$ 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 a batch 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 practicability 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 the study of this paper.

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 access 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 *not* 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. We give a brief exposition of the historical development of "associative" search algorithms in § 2.

In § 3 we study generalized hash functions as a means for answering partial-match queries. We derive a lower bound on their achievable performance, and precisely characterize the class of optimal hash functions. We then introduce a new class of combinatorial designs, called associative block designs. Interpreted as hash functions, associative block designs 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 efficiency gains obtainable by storing each record more than once).

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 the ith branching decision is made only according to the ith bit of the specific record being inserted or searched for, and not according to the results of comparisons between that record and another record 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 s, the average time required by any algorithm to answer a single partial match query $q \in Q_s$ must be at least

$$cn^{(k-s)/k},$$

where the average is taken over all queries $q \in Q_s$ and all files F of n k-bit records which are represented efficiently on a direct-access storage device. (That is, no more than $O(n \cdot k)$ bits of storage are used.)

2. Historical background.

2.1. Origins in hardware design. Associative search problems were first discussed by people interested in building associative memory *devices.* According to Slade [44], the first associative memory design was proposed by Dudley Buck in 1955. 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 [12]). Minker [32] has written an excellent survey of the development of associative processors.

2.2. Exact-match algorithms. New search algorithms for use on a conventional computer with random-access memory were also being rapidly discovered in the 1950's. 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 [30]. The use of binary trees for searching was invented in the early 1950's according to [28], with published algorithms appearing around 1960 (see, for example, Windley [46]—there were also many others).

Tries were first described about the same time by Rene de la Briandais [11]. Tries are roughly as efficient as binary trees for exact-match searches. We shall examine trie algorithms for performing partial-match searches in § 4.

Hash-coding (invented by Luhn around 1953, according to Knuth [28]) seems the best solution for many applications. Given b storage locations (with $b \geqq |F|$) in which to store the records of the file, a *hash function* $h : \Sigma^k \to \{1, 2, \cdots, b\}$ is used to compute the address $h(r)$ of the storage location at which to store each record r. 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 [28, § 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 is stored in list L_j (where $1 \leqq j \leqq b$) iff $h(r) = j$. Each bucket can now store an arbitrary number of records, so collisions are no longer a problem. To determine if an arbitrary record $r \in \Sigma^k$ is in the file, one need merely examine the contents of list $L_{h(r)}$ to see if it is present there. Since the expected size of each list is small, very little work need be done. Chaining can be implemented easily with simple linear linked list techniques (see [28, § 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 [26] proposed the use of k distinct hash functions h_i and k sets of lists L_j^i—for $1 \leqq i \leqq k$ and $1 \leqq j \leqq b$. Record r is stored in k lists—list $L_{h_i(r_i)}^i$ for $1 \leqq i \leqq k$. 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 associated with a unique list through the use of indices (search trees) instead of hash functions (see [20], [36]). Davis and Lin [10] describe another variant in which list techniques are replaced by compact storage of the record addresses relevant to each bucket. The above methods are often called *inverted list* 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 the number s of keys specified in the query is small. This is because the response to a query is the intersection of s lists of the inverted list system. 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 if the expected size $E(|q(F)|)$ of the response decreases. One might even hope to do 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 §§ 3 and 4, while nonlinear, easily outperform inverted list techniques. These algorithms do an amount of work that decreases approximately exponentially with s. When $s = 0$, *the whole file must, of course, be searched, and when* $s = k$, unit work must be done. In between, log(work) decreases approximately linearly with s.

J. A. Feldman's and P. D. Rovner's system LEAP [13] 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 use a very large number of short lists so that each query's response will be the union of some of the lists, instead of an intersection. Wong and Chiang [47] discuss this approach in detail. Note, however, that the requisite number of lists is at least $|\Sigma^k|$ 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 lists (most of them empty if $|F| < |\Sigma^k|$, 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 s is not allowed to exceed some fixed small value s' (for example, $s' = 3$). Then the number of lists required is $\binom{k}{s'} v^{s'}$. This is achieved by reserving a bucket for the response to each query with the maximal number s' of keys given; the response to other queries is then the union of existing buckets. Note that each record is now stored in $\binom{k}{s'}$ buckets, however! One can 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 [21], [22]. Gustafson assumes that each record r is an *unordered* list $\{r_1, r_2, \cdots\}$ of at most k' attribute values (these might be key words, where the records represent documents) chosen from a very large universe of possible attribute values. Let w be chosen so that $\binom{w}{k'}$ is a reasonable number of lists to have in the system, and let a hash function h map attribute values into the range $\{1, 2, \cdots, w\}$. Each list is associated with a unique w-bit word containing exactly k' ones, and each record r is stored on the list associated with the word with ones only in positions $h(r_1), h(r_2), \cdots, h(r_{k'})$. (If these are not all distinct positions, extra ones are added randomly until there are exactly k' ones.) A query specifying attributes $a_1, a_2, \cdots, a_t$ (with $t \leqq k'$) need only examine the $\binom{w-t}{k'-t}$ lists associated with words with ones in positions $h(a_1), h(a_2), \cdots, h(a_t)$. The amount of work thus decreases rapidly with t. Note that the query response is not merely the union of the relevant lists, since undesired records may also be stored on these lists. We are guaranteed, however, that all the desired records are on the examined lists. 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 on only one list (so updating is simple), 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 ours, however, so that the queries allowable in his system are a proper subset of our partial match queries—those with no zeros specified. (Convert each of his records into a long bitstring with ones in exactly k' places—each bit position corresponding to a permissible key word in the system.)

Terry Welch, in his Ph.D. thesis [45], 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 directory-less files, and derives a lower bound on the average time required to perform a partial match search with hash-coding methods that is much lower than the precise answer given in § 3. He also presents Elias's algorithm for handling best-match queries, which the author shows to be minimize the average number of lists examined in [40] and [41].

3. Hashing algorithms for partial-match queries. Given a set Σ^k of possible records, and a number b of lists (buckets) desired in a filing scheme, we wish to construct a hash function $h : \Sigma^k \to \{1, \cdots, b\}$ so that partial match queries can be answered efficiently (either on the average or in the worst case). A record r in our file F is stored in list L_j if and only if $h(r) = j$, with collisions handled by separate chaining. We define block B_j of the partition of Σ^k induced by h to be

$$B_j = \{r \in \Sigma^k | h(r) = j\},$$

so that

$$L_j = B_j \cap F.$$

If $|B_j| = |\Sigma^k|/b$ for all j, then we say that h is a *balanced* hash function.

To answer a partial-match query $q \in Q$, we must examine the contents of the lists with indices in

$$h(q) = \{j | (B_j \cap q(\Sigma^k)) \neq \text{null}\},$$

or equivalently,

$$h(q) = \bigcup_{r \in q(\Sigma^k)} \{h(r)\}.$$

Here we make the natural extension of h onto the domain Q.

The basic retrieval algorithm can thus be expressed

$$q(F) = \bigcup_{i \in h(q)} (L_i \cap q(\Sigma^k)),$$

or equivalently in pseudo-ALGOL:

```
Procedure SEARCH ($q$, $h$, $\{L_1, \cdots, L_b\}$);
comment SEARCH prints the response to a partial-match query $q \in Q$, given
    that the file $F \subseteq \Sigma^k$ is stored on lists $\{L_1, \cdots, L_b\}$ according to hash
    function $h$. ;
begin integer $i$; record $r$;
    for each $i \in h(q)$ do
        for each $r \in L_i$ do
            if $r \in q(\Sigma^k)$ then print ($r$)
end SEARCH;
```

The difficulty of computing the set $h(q)$ depends very heavily on the nature of the hash function h. It is conceivable that for some pseudo-random hash functions h, it is more time-consuming to determine whether $i \in h(q)$ than it is to read the entire list L_i from the secondary storage device. Such hash functions are, of course, useless, since one would always skip the computation of $h(q)$ and read the entire file to answer a query. We will restrict our attention to hash functions h for which the time required to compute the set $h(q)$ of indices of lists that need to be searched is negligible in comparison with the time required to read those lists.

We denote the average and worst-case number of lists examined by SEARCH to answer a partial-match query with s keys specified, given that the file was stored using hash function h, by

$$A_s(h)=|Q_s|^{-1}\sum_{q\in Q_s}|h(q)| \quad \text{and} \quad W_s(h)=\max_{q\in Q_s}|h(q)|,$$

respectively.

We shall denote the average number of lists examined, taken over all partial-match queries in Q, by $A(h)$. We denote the minimum possible average number of lists examined for a query $q\in Q_s$ by $A_{\min}(k, w, s, v)$ where h is assumed to be a balanced hash function mapping $\Sigma^k=\{0, 1, \cdots, v-1\}^k$ onto $\{1, 2, \cdots, b\}$, where $b=2^w$, w being a natural number. If v is omitted, $v=2$ is to be assumed.

Note that the number of lists examined does not depend on the particular file F being searched, but only on the hash function h being used.

3.1. Hash functions minimizing average search time. We first concentrate on the average number of buckets examined by SEARCH to calculate the response to a partial-match query q. That is, we consider the functions $A(h)$ and $A_s(h)$ and determine for which hash functions h are $A(h)$ and $A_s(h)$ minimized.

In § 3.1.1 we consider nonbinary records, (that is, where $|\Sigma|$ is relatively large), and show that a simple string-homomorphism hash function is optimal.

In § 3.1.2 we concentrate on binary records, for which a straightforward (ε-free) homomorphism technique would require using an exorbitant number of buckets (essentially one list per record in Σ^k). A relatively complicated combinatorial argument shows that selecting a subset of the record bits for use as address bits is optimal.

3.1.1. Optimal hash functions for nonbinary records. When k (the number of letters in each record) is less than or equal to w (the number of bits needed to describe a list index), then a very simple string-homomorphism scheme will provide efficient retrieval. This case arises frequently for nonbinary records.

We illustrate the principle by means of an example. Suppose we wish to construct a "crossword puzzle" dictionary for six-letter English words. Let $b=2^w$ be the number of lists used. Given a word (for example, "SEARCH"), we construct a w-bit list index (bucket address) by forming the concatenation:

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

of six $(w/6)$-bit values; here g is an auxiliary hash function mapping the alphabet Σ into $(w/6)$-bit values.

For a partial-match query with s letters specified, we have, in this case,

$$A_s(h)=W_s(h)=2^{w-s(w/6)}.$$

This approach is clearly feasible as long as $b\geqq 2^k$, since one or more bits of the list index can be associated with each attribute.

A similar technique has been proposed by M. Arisawa [2], in which the ith key determines, via an auxiliary hash function, the residue class of the list index modulo the ith prime (see also [34]).

For nonbinary records, the analysis fortunately turns out to be simpler than for binary records. In this section we prove that schemes that use the above string-homomorphism idea are, in fact, optimal.

Let the universe of records be Σ^k, where $\Sigma=\{0, 1, \cdots, v-1\}$ and $v>2$. Let $Q_{\min}(x, k)$ denote the minimum possible number of partial-match queries in Q which require examination of a list L whose corresponding block B has size x. The following inequality gives a lower bound for $Q_{\min}$.

$$Q_{\min}(x, k) \geqq \min\,(Q_{\min}(\max_i f_i, k-1) + \sum_{0 \leqq i < v} Q_{\min}(f_i, k-1)),$$

where the minimum is taken over all sets of nonnegative integers $f_0, \cdots, f_{v-1}$ such that $\sum_{0\leqq i<v} f_i = x$. (Here f_i denotes the number of records in B that begin with an i.) The term $Q_{\min}(\max f_i, k-1)$ is a lower bound on the number of partial-match queries beginning with a "*", and $Q_{\min}(f_i, k-1)$ is a lower bound on the number of partial-match queries beginning with "i", that require examination of L.

We can perform the analysis by passing to the continuous case. Define the function $Q'_{\min}$ as follows.

$$Q'_{\min}(x, k) = \inf\,(Q'_{\min}(\sup_{0\leqq z \leqq v} f(z), k-1) + \int_0^v Q'_{\min}(f(z), k-1)\,dz),$$

where the infimum is taken over all nonnegative functions $f(z)$ such that $\int_0^v f(z)\,dz = x$. The appropriate initial conditions in this case are $Q'_{\min}(x, 0)=1$ for $0 \leqq x \leqq 1$ (counting the totally unspecified query), $Q'_{\min}(x, 0)=\infty$ for $x>1$ (this forces the above definition of $Q'_{\min}$ to pick values for $f(z)$ so that all records are distinguished after all the positions are examined), and $Q'_{\min}(x, 0)=0$ otherwise. The function $Q'_{\min}(x, k)$ is obviously a lower bound for $Q_{\min}(x, k)$, since the definition for $Q'_{\min}(x, k)$ reduces to the above lower bound for $Q_{\min}$ (with equality holding) if $f(z)$ is defined $f(z)=f_i$ for $i \leqq z < i+1$.

The solution of the above recurrence for $Q'_{\min}$ is

$$Q'_{\min}(x, k) = \begin{cases} 0 & \text{if } x \leqq 0, \\ (x^{1/k}+1)^k & \text{if } 0 < x \leqq v^k, \\ \infty & \text{otherwise.} \end{cases}$$

The infimum is reached in the definition of $Q'_{\min}(x, k)$ when $f(z)$ is the following function:

$$f(z) = \begin{cases} x^{(k-1)/k} & \text{for } 0 \leqq z \leqq x^{1/k}, \\ 0 & \text{otherwise.} \end{cases}$$

When x is the kth power of some integer y, $1 \leqq y \leqq v$, the lower bound provided by $Q'_{\min}(x, k)$ is, in fact, achievable.

THEOREM 1. *If B is a subset of Σ^k containing $x = y^k$ records, for some integer y, $1 \leqq y \leqq v = |\Sigma|$, then the number $Q(B)$ of queries which require examination of the list L associated with B is minimized when*

$$B = \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_k,$$

where each $\Sigma_i \subseteq \Sigma$ is of size y.

Proof. $Q(B) = Q'_{\min}(x, k)$ in this case. □

Theorem 1 says that our crossword puzzle hashing scheme is optimal, as long as the auxiliary function g divides the alphabet Σ into four equal pieces. This is not possible for the English 26-letter alphabet (Greek puzzlists are in luck with their 24-letter alphabet, however); we conjecture that splitting Σ into four nearly-equal pieces would be optimal in this case.

3.1.2. The optimal hash functions for binary records. In this section we prove a conjecture due to Welch [45]: when $k > \log_2(b)$, an optimal hash function is one which merely extracts the first $\log_2(b)$ bits of each record for use as a list index. As we shall see later, this hash function is only one of many which minimize the average number of lists examined, some of which also do better at minimizing the worst-case behavior as well. Sacerdoti [43] has also suggested that this scheme may be practical for partial-match retrieval.

We shall only consider balanced hash functions in this section. This eliminates considering obviously "degenerate" hash functions mapping all the records into a single list (costing one list examination per search). Furthermore, if each record in Σ^k is equally likely to appear in the file (independent of other records), then the expected length of each list will be the same. A formal justification for the restriction to balanced hash functions will be given later on, after we examine more closely the average search time of optimal balanced hash functions.

The following theorem gives a precise characterization of which balanced hash functions h minimize $A(h)$, the average number of lists examined.

THEOREM 2. *Let $h : \{0, 1\}^k \to \{1, \cdots, b\}$ be a balanced hash function with $b = 2^w$ buckets for some integer w, $1 \leqq w \leqq k$. Then $A(h)$ is minimal if and only if each block B_i is a $q(\{0, 1\}^k)$ for some query $q \in Q_w$.*

The geometry of the sets $q(\{0, 1\}^k)$ thus is reflected in an appealing fashion in the optimal shape for the blocks B_i. The set of records in each optimal B_i can be described by a string in $\{0, 1, *\}^k$ containing exactly w bits and $k - w$ *'s.

The following corollary is immediate.

COROLLARY 1. *The hash function which extracts the first w bits of each record $r \in \{0, 1\}^k$ to use as a list index minimizes $A(h)$.*

The proof of Theorem 2 is unfortunately somewhat lengthy, although interesting in that we prove a little more than is claimed.

Let $B_i \subset \{0, 1\}^k$ be any block. Then by $Q(B_i)$, we denote $|\{q \in Q | (q(\{0, 1\}^k \cap B_i) \neq \emptyset\}|$, the number of queries $q \in Q$ which examine list L_i. Denote by $Q_{\min}(x, k)$ the minimal value of $Q(B_i)$ for any B_i of size x, $B_i \subset \{0, 1\}^k$. We now note that

$$\begin{aligned} A(h) &= |Q|^{-1} \cdot \sum_{q \in Q} |h(q)| \\ &= |\{(B_i, q) | (q(\{0, 1\}^k) \cap B_i) \neq \emptyset\}| \cdot |Q|^{-1} \\ &= \sum_{1 \leqq i \leqq b} Q(B_i) \cdot |Q|^{-1} \geqq b \cdot Q_{\min}(2^{k-w}, \mathrm{k}) \cdot |Q|^{-1}. \end{aligned}$$

To finish the proof of the theorem, we need only show that $Q(B_i) = Q_{\min}(2^{k-w}, k)$ if and only if B_i is of the form $q(\{0, 1\}^k)$ for some $q \in Q_w$. The rest of the proof involves four parts: calculation of $Q(B_i)$ for B_i of the proper form, calculation of

$Q_{\min}(2^{k-w}, k)$, the demonstration of equality, and then the demonstration of the "only if" portion.

Calculation of $Q(B_i)$. For simplicity of notation, we use the symbols x, y, z to denote either positive integers or their k-bit binary representation. Occasionally we may wish to explicitly indicate that some number t of bits is to be used; we denote this string by $x : t$ (so that $9 : 5 = 01001$). The length of a string x in $\{0, 1\}^*$ we denote by $|x|$. Concatenation of strings is represented by concatenation of their symbols; $0x$ denotes a zero followed by the string x. A string of t ones we denote by 1^t.

If x is a string, we denote by $\underline{x}$ the set of those $x+1$ strings of length $|x|$ which denote integers not greater than x. For example, $\underline{011} = \{000, 001, 010, 011\}$. If $x = 2^{k-w} - 1$, then $\underline{x : k}$ describes the set $q(\{0, 1\}^k)$ for $q = 0^w *^{k-w}$, which is in Q_w. Furthermore, $Q(q(\{0, 1\}^k))$ does not depend on which $q \in Q_w$ is chosen; this is always $2^w 3^{k-w}$ (since each $*$ in q can be replaced by 0, 1 or $*$, and each specified position optionally replaced by a $*$, to obtain a query counted in $Q(q(\{0, 1\}^k))$).

Thus $Q(B_i) = 2^w 3^{k-w}$ for $B_i = \underline{x : k}$ with $x = 2^{k-w} - 1$. To show this is optimal, it is necessary to calculate $Q(\underline{x})$ for arbitrary strings x.

LEMMA 1. (a) $Q(\text{nullstring}) = 1$; (b) $Q(\underline{0x}) = 2Q(\underline{x})$; (c) $Q(\underline{1x}) = 2Q(\underline{1^{|x|}}) + Q(\underline{x})$.

Proof. Take (a) to be true by definition. For (b), any query examining $\underline{x}$ can be preceded by either a 0 or a $*$ to obtain a query examining $0x$. Part (c) follows from the fact that $\underline{1x} = \underline{0(1^{|x|})} \cup \underline{1x}$; there are $2Q(\underline{1^{|x|}})$ queries starting with 0 or $*$, and $Q(x)$ starting with 1. □

The preceding lemma permits $Q(\underline{x})$ to be easily computed for arbitrary strings x; we list some particular values:

$x =$ null	0	1	00	01	10	11	
$Q(\underline{x}) =$ 1	2	3	4	6	8	9	
$x = 000$	001	010	011	100	101	110	111
$Q(\underline{x}) =$ 8	12	16	18	22	24	26	27

If x is the string $x_1 x_2 \cdots x_k$ and z_i denotes the number of zeros in $x_1 \cdots x_i$, then Lemma 1 implies that

$$Q(\underline{x}) = 2^{z_k} + \sum_{1 \le i \le k} x_i 3^{k-i} 2^{z_i+1}.$$

LEMMA 2. $Q(\underline{x1}) = 3Q(\underline{x})$.

Proof. Directly from the above formula for $Q(\underline{x})$ or by noting that if q is counted in $Q(\underline{x})$, then $q0$, $q1$ and q^* will be counted in $Q(\underline{x1})$. □

Using $p(x)$ to denote 2^{z_k} (where z_k is the number of zeros in x), we have also the following.

LEMMA 3. $Q(\underline{x : k}) = Q(\underline{x - 1 : k}) + p(x : k)$.

Proof. The only queries counted in $Q(\underline{x})$ but not $Q(\underline{x-1})$ will be those obtained by replacing an arbitrary subset of the zeros in x by $*$'s □

Now that we know quite a bit about $Q(q(\{0, 1\}^k))$ for $q \in Q_w$, we turn our attention to $Q_{\min}$.

A lower bound for $Q_{\min}(x, k)$. The following inequality holds for $Q_{\min}(x, k)$, the minimal number of queries $Q(B_i)$ for any $B_i \subset \{0, 1\}^k$, $|B_i| = x$:

$$Q_{\min}(x, k) \geqq \min\,(2Q_{\min}(x_0, k-1) + Q_{\min}(x_1, k-1)),$$

where the minimum is taken over all pairs of nonnegative integers x_0, x_1 such that $x_0 + x_1 = x$, $x_0 \geqq x_1$, and $x_0 \leqq 2^{k-1}$. To show this, let B_i contain x_0 records starting with a 0 and x_1 with a 1. Then $Q(B_i)$ must count at least $2Q_{\min}(x_0, k-1)$ queries beginning with a zero or a $*$, and at least $Q_{\min}(x_1, k-1)$ which start with a 1. (Nothing is lost by assuming $x_0 \geqq x_1$.) For $k = 1$, we have $Q_{\min}(x, 1) = Q(\underline{x-1 : 1})$, by inspection.

Showing $Q(B_i) = Q_{\min}(2^{k-w}, k)$ *if* $B_i = \underline{2^{k-w} - 1 : k}$. We will, in fact, prove the stronger statement that $Q(B_i) = Q_{\min}(x+1, k)$ if $B_i = \underline{x : k}$, by induction on k. Since $Q(\underline{x : k}) \geqq Q_{\min}(x+1, k)$ necessarily, with equality holding for $k = 1$, as we have seen, equality can be proved in general using our lower bound for $Q_{\min}(x, k)$ if we can show the following:

$$Q(\underline{x : k}) \leqq \min\,(2Q(\underline{y : k-1}) + Q(\underline{z : k-1})), \tag{1}$$

where the minimum is taken over all pairs of nonnegative integers y, z such that $y + z + 1 = x$, $y \geqq z$ and $y \leqq 2^{k-1}$. By induction, the right-hand side of (1) is a lower bound for $Q_{\min}(x+1, k)$. The case in (1) of $x = y$ corresponding to $x_1 = 0$ in our lower bound for $Q_{\min}(x, k)$ holds by Lemma 1 (b). To prove (1), we consider four cases, according to the last bits of y and z.

Case 1. $y : k-1 = y'1$, $z : k-1 = z'1$, *and* $x : k = x'1$. In this case, (1) is true by Lemma 2, since we know by induction that $Q(\underline{x'}) \leqq 2Q(\underline{y'}) + Q(\underline{z'})$.

Case 2. $y : k-1 = y'0$, $z : k-1 = z'1$, *and* $x : k = x'0$. If $p(x') \leqq 2p(y')$, then (1) is true since it is equivalent by Lemmas 2 and 3 to

$$3Q(\underline{x'-1}) + 2p(x') \leqq 6Q(\underline{y'-1}) + 4p(y') + 3Q(\underline{z'}),$$

but we know by induction that $Q(\underline{x'-1}) \leqq 2Q(\underline{y'-1}) + Q(\underline{z'})$. Otherwise if $p(x') > 2p(y')$, we use the fact that (1) says

$$3Q(\underline{x'}) - p(x') \leqq 6Q(\underline{y'}) - 2p(y') + 3Q(\underline{z'})$$

and we know that $Q(\underline{x'}) \leqq 2Q(\underline{y'}) + Q(\underline{z'})$ by induction.

Case 3. $y : k-1 = y'1$, $z : k-1 = z'0$, *and* $x : k = x'0$. Depending on the truth or falsity of $p(x') \leqq p(z')$, we use induction and the fact that (1) is implied by either

$$3Q(\underline{x'-1}) + 2p(x') \leqq 6Q(\underline{y'}) + 3Q(\underline{z'-1}) + 2p(z')$$

or

$$3Q(\underline{x'}) - p(x') \leqq 6Q(\underline{y'}) + 3Q(\underline{z'}) - p(z').$$

Case 4. *$y : k-1 = y'0$, $z : k-1 = z'0$, and $x : k = x'1$.* If $p(y') \leqq p(z')$, we use induction and the fact that (1) is implied by

$$3Q(\underline{x'}) \leqq 6Q(\underline{y'}) - 2p(y') + 3Q(\underline{z'-1}) + 2p(z').$$

Otherwise we use the fact that (1) is implied by

$$3Q(\underline{x'}) \leqq 6Q(\underline{y'-1}) + 4p(y') + 3Q(\underline{z'}) - p(z').$$

This completes the proof that $Q(\underline{x : k}) = Q_{\min}(x+1, k)$.

Showing $Q(B_i) = Q_{\min}(2^{k-w}, k)$ only if $B_i = q(\{0, 1\}^k)$ for some $q \in Q_w$. We need only that (1) holds with equality for $x = 2^{k-w}-1$ only if $y = z$, since this implies that $Q(B_i) > Q_{\min}(2^{k-w}, k)$ if B_i is not of the form $q(\{0, 1\}^k)$ for $q \in Q_w$. To see this, let $B_i = 0C \cup 1D$ for $C, D \subset \{0, 1\}^{k-1}$. Then note that if $B_i \neq q(\{0, 1\}^k)$ for some $q \in Q_w$, then either $|C| > 0$, $|D| > 0$, and $|C| \neq |D|$; or $|C| = |D|$ but at least one of C, D is not of the form $q(\{0, 1\}^{k-1})$ for any $q \in Q_{w-1}$.

Now $x : k = 0^w1^{k-w}$. If $y = z$, then (1) holds with equality by Lemma 1. To show (1) holds with equality only if $y = z$, suppose $y = 2^{k-w-1}+t-1$ and $z = 2^{k-w-1}-t-1$ for any t, $0 < t < 2^{k-w-1}$. Then (1) holding with strict inequality says:

$$Q(\underline{0^w1^{k-w}}) < 2Q(\underline{y : k-1}) + Q(\underline{z : k-1})$$

or

$$Q(\underline{01^{k-w}}) < 2Q(\underline{y : k-w}) + Q(\underline{z : k-w})$$

or

$$Q(\underline{01^{k-w}}) < Q(\underline{y : k-w+1}) + Q(\underline{z : k-w}).$$

Subtracting 2^{k-w-1} from both x and y, we get that (1) means

$$Q(\underline{01^{k-w-1}}) < Q(\underline{t-1 : k-w}) + Q(\underline{2^{k-w-1}-t-1 : k-w}).$$

It is simpler to note that the general statement

$$Q(\underline{x : k}) < Q(\underline{t-1 : k}) + Q(\underline{x-t : k})$$

is always true; in fact, it is implied directly by (1), Lemma 1 and the fact that $Q(\underline{x-t : k})$ is always positive. This completes our proof that $Q(B_i) = Q_{\min}(2^{k-w}, k)$ only if $B_i = q(\{0, 1\}^k)$ for some $q \in Q_w$, and also finishes our proof of Theorem 2. □

While Theorem 2 only counted queries in Q, the same result holds if we count queries in Q_s.

THEOREM 3. *Let h and b be as in Theorem* 1. *Then $A_s(h)$ is minimal for $0 < s < k$ if and only if each block B_i is a $q(\{0, 1\}^k)$ for some $q \in Q_w$.*

This can't be asserted in an "iff" manner for $s = 0$ or $s = k$, since $A_0(h) = b$ and $A_k(h) = 1$ independent of h.

Theorem 3 can be proved in the same manner as Theorem 2. We note here the differences, using $Q_s(B_i)$ and $Q_{\min,s}(x, k)$ to count queries in Q_s rather than Q.

LEMMA 4. (a) $Q_0(\underline{x})=1$, *for all x.* (b) Q_s $(\underline{\text{nullstring}})=0$ *for* $s\geqq 1$. (c) $Q_s(\underline{0x})=Q_s(\underline{x})+Q_{s-1}(\underline{x})$, *for* $s\geqq 1$ *and all x.* (d) $Q_s(\underline{1x})=Q_s(\underline{01^{|x|}})+Q_{s-1}(\underline{x})$, *for* $s\geqq 1$ *and all x.*

LEMMA 5. $Q_s(\underline{x})-Q_s(\underline{x-1})=\binom{z_k}{k-s}$, *where* z_k *denotes the number of zeros in x and* $|x|=k$.

LEMMA 6. $Q(\underline{x1})=2Q_{s-1}(\underline{x})+Q_s(\underline{x})$.

LEMMA 7. $Q_{\min,s}(x,k)\geqq\min(Q_{\min,s}(x_0,k-1)+Q_{\min,s}(x_0,k-1)+Q_{\min,s-1}(x_1,k-1))$ *where the minimum is taken over all pairs of nonnegative integers* x_0, x_1, *such that* $x_0+x_1=x$, $x_0\geqq x_1$ *and* $x_0\leqq 2^{k-1}$.

The proofs of these lemmas are omitted here (see [40]). The proof of Theorem 2 then proceeds along the same lines as that of Theorem 1; with (1) being replaced by

$$Q_s(\underline{x:k})\leqq\min(Q_s(\underline{y:k-1})+Q_{s-1}(\underline{y:k-1})+Q_{s-1}(\underline{z:k-1})).$$

We omit details here of the proof, as it varies little from the proof of (1). The "only if" portion of the proof is also similar, except that the inductive hypothesis has to be applied more than once (for varying s values). □

Calculation of $A_s(h)$. Now that Theorems 2 and 3 tell us what the optimal balanced hash functions $h:\{0,1\}^k\rightarrow\{1,\cdots,b=2^w\}$ are like, we can caloulate $A_s(h)$ easily, using the optimal h from Corollary 1 to Theorem 2 (using the first w bits of $x\in\{0,1\}^k$ as the hash value). We get

$$A_{\min}(k,w,s)=A_s(h)=\binom{k}{s}^{-1}\sum_{0\leqq i\leqq s}\binom{w}{i}\binom{k-w}{s-i}2^{w-i}\geqq b^{1-s/k},$$

where the first sum considers the ways in which i of the s specified bits fall in the first w positions; when this happens 2^{w-i} buckets must be searched. The inequality is a special case of a mean value theorem [24, Thm. 59]. In fact, $b^{1-s/k}$ is a reasonable approximation to $A_s(h)$, as long as w is not too small.

A better approximation may be obtained by replacing the hypergeometric probability

$$\binom{k}{s}^{-1}\binom{w}{i}\binom{k-w}{s-i}$$

by its approximation

$$\binom{w}{i}(s/k)^i(1-s/k)^{w-i}.$$

Then we have

$$A_{\min}(k,w,s)\approx\sum_i\binom{w}{i}(s/k)^i(1-s/k)^{w-i}2^{w-i}$$

$$=\sum_i\binom{w}{i}(s/k)^i(2-2\cdot s/k)^{w-i}=(2-s/k)^w.$$

For example, when half of the bits are specified in a query ($s=k/2$), we should expect to look at $(1.5)^w$ lists.

3.1.3. The optimal number of lists. In this section we use the results of the last section to derive the optimal number of lists (buckets) in a hash-coding partial-match retrieval scheme. We also give justification for the use of balanced hash functions.

The basic result of the preceding section was that the optimal shape for a block B is a "subcube" of $\{0, 1\}^k$; that is, $B = q(\{0, 1\}^k)$ for some partial-match query q, under the assumption that each partial-match query is equally likely. Let us now assume that each record $r \in \Sigma^k$ is equally likely to appear in F (equivalently, every file F of a given size is equally likely). In this case, the expected length of a list L_i is just $|F| \cdot |B_i| \cdot 2^{-k}$.

We use the following simple model for the true retrieval cost (rather than just counting the number of lists searched): α seconds are required to access each list L_i, and ρ seconds are required to read each record in L_i. The total time to search L_i is then $\alpha + \rho|L_i|$. Suppose $B_i = q(\{0, 1\}^k)$ for some $q \in Q_w$. The average expected time per partial-match query spent searching list L_i is then just

$$E((\alpha + \rho|L_i|) \cdot (2^w 3^{k-w}) \cdot 3^{-k}) = (\alpha + \rho|F| \cdot |B_i| \cdot 2^{-k}) \cdot (2/3)^w,$$

since there are $2^w 3^{k-w}$ queries which must examine L_i out of 3^k possible queries, and $(E|L_i|) = |F| \cdot |B_i| \cdot 2^{-k}$. To minimize the total average cost (the above summed over lists L_i), it is sufficient to minimize the cost per element of $\{0, 1\}^k$. Dividing the above by $|B_i| = 2^{k-w}$, we want to minimize

$$(2/3)^w 2^{-k}[\alpha \cdot 2^w + \rho|f|].$$

Taking the derivative with respect to w, setting the result to zero and solving for w, we get that the total expected cost per query is minimized when

$$w = \log_2 \left(\frac{-\rho|F| \ln (2/3)}{\alpha \ln (4/3)}\right) \cong \log_2 (\alpha^{-1}\rho|F| \cdot 1.408).$$

(The second derivative is positive, so this is a minimum.)

For example, if $\rho = .001$, $\alpha = .05$, $|F| = 10^6$ and $k = 40$, then the optimal hashing scheme would have $w \approx 15$, so that $2^{15} = 32{,}768$ buckets would be used. The average time per query turns out to be about 6.5 seconds.

Note that the optimal choice of w does not depend on k, due to the fact that the probability $(2/3)^w$ of examining L_i when $|B_i| = 2^{k-w}$ does not depend on k. Since our above analysis only considered a single arbitrary bucket, all buckets should have the same optimal size determined above. That is, the hash function should be balanced.

3.2. Minimizing the worst-case number of lists searched. The worst-case behavior of the hash function of Corollary 1 is obviously poor; if none of the specified bits occur in the first w positions, then every list must be searched. In this section we find that other optimal average-time hash functions exist which have much improved worst-case behavior, often approximately equal to the average behavior. We also consider a simpler strategy involving storing each record in several lists.

To obtain good worst-case behavior $W_s(h)$ for $h : \{0, 1\}^k \to \{1, \cdots, b = 2^w\}$, the hash function $h(x)$ must depend on all of the bits of x, so that each specified bit contributes approximately equally and independently towards decreasing the number of lists searched. We shall also restrict our attention to balanced hash functions satisfying the conditions of Theorems 2 and 3 so that optimal average time behavior is ensured. While we have no proof that these block shapes are necessary for optimal worst-case behavior, the fact that $W_s(h)$ is bounded below by $A_s(h)$ makes it desirable to keep $A_s(h)$ minimal.

3.2.1. An example. Let us consider, by way of introduction, an example with $k = 4$, $w = 3$. The following table describes an interesting hash function h; row i describes the query $q \in Q_3$ such that $B_i = q(\{0, 1\}^4)$. Thus $h(0110) = 6$ and $h(1110) = 4$, each $x \in \{0, 1\}^4$ is stored in a unique bucket. (This function was first pointed out to the author by Donald E. Knuth.)

TABLE 1
A hash function $h : \{0, 1\}^4 \rightarrow \{1, \cdots, 8\}$

		Bit position			
		1	2	3	4
	1	0	0	*	0
	2	1	0	0	*
Bucket	3	*	1	0	0
address	4	1	*	1	0
	5	1	1	*	1
	6	0	1	1	*
	7	*	0	1	1
	8	0	*	0	1

This can be interpreted as a perfect matching on the Boolean 4-cube, as indicated in Fig. 1, since each block contains two adjacent points and distinct blocks are disjoint. In general, we have the problem of packing $\{0, 1\}^k$ with disjoint $(k - w)$-dimensional subspaces.

Whereas the hash function of Corollary 1 would have all its *'s in the fourth column, here the *'s are divided equally among the four columns. As a result, we have $W_1(h) = 5$ instead of 8. For example, we need only examine buckets 1, 4, 5, 6 and 7 for the query **1*. Table 2 lists $W_s(h)$ and $\lceil A_s(h) \rceil$ for $0 \leqq s \leqq 4$; we see that we have reduced $W_s(h)$ so that $W_s(h) = \lceil A_s(h) \rceil$; no further reduction is possible.

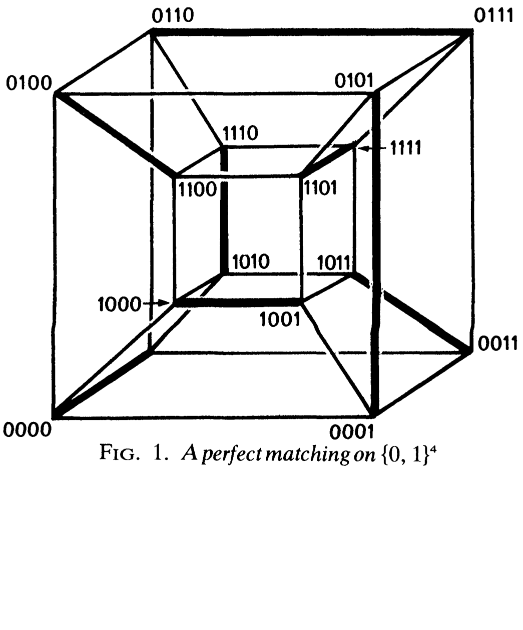

FIG. 1. *A perfect matching on* $\{0, 1\}^4$

TABLE 2

$W_s(h)$ and $A_s(h)$

s	0	1	2	3	4
$W_s(h)$	8	5	3	2	1
$\lceil A_s(h)\rceil$	8	5	3	2	1

3.2.2. Definition of ABD's. Let us call a table such as Table 1 an "associative block design of type (k, w)", or an ABD(k, w) for short. To be precise, *an* ABD(k, w) *is a table with $b = 2^w$ rows and k columns with entries from* $\{0, 1, *\}$ *such that $k > w$ and*

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

(ii) the rows represent disjoint subsets of $\{0, 1\}^k$. That is, given any two rows, there exists a column in which they contain differing digits.

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

Conditions (i) and (ii) ensure that $A_s(h)$ is minimal, by Theorem 3, since each row of the table represents a partial-match query in Q_w by condition (i), and by (ii) the corresponding sets $q(\{0, 1\}^k)$ are disjoint.

Condition (iii) attempts to restrict the class of ABD's to those hash functions with good worst-case behavior $W_s(h)$ by requiring a certain amount of uniformity in the utilization of each record bit by h. In fact, (iii) implies that $W_1(h)$ is minimal by (i) of Theorem 4, to follow. More stringent uniformity conditions are conceivable, perhaps involving the distribution of t-tuples within each t-subset of columns, but (iii) alone is enough to make the construction of ABD's a difficult combinatorial problem, comparable to the construction of balanced block designs (see [9]). Furthermore, with the exception of a construction due to Preparata based on BCH codes for the case $w = k - 1$, the existence of ABD's of arbitrary size does not seem to be answered by any previous results of combinatorial design theory. (See [40], [41], however, for an interpretation of ABD's as a special case of group-divisible incomplete block designs, defined in [5].)

3.2.3. Characteristics of ABD's. The following lemma gives some additional characteristics of ABD's that are implied by the definition.

THEOREM 4. *An* ABD(k, w)

(i) *has exactly $b \cdot w/2k$* 0's (*or* 1's) *in each column,*

(ii) *has $\binom{w}{u}$ rows which agree in exactly u positions with any given record* $r \in \{0, 1\}^k$, *for any u, $0 \leqq u \leqq w$,*

(iii) *is such that*

$$k \cdot \left(\frac{b \cdot w}{2k}\right)^2 \geqq \binom{b}{2},$$

or equivalently,

$$\frac{k}{w} \leqq \frac{w}{2} \cdot \left(\frac{b}{b-1}\right).$$

Proof. (i) There are as many records in $\{0, 1\}^k$ having a 0 in a given column as there are having a 1. A row of the ABD containing a * in that column represents a block B containing an equal number of each type. Rows containing a digit in that column represent blocks containing 2^{k-w} records of that type.

(ii) Let a_u denote the number of rows in the ABD which agree in exactly u positions with the given record r. We have that a_u is nonzero only if $0 \leqq u \leqq w$, since any row of the ABD contains w digits (*'s don't "agree" with digits). Also $a_0 = 1$, since the complement of r is stored in a unique list. Furthermore,

$$a_u = \binom{k}{u} - \sum_{0 \leqq v \leqq u} a_v \binom{k-w}{u-v},$$

since of the $\binom{k}{u}$ records that agree with r in u positions,

$$\sum_{0 \leqq v < u} a_v \binom{k-w}{u-v}$$

are accounted for by rows which agree with r in v positions, for some $v < u$ (by replacing $u - v$ of the $k - w$ stars in that row by digits which agree with r, and the other $k - w - u + v$ stars by digits complementary to r). Each record remaining must be in a separate list, since a row agreeing with r in u places can represent at most one record agreeing with r in u places, and any row agreeing with r in more than u places can represent none. The formula $a_u = \binom{w}{u}$ is the solution to the above recurrence.

(iii) Between each pair of rows in the ABD there must be at least one column in which they contain differing digits. There must be at least $\binom{b}{2}$ such row-row-column differences. On the other hand, each column can contribute at most $(bw/2k)^2$ such differences by (i) of this theorem. □

Parts (i) and (iii) of the above theorem can be used to restrict the search for ABD's. Note that (i) implies that $bw/2k$ must be integral, so that no ABD(5, 4)'s exist, for example. Part (iii) implies that to achieve large (record length)/(list index length) ratios k/w, we must let w grow to at least $2k/w$, approximately. For $k \leqq 20$ the above restrictions imply that the ABD(k, w)'s could only exist for the following (k, w) pairs:

$$\begin{array}{ll}
(4, 3), & (8, w) \quad \text{for } 4 \leqq w \leqq 7, \\
(10, 5), & (12, 6), (12, 9) \\
(14, 7) & (16, w) \quad \text{for } 6 \leqq w \leqq 15, \\
(18, 3t) \quad \text{for } 2 \leqq t \leqq 5, & (20, 10), (20, 15).
\end{array}$$

By an extension of the reasoning used to show (iii), an ABD(8, 4) has been shown not to exist, so that an ABD(8, 5) would be the next possible size after our ABD(8, 4) of § 3.2.1 for which existence is possible. To date, the existence of an ABD(8, 5) has not been settled; ABD(8, 6)'s and ABD(8, 7)'s are shown to exist in the following section.

3.2.4. ABD construction techniques. We present here several direct construction techniques which provide infinite classes of ABD's. The general question of the existence of an ABD of arbitrary type (k, w) seems to be extremely difficult; the positive nature of the very partial results obtained here suggests, however, that ABD's are not scarce.

We first present a simple infinite class of ABD's. The construction here is due to Ronald Graham. Franco Preparata has discovered another construction for a class of the same parameters, based on cyclic BCH erro-correcting codes [38].

THEOREM 5. *An* ABD$(2^t, 2^t - 1)$ *exists for* $t \geqq 2$.

Proof. We extend our notation for an ABD; a row containing r "–" 's will represent 2^r rows of the actual ABD obtained by independently replacing each – with a 0 or 1. The construction has two parts:

(i) Rows 1 to $t+1$ have –'s in positions $t+2$ to k. Row i for $1 \leqq i \leqq t+1$ has its star in column i, the remaining columns contain digits. (For example, columns 1 to $t+1$ of these rows might contain cyclic shifts of $*10^{t-1}$.)

(ii) Row i for $t+2 \leqq i \leqq 2k - t - 1$ contains digits in columns 1 to $t+1$, a * in column $t+1+\lfloor (i-t)/2 \rfloor$, and –'s elsewhere. The digits used are arbitrary except they must satisfy part (ii) of the ABD definition. It is easy to check that this is an ABD. □

We present in Table 3 an ABD(8, 7)($t = 3$) constructed this way.

TABLE 3
An ABD(8, 7)

	1	2	3	4	5	6	7	8
1	*	1	0	0	–	–	–	–
2	0	*	1	0	–	–	–	–
3	0	0	*	1	–	–	–	–
4	1	0	0	*	–	–	–	–
5	0	0	0	0	*	–	–	–
6	0	1	0	1	*	–	–	–
7	0	1	1	1	–	*	–	–
8	1	0	1	0	–	*	–	–
9	1	0	1	1	–	–	*	–
10	1	1	0	1	–	–	*	–
11	1	1	1	0	–	–	–	*
12	1	1	1	1	–	–	–	*

Graham has also constructed an ABD(16, 13) with a similar two-part method. This design is given in Table 4.

TABLE 4
An ABD(16, 13)

*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--------***

The ABD's of Theorem 5, while interesting as a solution to a combinatorial problem, are essentially useless as hash functions since the number of buckets is unacceptably large. We wish to have ABD's such that the ratio k/w does not tend to 1. The following theorem does just that.

THEOREM 6. *Given an* ABD(k, w) *and an* ABD(k', w') *such that* $k/w = k'/w'$, *one can construct an* ABD($k+k'$, $w+w'$).

Proof. For each possible pair of rows (R_1, R_2) with $R_1 \in$ ABD(k, w), $R_2 \in$ ABD(k', w'), let the concatenation R_1R_2 be a row of the ABD($k+k'$, $w+w'$). This is easily shown to be a legal ABD. □

We can now form an ABD(8, 6) or an ABD(12, 9) from the design of Table 1. Table 5 gives the ABD(8, 6) so constructed.

TABLE 5
An ABD(8, 6)

	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

Theorem 5 allows us to construct an infinite family of ABD's of type $(4t, 3t)$, for $t \geqq 1$, using the ABD of Table 1. This is approaching utility (an ABD(16, 12) is conceivably useful, say) but we need "starting" designs with large k/w to obtain a family with large k/w. On the other hand, we know by Theorem 4 (iii) that designs with large k/w must have k at least $2(k/w)^2$ approximately. Unfortunately, these tables get rapidly unmanageable by hand. Computer searches for an ABD(8, 5) or an ABD(10, 5) also ran out of time before finding any. The question as to whether ABD(k, w)'s existed with $k/w > 4/3$ thus remained open until the following theorem, showing that k/w can be arbitrarily large, was discovered.

THEOREM 7. *Given an* ABD(k, w) *and an* ABD(k', w') *one can construct an* ABD(kk', ww').

Proof. Each row R of the first ABD generates $2^{w(w'-1)}$ rows of the resultant ABD, as follows. The set of rows of the ABD(k', w') is arbitrarily divided into equal-sized subsets, A_0 and A_1. Each character x of R is replaced by a string of k' characters: if $x = *$, x is replaced by $*^{k'}$, otherwise x is replaced by some row in A_x. The w digits of R are replaced independently in all possible ways by rows from the corresponding sets A_0 and A_1.

This generates a table with $2^{ww'}$ rows of length kk', each row having $(k-w)k' + w(k'-w') = kk' - ww'$ $*$'s. Any two rows of the resultant ABD must contain differing digits in at least one column, since rows replacing differing digits must 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 by two (differing) rows from the second ABD. The number of $*$'s in each column turns out to be $2^{ww'}(kk'-ww')/kk'$, as required, so that we have created an ABD(kk', ww'). □

The 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$(4^t, 3^t)$ for $t \geqq 1$) from the ABD(4, 3) of Table 1. The following table illustrates the rows generated for an ABD(16, 9).

TABLE 6
Rows of 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
```

The preceding theorem allows us to construct an ABD$(4^t, 3^t)$ for $t \geqq 1$, for example, from our ABD(4, 3). While this does provide large k/w designs, they are quite likely not the smallest designs for the given ratio, since by Theorem 4 (iii) we might hope to have w grow linearly with (k/w), whereas here it grows like $(k/w)^{\log_{4/3}(3)} > (k/w)^4$. At present, however, it is our only way of constructing large k/w designs.

3.2.5. Analysis of ABD search times. Let us examine the worst-case number of lists examined $W_s(h)$ for hash functions associated with the ABD's of the last section.

Consider first the hash function h associated with an ABD$(k+k', w+w')$ which was created by the concatenation (Theorem 6) of an ABD(k, w) and an ABD(k', w') (with associated hash functions g and g', respectively). Then

$$W_s(h) = \max_{u+v=s} W_u(g) \cdot W_v(g')$$

for $0 \leqq s \leqq k+k'$, $0 \leqq u \leqq k$, $0 \leqq v \leqq k'$. For example, the ABD(8, 6) created from two of our ABD(4, 3)'s has $W_s(h)$ as in Table 7.

TABLE 7
Performance of an ABD(8, 6)

s	0	1	2	3	4	5	6	7	8
$W_s(h)$	64	40	25	16	10	6	4	2	1
$\lceil A_s(h) \rceil$	64	40	25	15	9	6	4	2	1

Also shown are the values of $\lceil A_s(h) \rceil$, which is a lower bound for $W_s(h)$. 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 $W_s(g)$ with itself m times, yielding an ABD(mk, mw). Consider a partial match query $q \in Q_s$. Let y_i be the number of k-column blocks which have exactly i specified bits, for $0 \leqq i \leqq k$, so that

$$\sum_{0 \leqq i \leqq k} y_i = m \quad \text{and} \quad \sum_{0 \leqq i \leqq k} i y_i = s.$$

We also have, of course, the condition that

$$y_i \geqq 0 \quad \text{for} \quad 0 \leqq i \leqq k.$$

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

$$W_s(h) = \max \prod_{0 \leqq i \leqq k} W_i(g)^{y_i},$$

where the maximum is taken over all sets of integers $y_o, \cdots, y_k$ satisfying the three conditions on the y_i's given above.

Let $W'_s(h) = \log W_s(h)$ for any h and for all s, so that

$$W'_s(h) = \max \sum_{0 \leqq i \leqq k} W'_i(g) \cdot y_i,$$

yielding an integer programming problem in $(k+1)$-dimensional space. Since we desire 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:

$$\text{maximize } W'_s(h) = \sum_{0 \leqq i \leqq k} W'_i(g)x_i,$$

subject to

$$\sum_{0 \leqq i \leqq k} x_i = 1,$$

$$\sum_{0 \leqq i \leqq k} ix_i = s/m,$$

$$x_i \geqq 0 \quad \text{for} \quad 0 \leqq i \leqq k.$$

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 $W'_s(g)$ is a concave function, we have

$$i = \lfloor s/m \rfloor = j-1$$

and

$$W'_s(h) = W'_i(g)(j-s/m)/(j-i) + W'_j(g)(s/m-i)/(j-i).$$

This is the general solution. When s/m is a multiple of $1/k$, then only $x_{sk/m}$ is nonzero, equal to one. This solution does not apply when $W'_s(g)$ is not concave. (For example, $W'_s(g)$ for our ABD(4, 3) is not quite concave, since $W_3(g) = 2$ is a little too large. This convexity is the cause of the discrepancies of Table 4.) One can show, by a combinatorial argument, that if $W_s(g) = A(k, w, s)$ for $0 \leqq s \leqq k$, then $W_s(h) = A(mk, mw, s)$ for $0 \leqq s \leqq mk$ 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 (of type (k, w)) used in the construction.

Under this assumption (for which I have no proof) the worst-case behavior of an ABD(16, 9) created from two of our ABD(4, 3)'s can be calculated to be as given in Table 8.

TABLE 8
Performance of an ABD(16, 9)

s	0	1	2	3	4	5	6	7	8
$W_s(h)$	512	368	272	224	176	116	76	56	36
$\lceil A_s(h) \rceil$	512	368	263	186	131	91	63	43	30

s	9	10	11	12	13	14	15	16
$W_s(h)$	33	24	16	8	5	3	2	1
$\lceil A_s(h) \rceil$	20	14	9	6	4	3	2	1

We see that while $W_s(h)$ approximates $A_s(h)$ reasonably well, performance has been degraded somewhat. We conjecture that better ABD(16, 9)'s exist. (It is a situation reminiscent of the fact that recursive or systematic constructions of error-correcting codes tend not to work as well as a random code.) An exhaustive search by computer for better designs appears to be infeasible, so that a better construction method is needed. (A sophisticated backtracking procedure was unable to determine whether an ABD(8, 5) exists or not, using one hour of computer time.)

While it is not too difficult to calculate $W_s(h)$ for small ABD's constructed by insertion (assuming that our conjecture about the nature of the worst case is correct), the asymptotic analysis seems difficult. In any case, the ABD's so constructed yield large k/w designs with significantly improved $W_s(h)$.

3.2.6. 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 Theorem 4, so that an ABD(3, 2) can not exist). The "design" in Table 9 yields reasonably good worst-case performance.

TABLE 9
An "irregular" (3, 2) design

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

Here bucket 4 contains both records 010 and 101. This hash function has worst-case behavior as in Table 10.

TABLE 10
Behavior of the previous design

s	0	1	2	3
$W_s(h)$	4	3	2	1

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 a k/w ratio of 2 (Table 11).

TABLE 11
An "irregular" (4, 2) design

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

The above hash function has worst-case behavior shown in Table 12.

TABLE 12

Behavior of the irregular (4, 2) *design*

s	0	1	2	3	4
$W_s(h)$	4	4	3	2	1

3.2.7. Conclusions on ABD's. Associative designs minimize the average retrieval time for partial-match queries, since the blocks have the optimal shape specified in Theorems 2 and 3. In addition, they reduce greatly the worst-case retrieval time, often nearly to the average time. While the recursive or iterative nature of our ABD construction techniques lends itself to simple implementation, the complexity of the hash function computation may be such that using ABD's is justified only when the file is stored on slow secondary storage devices.

In summary, ABD's can be used to minimize the worst-case performance of hashing algorithms with no increase in either the average retrieval time of 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 2^{30} 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 ($=100/20$) independent filing systems, and let each record be filed once in each system. Each bucket system will have 2^{20} lists. The first system will use the first 20 bits of each record as its list index, the second system will use the second 20 bits of the record, and so on.

Now suppose we have a query $q \in Q_s$. At least one of the five systems will have at least $s/5$ bits specified for its list index—so we can use this system to retrieve the desired records. The number of buckets searched is not more than $2^{20-\lceil s/5 \rceil}$.

In general, if $b = 2^w$ is the number of buckets per system, and we have k-bit records to store (records with nonbinary 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:

$$W_s(h) \leqq 2^{w - \lceil ws/k \rceil}.$$

This surpasses even the best achievable *average* 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., $s = k/2$), then at most sqrt(b) $= 2^{w/2}$ buckets 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 methods chosen for each subfield.

4. Trie algorithms for partial-match queries. The results of § 3, that an optimal block shape for a hashing algorithm for partial-match retrieval is a "subcube" of Σ^k, suggests using tries as an alternative data structure. Since the set of records stored in each subtrie of a trie is a subcube of Σ^k, recursively split into smaller subcubes at each level, tries might perform as efficiently as the optimal hash functions of the preceding section.

4.1. Definition of tries. Tries were introduced by Rene de la Briandais [11], and were further elaborated on by E. Fredkin [15], D. E. Knuth [28, § 6.3], and G. Gwehenberger [23].

A trie stores records at its leaves (external nodes). Each internal node at level i (the root is at level 1) specifies an $|\Sigma|$-way branch based on the ith character of the word being stored. As an example, consider the file

$$F=\{000, 001, 100, 101, 111\}$$

of three-bit binary words (that is, $\Sigma=\{0, 1\}$). They would be stored in a trie shown in Fig. 2. Here a left branch is taken when the corresponding bit is zero; a right branch is taken otherwise. A record is stored as a leaf as soon as it is the only record in its subtree; this reduces the average path length from the the root to a leaf in a random binary trie from k to about $\log_2|F|$ (see [28, § 6.3]).

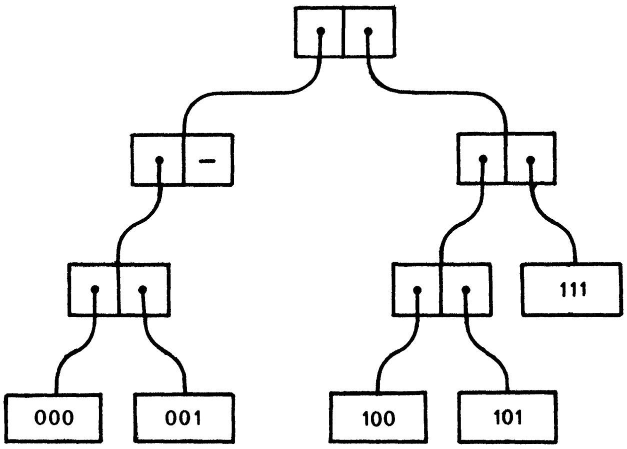

FIG. 2. *A trie*

A link to an empty subtree, such as that for records of type 01* in the figure, is indicated by the null link "–". Useless internal nodes having but one nonnull link can be eliminated by storing at each internal node the character position on which to branch. This refinement, introduced by D. R. Morrison [33], shall not be pursued further here. Knuth has shown that the number of internal nodes will be roughly $|F|/(\ln 2) \approx 1.44|F|$, if $k \gg \log_2 |F|$, for random binary tries; here the number of such useless nodes will not be excessive.

4.2. Searching tries. Given a partial match query $q = (q_1, \cdots, q_k)$, where $q_i \in (\Sigma \cup \{\text{"*"}\})$ for $1 \leqq i \leqq k$, and a trie T with root node N, the following procedure prints all records in T satisfying q. The initial call has the argument *level* equal to one. If M is an internal node of T, the M_c denotes the root of the subtree corresponding to the character $c \in \Sigma$, or *null* if no such subtree exists.

```
**Procedure** Triesearch ($N$, $q$, level);
**begin**
**if** $N$ is a leaf containing record $r$ **then**
    **begin if** $r$ satisfies $q$ **then** print ($r$) **end**
**else if** $q_{\text{level}} \neq$ "*" **then**
    **begin** if $N_{q_{\text{level}}} \neq$ **null** then Triesearch ($N_{q_{\text{level}}}$, $q$, level + 1) **end**
**else for** $c \in \Sigma$ **do**
    **if** $N_c \neq$ null **then** Triesearch ($N_c$, $q$, level + 1)
**end** Triesearch
```

4.3. Analysis of Triesearch. We will restrict our attention to binary tries in this section; the analysis for general $|\Sigma|$-ary tries would be similar. Since for nonbinary alphabets, record r could be encoded in binary by concatenating binary representations of each character r_i (for $1 \leqq i \leqq k$), this entails no real loss in generality; binary tries can be used to store arbitrary data. In fact, Theorem 1 suggests that this might be even a good idea, since we could then branch on the first bit of the representation of each character of r before branching on the second bit of each, and thus obtain record-spaces for each subtree more closely in agreement with Theorem 1. Bentley [3] has examined a similar approach in more detail; we shall not pursue it further here.

Our cost measure $c(n, s, k)$ shall be the average number of internal and external nodes examined by Triesearch to respond to a partial match query $q \in Q_s$, given that the trie contains n k-bit records.

Consider an arbitrary node M at level $w+1$ in a trie. There are at most 2^w nodes at this level. Let $m_1 \cdots m_w$ denote the common prefix of the records in the subtree with root M; the bits $m_1 \cdots m_w$ specify which w branches to take to get from the root of the trie to M. Finally, let $p(n, w, k)$ denote the probability that in a random trie there is a node M at level $w+1$ with prefix $m_1 \cdots m_w$ (this is independent of the actual values of $m_1 \cdots m_w$ if each n-record file F is equally likely). Note that there will be such a node if and only if the number of records with prefix $m_1 \cdots m_w$ is not zero and the number with prefix $m_1 \cdots m_{w-1}$ is not one. Thus

$$p(n, w, k) = \binom{2^k}{n}^{-1} \left[\binom{2^k}{n} - \binom{2^k - 2^{k-w}}{n} - \binom{2^k - 2^{k-w+1}}{n-1} \cdot 2^{k-w} \right].$$

Assuming that $n \ll 2^k$, we may approximate the latter two terms by the probability of having zero (resp., one) success in 2^{k-w} (resp., 2^{k-w+1}) trials, where the probability of success is $n2^{-k}$. Using the Poisson approximation to the binomial distribution, we have

$$p(n, w, k) \cong 1 - \exp(-n2^{-w}) - n2^{-w} \exp(-n2^{-w+1}).$$

This expression is independent of k, as we might expect, since once enough bits of a record are known to distinguish it, it is stored as a leaf in the trie, independent of the total record length. The function $p(n, w, k)$ is very nearly a step function; it is approximately 1 for $w < \log_2(n)$, going very quickly to 0 for $w > \log_2(n)$, passing the value $\frac{1}{2}$ at $(\log_2(n) - .0093)$, approximately.

Thus probability that a node M at level $w+1$ will be examined for a partial match query $q \in Q_s$ is just $A_{\min}(k, w, s)$, where $A_{\min}(k, w, s)$ is the function defined in § 3.12. Since there are 2^{-w} nodes at level $w+1$ in a complete binary tree, we get that the total expected cost of Triesearch is

$$1 + \sum_{1 \leqq w \leqq k} p(n, w, k) \cdot A_{\min}(k, w, s).$$

$$\cong \sum_{1 \leqq w \leqq k} (1 - \exp(-n2^{-w}) - n2^{-w} \exp(-n2^{-w+1}))(2 - s/k)^w.$$

Substituting $w = \log_2(n) + z$, we get that the above is equivalent to

$$\sum_{-\log_2(n) \leqq z \leqq k - \log_2(n)} (1 - \exp(-2^{-z}) - 2^{-z} \exp(-2^{1-z}))(2 - s/k)^{\log_2(n) + z}.$$

For $z \leqq -1$, $p(n \log_2(n) + z, k) \geqq .82$, so the sum for negative z is $ck(k-s)^{-1}(2 - s/k)^{\log_2(n)}$ for some c, $.82 \leqq c \leqq 1.00$. For $z \geqq 0$, the sum is maximized when $s/k = 0$. However, even in this case the sum is not more than 1.54 $(2 - s/k)^{\log_2(n)}$, by numerical calculation, so that the total cost of the algorithm is approximately

$$ck(k-s)^{-1}(2 - s/k)^{\log_2(n)},$$

where the constant c of proportionality is less than 2.54.

4.4. Conclusions on tries. Tries are roughly as efficient as the optimal hashing algorithms for random data for which the number of lists used is $|F|$. For the usual situation involving highly nonrandom data, tries are probably the best practical choice, since the tries will store any data efficiently, whereas a hashing algorithm which selects record bits to use as a list index might result in a large number of empty lists and a few very long lists. For nonrandom data, an interesting problem arises if the branching decision may be made on any of the untested bits; we wish to choose the bit on which to split the subfile that will yield the best behavior. (Note that it is the most unbalanced tries which perform best.) For this modification, it may also be possible to take into consideration the probability that any given bit may be queried.

5. Conclusions. The hashing and trie-search algorithms presented perform more efficiently than any previously published partial-match search algorithms.

Retrieving from a file of n k-bit words all words that match a query with s bits specified takes approximately $n^{\log_2(2-s/k)}$ time, a little more than $n^{1-s/k}$, our conjectured lower bound on the time required by any algorithm. The main open problems are the proof or disproof of this conjecture, the existence questions for ABD's of general parameters, and the generalization of the results of this paper to handle nonrandom data with nonuniform query distribution probabilities.

Acknowledgments. I would like to thank the many people who provided constructive criticism of this research while in progress, with particular thanks to N. G. de Bruijn, Robert W. Floyd, Ronald Graham, David Klarner, Donald Knuth, Malcolm Newey and Franco Preparata. I would also like to thank Terry Bowes and Allison Platt for typing the manuscript and drawing the figures.

REFERENCES

[1] C. T. Abraham, S. P. Ghosh and D. K. Ray-Chauduri, *File organization schemes based on finite geometries*, Information and Control, 12 (1968), pp. 143–163.

[2] M. Arisawa, *Residue hash method*, J. Inf. Proc. Soc. Japan, 12 (1971), pp. 163–167.

[3] J. L. Bentley, *Multidimensional binary search trees used for associative searching*, unpublished manuscript, Dept. of Computer Sci., Univ. of North Carolina, Chapel Hill.

[4] J. L. Bentley and R. A. Finkel, *Quad trees: A data structure for retrieval on composite keys*, Acta Informat., 4 (1974), pp. 1–10.

[5] R. C. Bose and W. S. Connor, *Combinatorial properties of group divisible incomplete block designs*, Ann. Math. Statist., 23 (1952), pp. 367–383.

[6] R. C. Bose, C. T. Abraham and S. P. Ghosh, *File organization of records with multi-valued attributes for multi-attribute queries*, Combinatorial Mathematics and its Applications, UNC Press, 1969, pp. 277–297.

[7] R. C. Bose and Gary G. Koch, *The design of combinatorial information retrieval systems for files with multiple-valued attributes*, SIAM J. Appl. Math., 17 (1969), pp. 1203–1214.

[8] D. K. Chow, *New balanced-file organization schemes*, Information and Control, 15 (1969), pp. 377–396.

[9] W. S. Connor, Jr., *On the structure of balanced incomplete block designs*, Ann. Math. Statist., 23 (1952), pp. 57–71.

[10] D. R. Davis and A. D. Lin, *Secondary key retrieval using an IBM 7090-1301 system*, Comm. ACM, 8 (1965), pp. 243–246.

[11] R. de la Briandais, Proc. Western Joint Computer Conference, 15 (1959), pp. 295–298.

[12] P. J. Denning, *Virtual memory*, Comput. Surveys, 2 (1970), pp. 153–189.

[13] J. A. Feldman and P. D. Rovner, *An ALGOL-based associative language*, Comm. ACM, 12 (1969), pp. 439–449.

[14] W. Feller, *An Introduction to Probability Theory and its Applications*, vol. 1, John Wiley, New York, 1950.

[15] E. Fredkin, *Trie memory*, Comm. ACM 3 (1960), pp. 490–500.

[16] S. P. Ghosh and C. T. Abraham, *Application of finite geometry in file organization for records with multiple-valued attributes*, IBM J. of Res. Develop., 12 (1968), pp. 180–187.

[17] S. P. Ghosh, *Organization of records with unequal multiple valued attributes and combinatorial queries of order 2*, Information Sci., 1 (1969), pp. 363–380.

[18] ———, *File organization: Consecutive storage of relevant records on a drum type storage*, IBM Res. Rep. RJ 895, 1971.

[19] ———, *File organization: The consecutive retrieval property*, Comm. ACM 15, (1972), pp. 802–808.

[20] H. J. Gray and N. S. Prywes, *Outline for a multi-list organized system*, Paper 41, Ann. Meeting of the ACM, Cambridge, Mass., 1959.

[21] R. A. Gustafson, *A randomized combinatorial file structure for storage and retrieval systems*, Ph.D. thesis, Univ. of South Carolina, Columbia, 1969.

[22] ———, *Elements of the randomized combinatorial file structure*, Proc. of the Symposium on Inf. Storage and Retrieval, ACM SIGIR, Univ. of Maryland, April 1971, pp. 163–174.

[23] G. Gwehenberger, *Anwendung einer binären Verweiskettenmethode beim Aufbau von Listen*, Elektron. Rechenanl., 10 (1968), pp. 223–226.

[24] G. H. Hardy, J. E. Littlewood and G. Pólya, *Inequalities*, Cambridge University Press, London, 1959.

[25] D. Hsiao and F. Harary, *A formal system for information retrieval from files*, Comm. ACM, 13 (1970), pp. 67–73.

[26] L. R. Johnson, *An indirect chaining method for addressing on secondary keys*, Ibid., 4 (1961), pp. 218–222.

[27] J. R. Kiseda, H. E. Peterson, W. E. Seedback and M. Teig, *A magnetic associative memory*. IBM J. Res. Develop., 5 (1961), pp. 106–121.

[28] D. E. Knuth, *The Art of Computer Programming. Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1972.

[29] G. G. Koch, *A class of covers for finite projective geometries which are related to the design of combinatorial filing systems*, J. Combinatorial Theory, 7 (1969), pp. 215–220.

[30] J. Mauchly, *Theory and Techniques for the Design of Electronic Digital Computers*, G. W. Patterson, ed.; vol. 1, 1946, §§ 9.7–9.8; vol. 3, 1946, §§ 22.8–22.9.

[31] J. Minker, *An overview of associative or content addressable memory systems and a KWIC index to the literature*, Tech. Rep. TR-157, Univ. of Maryland Computer Center, College Park, 1971.

[32] ———, *Associative memories and processors: A description and appraisal*, University of Maryland and Auerbach Corp. Tech. Rep. Tr-195, 1972.

[33] D. R. Morrison, *PATRICIA—Practical algorithm to retrieve information coded in alphanumeric*, J. Assoc. Comput. Mach., 15 (1968), pp. 514–534.

[34] S. Nishihara, S. Ishigaka and H. Hagiwara, *A variant of residue hashing method of associative multiple-attribute data*, Rep. A-9, Data Processing Center, Kyoto Univ., Kyoto, Japan, 1972.

[35] W. W. Peterson and E. J. Weldon, *Error-Correcting Codes*, MIT Press, Cambridge, Mass., 1972.

[36] N. S. Prywes and H. J. Gray, *The organization of a Multilist-type associative memory*, IEEE Trans. on Communication and Electronics, 68 (1963), pp. 488–492.

[37] N. S. Prywes, *Man-computer problem solving with Multilist*, Proc. IEEE, 54 (1966), pp. 1788–1801.

[38] F. Preparata, Personal communication.

[39] D. K. Ray-Chauduri, *Combinatorial information retrieval systems for files*, SIAM J. Appl. Math., 16 (1968), pp. 973–992.

[40] R. L. Rivest, *Analysis of associative retrieval algorithms*, Ph.D. thesis, Computer Science Dept., Stanford Univ., Stanford, Calif., 1974.

[41] ———, *On the optimality of Elias's algorithm for performing best-match searches*, IFIP Proc., Stockholm, Aug. 1974, pp. 678–681.

[42] J. A. Rudolph, *A production implementation of an associative array processor—STARAN*, Proc. Fall Joint Comp. Conf., 1972, pp. 229–241.

[43] E. D. Sacerdoti, *Combinatorial mincing*, Unpublished manuscript, 1973.

[44] A. E. Slade, *A discussion of associative memories from a device point of view*, American Documentation Institute 27th Ann. Meeting, 1964.

[45] T. A. Welch, *Bounds on the information retrieval efficiency of static file structures*, Project MAC Rep. MAC-TR-88, Mass. Inst. of Tech., Cambridge, Mass., 1971; Ph.D. thesis.

[46] P. F. Windley, *Trees, forests, and rearranging*, Comput. J., 3 (1960), pp. 84–88.

[47] E. Wong and T. C. Chiang, *Canonical structure in attribute based file organization*, Comm. ACM, 14 (1971), pp. 593–597.