PARTIAL-MATCH RETRIEVAL USING TRIES, HASHING, AND SUPERIMPOSED CODES

by

Jeannette Marie Wing

SUBMITTED IN PARTIAL FULFILLMENTS OF THE REQUIREMENTS
FOR THE DEGREES OF
BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1979

Signature of Author Jeannette M. Wing
Department of Electrical Engineering and
Computer Science, May 4, 1979

Certified by Ronald L. Rivest
Thesis Supervisor (Academic)

Certified by John F. Reiser
Company Supervisor (VI-A Cooperating Company)

Accepted by ________________________
Chairman, Departmental Committee on Graduate Students

PARTIAL-MATCH RETRIEVAL USING TRIES, HASHING, AND SUPERIMPOSED CODES

by

Jeannette Marie Wing

Submitted to the Department of Electrical Engineering and Computer Science on May 4, 1979 in partial fulfillments of the requirements for the Degrees of Bachelor of Science and Master of Science.

ABSTRACT

We analyze and implement for partial-match retrieval three file organization methods: tries, extendible hashing, and superimposed coding. We compare predicted values of storage and retrieval times derived from mathematical formulae with measurements based on implementations done in a UNIX environment on a PDP 11/45 computer. The Suffolk County, NY telephone directory serves as the database.

Tries require less storage than predicted; extendible hashing, more; superimposed coding, the same. Actual retrieval times for tries and superimposed coding are twice the predicted; that for extendible hashing is three times the predicted.

Tries and extendible hashing are similar in structure and can be used to speed up retrieval time for superimposed coding at the expense of storage for an additional index. We discuss the relative merits of each method for partial-match retrieval and note many issues that arise in the implementations.

Thesis Supervisors: Ronald L. Rivest, Associate Professor of Computer Science
John F. Reiser, Member of Technical Staff, Bell Laboratories

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I am deeply grateful to the members of the Interactive Computer Systems Department at Bell Laboratories for their help throughout the progress of my thesis.

John D. Gabbe provided invaluable assistance in my understanding the working software version of superimposed coding for partial-match retrieval. Thomas B. London gladly answered questions about superimposed coding, UNIX, and C.

Both Matthew S. Hecht and Howard P. Katseff's general advice and interest; Ken Swanson's patience and help in maintaining the system; the fellow PDP 11/45 users' tolerating my use of the machine and disk drives; are all greatly appreciated.

I also wish to thank Charles S. Roberts for giving me the opportunity to work in his department, for introducing me to my thesis topic, and for his guidance through my CO-OP assignment.

Finally, I wish to express sincere thanks to my advisors: Professor Ronald L. Rivest for his perceptive suggestions, and John F. Reiser for his technical help and for his constant moral support and encouragement.

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Given the class of queries a user makes in an information retrieval system, the engineer organizes the database within storage constraints to yield an acceptable response time. Thus, two questions arise:

1. What kind of queries will the user make?

2. How should the database be organized?

This thesis addresses only partial-match queries. It analyzes three database organization techniques for partial-match retrieval: tries, extendible hashing, and superimposed coding. We compare, in terms of storage and time requirements, the performance predicted by analytical models with the performance measured from implementations of these techniques.

Previous studies of partial-match retrieval algorithms analyze search time in theoretical terms, like the number of nodes visited in a trie or buckets accessed in hashing. However, storage requirements modeled in terms of disk space, and retrieval time modeled in terms of access time help a user decide how to organize the database for a particular application.

1.1 Definitions

1.1.1 Files, Records, Attributes

A *record* r is defined to be an ordered k-tuple $(r_1, r_2, \ldots, r_k)$ of values (or *keywords)* chosen from some finite set Σ. Each coordinate of the k-tuple is referred to as a *key, attribute,* or *field.* Let $\Sigma = \{0, 1, \ldots, v-1\}$ so that Σ^k is the set of all k-letter words over the alphabet Σ, and has size v^k. A *file* F is defined to be any nonempty subset of Σ^k. Throughout this thesis we let N denote the size of the file.

1.1.2 Queries, Partial-Match 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 a (perhaps null) subset of *F*.

Two classes of queries are intersection queries and best-match queries. Intersection queries ask for records that fall within a specified subset, whereas best-match queries ask for all the nearest neighbors in the file [Ri,74].

The class of intersection queries contains exact-match, single-key, partial-match and range type queries. Table 1.1 presents a sample file and Table 1.2 illustrates each of these types of intersection queries. The keywords of each record are for the attributes *name, house number, street,* and *town.* Let the retrieved set be the set of record numbers corresponding to the response $q(F)$ to the query q.

An exact-match query asks for a specified record from a file. A single-key query asks for all records having a specified value for a certain attribute of the record. A partial-match query asks for all records having a specified value for one or more attributes of a record. More generally, a range query asks for all records having a value within a given range of values specified for each attribute of the record.

Since we only consider partial-match queries in this thesis, we will use the following notation for the formal definition of a partial-match query. A partial-match query q with s keys specified for some s in the range $0 \leqslant s \leqslant k$ is represented by a record $\hat{r} \in R$ with k-s keys replaced by the special symbol "*" which means "unspecified". If $\hat{r} = (\hat{r}_1, \ldots, \hat{r}_k)$, then for k-s values of j attributes, $\hat{r}_j = *$. The set $q(\Sigma^k)$ is the set of all records agreeing with $\hat{r}$ in the specified positions. Thus

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

Record Number	Name	House Number	Street	Town
1	Appleby Lloyd	15	Bermann Way	Middletown
2	Barone Paul	14	Knox Lane	Englishtown
3	Barone Ronald	9	Tanglewood Road	Englishtown
4	Barone Sandra	111	Newark Avenue	Bradley Beach
5	Fox Norman	84	Tower Hill Avenue	Bradley Beach
6	Gillies David	8	Monmouth Road	Middletown
7	Hagg B	52	Maxwell	Englishtown
8	Lanzo O	612	Lorillard Avenue	Bradley Beach
9	Lanzo P J	8	Hill	Hazlet
10	Richey Marjorie	85	Spruce Drive	Hazlet

Table 1.1. A sample file of 10 records.

Query Type	*Query q*	*Retrieved Set q(F)*
Exact-match	Retrieve the record for a "Sandra Barone" who lives on "111 Newark Avenue" in "Bradley Beach".	{4}
Single-key	Retrieve all records for those people whose last name is "Barone".	{2,3,4}
Partial-match	Retrieve the records for all those people whose last name is "Barone" and who live in "Englishtown".	{2,3}
Range	Retrieve all records for those people whose house numbers are greater than "100" and live in "Bradley Beach".	{4,8}

Table 1.2. Query types and sample queries.

and

$$q(F) = F \cap q(\Sigma^k).$$

That is, the response to the query q is the intersection of the file F and the set of all records (not necessarily in the file) that match the query.

1.2 File Organization Techniques

Familiar techniques of organizing a file include storing records in a sequential file, in inverted lists, in a tree structure, or in buckets via hashing.

We can retrieve a record from a sequential file by linear search or by binary search. But since binary search is possible only for the attribute type by which the file has been ordered, a partial-match retrieval requires a linear search of $O(N)$ records, where N is the number of records in a file.

If the database is too large for linear search to be practical, then we can use inverted lists. An inverted list is similar to an index usually found in the back of a textbook. We make an index for an attribute from values found for that attribute in the records. Each value heads a list of records, or pointers to records, which have that value for that attribute. Thus, the inverted lists in an index of a textbook would be the lists of page numbers, each list headed by a keyword found in the text. The time required to retrieve a record via an index is the time to find the value of an attribute and depends on the number of values kept in the index.

We can use tree structures to yield on $O(\log N)$ search time for exact-match queries. Bayer and McCreight [BM] introduce B-trees that use multiway branching to maintain a balanced tree efficient for searching and updating. They analyze the storage utilization and the costs to retrieve, insert, and delete a single key. They measured the performance of B-trees under varying retrieval, insertion, and deletion conditions. Bayer [Ba] similarly studies symmetric binary B-trees. Wedekind [KK] compares B-trees to indexes and Aho et. al. [AHU] discuss 2-3 trees, a specific case of B-trees.

Another tree structure is a "trie", a data structure introduced by de la Briandais [dlB]. A trie is a type of radix-search tree whose internal nodes at level i specify a $|\Sigma|$-way branch on the i^{th} character of the word being stored. Rivest [Ri, 74, 76] proves that in partial-match retrieval for s letters specified of k-letter words, a trie-search algorithm requires an average time $O(N^{(k-s)/k})$. Burkhard [Bu, 76, 77a, 77b] gives bounds on the worst case performance and an explicit expression for the average performance on a class of partial-match file trie designs.

Hashing, or key-to-address transformation, is a technique that provides an $O(1)$ search time for exact-match queries. Both Rivest and Burkhard analyze the performance of hashing-search for partial-match queries in the aforementioned references.

Fagin et al [Fa] introduce extendible hashing as an alternative to conventional hashing because it has a dynamic structure which makes hash tables extendible and keeps radix-search trees balanced. Extendible hashing is particularly attractive if the database grows and/or shrinks.

Similarly, Larson [La] analyzes the storage requirements for extendible hashing which he calls "dynamic" hashing. Because of the close relationship between tries and extendible hashing, Larson uses the results for storage for tries from Knuth [Kn]. A user subroutine, recently implemented in the UNIX® environment, offers a version of extendible hashing for database management [RT].

A less familiar technique used for partial-match retrieval is superimposed coding. Mooers [Mo] describes a simplified superimposed code used in Zatocoding retrieval systems in the early 1950s. Each record corresponds to a notched-edge card. Long metal needles are mechanically used to select cards (records). The query determines which needles to use in selection. The subset of records that drop from the file of notched-edge cards includes all the records ("good" drops) that match the query. This subset can also include "false" drops. We further discuss this method of partial-match retrieval in Section 2.

Superimposed coding is experimentally used on a directory assistance retrieval system for partial-match queries at Bell Laboratories [Ga]. I base my predictions and measurements for superimposed coding on this system. Roberts [Ro] presents an algorithm to generate the superimposed codes without the use of a stored code dictionary and analyzes its false-drop probability.

1.3 Storage and Time Parameters

1.3.1 Secondary Storage

Current media for secondary storage of large files are magnetic tape, disk, and drum.

Magnetic tape parameters are the *width, length, density* of the tape, and the number of *tracks.*

Disk and drum parameters are the number of *disks,* (or surfaces), the number of *cylinders* (which equals the number of *tracks* per disk surface), and the number of *sectors* per track. A *block* is a software-defined portion of a track equal to an integral number of sectors. It is the unit of information actually transferred between secondary storage and main memory. This thesis presents storage requirements in terms of disk parameters only.

1.3.2 Time Parameters

The average time to reach a specific position on disk or drum storage is the *random-access time* which consists of a *seek time* and a *rotational delay.*

When the proper position is reached, the rate at which the actual data is read from or written to secondary storage is the *transfer rate.* For example, on disks the transfer rate is a function of rotational speed and track density. If the transfer rate is t bytes/msec. and the block size is B bytes than the block transfer time is $btt = B/t$ msec.

1.4 Thesis Outline

This thesis examines the storage requirements and the retrieval time of using tries, extendible hashing search, and superimposed coding for partial-match retrieval in terms of storage and time parameters. The times to do insert and delete operations are also discussed.

Section 2 discusses in detail the three file organization techniques. Section 3 presents a comparison among a theoretical model, a modified theoretical model, and an implementation model for the storage requirements for each of the three techniques. It explains the assumptions of the theoretical model and how the implementation decisions differ from these assumptions and lead to a modified theoretical model that is used to predict the results of the actual implementation. Similarly, Section 4 presents analyses for retrieval time; Section 5 discusses the results from Sections 3 and 4, and some of the issues that arose in the implementations. Section 6 summarizes the conclusions of the thesis and suggests topics for future study.

2. STORAGE STRUCTURES OF TRIES, EXTENDIBLE HASHING, SUPERIMPOSED CODING

2.1 Tries

A trie, a type of tree, stores records at its external nodes, or *leaves.* An *internal node* at level i specifies a $|\Sigma|$-way branch based on the i^{th} character of the key being stored. The root is at level one. The trie nodes collectively form the *trie index.* In a simple trie, only one record is stored at each leaf. More generally, a leaf can have a capacity of l records in the range $1 \leqslant l \leqslant$ c where c is a fixed maximum capacity.

For example, let $\Sigma = \{0,1\}$, and $k = 4$, so that Σ^k is the set of all tetragrams of the binary digits 0 and 1. Let the file $F = \{0000,0001,0010,1001,1010,1011,1111\}$ and let the maximum capacity of a leaf be $c = 2$. Then we store the records in F in a trie as shown in Figure 2.1. An internal node is denoted by a rectangle; a leaf, by a circle. At an internal node, a left branch is taken when the corresponding digit is 0, a right branch is taken otherwise.

Instead of generating all internal nodes for all possible records in Σ^k, we store a record, such as 1111 in the figure, as a leaf as soon as it is one of at most 2 (because c=2) records in its subtree. Also, a null link "-" indicates an empty subtree, such as that for records 01** in Fig. 2.1. (Recall that "*" is used for an unspecified value.)

Because a leaf can store at most c records, when we insert a record into a leaf that already holds the maximum number of records, we must generate a new internal node and subdivide the c+1 records accordingly. Thus, we generated the leftmost internal node of the third level of the trie in Fig. 2.1 upon inserting the record 0010. This property of tries is identical to a property in extendible hashing which makes the two file-organization techniques similar. Larson [La] and Fagin et al , [Fa] note this connection between tries and extendible hashing.

2.2 Extendible Hashing

The storage for the file is organized into two levels: the *directory* and the *leaves.* The

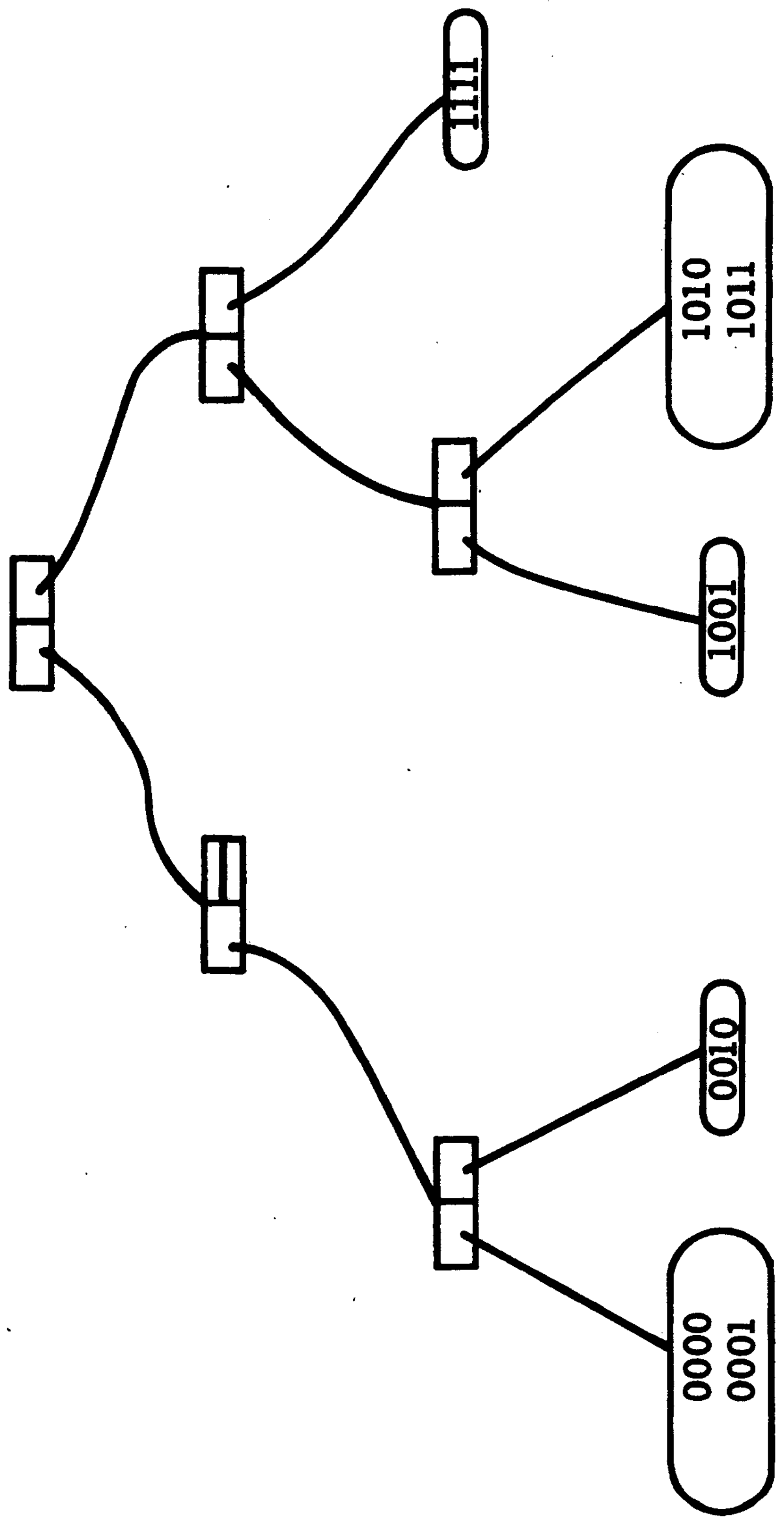

FIGURE 2.1. A TRIE

directory contains pointers to the leaves and the leaves store the records. Each leaf has a maximum capacity of c records.

Let h, a fixed hash function, be a mapping from the records in file F into the set B of infinite binary sequences. That is,

$$h : F \rightarrow B$$

such that

$$h(r) = (b_0, b_1, b_2, \ldots), \quad b_i \epsilon \{0,1\}, \quad i = 0,1,2,\ldots, \; r \epsilon F.$$

If r is the record to be stored then let $r' = h(r)$ be the *pseudokey* associated with r. For practical purposes and in the following discussion, we truncate each pseudokey to a moderate bit vector of fixed length.

The directory not only contains the pointers to the leaves but also a depth d which never exceeds the length of a pseudokey. This depth d determines the number of pointers (not necessarily distinct) in the directory to be 2^d.

How are these pointers organized? Informally, the pointers are laid out as follows. The first pointer in the directory points to a leaf that stores all records whose pseudokeys begin with d consecutive zeros. The second pointer points to a leaf that stores all records whose pseudokeys begin with the d bits 0...01. The next pointer is for all records whose pseudokeys being with the d bits 0...010, and so on lexicographically, to the final pointer for all records whose pseudokeys begin with d consecutive ones.

These pointers are not necessarily distinct as noted before. That is, neighboring pointers may point to the same leaf. What determines to which leaf a pointer points? Each leaf also has a local depth $d' \leqslant d$. The local depth d' for the leaf indicates the number of bits which are lexicographically the same in the pseudokeys for the records stored in that leaf.

For example*, if $d = 3$, the directory looks like the left side of Fig. 2.2. The 000 and 001 pointers point to the same leaf in the figure because all records whose pseudokeys begin with the two bits 00 hash to that leaf. The local depth d' for that leaf is correspondingly 2.

More formally, let f be the function mapping the set of natural numbers to the set of the d-bit binary representations of the natural numbers. That is, $f(n)$ is the d-bit binary representation of n where n is a natural number. Then the i^{th} entry in the directory points to the leaf storing all records whose pseudokeys begin with the leading d' bits of $f(i-1)$ where d' is the local depth of that leaf. (A quick check in the example shows that for $i = 1$, the first entry points to the leaf, with $d' = 2$, storing records whose pseudokeys begin with the leading 2 bits 00 of $f(0)$, as in the figure).

To do a simple insertion of a record r, first we compute $r' = h(r)$. Then, using the leading d bits of r', we find the entry of the directory which contains the pointer to the leaf where r is to be stored. Finally, we access that leaf from secondary storage and insert r in the leaf.

Two events may occur upon inserting a record in a leaf. The first occurs when an insertion causes a leaf p to overflow, that is, exceed the maximum capacity of a leaf. We allocate a new sibling leaf p' and *split* the pointers pointing to p between p and p'. We then increase the local depth of p by one and set the local depth of p' equal to this. Finally, we distribute all the records originally stored in p' between p and p'. Since the local depth of p has been

* This example and subsequent ones for extendible hashing are taken from [Fa].

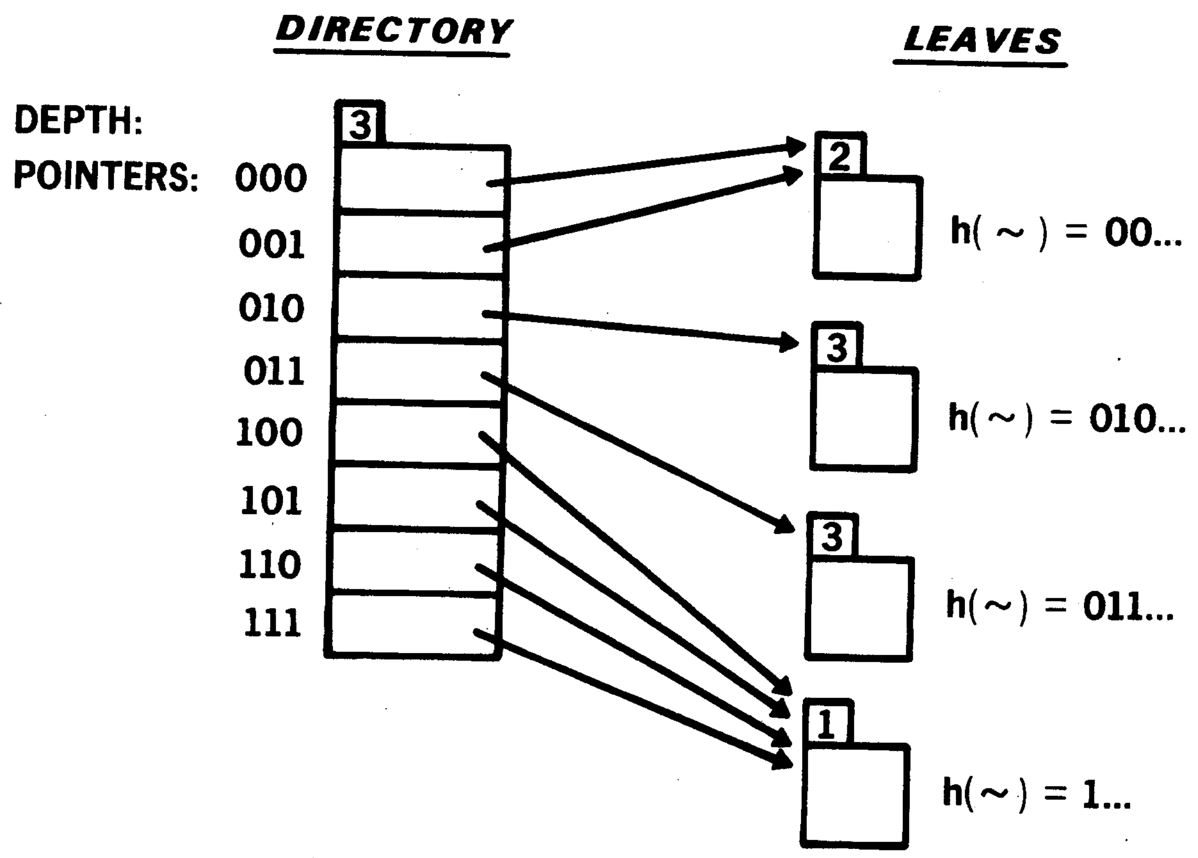

FIGURE 2.2. A DIRECTORY AND LEAVES

increased, using an additional bit from the pseudokeys of the records hopefully distributes the records between p and p' and not to just one of them. Figure 2.3 illustrates what happens when the leaf pointed to by the pointers 100,101,110,111 in Figure 2.2 overflows.

The second event occurs when an insertion overflows a leaf p and the increased (by one) local depth p' exceeds the directory depth d. Then, the directory *doubles* in size. We must then increase the depth of the directory by one and appropriately copy the pointers from the old directory to the new. That is, we only need to access the leaf p that overflowed and no others. Figure 2.4 illustrates how a directory doubles upon overflowing the leaf p pointed to by the 010 pointer in Figure 2.3. We allocate a new leaf p' and increase the depth of the directory and the local depth of p each by one. Pointers are copied in a straightforward manner.

Two similar events may occur upon deletion of a record: pointers *merge* and/or the directory *halves*.

2.3 A Brief Digression: Tries Versus Extendible Hashing

We now show the similarity between tries and extendible hashing. If we collapse the levels of a trie into a single level, this level is exactly the directory of the extendible hashing scheme. The leaves of a trie with capacity c are exactly the leaves in the extendible hashing scheme, also with capacity c. The records stored in the trie of Figure 2.1 would be stored in the directory and leaves of Figure 2.5 if the hashing function h is the identity function. That is, $h(r) = r$ for all $r \in F$. The number of levels of internal nodes in a trie equals the depth of the corresponding directory. The level at which a leaf of a trie is found equals the local depth of the corresponding leaf in the extendible hashing scheme. If the capacities of a trie leaf and a hashing leaf are the same then the contents of a trie leaf and its corresponding hashing leaf are identical.

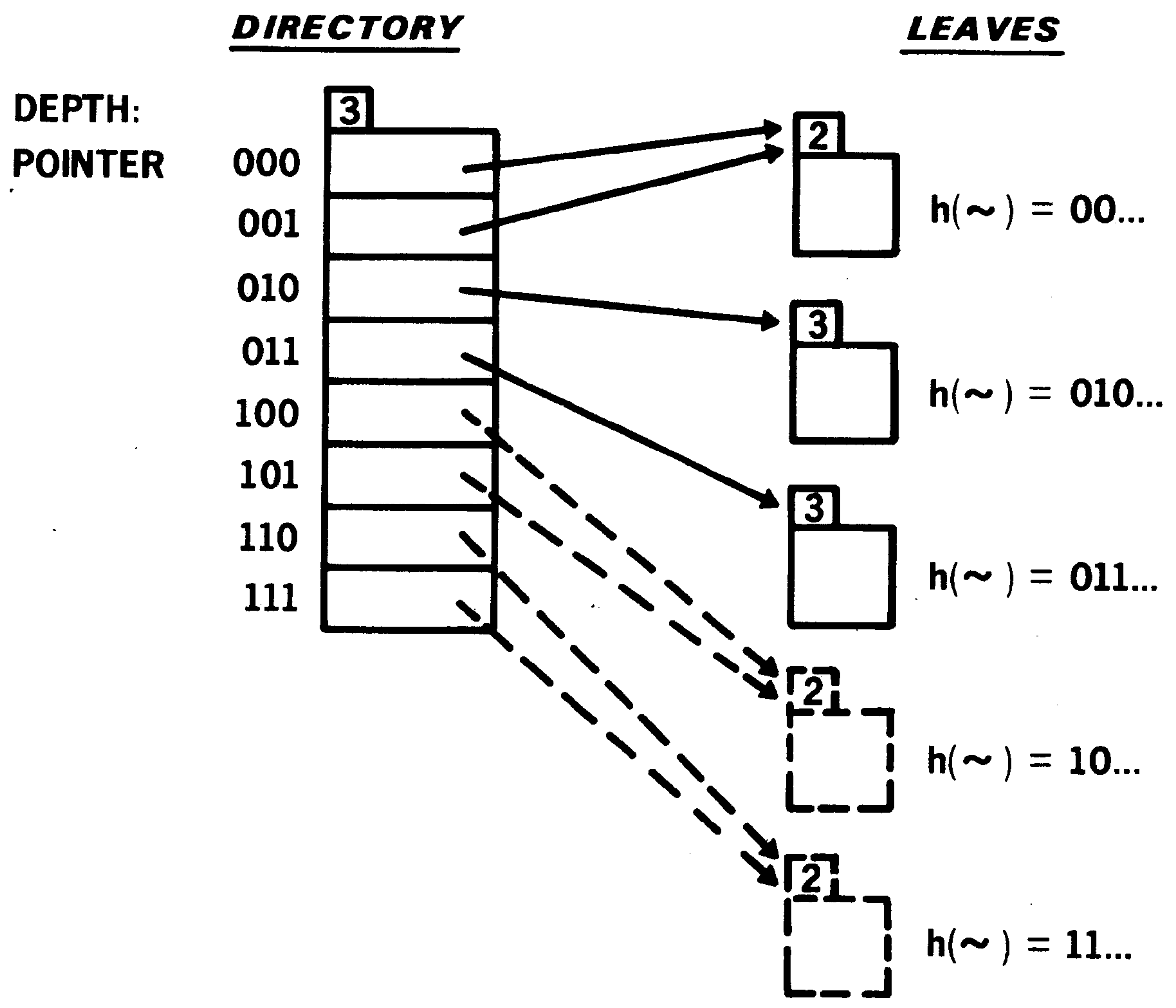

FIGURE 2.3. SPLITTING POINTERS

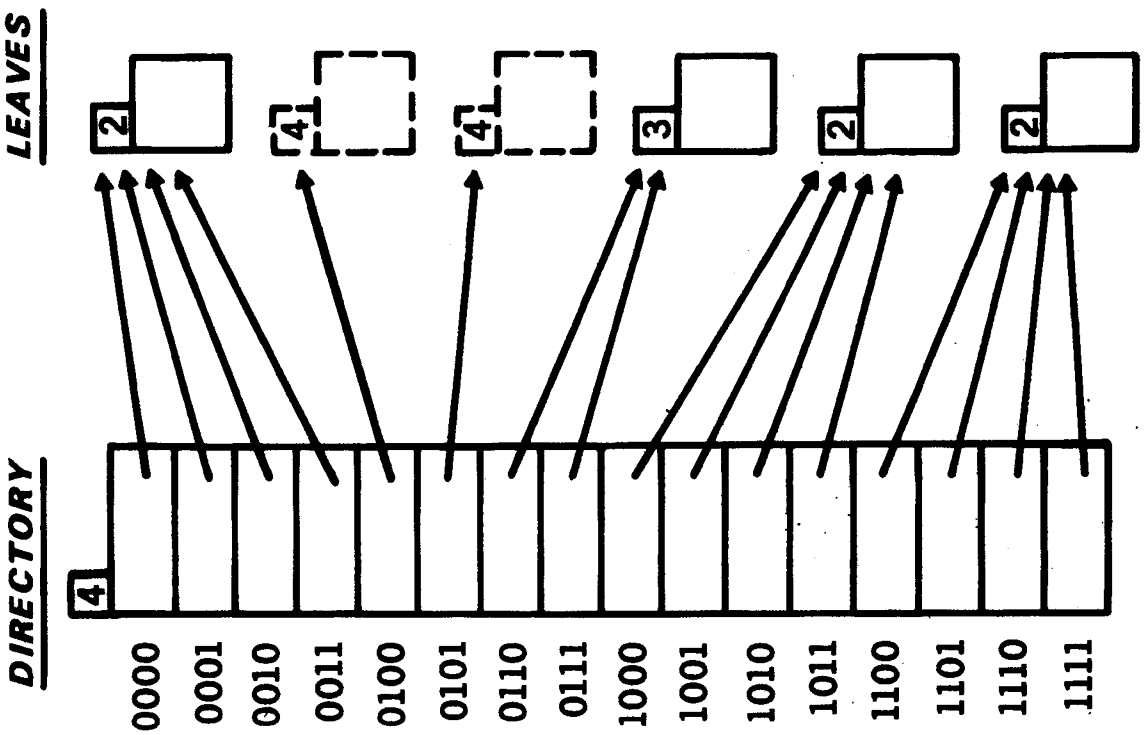

FIGURE 2.4. DOUBLING THE DIRECTORY

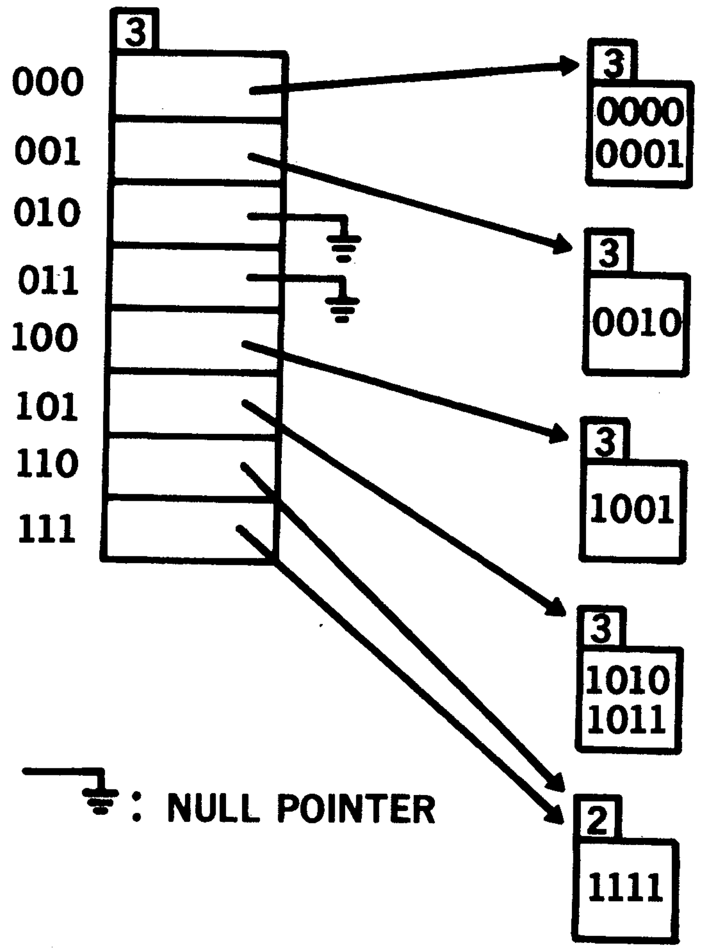

FIGURE 2.5. FIGURE 2.1 CONVERTED INTO A DIRECTORY AND LEAVES à la EXTENDIBLE HASHING

The difference, of course, is in the hashing function. If the identity function was not chosen for the above example, then the distribution of the records among the leaves might be different. Thus, one advantage of extendible hashing is that if the distribution of the records in the input file is itself not uniform, (which would lead to an unbalanced trie) then the distribution of the records after hashing could be uniform if a suitable hashing function is chosen.

Another advantage of extendible hashing is that the scheme guarantees at most two secondary storage accesses: one to the directory and one to the leaf. In fact, if the directory is small enough to fit into main memory, only one secondary storage access is needed. The number of secondary storage accesses in the worst case for a trie equals the number of internal nodes touched in traversing the path from the root to a leaf of the trie.

2.4 Superimposed Coding

Superimposed coding is a technique similar to hashing. However the terminology used in describing superimposed coding differs from that used for hashing.

We code each keyword r_i of a record $r = (r_1,...,r_k)$ into a binary bit vector of fixed length b using a coding algorithm C. This bit vector is called a *binary code word (bcw)*. We superimpose (logical inclusive OR) the bcw's (one for each keyword) of a record to form a *superimposed code word (scw)* for that record. Thus, we generate an scw for each record in the file F and keep each scw, along with a pointer to the corresponding record, in a secondary file S, the superimposed code word file.

Then, for a query $q = (q_1, \ldots, q_s)$, using the same coding algorithm C, we code the s specified keywords into S bcw's. We superimpose these bcw's to form a *query mask* and logically AND this mask with all the scw's in S. Finally, we put all scw's in S that have the same bits set as in the query mask in a *dropped* subset S'.

For example, Figure 2.6a shows a sample file of names. The keywords in the file are {John, Smith, Dave, Wing, Mary, Jane, Mark, Bell, Labs}. Let the coding algorithm set for each keyword in a record 3 random bits of a bit vector of length 10. Figure 2.6b shows the bcw's corresponding to each of the keywords. Then the two appropriate bcw's are OR-ed to form an scw for each record as shown on the right in Figure 2.6a. Bits that are not set are left blank for readability.

Suppose the query q_1 is {Smith}. Since the bcw for *Smith* sets bits 2, 3, and 10, we form the query mask 0110000001. We AND this query mask with each of the scw's in S and drop the subset $S' = \{0111000011, 0111011001\}$ corresponding to the records {John Smith, Mark Smith} respectively. However if the q_2 is {Jane}, the query mask is 1000100001 and the records {Dave Wing, Mary Jane} are retrieved. *Dave Wing* is a *false drop* in this case. Addition of the keyword *Mary* to the query q_2 will uniquely identify the record *Mary Jane.* However, uniqueness is not guaranteed: *Dave Wing* always falsely drops for *Bell Labs* since all the bits set by *Bell Labs* are a subset of the bits set by *Dave Wing.* Knuth gives a more extensive (and more appetizing) example of superimposed coding than the one presented here [Kn].

The records not dropped cannot possibly match the query. Let $S_g' \subset S'$ be the set of the scw's of all the records that match the query ("g" for "good") and $S_f' \subset S'$, the set of all records that do not match the query ("f" for "false"). Then $S' = S_g' \cup S_f'$ and $S_g' \cap S_f' = \emptyset$. Those records whose scw's are in $S - S'$ are guaranteed to not match the query.

Let F' be those records represented by the scw's of S'. Only the records in F' need to be retrieved from F. A search of F' is necessary to determine which records actually satisfy the query.

The number of false drops is determined by the length of the scw, the weight (number of bits set) of the bcw's, and the number of keywords per record. Increasing the length of the scw reduces the number of false drops, but increases the storage required for the

F		S										
		Superimposed Code Word										
Record Number	Name	bits	1	2	3	4	5	6	7	8	9	10
1	John Smith			1	1	1					1	1
2	Dave Wing		1		1		1			1	1	1
3	Mary Jane		1		1	1	1		1			1
4	Mark Smith			1	1	1		1	1			1
5	Bell Labs		1		1		1			1		

Figure 2.6a. A file *F* of names and the corresponding superimposed codeword file *S*.

					Binary Code Word						
Keywords in F	bits	1	2	3	4	5	6	7	8	9	10
John			1		1					1	
Smith			1	1							1
Dave		1		1					1		
Wing						1				1	1
Mary				1	1			1			
Jane		1				1					1
Mark					1		1	1			
Bell		1				1			1		
Labs				1			1			1	

Figure 2.6b. The bcws of the keywords of

file *F* of Figure 2.6a.

scw file. Optimally, the weight of an scw should be $b/2$, where b is the length of the scw [Ro]. Too large a weight for bcw's will cause too many bits to be set in the scw so that the number of false drops is large. However, too small a weight for bcw's will cause too few bits to be set in the scw, decreasing its effective length and also increasing the number of false drops. Thus, the coding algorithm is selected so that each scw will have approximately half of its bits set.

One advantage of superimposed coding is that operations such as retrieve and update are done on bit vectors. This lends itself readily to implementation in hardware, where parallel processing can be used to decrease the response time to a query. Both a software [Ga, 77] and a hardware [Ah] implementation exist at Bell Laboratories as an experiment to computer-aided directory assistance.

3. STORAGE

Sections 3 and 4 discuss the storage and retrieval time, respectively, for each of the three file organization techniques: tries, extendible hashing, and superimposed coding.

For tries and extendible hashing, we introduce a general model, develop a modified model, and present the results of the actual implementation. We compare predictions based on the modified model with measurements taken from the implementations. For superimposed coding we do not need a modified model and base our predictions on the general model.

Reasons discussed in these two sections and in Section 5 justify implementing a modified version of the general model for tries and extendible hashing. In fact, we modify the extendible hashing model sufficiently to warrant calling the implementation (simply) "hashing".

3.1 Tries

For partial-match retrieval, we build a trie from the set of all κ keywords, of a single attribute, found in the records of the file. Thus, we build a trie for each of the name, street, and town attributes of a telephone directory. We keep a list of records (or pointers to records) at each leaf of the trie and intersect these lists to obtain the records which match the query.

The keywords are words of finite length whose characters are from the alphabet Σ. The total space required for trie storage consists of the space used for the internal nodes plus the space used for the leaves. First, we will discuss the space for the internal nodes, then, the space for the leaves.

3.1.1 Internal Nodes

Assume that we build an M-ary ($M=|\Sigma|$) trie for κ keywords, and such that at most c records are stored at each leaf. Recall that c is the maximum capacity of a leaf. Knuth shows that for such a trie, the expected number of nodes is

$$\frac{\kappa}{c \cdot \ln M} - \kappa \cdot g(\kappa) - O(1) \tag{3.1}$$

where $g(\kappa)$ is a periodic function which can be ignored for small M and c because its absolute value is very small.

Various statistics of the file (see Appendix A) used for the implementation of trie-retrieval, such as the word and character frequencies, make the above assumptions for an M-ary trie too general. For instance, the word "INCORPORATED" occurs over 10,000 times in the name field of the single-line listings. We would need $c = 10{,}000$ and an internal node at level 12. But with a capacity of 10,000 we would waste storage for the many keywords that occur infrequently. Also, we might generate all internal nodes necessary for each keyword occurring not greater than 10,000 times, regardless of the length of the keyword. In the worst case, for example, if a word, 20 characters in length occurring once in the database, differs only on the 20th character from a word occurring c times in the database, we would have to generate a node at each of 20 levels and place the single record at the leaf on the 21^{st}. Clearly, an impractical and unnecessary amount of storage is used.

Thus, we add the following restriction to the model. We build a trie with at most d levels and use only d characters of each keyword for the trie index. For instance, we use only the first three characters of each keyword in the actual implementation. This implies that we do not store records at leaves once at most c records are at a leaf, but that we store records with the same leading trigram at the same leaf.

With this restriction to the general model, the number of nodes generated for a d-level trie in the worst case is:

$$\sum_{i=1}^{d} M^{i-1} = \frac{1-M^d}{1-M} \tag{3.2}$$

This assumes that for each node at level i, there is an M-ary branch to the nodes at level $i + 1$.

However, in practical implementations, a full trie is likely to be unnecessary. Instead, at each level i of the trie, only a fraction f_i of the possible i-grams will occur in the file. If the branching factor at level i is denoted M_i then there are at most $\prod_{j=1}^{i} M_j$ possible i-grams at level i. Let the number of nodes at level $i+1$ be n_{i+1}. Then $n_{i+1} = M_i \cdot n_i$, where each $M_i = \lfloor f_i M \rfloor$ for some fraction f_i of M. Thus, the total number of internal nodes is

$$\sum_{i=1}^{d} n_i = \sum_{i=0}^{d-1} f_i \cdot M \cdot n_i \tag{3.3}$$

where $f_0 = 1$, $n_0 = 1/M$ so that the number of nodes at level 1 is one and $M_1 = M$.

We approximate values for the f_i from the statistics of the file. [See Section 5.1.1 for a discussion of the need to take statistics.] The amount of storage actually required depends on the size of each node and the representation of the trie. In the implementation, we allow a maximum of $m \leqslant M$ characters for each internal node, thus saving space because many nodes require fewer than M characters. We also keep a pointer to an overflow node at each node because some nodes require more than m characters. Figure 3.1 gives an example for $d = 3$, $m = 4$, $M = 8$, and $F = \{$aagb, ababb, acad, ada, aeag$\}$. Call these nodes of size m, m-nodes, so that an *internal* node of a trie is comprised of at most $\left\lceil \frac{M}{m} \right\rceil$ m-nodes.

The number of m-nodes at level i depends on the branching factor at level i. It is the product of the number of internal nodes at level i and the number of m-nodes needed to support the branching factor at level i. That is, the number of m-nodes at level i is is $n_i \cdot \left\lceil \frac{f_i M}{m} \right\rceil$. Recall that the number of nodes at level $i+1$ depends on the branching factor at level i.

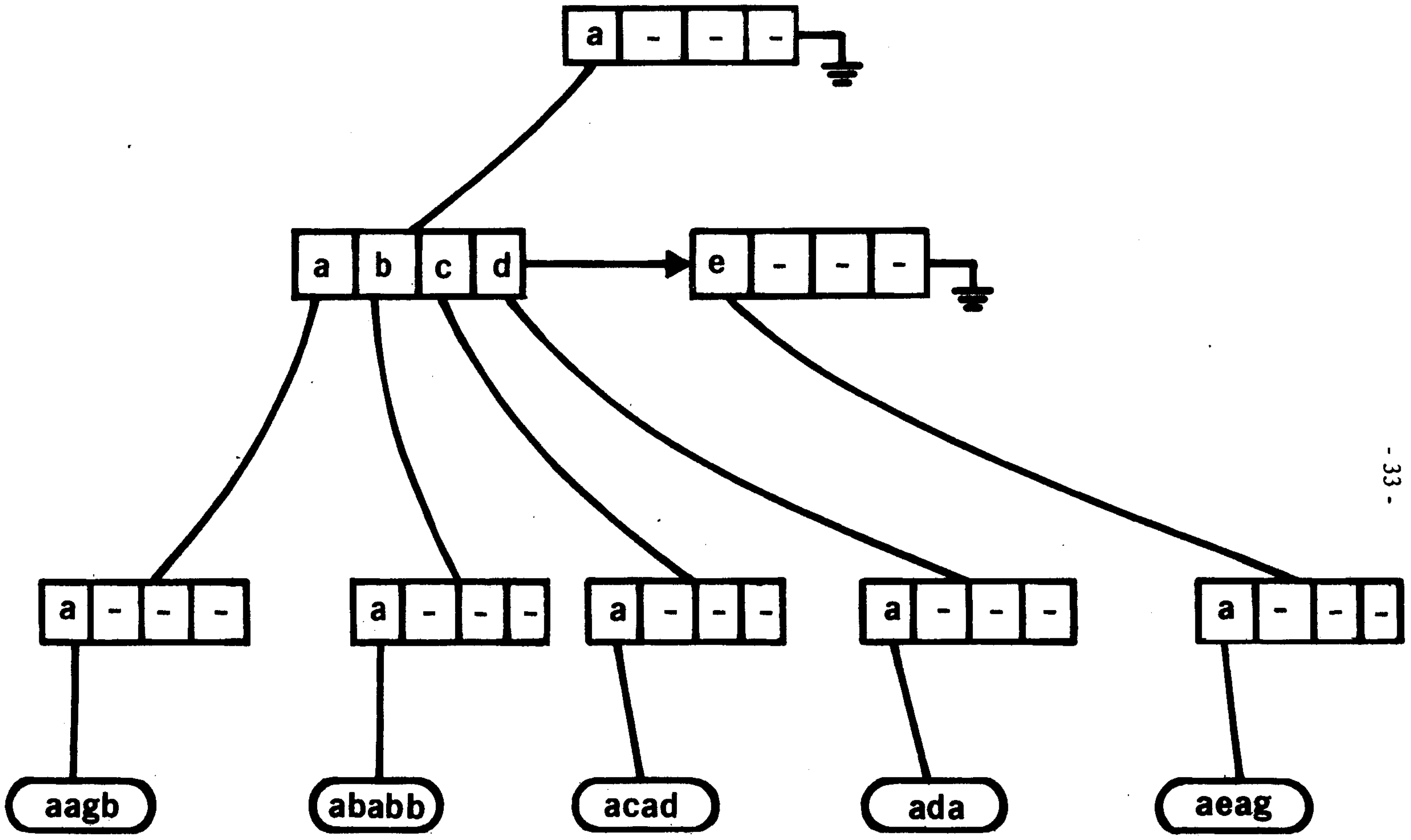

FIGURE 3.1. A TRIE WITH AN OVERFLOW NODE AT LEVEL 2

Also, we assume the branching factor is the same for each internal node at any given level. Thus, the total number of m-nodes is:

$$\sum_{i=1}^{d} \left\lceil \frac{f_i M}{m} \right\rceil n_i \ . \tag{3.4}$$

The total amount of storage, in bytes, is therefore

$$b_m \sum_{i=1}^{d} \left\lceil \frac{f_i M}{m} \right\rceil n_i \ . \tag{3.5}$$

where b_m is the number of bytes per m-node.

For the implementation, $M = 37$, $m = 14$, $d = 3$, and $b_m = 46$. For the name field, $f_1 = 1$, $f_2 = 2/5$, $f_3 = 1/6$. Equation 3.4 predicts that the total number of m-nodes is 632, requiring 29,072 bytes of storage. We actually allocate 616 m-nodes using 32,256 bytes of storage. We use more storage than predicted even though we allocate fewer m-nodes than predicted because of the way we packed the m-nodes in secondary storage. Also, the number of actual m-nodes is not equal to the predicted number because the f_i are approximations and the branching factor is not, in fact, uniform at any given level as assumed in the model. Table 3.1 summarizes these results plus those for the street and town fields. Note that the f_i change from field to field.

3.1.2 Leaves

Assuming each leaf with capacity c is 100 percent full, we need c times the size of a record to store records at the leaves.

Field	m-nodes		bytes	
	Predicted	Actual	Predicted	Actual
Name $f_1=1$ $f_2=2/5$ $f_3=1/6$	632	616	29072	32256
Street $f_1=1$ $f_2=1/4$ $f_3=1/10$	410	396	18860	22528
Town $f_1=2/3$ $f_2=1/7$ $f_3=1/20$	170	152	7820	9728

Table 3.1. Storage for internal nodes for tries

However, because of the similarity between tries and extendible hashing, we can apply Larson's results on the capacity of each hash bucket to the capacity of a trie leaf. Larson shows that for $c \geqslant 9$, an average of 69.3 percent of the leaf is used and that the expected number of allocated leaves a is

$$E(a) \approx \left(\frac{N}{c \cdot \ln 2}\right)\left[1-c\left(1-\ln 2-2\sum_{i=2}^{c}\frac{2^{-i}}{i(i-1)}\right)\right]. \qquad (3.5)$$

Thus the expected amount of storage used for leaves is $(.693)E(a)\cdot$ (size of a record).

One modification to the general model of a trie proves useful. Instead of storing the records themselves at the leaves, we store pointers to the records. Using pointers has two advantages. One is that we save storage. We need only store each record once in a separate file instead of k times for the k keywords in the record. This advantage arises when we do not have a single primary key for each record as is the case for partial-match retrieval. The other advantage is that using pointers is more amenable to partial-match queries than exact-match or single-key queries. A leaf is a list of pointers to the records that have the keyword whose insertion in the trie led to that leaf. For each keyword in the query, we retrieve the corresponding list. Then we intersect lists of pointers (instead of lists of records) to find the records which match the query.

With this modification, if the size of the file is N, the number of keywords in each record is k, and the number of bytes per pointer is b_p, then we require $N\cdot\kappa\cdot b_p$ bytes of storage for the lists.

These lists vary in size. In practice, we could represent a list as a linked list or as an array of fixed maximum size. An intermediate approach is to use bucket chaining where each bucket is of fixed size of B entries. This parallels the method of linked m-nodes used for internal nodes of the trie. That is, we use linked buckets for the leaves of a trie. This implementation decision leads to the following model for the total amount of leaf storage:

$$\frac{N\cdot k}{B}\, b_B\,. \tag{3.6}$$

where b_B is the size of a bucket in bytes.

For the actual implementation of the trie for the name field, $N = 50,000$, $k = 4$, $B = 8$, *and* $b_B = 36$. Then Equation 3.6 predicts a total of 25,000 buckets using 900,000 bytes of storage. We actually allocate 21,448 buckets using 784,384 bytes. Table 3.2 shows the results for the street and town fields as well. Note that the average number of keywords is different for each field. Since these average numbers of keywords are approximations the difference between the actual and predicted number of buckets is not unexpected.

Field	buckets		bytes	
	Predicted	Actual	Predicted	Actual
Name (k=4)	25000	21448	900000	784384
Street (k=3)	18750	13524	675000	494592
Town (k=2)	12500	8470	450000	309760

Table 3.2. Storage for leaves for tries

3.2 Extendible Hashing

Extendible hashing requires storage for the directory and the leaves. The similarity between tries and extendible hashing makes the analysis of the storage requirements for extendible hashing similar to that for tries.

Larson represents a directory by a forest of binary hash tries. As shown in Section 2, these tries can be collapsed into one level which is exactly the "directory" introduced by Fagin, et al [Fa]. Their analysis of storage requirements considers two probabilistic models of the inputs: the Poisson model and the Bernoulli model. For the Poisson model, the number of records is a Poisson distributed random variable; for the Bernoulli model, the number of records has a deterministic value. The results for both these models are the same and are discussed in Section 5.1.2. In the following sections we discuss first the storage for the directory, then the storage for the leaves.

3.2.1 Directory

Using Larson's results for the expected number of external nodes for his binary tries, the expected number of distinct directory entries is $\frac{N}{c \cdot ln2}$ where N is the number of records and c is the capacity of a leaf. Assuming that more than half of the directory entries are distinct, then the depth of the directory is $\left\lceil \log \frac{N}{c \cdot ln2} \right\rceil$ (logarithm to the base 2). If each directory entry is b_e bytes, the size of the directory is

$$2^{\left\lceil \log \frac{N}{c \cdot ln2} \right\rceil} \cdot b_e + b_d.$$

where b_d is the number of bytes to store the depth of the directory.

The expression for the depth of the directory merits more analysis than presented here. In fact, the "birthday paradox" [Kn] helps justify the given expression for large enough c. Section 5.1.2 discusses this issue in more depth (no pun intended).

As with the case for tries, we adapted to the peculiarities of the database by modifying our approach to extendible hashing. For instance, a word like "INCORPORATED" would hash to the same leaf over 10,000 times since it occurs over 10,000 times in the database. Since the blocksize on the UNIX operating system is 512 bytes, the total number of bytes for a leaf could not be greater than 512.

At first, we considered keeping a separate file for those words that occur frequently in the database. However, this file would be very large since for just the name field, 88 words occur over 300 times.

Instead, we decided to link buckets together as done with trie leaves. Thus, an entry in a directory points to a leaf which is a linked list of buckets. Note that we lose one of the advantages of extendible hashing: we now may need more than two secondary storage accesses to retrieve a record.

In extendible hashing a directory may double or halve upon overflowing or emptying a leaf. Since a leaf no longer has a fixed size, the directory can theoretically never double or halve. Instead, we give the directory a fixed depth *d*. Thus, the predicted and actual values for the size of the directory are the same. In the implementation $d = 12$ so that the directory for each of the name, street, and town fields has 4096 entries at 2 bytes per entry plus 2 bytes for storing the depth of the directory.

We could implement extendible hashing using linked buckets by keeping a sum of the number of entries in the buckets that are linked together. Once this sum exceeds a predetermined number then the directory doubles. Similarly, once this sum falls below a certain level the directory halves. This option requires a storage manager to dynamically allocate and free buckets.

3.2.2 Leaves

For the general model of extendible hashing, Larson derives an expression for the expected number of leaves allocated. It is the difference between the expected number of directory entries (Larson's term is "external nodes") and the expected number of directory entries which contain duplicate pointers. Larson's term for this is "empty buckets." He derives the approximation in Equation 3.5 as the expected number of leaves allocated. Fagin, et al discuss the number of leaves in a probabilistic argument (see Section 5.1.2).

However, since we store leaves for hashing in the same manner as we do for tries, we can use Equation 3.6 repeated below. Again because of partial-match retrieval and storage constraints, we keep pointers to records in the leaves.

If N is the size of the file, k is the number of keywords, B is the number of entries in a bucket and b_l is the size of the bucket in bytes, the number of bytes to store the leaves is

$$\frac{N \cdot k}{B} b_l \tag{3.6}$$

Table 3.3 presents the results from building a hash directory for each of the name, street, and town fields with varying k, $N = 50{,}000$, $B = 84$, and $b_l = 512$.

Two reasons for the difference between the predicted and actual values are that the buckets are not 100 percent full and that the k's given are approximations.

	buckets		bytes	
Field	Predicted	Actual	Predicted	Actual
Name (k=4)	2380	4861	1219047	2488832
Street (k=3)	1785	2880	914285	1474560
Town (k=2	1190	1009	609523	516608

Table 3.3 Storage for leaves for hashing

3.3 Superimposed Coding

Superimposed coding only requires the additional superimposed code word (SCW) file in addition to the records themselves. If each scw is b bits in width and the size of the file is N, then the total number of bytes of storage for the scw file is

$$\frac{N \cdot b}{8} \tag{3.7}$$

where 8 is the number of bits per byte. This assumes the width of an scw is a fixed size.

Gabbe et al, [Ga] use a larger input file in implementing superimposed coding than we use for tries and extendible hashing. The number of records is 58,587 and includes multi-line and structured listings (e.g., U. S. Government). As a result, they use two code word sizes: 127 bits for most (72 percent) of the listings and 191 bits for multi-line and structured listings. Thus the index file occupies 1,061,303 bytes, or about 1 megabyte. If each listing occupies an average of 80 bytes, then the superimposed code word index is approximately 20 percent of the file of records itself.

3.4 Summary

Table 3.4 summarizes the models used to predict the amount of storage required for tries, extendible hashing, and superimposed code words. The expression given for extendible hashing is for true extendible hashing and not for our implementation. If we do not store the actual records at the leaves of a trie or at the leaves in extendible hashing then the size of the file must be added to each of the corresponding expressions. This term is included in the expression for superimposed coding.

Tries	$b_m \sum_{i=1}^{d} \left\lceil \frac{f_i M}{m} \right\rceil n_i + \frac{N \cdot k}{B} b_B$ (storage for 1 trie)
	where b_m = number of bytes per m-node b_B = number of bytes per bucket B = number of entries in a bucket d = number of levels of the trie f_i = fraction of i-grams that occur of all possible i grains k = average number of keywords in a record m = maximum branching factor of a m- node M = maximum branching factor of an internal node n_i = number of internal nodes at level i N = number of records in the file

Extendible Hashing	$2\left\lceil \log\frac{N}{c\cdot\ln 2}\right\rceil\cdot b_c \quad + b_d + \frac{N}{c\cdot\ln 2}\left\{1 - c\left(1 - ln2 - 2\sum_{i=2}^{c}\frac{2^{-i}}{i(i-1)}\right)\right\}b_l$
	where $b_d =$ number of bytes to represent an integer (for the depth of the directory) $b_c =$ number of bytes per directory entry $b_l =$ number of bytes per leaf $c =$ maximum capacity of a leaf $N =$ number of records in the file

Superimposed Coding	$\frac{N\cdot b}{8} + N\cdot b_r$
	where $b =$ number of bits per superimposed code word $b_r =$ number of bytes per record $N =$ number of records in the file

Table 3.4. Summary of storage requirements for tries, extendible hasing, superimposed coding (units in bytes).

4. RETRIEVAL TIME

4.1 Tries

4.1.1 General Model

For the general model of a trie, Rivest [Ri,74] gives a recursive algorithm for retrieving a record from a trie. He derives expressions for upper and lower bounds for searching in a full trie in [Ri,74] and an expected cost for searching in a binary trie in [Ri,76] in terms of the number of internal nodes examined by the retrieval algorithm. These expressions are $O(N^{(k-s)/k})$ for N records in the file, k keywords per record, and s specified keywords per query.

For a M-ary trie with a capacity of s records for a file of N records, Knuth [Kn] gives an expression for the average number of characters examined in a successful search to be

$$\frac{lnN + \gamma - H_{c-1}}{lnM} + ½ - \hat{g}(N) + O(N^{-1}) \tag{4.1}$$

and the average number of comparisons made in a successful search to be

$$1 + ½(1-\frac{1}{M})\left[\frac{c-1}{lnM} + \tilde{g}(N)\right] + O(N^{-1}) \tag{4.2}$$

where γ is Euler's constant, H_i is the i^{th} harmonic number, and $\hat{g}(N)$ and $\tilde{g}(N)$ are periodic functions similar to $g(N)$ mentioned in Section 3.1.

4.1.2 Modified Model: Retrieval Algorithm

For the modified model of the trie used for the implementation of trie retrieval, we fixed the number of levels of the trie and we let the leaves store a varying number of pointers to records. Given a query $q = (q_1, \ldots, q_k)$, for each query keyword q_i we search the trie on d of the letters in the keyword and return the leaf storing a list of pointers to all records containing those d letters in the keyword. Then we intersect the returned lists to yield a final list of pointers to records. We retrieve each of these records from the file and match each of the actual query keywords with the keywords of the record, since we only use d letters of the query keyword search. Thus, we use the following algorithm for an M-ary trie with m-nodes used to represent an internal node. Note that an m-node stores at most m characters and m "child" pointers to subtries and an "onode" to an overflow m-node. Initially, *trieretrieve* is called with t as the root of the trie and query q.

```
Procedure trieretrieve (q,t):
begin
          for $q_i \in$ q  = $(q_i,\ldots,q_k)$
               $l_i$ = triesearch ($q_i$, t);

          $l_{final}$ = intersect ($l_1,\ldots,l_k$);
          for $r_i \in l_{final}$
               if (match ($r_i$,q) = true) print $r_i$)
end trieretrieve

Procedure triesearch (keyword,t):
begin
          j=1;
          while j<d+1 begin
                    i=1;
                    while i<m+1 begin
                              if (keyword$_j$ = t→char$_i$)
                                        if(j=d) return (t→ child$_i$)
                                        else begin
                                                  t=t→child$_i$;
                                                  j=j+1;
                    i=0;
                end
            i=i+1;
        end
        if (i=m)
            if(t→onode != null) begin
                t=t→onode;
                j=j-1;
            end
            else begin
                print ("no such keyword");
                return (null);
            end
        j=j+1;
    end
end triesearch
```

4.1.3 Modified Model: Time Analysis

The intersection is done optimally if we sort the lists and intersect them in increasing size order [AMP]. The *intersect* and *match* routines are done in main memory. Thus, the retrieval time is bounded by the number of records actually retrieved plus the product of the time to execute a *triesearch* and the number of times we call it. We call it once for each query keyword.

The time to execute each *triesearch* is bounded by the number of levels of the trie. Thus, if examining an internal node requires one secondary storage access, we will need d secondary storage accesses to search down a trie of d levels. Then we need an additional access to retrieve the list of pointers to records.

We model the retrieval time T_R to be:

$$T_R = r_q T_a + k(d+1)T_a + T_I + r_q T_M \tag{4.3}$$

where T_a is the time to do a secondary storage access; T_I is the time to do the intersection of lists; T_M is the time to do a match; r_q is the number of records that must be retrieved; k is the number of query keywords; and d is the number of levels in the trie. Since T_I and T_M are on the order of one or two milliseconds, actual retrieval time is bound by the first two terms of T_R to yield

$$T_R \doteq [r_q + k(d+1)]T_a \tag{4.4}$$

We can break T_a down into the sum of three components: seek time s, rotational latency r, and block transfer time btt. Now we can predict the time needed to retrieve the records satisfying a query of k keywords.

For the actual implementation the secondary storage device used is a Digital Equipment Corporation RP02 disk. Average seek time is 47 msec., and average latency is 12.5 msec. The transfer time is 7.5 μsec/word, yielding a transfer time of 1.92 msec/block. Thus $s+r+btt$ = 61.4 msec. For an average of 2 keywords/query and a median of only one record ($r_q=1$) actually retrieved to satisfy a random query, Equation 4.4 yields $9T_a$ or 0.55 seconds.

We tested two sets of queries. One set contained random trigrams grouped in queries of one, two, and three trigrams per query. However, two random trigrams in a single query rarely resulted in any records matching the query. Thus, we did not pursue timing this set of test queries.

We made the second set of queries in the following manner. The first query consists of a randomly selected trigram. The second query consists of the same random trigram plus a different trigram actually present in one of the records retrieved by the first query. The third query added a third trigram (if present) to the second query. We repeated this procedure until we had 20 queries.

We then measured the time to perform these 20 queries. We made twelve trials: for the first six, we suppressed the actual printing of the records and for the remaining six, we printed them to verify that the retrieval was done correctly. Table 4.1 shows the results of the first six timing experiments.* It presents the total times for secondary storage access. Appendix B presents the results from the *time* command, and the results of user and system times from the *times* subroutine.

The average I/O time per query is about two and a half times the predicted. The UNIX operating system performs an additional disk access per user read request in maintaining the file system. This accounts for the difference in retrieval time between the actual and predicted values.

4.2 Extendible Hashing

4.2.1 General Model: Retrieval Algorithm and Retrieval Time

Retrieval in an extendible hashing scheme requires at most two secondary storage accesses: one for the directory and one for the appropriate leaf. If the directory can be read into main memory initially and kept resident, then a retrieval will require only one secondary storage access.

For partial-match retrieval, each keyword in the query takes at most two accesses. We compare the records on the retrieved leaves for an intersecting set of records that satisfy the query. Or similarly if we keep pointers to records in the leaves, we intersect the lists of pointers as with the leaves of a trie discussed in the previous section.

We use the following partial-match retrieval algorithm for extendible hashing. Let *directory*, an array of pointers, be the hash directory, and let the query be *q*.

* Using the UNIX *time* command, we measured the total elapsed ("real"), user, and system times in executing the entire process to perform the 20 queries. This includes opening and closing files and other initialization and cleanup procedures. Also we used the UNIX *times* subroutine to measure the user and system times of executing just the retrieval for the 20 queries. The time spent doing secondary storage accesses (I/O time) is approximately the difference between the elapsed time and the sum of the user and system times.

Trial	Total I/O time for 20 queries (in seconds)	Average I/O time/query (in seconds)
1	27.8	1.4
2	26.5	1.3
3	27.5	1.4
4	26.5	1.3
5	29.3	1.5
6	28.9	1.4
Average	27.7	1.4

Table 4.1. I/O time for tries for 20 queries and per query. The predicted I/O time is 0.55 seconds per query.

```
**Procedure** *hashretrieve* (q) **begin**
    **for** $q_i \in q = (q_1,...,q_k)$
        $l_i$ = *hashsearch* $(q_i)$
    $l_{final}$ = *intersect* $(l_1, \ldots, l_k)$;
    **for** $r_i \in l_{final}$
    **if** (*match* $(q,r_i)$ = **true**) **print** $(r_i)$;
**end**
**end** *hashretrieve*

**Procedure** *hashsearch* (keyword): **returns** leaf
**begin**
    hashed_keyword = $h$ (keyword);
```

```
        directory_entry = entry (hashed_keyword);
        leaf_pointer = directory [directory_entry];

        read leaf_pointer into buffer;
        leaf = search (buffer, keyword);
end
end hashsearch
```

The efficiency of the *search* and *intersect* subroutines depends on how the entries on the leaves are kept. If they are sorted then binary search yields an 0(log n) time where n is the number of entries on a leaf. Also (as mentioned in the previous section) the *intersect* subroutine can be optimally fast if we intersect sorted lists in order of increasing size. We execute the *intersect, match, h, entry* and *search* subroutines all in main memory.

Thus the retrieval time is bounded by the time necessary to perform at most two secondary storage accesses per query keyword. This yields the following expression for partial-match retrieval time for extendible hashing:

$$T_R = r_q T_a + \alpha \cdot k T_a + k T_S + T_I + r_q T_M \tag{4.5}$$

where k is the number of query keywords, T_a is the average access time, and r_q is the number of records actually retrieved. T_a is the sum of the seek time, rotational latency and the block transfer time. T_S, T_I, T_M, are the times to perform the *search, intersection,* and *match* subroutines, respectively. The times to execute h and *entry* for each query keyword are not significant.

The coefficient α in the second term in Equation 4.5 is 1 if we keep the directory resident in main memory; otherwise it is 2.

Retrieval time is bounded by the number of secondary storage accesses leading to the following expression:

$$T_R \doteqdot (r_q + \alpha \cdot k) T_a \tag{4.6}$$

4.2.2 Modified Model: Implementation Results

The actual implementation does not guarantee at most two secondary storage accesses. Depending on the number of entries in one leaf bucket and the number of pairs of keywords and record pointers that hash to the same bucket, additional secondary storage accesses may be necessary to find the appropriate hashed keyword in the leaf. However, the expected number of entries in one leaf bucket is less than the maximum size chosen for a leaf bucket, so the expected number of secondary storage accesses is still at most two per query keyword.

Table 4.2 presents the results from six trials on the same set of queries as those used for tries. Using the same values for s, r, and *btt* as for tries, Equation 4.6 yields a time T_R of 0.25 seconds per query, where $\alpha=1$ since we keep the directory in main memory, k is an average of two keywords per query, and $r_q=2$. (One record satisfies the random query; one record does not but is retrieved because of keyword collisions.) Appendix B includes the breakdown of these numbers in terms of user time and system time.

The actual I/O per query is about three times more than predicted because the use of linked buckets and because of keyword collisions. In particular, for one query for which one keyword is "INCO", we read 98 buckets ($c=98$) instead of the expected value of 1. Also for two other queries we retrieved (r_q) 38 and 64 records respectively when actually only one record satisfied each query. Finally, the additional I/O performed by the UNIX operating system contributes to the actual I/O retrieval time per query. All these factors account for the difference between the actual and the predicted results.

4.3 Superimposed Coding

4.3.1 Retrieval Algorithm

The following procedure presents the steps required to do retrieval via superimposed coding given a query q, and superimposed code word file S and file F of records.

```
procedure scretrieve (q)

    qmask = 0;
    begin
        for $q_i \epsilon q = (q_1,\ldots,q_k)$
          $qmask = qmask \vee h(q_i)$ /* $\vee$ is bitwise inclusive "OR" */
        for $(s_i \epsilon S)$
          if $(s_i \wedge qmask)$ then
          put $r_i$ in $F'$ /* " $\wedge$ is bitwise "AND" */
        for $(r_i \epsilon F')$
          if $(match(r_i,q)$ = true) print $(r_i)$
    end
end scretrieve
```

The time to retrieve records is bounded by the number of secondary storage accesses needed to compare the query mask with the scw's of the superimposed code word file.

The comparison can be done in two ways. Since the superimposed code word file is just an $N \times b$ array of 0's and 1's, it can be stored by row or by column. (Here N is the number records and b is the width of an scw.) If the SCW file is stored by row, then each row vector represents a record in the file and is b bits in width. Retrieval requires the query mask to be compared with all N of the records making b bit comparisons for each record.

However, if the SCW file is stored by column, then each column vector represents one of the b positions in an scw and is N bits in length. Retrieval requires only $b' \leqslant b$ comparisons of bit vectors with the query mask where b' is the number of 1's in the query mask. Figure 4.1 illustrates this for the query {Smith} and the example given in Section 2.4. Only the bit vectors corresponding to rows 2, 3, and 10 need to be retrieved for the records {John Smith, Mark Smith} to be dropped. Therefore, we do not need to look at all $N \cdot b$ bits in the SCW file, just $N \cdot b'$ bits. Roberts [Ro] calls this method of storing the SCW file "bit-sliced organization."

4.3.2 Retrieval Time

Using bit-sliced organization, we would store N bits per bit vector. Thus, we model the retrieval time T_R to be:

Trial	Total I/O time for 20 Queries (in seconds)	Average I/O time/query (in seconds)
1	16.2	.8
2	15.4	.8
3	16.2	.8
4	15.7	.8
5	14.8	.7
6	16.6	.8
Average	15.8	.8

Table 4.2. I/O Time for Hashing for 20 Queries and per query. The predicted I/O time per query for extendible hashing is 0.25 seconds per query.

$$T_R = T_Q + T_B + r_q T_a + r_q T_M \tag{4.7}$$

where T_Q is the time to make the query mask, T_B is the time to retrieve bit vectors from the SCW file, T_a is the time to retrieve a record from secondary storage, T_M is the time to match a retrieved record with the query. The retrieval time is bounded by secondary storage accesses so Equation 4.7 reduces to

$$T_R \doteqdot T_B + r_q T_a \tag{4.8}$$

bit number	SCW File record number 1	2	3	4	5	query mask
1	0	1	1	0	1	0
2	1	0	0	1	0	1
3	1	1	1	1	1	1
4	1	0	1	1	0	0
5	0	1	1	0	1	0
6	0	0	0	1	0	0
7	0	0	1	1	0	0
8	0	1	0	0	1	0
9	1	1	0	0	0	0
10	1	1	.1	1	0	1

Figure 4.1. If the SCW file is stored by column, only the three bit vectors corresponding to bit positions 2, 3, and 10 need to be retrieved for the records 1 and 4 to be dropped.

The time T_B can be represented as follows:

$$T_B = \frac{N}{8 \cdot b_s \cdot t} \cdot (r + btt) \cdot q + s \cdot q \qquad (4.9)$$

where b_s is the number of bytes per sector, 8 is the number of bits per byte, t is the number of sectors per track, q is the average weight of the query mask, and s, r, and btt are seek time, rotational latency, and block transfer times, respectively. We derive the factor $\frac{N}{8 \cdot b_s \cdot t}$ from the number of tracks necessary to store the SCW file. If there are N records, then each bit vector is N bits in length.

4.3.3 Implementation Results

Equation 4.8 yields a time T_r of 5.0 seconds per query, where N is 60,000 records, b_s is 512 bytes/sector, t is 10 sectors/track, q is 73 bits, r_q is 1, and s, r, and btt are 47, 12.5, and 1.92 msec. respectively.

Table 4.3 presents the results from six trials on the same set of queries as these used for tries and hashing. Appendix B includes a breakdown of these numbers in terms of user and system times.

The implementation for superimposed coding uses 11 UNIX processes, not all of which can fit into main memory at once. Thus about a 2-second overhead for swapping processes in and out of main memory accounts for part of the difference between the actual and predicted values.

4.4 Summary

Table 4.4 summarizes the retrieval times for tries, hashing, and superimposed coding.

Trial	Total I/O time for 20 Queries (in seconds)	Average I/O time/query (in seconds)
1	179.0	9.0
2	179.5	9.0
3	177.8	8.9
4	183.3	9.2
5	192.9	9.7
6	179.8	9.0
Average	182.1	9.1

Table 4.3. I/O time for superimposed coding for 20 queries and per query. The predicted I/O time is 5.0 seconds per query.

Tries	$T_R = r_q T_a + k(d+1) T_a + T_I + r_q T_M$
Hashing	$T_R = r_q T_a + \alpha \cdot k T_a + k T_S + T_I + r_q T_M$
Superimposed Coding	$T_R = T_Q + T_B + r_q T_a + r_q T_M$.

where

$\alpha =$	1 if the hashing directory is kept in main memory; 2, otherwise
$d =$	the number of levels in a trie
$k =$	the number of query keywords
$r_q =$	the number of retrieved records
$T_a =$	access time (seek + rotational latency + block transfer time
$T_B =$	time to retrieve bit vectors from the SCW file
$T_I =$	time to intersect lists
$T_M =$	time to match a retrieved record with the query
$T_Q =$	time to make a query mask

Table 4.4. Summary of retrieval times for tries, hashing, and superimposed coding.

5. DISCUSSION

5.1 Storage and Retrieval Time

5.1.1 Tries

Are frequency counts necessary?

The present model for the amount of storage needed for tries depends on the statistics of the data base. This dependency has its drawbacks if a general model is desired but in practice it is not an unreasonable task to take statistics on the database. In fact, the database organizer often computes statistics such as character and keyword frequency counts depending on the application.

The question arises whether the model can be extended for a file from several thousands to over a million records and whether keeping the frequency counts of digrams and trigrams occurring in such a file is plausible. Sorting the keywords (as done in this study, for convenience) can be avoided by keeping approximate counts of these large numbers [Mor]. Thus we only need one pass through all keywords in the records.

Also, depending on the statistics of the database, we may choose to build the trie index on the first *d* consonants of each keyword or any combination of *d* letters of each keyword where *d* is the number of levels of the trie index. The objective is to distribute the records somewhat uniformly across the leaves of the trie, so that the entire trie is balanced. (This hints again at the similarity between using tries and hashing.)

Implementation issues.

Because of the limited address space of the PDP 11/45, we could not construct the entire trie index in main memory. Thus, in constructing and retrieving from the tries (one each for the name, street, and town fields) we simulate a cache memory scheme using a least recently used algorithm for buffer replacement. Since we chose the maximum number of characters stored in an *m*-node to be 14, the maximum expected numbers of *m*-nodes at levels one, two, and three are 3, 2, and 1, respectively. One buffer of 512 bytes can hold at most 11 *m*-nodes. Thus we waste storage in fragmentation, and the number of actual bytes used to store the trie index is greater than the predicted value.

Similarly, we use separate buffers and UNIX files to store the pointers to records in the leaves of the trie. Because the leaves are stored by linked buckets, we make additional disk accesses when a leaf overflows into the next disk block. That is, 14 buckets can fit onto one UNIX disk block and 8 pointers can fit into one bucket. If a leaf points to a bucket which is linked to another bucket on to a different block then both blocks may have to be read into main memory. We chose the size of the bucket to approximate the average number of records that would have any random trigram occurring in the file. Clearly a trigram like "INC" (as in "INCORPORATED") will generate a leaf that spans several disk blocks and thus if "INC" occurs in a random query (as it does in two of the test set of 20 queries) the number of disk accesses can dramatically increase. This also accounts for part of the difference between the actual and predicted values for retrieval time.

5.1.2 Extendible Hashing

Depth of the directory.

Hashing is attractive because of its $O(1)$ exact-match retrieval time and because a uniform distribution of hashed keywords can result if a good hash function is chosen.

However, collisions can hamper the effectiveness of the hashing technique and in fact, can be devastating in the case of extendible hashing.

The problem of clustering occurs if there is a collision in the (full) hash value when a leaf overflows. This causes the directory to double in size (because the records hash to the same directory entry) until in the worst case the maximum size of the directory is reached. In other words, just redistributing one leaf's entries may generate many directory entries which contain duplicated pointers.

What then is the depth of the directory? We know from the "birthday paradox" that if 23 or more people are present in a room then the probability of a birthday coincidence exceeds 1/2. So out of 365 possible directory entries to which a record could hash, after only 23 insertions, chances are greater than one in two that a collision will occur. Knuth [Kn., exercise 6.4.4] poses this problem for 3 people and Pinzka [Pi] extends it for the cases of 3, 4, and 5 people. Using a Poisson approximation, Pinzka concludes that 88, 187, and 314 people must be present for the probability to exceed one half that 3, 4, and 5 people, respectively, have the same birthday.

For the Poisson model, if the number of records n is the Poisson distributed random variable then

$$P(n=N) = \frac{\nu^N e^{-\nu}}{N!}$$

where N is the number of records in the file and ν is the average number of records. Assuming that all leaf pages appear at two successive levels, Fagin et al [Fa] show

1. the number of directory entries is

$$2^{\left\lceil \log \frac{\nu}{c \cdot \ln 2} \right\rceil}$$

where c is the size of a leaf and

2. the average number of directory entries is

$$\frac{\nu}{c \cdot (\ln 2)^2} .$$

They show for practical values of c and ν that indeed all leaf pages appear at two successive levels with probability nearly one for this model. However, they note that if c remains fixed and $\nu \rightarrow \infty$ they have no conclusive estimates for the average number of directory entries or for the depth of the directory.

The probability bounds they derive for the Poisson model hold for the Bernoulli model as well.

From the above discussion based on the "birthday paradox" and the Poisson model developed by Fagin, et. al., we suggest that the expression $\left\lceil \log \frac{N}{c \cdot \ln 2} \right\rceil$ given in Section 3.2.1 as the depth of the directory may be off by a factor of 2.

Implementation issues.

We chose the hashing function from a class of universal hash functions [CW] using a pseudo-random number generator.

For the name field, the number of directory entries which actually point to leaves is 3757, only 339 (or 8 percent) shy of the maximum possible 4096 entries. Thus, the hash function is "good" in that any clustering that arose was eventually smoothed out.

Because of the large size of each bucket (512 bytes) the buckets are only about 50 percent full. Again, wasted storage in each bucket explains why more actual buckets are required than predicted.

Comparing the name field to the street and town fields, the number of nonnull directory entries decreases whereas the number of actual buckets approximates the predicted value better. This is not surprising since few, dense lists are expected in the street and town fields because fewer keywords occur in those fields than in the name field and these keywords occur more often in the file.

The same issue that arises for tries of linked buckets extending over more than one disk block holds for the implementation of hashing. This is noted in Section 4.2.2 in the presentation of the timing results.

5.1.3 Superimposed Coding

Width of an scw and the coding function.

The additional storage for the SCW file depends on the number of records in the file and the width of an scw. The width and the coding function are related to each other and to the number of records in the file. The choices for the width and the coding function can be made by taking statistics on the database. Ideally the coding function should set half of the bits in an scw. Hence, the width of an scw should be chosen so that the records are not coded to an scw that is too sparse or too dense.

Other approaches include varying the width of an scw, or varying the weights each keyword contributes to an scw. Also keywords that occur frequently should be given a smaller weight than average.

Implementation issues.

Two different code widths are used: one (127 bits) for single line listings and one (191 bits) for long listings including structured listings like that for U.S. Government.

The coding function considers only the first four characters of each keyword, with each character contributing a varying number of bits to the scw depending on the character's position. For most words the first character contributes 4 bits and the second, third, and fourth characters each contribute 3 for a total of 13 bits per keyword. However, for example, in the case of the first letter being a "C" then only 1 bit is set in the scw since "C" occurs as the first letter about 26,000 times in less than 60,000 records. Thus, the coding uses the knowledge of character and word frequencies in the database.

To decrease the search time through the SCW file, we can order the code words. Then a binary search can decrease the search through the file from $O(N)$ to $O(\log N)$. However, ordering the SCW file increases the time to perform insert, delete, and update operations, requires additional overload for storing a record number besides an scw number (currently they are the same), and is less amenable to a changing coding function (since then the entire SCW file would have to be resorted).

5.2 Insert and Delete Operations

A complete analysis of a file organization technique should include a study of the insert, delete and update times. An update can be regarded as an deletion followed by an insertion, although this is wasteful if only a small modification is made to a record. Depending on the application, on whether the file is dynamic or static, and on the typical requests of the user, inserts, deletes, and updates can be performed as they enter the retrieval system or they can be batched and all performed at once.

5.2.1 Tries

Insertion of a record into a trie requires the same "trie" walk as a retrieval. If the record is not in the leaf then we insert it. How the leaves are stored and if they are kept sorted determines how simple this operation is. If we keep an unsorted linked list, then we simply add the record to either end of the list. If we keep a sorted list then one insertion to the beginning of the list requires moving all records to make room for the inserted one.

We did not measure insertion times but we did record the times to build the entire trie. For the name field alone, it took almost 15 hours (real time)* to construct the trie index

* And over 11 hours of CPU time alone.

and leaves for about 50,000 records. The records were processed so that partially sorted keyword and record identifier pairs were input to the program that constructed the tries.

Deletion of a record raises similar problems. First a "trie" walk is done, followed by a search in the list for the record and a reorganization of the list.

If we store characters alphabetically in each internal node and if we maintain sorted lists then a retrieval is easy but an update is tedious.

5.2.2 Extendible Hashing

One of the attractive features of extendible hashing is its flexibility for insertions and deletions. An insertion may cause a leaf to overflow which may then cause the directory to double. But if the directory doubles, only pointers need to be copied to the appropriate directory entries. The leaves are not affected at all. Similarly, a deletion may cause a leaf to become empty, which may then cause the directory to halve. Still, only the pointers change, not the leaves themselves.

In either case we search for a record on a leaf. Thus the size of the leaf is important, not only because it affects how often the directory doubles or halves but because it influences the search technique that we use to find the record. Again, if we keep the leaves sorted, a binary search is appealing, although if any one leaf has few entries, then a linear search is just as acceptable.

Because of the dynamic nature of the directory in extendible hashing, an update is not as expensive as in the case of tries.

5.2.3 Superimposed Coding

One advantage of superimposed coding is the ease of inserting and deleting records. For an insertion, a record's scw is generated and added to the (end of the) SCW file. For a deletion, the record's scw is removed from the SCW file. Deletion may be accomplished "in place" by setting the scw to all zeroes. This wastes some space (until the next reorganization) but may be preferred if deletions are infrequent.

However if the SCW file is stored in a bit-sliced organization then inserting and deleting are not so easy. Since the SCW is stored by column and we want to update by row, we have the canonical problem of performing a row operation on a matrix stored by column. Also if the SCW file is ordered, these insertions and deletions must maintain the order which introduces the same problems as mentioned for sorted lists for tries.

5.3 Comparison

5.3.1 Summary

Table 5.1 presents a summary of the predicted and actual values for storage and retrieval time for all three methods.

	Predicted		Actual	
	Storage (in bytes)	Retrieval Time (in seconds)	Storage (in bytes)	Retrieval Time (in seconds)
Tries	2,080,752	.55	1,653,248	1.4
Hashing	2,767,437	.25	4,504,582	.8
Superimposed Coding	1,061,303	5.0	1,061,303	9.1

Table 5.1. Comparison of tries, extendible hashing, and superimposed coding: predicted and actual values for storage and retrieval times.

Note that the numbers for storage for tries include the trie indices and leaves for all three fields: name, street, and town. Similarly, for hashing, the numbers for storage include the directories and leaves for all three fields. Since we perform partial-match retrieval across the three fields and since the storage for superimposed coding inherently includes information from all three fields, storage for all three fields should be included. The storage numbers do not include the amount of storage required by the file (the Suffolk County directory) itself. This is an appropriate omission in the case of partial-match retrieval because each record is expected to appear many times in any index. Hence each index will use pointers rather than storing the actual records many times.

The storage actually required by tries and superimposed coding are not significantly different. In fact, hindsight reveals more efficient ways of packing nodes and leaves for tries so that storage for tries would be nearly the same as that for superimposed coding. Storage required by hashing is much greater than either because the leaves are only 50 percent full.

We only specified keywords in the name field for the random queries used to measure retrieval time. This is representative of queries in the context of directory assistance. Also the differences in time are not expected to be significant from field-to-field or across fields. The actual retrieval times listed in Table 5.1 are the averages over the set of queries.

Overall, using tries is a reasonable compromise between the fast but storage-wasting extendible hashing method and the slow but storage-saving superimposed coding method for file organization. However, if space is not a constraint, and if ease of performing insert and delete operations is desired, extendible hashing may be preferred.

5.3.2 Implementation Issues

Retrieval time for superimposed coding appears much worse than that for tries or hashing. To fairly compare superimposed coding to the other two techniques, important differences between the implementations must be considered.

First, the purpose of the implementation using superimposed codes for partial-match retrieval was not for speed of retrieval but for applicability of this retrieval technique to directory assistance.

The implementation for superimposed coding includes structured listings (e.g. U.S. Government) whereas those for tries and hashing do not. The superimposed coding program uses 11 UNIX processes and message handling capabilities which slow down the performance of the overall system significantly. The process that performs the search is actually run twice, once for short listings and once for long listings. The process that performs the match between the dropped records and the query does a best-match for each keyword so that eventually records output to the user are ordered with best matches first. Finally, the retrieval system includes a feedback mechanism that returns the number of dropped records after each character is typed in by the user.

One advantage that the superimposed coding implementation has over the other two is that raw I/O (and not UNIX I/O) is used. This takes advantage of the SCW file being stored in a bit-sliced organization so multiple units of a UNIX block size can be read in at once.

Roberts [Ro] developed a program, the precursor to the current implementation, using superimposed codes that was written specifically with the goal of measuring retrieval time. He used the same database - the single listings - as that used for tries and hashing. Measurements taken by Roberts show that a lower bound for actual retrieval time is 1.5 seconds per query. This is significantly lower than those measurements taken on the current implementation.

Whereas Roberts' version considered speed of retrieval and minimized the number of features, the current implementation considered the number of features without major concern for retrieval time. Thus, actual retrieval time per query for superimposed coding probably lies in between Roberts' result and our result. The general conclusion is the same: retrieval time is greatest for superimposed coding.

6. CONCLUSIONS

6.1 Summary

As a result of analyzing and implementing three file organization methods (tries, extendible hashing, and superimposed coding) for partial-match retrieval, the following conclusions can be made:

1. Storage: The actual amount of storage used is less than predicted for tries, more than predicted for hashing, and as predicted for superimposed coding. Tries and superimposed coding use a comparable amount of storage. Hashing requires more than either.

2. Retrieval time: All three actual average retrieval times are higher than the predicted. For tries, the actual value is about twice the predicted; for hashing, about three times the predicted; for superimposed coding, about twice the predicted. Retrieval via hashing is fastest, then tries, then superimposed coding.

3. As partial-match retrieval techniques: Tries and superimposed coding are suited for partial-match retrieval on characters within keywords whereas extendible hashing is not. For tries, one can specify up to *d* characters for each keyword where *d* is the number of levels in the trie index. For superimposed coding, using the appropriate coding function one can also partially specify the characters of a keyword.

   For extendible hashing, for a random hash function, one cannot partially specify the characters of a keyword but must specify all characters that the hash function uses. That is, "HE" does not necessarily hash to the same directory entry as "HEL" so the user cannot just input "HE" if "HEL" occurs as part of the keyword.

   Both tries and extendible hashing are better techniques for single-key queries rather than for partial-match queries. For single-key queries, we can store the entire record in a leaf instead of a pointer to the record. Partial-match requires redundancy in the leaves.

4. Tries and extendible hashing are similar in structure. The levels of the trie index can be collapsed into a single level - the directory of extendible hashing. The difference is that the hashing function may distribute the records of a file more uniformly throughout the leaves.

5. Implementation decisions cloud the issues. Unfortunately, implementations must conform to real constraints like the size of main memory and disk access time. (Things could be a lot different if we used a machine with a larger address space, say 2^{24} bytes, and sufficient real memory.) This especially is true with extendible hashing since the UNIX page size limits the size of leaf.

6.2 Suggestions for Future Work

6.2.1 Best-Match Queries

We applied the file organization techniques presented in this thesis to partial-match retrieval. Suppose instead, we want those records which also fall as "nearest neighbors" to the response to a partial-match query; that is, we specify a best-match query. How easy is it to adapt what we know about tries, extendible hashing, and superimposed codes to best-match queries?

Retrieval from tries readily adapts to best-match queries. The siblings of the nodes traversed for a partial-match query will lead to the leaves which contain records that are nearest neighbors to the records satisfying the partial-match query.

Neither extendible hashing nor superimposed coding readily adapts to best-match queries. Because query keywords are encoded either by a hash function or into a query mask, a nearest neighbor search would require the information retrieval system to be smart enough to generate all other queries that might satisfy the input query.

The motivation for the capability of satisfying best-match queries stems from the directory assistance problem of alternate spellings for a name. For example, in a typical telephone directory, under "Brown" often one also reads "See also Braun". Also, if a user overspecifies a name like "John Doe" when "J. Doe" is listed, we would like to retrieve records for all "J. Doe"'s. Similarly, the thesaurus problem arises because professions are frequently listed in more than one way: doctor and physician, lawyer and attorney. These problems can be solved in an engineering fashion, but for instance, can we solve it in the coding function itself for superimposed coding?

6.2.2 Structured Listings

The implementations for retrieval from tries and extendible hashing avoid structured listings because the problems that arise in including them merely hide the issues addressed in this thesis. However, structured listings do pose interesting problems for superimposed coding since varying the code word widths affects the overall performance of the retrieval system. Another approach might be to encode the information in the scw that a record is from a structured listing.

6.2.3 Hardware Implementation

For superimposed coding, if we do the search through the SCW file in hardware, the retrieval time is no longer bounded by the search time. In fact, preliminary analyses done by Ahuja [Ah] at Bell Laboratories show that with the help of special-purpose hardware one can process about 7200 queries per minute as opposed to 6 queries per minute with the current software implementation.

If time is a crucial factor such as in directory assistance where one may reasonably require about 400 queries per minute, then special-purpose hardware is essential in the information retrieval system. Perhaps similar hardware applications can be made for tries and extendible hashing.

6.2.4 Combining the Methods

The search through the SCW file slows down the retrieval time for superimposed coding. If we add a trie index or an extendible hashing directory as a level of indirection to the

SCW file, then we can reduce the number of scw's actually compared to the query mask. Retrieval time should improve at the expense of additional overhead for the extra level of indirection.

6.3 Final Thoughts

Throughout this thesis, we allude to many problems that arise from implementation decisions. We tend to attack these problems based on engineering intuition and not theoretical results. Such is the state of affairs in the database and information retrieval business. It is often necessary to obtain an intuitive grasp from an actual implementation before one can make meaningful generalizations.

APPENDIX A

The January 1975 edition of the Suffolk County, New York white pages telephone directory serves as the common database for the thesis' implementations. For tries and extendible hashing, only the 47,466 single line listings are used; for superimposed coding, the 2187 structures, (e.g., U.S. Government) are also included.

Table A.1 presents the statistics on word frequencies of the five most frequent words in the name, street, and town fields [Be].

Field: Number of Distinct Words	Word	Frequency (per 1000 listings)
Name: 23,026	incorporated	221
	company	83
	corporation	73
	service	50
	A	37
Street: 5328	avenue	225
	road	224
	main	100
	highway	73
	E	56
Town: 603	Huntington	84
	E	56
	station	48
	Farmingdale	48
	Islip	45

Table A.1. Frequency counts of the five most frequent words in the name, street, and town fields.

Table A.2 presents the number of distinct uni-, di-, and trigrams for each field.

Field	Unigrams	Digrams	Trigrams
Name	36	467	2919
Street	38	339	1325
Town	24	125	249

Table A.2. Number of distinct uni, di-, and trigrams.

APPENDIX B

We use the UNIX *time* and *times* subroutine to compute the I/O time for retrieval. *Time* returns three numbers: real (R), user (U), and system (S) time. *Times* returns a more precise estimate of the user and system times in units of 1/60 second.

Real time is the actual clock time that a user spends waiting for a response. User time plus system time gives the total CPU time. Thus to approximate I/O time, the following equation is used:

$$I/O\ time = R - (U + S)$$

where R is real time, S is system CPU time, and U is user CPU time.

Time was used to time total time elapsed for the process that runs 20 queries and retrieves the proper records. It includes forking UNIX processes, initializations, opening and closing UNIX files. For superimposed coding, it includes time for sending and receiving messages through the UNIX pipes. *Times* was used for tries and extendible hashing to yield more precise times for the retrieval of records in response to the test set of 20 random queries.

Finally, the times in seconds to read the directory for extendible hashing into main memory before any retrievals are executed. The times are 6.0 (real), 3.0 (user), and 0.4 (system). These times are considered in calculating the I/O time for 20 queries to yield the following:

$$I/O\ time = R - (U + S) - [R_H - (U_H + S_H)]$$

where R_H, U_H, and S_H are the real, user, and system times to read in the hash directory.

We performed six trials. The results are shown in Tables B.1, B.2, and B.3 for tries, extendible hashing, and superimposed coding respectively.

Trial	*time*(in sec.) for 20 queries	*times*(in 1/60 sec.) for 20 queries	*times*(in sec.) for 20 queries	I/O time (in sec) for 20 queries	I/O time (in sec.) per query
R	40.0				
1 U	9.9	595	9.9	27.8	1.4
S	2.3	109	1.8		
R	40.0				
2 U	9.9	594	9.9	26.5	1.3
S	3.6	192	3.2		
R	41.0				
3 U	10.9	656	10.9	27.5	1.4
S	2.6	129	2.2		
R	40.0				
4 U	9.5	571	9.5	26.5	1.3
S	4.0	208	3.5		
R	42.0				
5 U	9.4	567	9.4	29.3	1.5
S	3.3	170	2.8		
R	41.0				
6 U	9.5	570	9.5	28.9	1.4
S	2.6	132	2.2		

Table B.1 Timing results for tries.

Trial	*time*(in sec.) for 20 queries	*times*(in 1/60 sec.) for 20 queries	*times*(in sec.) for 20 queries	I/O time (in sec) for 20 queries	I/O time (in sec.) per query
R	27.0				
1 U	6.0	169	2.8	16.2	0.8
S	2.2	84	1.4		
R	26.0				
2 U	6.1	172	2.9	15.4	0.8
S	1.9	75	1.3		
R	27.0				
3 U	6.4	192	3.2	16.2	0.8
S	1.8	64	1.1		
R	27.0				
4 U	6.1	173	2.8	15.7	0.8
S	2.6	175	1.9		
R	26.0				
5 U	6.1	169	2.8	14.8	0.7
S	2.5	98	1.6		
R	28.0				
6 U	6.2	179	3.0	16.6	0.8
S	2.6	112	1.9		

Table B.2. Timing results for extendible hashing.

Trial	*time*(in sec.) for 20 queries	I/O time (in sec.) for 20 queries	I/O time (in sec.) per query
R	186.0		
1 U	0.5	179.0	9.0
S	6.5		
R	186.0		
2 U	1.0	179.5	9.0
S	5.5		
R	184.0		
3 U	0.3	179.5	8.9
S	5.9		
R	190.0		
4 U	1.1	183.3	9.2
S	5.6		
R	200.0		
5 U	0.7	192.9	9.7
S	6.4		
R	186.0		
6 U	0.6	179.8	9.0
S	5.6		

Table B.3. Timing results for superimposed coding.

REFERENCES

[Ah] S. R. Ahuja, "Associative/Parallel Processor for Partial Match Retrieval Using Superimposed Codes", Bell Laboratories, Holmdel, NJ, (to be published).

[AHU] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, 1976.

[AMP] A. R. Aronson, J. Minker, and A. J. Perlis, "Optimizing Boolean Expressions for Compilers and Data Retrieval", April, 1977.

[Ba] R. Bayer, "Symmetric Binary B-Trees - Data Structure and Maintenance Algorithms", Acta Informatica (1972), pp. 290-306.

[BM] R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indexes", Acta Informatica (1972), pp. 173-189.

[Be] J. D. Beyer, J. D. Gabbe, and R. E. Miller, "Suffolk Telephone Directory Data Base for Information Retrieval Research", Bell Laboratories, Internal Memorandum.

[Bu,76] W. A. Burkhard, "Hashing and Trie Algorithms for Partial-Match Retrieval", ACM Trans. on Data Base Systems, 1, (1976), pp. 175-187.

[Bu,77a] W. A. Burkhard, "Associative Retrieval Trie Hash-Coding", J. Comput. and System Sci., 15, (1977), pp. 280-299.

[Bu,77b] W. A. Burkhard, "Nonuniform Partial-Match File Designs", Theoret. Comput. Sci., 5, (1977), pp. 1-23.

[CW] J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions", Proceedings of Ninth Annual SIGACT Conference, May, 1977.

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

[Fa] R. Fagin et. al., "Extendible Hashing - A Fast Access Method For Dynamic Files", IBM Research Report RJ2305, July, 1978.

[Ga] J. D. Gabbe et. al., "QUIK: A Novel Approach to Computerized Directory Assistance Services", Bell Laboratories, Internal Memorandum.

[Ga,77] J. D. Gabbe et. al., "Applications of Superimposed Coding to Partial-Match Retrieval", COMPSAC 78 Conference Proceedings, New York.

[KK] J. W. Klimbie and K. L. Koffeman (editors), "Data Base Management", Proc., 1974, IFIP TC-2 Conf. Congress, North Holland, 1974.

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

[La] P. A. Larson, "Dynamic Hashing", BIT 18(1978), pp. 184-201.

[Mo] C. N. Mooers, "Zatocoding and Developments in Information Retrieval", ASLIB Proceedings 8, 1 (February, 1956), pp. 3-22.

[Mor] R. Morris "Counting Large Numbers of events in Small Registers", CACM October, 1978, Vol. 21, No. 10.

[Pi] C. F. Pinzka, "The American Mathematical Monthly", Vol. 67, No. 7, 1960, p. 830.

[Ri,74] R. L. Rivest, "Analysis of Associative Retrieval Algorithms", TR STAN Cs-74-415, Computer Science Department, Stanford, CA, 1974.

[Ri,76] R. L. Rivest, "Partial-Match Retrieval Algorithms", SIAM J. Computing 5, 1976, pp. 19-50.

[RT] D. M. Ritchie and K. Thompson, "UNIX Programmer's Manual", Sixth Edition, 1975.

[Ro] C. S. Roberts, "Partial-Match Retrieval Via the Method of Superimposed Codes", Computing Science Technical Report #64, Bell Laboratories, Holmdel, NJ, 1977.