HEURISTICS FOR CONSTRUCTING NEAR-OPTIMAL TRIES FOR PARTIAL MATCH RETRIEVAL

Christina Lucia Soochan

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

April 1980

HEURISTICS FOR CONSTRUCTING NEAR OPTIMAL TRIES FOR PARTIAL MATCH RETRIEVAL

CHRISTINA LUCIA SOOCHAN

ABSTRACT

The construction of tries for storage and retrieval of records for partial match queries is studied. The main objective has been to improve the average retrieval time and lower the worst case behaviour under given query distributions. If the queries are uniform over the query space the characteristics of the records alone influence the shape and size of the trie, whereas when non-uniform distribution of queries is given, it is shown how this can influence the selection, the order of testing of the attributes and hence the shape, size and unbalance of the resulting collapsed order containing trie. The tries are compared on the basis of their performance i.e. the average number of buckets (leaf nodes) examined to answer a partial match query. Superimposed coding schemes and concatenation of codes are employed to transform non-binary files into binary files. Both sequential and trie based search of the coded data base are studied and their performance compared. The results obtained by exhaustive construction of tries for records with a small number of binary attributes (3 - 6) and simulation for large files indicate that the constructed tries are close to optimal.

To my husband Louis,

mom and dad and grandpa.

ACKNOWLEDGEMENTS

I wish to express my gratitude to Dr. Alagar without whom this thesis could never have been written. His support has seen me through the most difficult moments during my stay at Concordia. His advice and knowledge have helped me initiate, pursue and complete this research.

To Bilal, Khalid, Pierrette and Vivian, thank you for your great help. To all my friends and colleagues in the Department of Computer Science, thank you for your encouragement.

Special thanks to George, Larry and Shelley and forgive me for being so demanding at times.

Last, but not least, thank you, Louis, for your patience and understanding.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

Industrial and scientific developement of the last few decades has caused accumulation of large amounts of information in every aspect of life. In governments, schools and in small businesses there is a growing need to automate the storage and retrieval of information. As the volume and complexity of data has grown over the years, efforts have been channelled towards reducing the physical storage required by the information and the time required to access it. With the advent of electronic computers a whole new field of research that addresses this problem has evolved. This we call data base studies.

In data base studies it is important to have a good file design as well as efficient algorithms for performing certain tasks on the information stored in the data base, such as accessing the data and updating it.

In several problems in computer science the design and evaluation of algorithms can be done independent of the characteristics of the storage media and the structuring of the file, as is the case in some numerical applications (e.g. solutions to differentiral equations, integration, etc.). In commercial and administrative data bases the requirements are different and a good file design is of paramount importance.

In this thesis we address ourselves to both file design and algorithm design when searches have to be made in a data base for queries of a special type. Each record stored in a data base has a unique key which identifies it. This is called a primary key. For instance, the primary key field in a file of taxpayers' records is the social insurance number. Besides the primary key field, a record may contain other fields, generally known as "secondary keys" or "attributes". A secondary key can be viewed as a k-tuple $(i_1, \ldots, i_k)$ where $i_1, \ldots, i_k$ is a subset of the attributes of a record. In the example above, the attributes of each record, besides the social insurance number, may be:

NAME, ADDRESS, YEAR OF BIRTH, SEX, MARITAL STATUS, OCCUPATION, EMPLOYERS'S NAME and SALARY.

There are many algorithms to perform efficient searches on the primary key of a file. As an example, if the records in a file are sorted on the primary key value and the size of the file allows it to be kept in core, then the cost of searching for a record using binary search is at most $\log_2 N$ primary key comparisons, where N is the number of records.

A small file may also be stored in a binary search tree structure, where each node contains a record and pointers to the left and right sons. A left son is a record whose primary key value is "smaller" than the primary key value of the record in the father node. The right son is a record whose primary key value is "greater" than that of the father node. Here "smaller" and "greater" are used in the lexicographical sense.

The advantage of a binary search tree over the sorted sequential file is that insertions and deletions are easier to perform in a binary search tree. Searching for a record whose primary key value is known is just a matter of comparing its primary key value with that in the root node, taking the appropriate branch and repeating the procedure until the record is found or there are no more branches. For insertion of a record, the same procedure is followed and the record is inserted as the son of the corresponding node. For deletion the node to be deleted is found with the search algorithm and the links are restructured.

In the case of large files which cannot be kept in core, a B-tree structure can be used. Without going into much detail, we describe the search method on a B-tree. A node in a B-tree contains several primary key values and pointers. The primary key values within a node are ordered increasingly and they alternate with pointers to secondary storage locations and thus provide a partitioning of the file into several subtrees. In order to locate a record with a given key, the root of the B-tree is searched for the occurrence of the first key which is greater or equal to the given key. The address associated with this key is the starting address of a block of secondary storage in which the required record may be found (if it exists).

There are however situations in which it is necessary to perform searches based on secondary keys. As an example consider a file of student records in a university. While the primary key is the student number, a typical query to this file may be a request to search for "all the students with unpaid fees", or "all the students in Computer Science major who will graduate in May 1980".

Clearly the standard methods of whole key comparisons cannot be used in such situations and the complexity measures for these search problems involving secondary keys are different and depend on the design and storage medium of the file.

Generally a data base that allows searching on secondary keys will have an index mechanism which should decrease the cost of conducting transactions. A naive approach is to provide a separate index for each attribute in the file, such as an inverted file or a multilist structure. Alternately attributes may be combined or query types restricted.

Although an inverted file is invariably necessary in a system that allows general boolean queries, the time and space required to maintain and update the file suggest that this approach is not worthwhile in more specific situations. In this thesis we study only restricted query types and our methods can be looked upon as optimal indexing strategies for accessing a file.

It has been shown by Comer [10] that in general it is difficult to select a subset of the attributes for indexing a file, which would be optimal for a given set of transactions. Schkolnick [28] has given an algorithm for selecting a near-optimal subset of attributes for indexing a file under the following assumptions:

1) attributes are not to be combined;
2) the statistical properties of the transactions (i.e. the probabilities of querying) are known.

However the users' requests may change in time and thus alter the distribution of queries. Hence a system such as the one proposed by Schkolnick must keep statistics about the transactions in the recent past (folowing the last recomputation of the index set), which would enable the recomputation of indices for future use. It seems therefore that there is no optimal solution that is permanent over the entire life of a data base in the most general situations.

In this thesis we consider file designs for data bases that allow only partially specified queries (i.e. queries in which not all of the attributes are specified). Once again the justification for this is that the more general situations fall in the category of intractable (NP-complete) problems. Moreover partially specified queries are quite common. We give below some important instances of partial match retrieval problems.

EXAMPLE 1. As a practical example of a system permitting partially specified queries, consider a data base of a telephone directory. In this data base each record consists of NAME, ADDRESS and TELEPHONE_NUMBER. The TELEPHONE_NUMBER is the primary key for the file, but normally the queries specify the NAME only, or the NAME and ADDRESS. Under certain circumstances one may know the NAME, ADDRESS and the first 3 digits in the telephone number and wish to find the whole number. In this case we may consider the telephone number as being made up of two fields and so any one of them may be specified in a partially specified query. For example, suppose the query is

(NAME = SMITH, EXCHANGE = 935)

where EXCHANGE represents the first 3 digits of the telephone number. The response to this query will possibly contain several records, all of which satisfy tne NAME and EXCHANGE fields.

EXAMPLE 2. Consider a data base used by the Canadian Taxation Bureau. Each record contains information about a taxpayer and the primary key is the SIN (social insurance number). Besides the primary key each record stores some pertinent information on each taxpayer such as NAME, ADDRESS, MARITAL_STATUS, OCCUPATION, INCOME, NUMBER_OF_DEPENDENTS, etc. A subset of these attributes such as NAME, OCCUPATION, MARITAL_STATUS are secondary keys and may occur in queries. Frequently it is required to extract information on certain categories of taxpayers (e.g. OCCUPATION=LAWYER, INCOME=$50,000 per year). Typically these queries are partially specified, hence there is the need for a good file design capable to handle such requests.

EXAMPLE 3: Another example of a data base permitting partially specified queries is a file of licence numbers of stolen cars used by Police and Customs and Immigration departments. Suppose that a licence plate number has 7 digits. Each record in the file has the licence number as the primary key, and the secondary keys are the individual digits in the number. One can imagine the situation when search is to be done given a licence plate number in which only some of the digits are known. This might be the case when a witness to a hit and run accident reports to the police trying to recall the licence number of the car that caused the accident, but cannot remember all the digits. A similar example is a file of lost or stolen credit cards used by a department store.

In the examples given above, all the data bases allow partially specified queries, as well as fully specified queries.

In general all the queries permitted by an information retrieval system do not occur with the same probability. Thus the system must handle the most probable queries in an optimal manner in order to optimize the total cost.

In this thesis we examine several algorithms for peforming partial match searches (i.e. searches for partially specified queries) on a direct access file and we make a comparative study on the achievable efficiency of the algorithms.

Partial and best-match queries are usually considered paradigms of associative queries. Several researchers have addressed this problem in the past. Most notably Rivest [25] and Burkhard [7] have considered hashing and trie-based methods.

Rivest [25] has introduced several classes of ABD's (associative block designs) in which hash functions are used to select record bits to use as list index. This has suggested the use of tries as alternative data structures, which have later been studied by Burkhard [9] as well. However the studies conducted by Rivest and Burkhard were restricted to uniform query patterns.

We report in this thesis results on the construction and performance of tries for non-random data and non-uniform query patterns. Rivest [25] has suggested this as an open problem. The concept of non-uniformity is the realistic approach to everyday transactions. Thus our methods can be looked upon as probabilistic retrieval and probabilistic analysis for partial match queries.

The thesis is organized in the following way. In chapter 2 we give a general outline of an information retrieval system. In chapter 3 we discuss partial match query types and review the past work done in this area. In chapter 4 we define modes of retrieval, complexity measures consistent with these modes of retrieval and probabilistic notions concerning querying.

Chapter 5 contains a discussion of tries as general data structures, giving the construction and search of tries. We will emphasize the difficulty of constructing a trie to satisfy a given set of constraints.

Four methods of trie construction, similar in conception, yet different in the way they influence the shape and structure of the resulting tries are given in chapter 6.

In chapter 7 we conduct a comparative study of two methods of coding a file and retrieval on a coded file. These are:

1) superimposed coding with sequential search versus superimposed coding with trie construction and search.
2) Concatenation of codes (partitioned hashing) with random access versus concatenation of codes and trie search.

Finally in chapter 8 we compare our results with the best previously known results.

A significant contribution of this thesis is to show tries as useful file designs for partial match queries even when queries are non-uniform.

The results presented in this thesis are mainly empirical, yet exhaustive. Statistics have been gathered on the performance of these methods for different assumptions on the probability distribution of the query space.

The results show that the average retrieval cost in any method prevails almost everywhere over the query space and the probability of worst-case behaviour is very low (see chapter 4 for probabilistic notions).

Hence we are lead to conclude that the tries constructed in chapter 6 are either optimal or near-optimal.

CHAPTER 2

QUERY TYPES, FILES AND NOTATIONS

In this chapter we formalize the concepts of files and query types. An information retrieval system consists of several components, namely:

1) a collection of information, called a file;

2) a procedure for storing the file in a given storage medium;

3) an access mechanism which enables accessing and reading a file;

4) a search algorithm consistent with the file design;

5) a user who is armed with a piece of information and is seeking more information from the system.

First we present some formal definitions of files and query types. In the next chapter we will discuss file design and search methods.

A file F is a collection of N distinct records where each record R is an ordered k-tuple $(r_1, r_2, \ldots, r_k)$. A component r_i of R is called a key or attribute. Each component r_i can take values from a domain D_i, $1 \leq i \leq k$. Each domain D_i is an alphabet containing d_i values, such that the set U formed by the cartesian product $D_1 \text{ X } D_2 \text{ X} \ldots \text{X } D_k$ has $m = \prod_{i=1}^{k} d_i$ elements, and it represents the universe of k-component records, each component having values from the corresponding domain. Thus a file F is a nonempty subset of U, with N $\lesssim$ m records.

As an example, suppose that k=3 and

$$D_1=D_2=D_3=\{A,B,C,....Z\}.$$

Then a file F represents a collection of 3-tuples of letters and U contains all the possible 3-tuples of letters. The cardinality of this set is $26^3=17576$.

This model of the file is the well known relational model. See Hsiao and Herary [18] for a generalized file model, where a record has been considered as an unordered collection of (attribute, value) pairs.

The user of an information retrieval system may or may not have a priori knowledge of the information in the system. The user communicates with the system in a form predetermined by the system designers. Every request from the user asking for information from the data base can be considered as a query. The information retrieval system receives the queries, analyzes them, searches the data base by possibly making use of an auxiliary file and outputs an answer to the given query.

The structure and format of a query are the concerns of a query language designer. A good query language must be based on the data description model and should avoid machine dependent concepts.

In this thesis we do not discuss the design of a query language; our only interest is to attempt a classification of the queries on the basis of the information that has to be manipulated in the data base.

In general a query must have two parts. One of them is called the "qualification part" and the other is known as "target part". The qualification part is the piece of information which the user has and the target part specifies the type and amount of information that the user wants to have from the system. For example consider the query "give the names of Computer Science faculty members with more than 10 years experience". The part of the query "Computer Science faculty members with more than 10 years experience" is the qualification part and "names" is the target part. The response to this query is a subset (may be empty) of the Computer Science faculty member file.

The complexity of a query depends upon the number of attributes and the mode of specification of the value of each attribute. A query in which only one attribute is specified in the qualification part is a simple query. If a single value is specified we can say the query is much simpler than when the mode of specification is a range. The query "give the names of Computer Science faculty members with more than 10 years experience" is an example of a range query; the query "give the names of faculty members who are unmarried" is an example of the simplest type of query.

The qualification part in a query is a predicate expression and hence all predicate calculus operations can be extended to a query. The qualification part may specify several attributes and for each attribute a value, a set of values or a range may be specified. The following example illustrates a few kinds of queries which may be asked in a relational data base.

EXAMPLE 1. Consider a sample student file. The attributes are:

NAME, AGE, SEX, COURSE, GRADE.

Let the data in the file appear as:

NAME	AGE	SEX	COURSE	GRADE
JOHN	20	M	CS211	B
MARY	19	F	CS231	A
MIKE	20	M	CS231	C
HARRY	21	M	CS211	A
HELEN	20	F	CS241	B
BILL	22	M	CS241	A
HELEN	18	F	CS211	B

The following queries are asked:

	TARGET	QUALIFICATION	RESPONSE
1.	NAME∧GRADE	(AGE=20)∧(COURSE=CS211)	(JOHN,B)
2.	AGE∧SEX	(COURSE=CS231)∧(GRADE=B)	------
3.	GRADE∧AGE	(NAME=HELEN)	(B,20),(B,18)
4.	GRADE∧AGE	(NAME=HELEN)∧¬(COURSE=CS211)	(B,20)
5.	NAME∧SEX	(AGE≥20)	(JOHN,M) (MIKE,M) (HARRY,M) (HELEN,F) (BILL,M)

In this example the qualification part of the queries is represented by a predicate expression. The predicates (e.g. NAME=HELEN) are connected with boolean operators "∧"(and), "∨"(or), "¬"(not).

Below we give a classification based on the qualification parts of queries.

Let Q represent the set of queries that are allowed in an information retrieval system consisting of a file F of N records. Given a query $q \in Q$, $q(F)$ denotes the set of records representing the proper response to q ; $q(F)$ is a subset of F and it may be empty.

There are two major classes of queries: intersection queries and best-match queries. The most general class is that of intersection queries. Such queries have as a common characteristic the definition of the response. A record in F may be retrieved if and only if it is also in a subset $q(U) \subset U$ so that $q(F)=F \cap q(U)$, or, more concisely,

$$R \in q(F) \iff q(F)=F \cap q(U)$$

where U is the universe of records (see Rivest [25]).

In the special case when F=U, it can be seen that $q(F)=q(U)$ which is consistent with the above definition.

Naturally the presence of a record $R \in F$ in $q(F)$ is quite independent of the other records in the file.

There are several important subclasses of the class of intersection queries, namely:

1) exact match queries;
2) single key queries;
3) partial match queries;
4) boolean queries;
5) range queries.

Of the second class, the best-match queries with restricted distance are more common. Below we explain briefly each class of queries.

1. EXACT MATCH QUERIES

Such a query specifies a value for each of the k attributes and the response consists of at most one record from the file.

Thus an exact match query asks f8r the presence or absence of a specific record.

2. SINGLE KEY QUERIES

The query specifies a key (attribute) and a value and it requires retrieval of all the records with the same value for the key that is specified.

3. PARTIAL MATCH QUERIES

The query type based on a single key mentioned above is a particular case of the partial match query type. A query that specifies s attributes, $s \leq k$, is called a partial match query. The response to such a query will contain all the records in the file which match exactly the s specified attributes. A partial match query is usually represented by a k-tuple in which s keys have values from their respective domains, while the remaining (k-s) key positions have "*"s, meaning a "don't care" condition. This thesis is concerned with file designs and retrieval for this kind of queries.

The set of all partial match queries with s specified attributes will be represented by Q_s.

In the special case where $D_1 = D_2 = \ldots\ldots = D_k = \Sigma$ and the cardinality of the alphabet Σ is σ, we have $|Q_s| = \sigma^s \binom{k}{s}$ for $0<s<k$ and the universe $U = \Sigma^k$, $|\Sigma^k| = \sigma^k$. Furthermore if $\Sigma = (0,1)$, such that a record is a k-bit string, we note that $|Q_s| = 2^s \binom{k}{s}$ and $|\Sigma^k| = 2^k$.

4. BOOLEAN QUERIES

A boolean query is defined by a boolean function of the atttributes. Boolean queries are a generalization of the other types of intersection queries since any intersection query may be expressed as a boolean query.

5. RANGE QUERIES

They are similar to partial match queries in the sense that $s \leq k$ attribute positions are specified but, instead of specifying a value for each of the s attributes, a range of values may be given for each one.

The response to any intersection query must have a recall factor of 1, meaning that all the records in the file which agree with a query will be contained in the system's response.

6. BEST-MATCH QUERIES

For this class of queries the response contains those records which are "closest" to the query in some sense. For best-match queries with restricted distance a function d is defined on the universe U. Such a query specifies a set of k keys and a distance δ, indicating that the response must contain all those records in F which have at least $k - \delta$ keys agreeing with those in the query. This distance function is the Hamming metric.

The notion of best-match can be applied to partially specified queries as well if we allow no more than δ out of the s specified attributes to differ in the response.

Some examples of the foregoing query types are given below.

EXAMPLE 2. Let us assume that $k=3$ and $D_1=D_2=D_3=\Sigma=\{A,B,C,...Z\}$. Thus $U=\Sigma^3$ and it represents all possible 3-letter words.

Let the file F contain 8 words from Σ^3:

F= (DOG, LOG, BAT, NAP, DIG, TAB, SAG, BUT)

i. exact-match queries:

a) q=(DIP)--> no match

b) q=(NAP)--> match

ii. single key queries:

a) q=(all the words in F starting with D)
--> (DOG, DIG)

b) q=(all the words in F with second letter E)
--> no match

iii. partial match queries:

a) s=2, q=(*OG) --> (DOG, LOG)

b) s=1, q=(*A*) --> (BAT, NAP, TAB, SAG)

iv. boolean queries:

a) q=(all the records in F with first letter B
and second letter not A) --> (BUT)

b) q=(all the records in F with first letter (D or S)
and second letter not (G or P) --> no match

As an example of range queries, consider the file F to be a set of records of employees of a company. Two of the keys are SALARY and SENIORITY. A query may be formulated as :

q= (all the records for employees with
($15,000 ≤ SALARY ≤ $20,000) and
(2 years ≤ SENIORITY ≤ 3 years))

For best-match queries what better example can there exist than queries asked in a dating service? Suppose that the file of male candidates contains their physical, psychological and vocational description. A query might request a person of (AGE=40, HEIGHT=5'10", WEIGHT=160 LB, PERSONALITY=kind, CAREER=professional), such that no candidate should differ in more than two aspects given in the query profile.

From the examples shown above it is obvious that the size of the response (i.e. the number of records in the response) obtained to a query depends very much on the type of query (apart from the type of records).

Of all query types, only the exact match queries produce a unique response, but most of the time formulation of such a query may be impossible or undesirable. The amount of work that a system has to do to respond to a query varies from one query type to the other. For example, each intersection query requires a total recall, that is every record in F satisfying the requirements must be retrieved. If the requirements of the user are specified rather loosely in the query, then the system invariably has to do a lot of work in order to meet the user's request. The classification given here is on the line suggested and outlined by Rivest [25]; by no means is this exhaustive, yet it covers a wide range of query types. There may be however query instances such as:

1) "Find the names of all students whose reports are less than the average report of the students in year II computer science".
2) "Find the minimum height of all students 10 years younger than the instructor for CS491".
3) "Find all train stations within 200 miles of Montreal".

These queries demand a computation to be performed on the value of the attributes specified in their qualification part. Naturally these queries are more complex and probably would be relevant in more specific data bases.

All information retrieval systems are set up to answer a class of queries. In some cases the class of queries may be general and in others they are restricted. Usually the restrictions are imposed from the point of view of efficiency and practical considerations.

The simplest query types have been long studied and efficient algorithms are available. The most general intersection queries and computational queries always require a lot of computing time. Queries which specify only a subset of the attributes are more common in Data Base Management

Systems and library information systems. In this thesis we consider queries of such restricted type.

CHAPTER 3

FILE DESIGNS AND RETRIEVAL STRATEGIES

Looking for an object or a set of objects to satisfy a particular identifier or a given set of characteristics is an old and frequently encountered problem. It has often been said that in order to find an object one must know not only where it can be but also where it cannot be located. Thus to look for a book written by D.E. Knuth one must search under "Computer Science" categories and not under "The Art of Cooking".

Although it is difficult to state a general theory of searching, under certain realistic assumptions and restrictions, efficient computer oriented search methods can be given for well tailored problems.

In the previous chapter we have defined the general form of a file. Thus in the context of retrieval we assume that a record has several fields and a query is of one of the types mentioned in chapter 2. In general we may say that a query is either "key based", in the sense that the primary key that identifies the record uniquely appears in the qualification part, or "characteristic (attribute) based", in the sense that (attribute, value) pairs occur in the qualification part. We shall discuss file designs and measures of complexity for the associated retrieval methods for these two instances.

A search is initiated by a user who has a search key - the value of a primary key field of a target record. Search process is dependent on the file design, storage medium and other auxiliary tables of information. Typically the search key would be compared with a subset of the file before the target record is retrieved. The time that elapses from the instant the key specified by the user is transmitted to the storage medium containing the file, to the instant the record is read completely from the file is called search time. To a large extent the search time depends on:

1) the file design and storage medium;

2) the algorithm used for searching;

3) the access mechanism for the storage medium.

In key based retrieval on the file stored in the internal memory of a computer it is usual to measure the search time in terms of the number of comparisons made on the primary key fields of records. Such a measure is meaningful when whole key comparisons are feasible and are actually done. When the file is stored externally on typical secondary storage devices such as magnetic tape or disk, the search time would be dominated by the number of accesses to secondary storage; hence the search time should be measured in terms of the number of secondary storage accesses.

The significance of the search time is also dependent on the mode of retrieval.

An information retrieval system can accumulate enough queries to make one pass over the entire file and answer all the queries. This mode is known as the batch mode. When the user sits at a terminal and interactively uses the system, it is important to minimize the waiting time. This is known as on-line mode and the usability of an interactive system depends on the design that would minimize the on-line measure.

When the file is large and is stored on a secondary storage device such as disk, the time taken to search for a set of items is dependent on two quantities, namely:

1) the number of accesses (i.e. read/write commands)

2) the transfer rate of data.

In this thesis we consider only the number of accesses required to answer a characteristic based query. We do not consider the transfer rate for it would be machine dependent. We do not consider the amount of storage required, nor the time required for updates. However, for one of our methods we remark how the update operations can easily be performed (see chapter 6).

We will give a brief description of primary key based retrieval methods followed by search methods based on secondary keys.

Sequential search is well understood. When the records are in core the task of searching for a record for a given search key involves examining the keys of each record starting from the first one, until the target key is found and the record is retrieved. When all the search keys are assumed to be equally likely to occur and the search is initiated from the first record in the file, it can be shown that the average search time is proportional to the number of records in the file. The situation remains the same when the records are stored in a magnetic tape and serial search is performed.

If the file is sorted (in either ascending order or descending order) on the primary key, a technique called binary search may be used to locate a given key in the file. The binary search algorithm operates by comparing the given key with that of the record in the middle of the file and determining in which half of the file the key may be located. The procedure is repeated until the key is found or it is decided that it is not in the file at all. The maximum number of key comparisons for binary search is $\log_2 N$ where N is the number of keys (records) in the file.

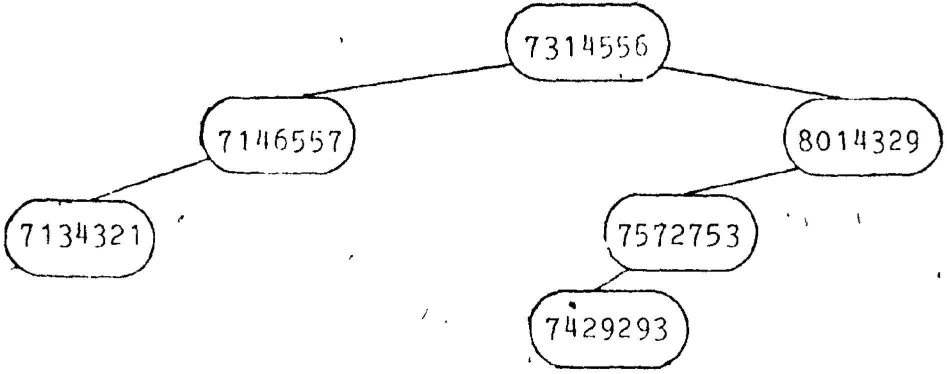
A major drawback of a sequential sorted file is that for every insertion or deletion the records in the file have to be moved such as to maintain the file sorted. This is a time consuming operation.

For a small but dynamically changing file, an explicit binary tree structure allows fast insertions and deletions of records and also enables efficient search. Construction of a binary search tree does not involve sorting the physical file, but rather setting up links between records. A record is either the "root" of the tree, or the left (right) "son" of some other record. The left son of a record has a primary key value smaller than that of its "father", while the right son has a primary key value larger than that of its father. In a general binary search tree one of the sons may have the same key value as the father, but we assume that all primary keys in the file are distinct. Note again that "smaller" and "larger" refer to the lexicographical order of alphanumeric keys.

As an example consider a file of student records having the student number (STUDENT #) as the primary key. Two additional fields (LEFT, RIGHT) are set up in each record to point to their sons.

ADDRESS	STUDENT #	NAME	COURSE	GRADE	LEFT	RIGHT
1	7314556	MIKE	CS211	A	3	2
2	8014329	JANE	CS241	A	5	-
3	7146557	MARK	CS211	B	4	-
4	7134321	ERIC	CS225	C	-	-
5	7572753	JOAN	CS241	B	6	-
6	7429293	CARL	CS211	C	-	-

When representing the tree nodes, only the primary key values will be shown.



An inorder traversal of this binary search tree will produce a file of records sorted in increasing order by the STUDENT # field.

There are algorithms for constructing an "optimum" binary search tree for a file. For non-uniform files (where not all the records have the same probability to be queried), the most frequently requested records should be placed as high as possible in the tree (near the root).

When the frequencies of searching on a given binary search tree are known, efficient algorithms exist for constructing optimal trees. Such methods are applicable in problems like linguistic analysis and searching in a census file. (See Knuth [22]).

There are situations when certain conflicting requirements have to be met, namely to minimize search time and insertion/deletion time. Search trees that accomodate frequent insertions and deletions which still retain optimal search time are known as AVL-trees (refer Knuth [22]).

Optimal trees are useful when the file is stored internally. For large files on which insertions and deletions are rare, a three-level tree is appropriate, where the first level of branching determines the cylinder number, the second level determines the appropriate track and the third level of branching gives the records. Such a method of organization is called an index sequential method [14] and search based on such a three-level tree is known as index sequential search method.

For illustration of the index sequential method, consider a file, sorted on the primary key field. The keys are as follows:

13, 16, 19, 21, 27, 30, 49, 53, 55, 58, 62, 67, 71, 74, 76, 92, 98.

Four groups of keys are created, with sizes $n_1=4$, $n_2=3$, $n_3=5$, $n_4=5$. The following table shows the groups of keys and their sequential indexes:

KEYS	GROUP SIZE	SEQUENTIAL INDEX
13,16,19,21	4	21
27,30,49	3	49
53,55,58,62,67	5	67
71,74,76,92,98	5	98

Suppose the search key is 74. This key is compared with each of the sequential indexes. In this case 74 is larger than 67 and smaller than 92. Thus it is determined that it can only be in the fourth group. Then 74 is compared with each of the keys in this group and the second comparison reveals equality. Had the search key been 75, for example, then upon comparison with the third key in the fourth group (76), the search would terminate unsuccessfully. Six comparisons were necessary to locate the key 74, and seven to determine the absence of the key 75. These numbers should be compared with 14 comparisons for the key 74 and 15 comparisons for the key 75 in serial search. Thus index sequential search achieves a considerable reduction in the number of comparisons in this case. It is not always better than binary search, however. For this example, binary search of key 74 would have required 4 comparisons. Using the index sequential search method for the key 16 requires 3 comparisons, while binary search requires 4 comparisons.

If there are M groups of keys and there are N keys in total, under the assumption of uniform querying, the average number of comparisons for index sequential search can be shown to be f(M)+f(N/M) where f(M) is the average number of comparisons on the index and f(N/M) is the average number of comparisons on a list of keys. The function f(n) is defined as:

a) $f(n) = (n + 1)/2$ for serial search

b) $f(n) = \log_2 n$ for binary search

For sequential search of both index and list the result is [M+N/M+2]/2; when both index and list are searched using binary search, the result is $\log_2 N$, the same as if the file had been stored sequentially sorted and binary search had been performed on it. (See S.P. Ghosh [14] for proof.)

The problem of inserting new records in a file organized by the index sequential method does not lend itself to a satisfactory solution. Usually some tracks are reserved as overflow areas and appropriate pointers are set up indicating the groups to which the overflow records belong.

An efficient data structure that almost guarantees efficient search as well as efficient updates has come to be known as a B-tree. (See Knuth [22].) A node in a B-tree contains j keys and j+1 pointers as follows:

P_0	K_1	P_1	K_2	P_2	....	K_j	P_j

P_0 points to a node which contains keys smaller than K_1; P_t points to a node which contains keys smaller than K_{t+1} for $1 \leqslant t \leqslant j-1$. Finally P_j points to a node containing keys larger than K_j. The properties of a B-tree of order m are:

a) every node has at most m sons;

b) every node except the root and leaf nodes has at least m/2 sons;

c) the root has at least 2 sons (unless it is a leaf node);

d) all leaves appear on the same level and carry no information;

e) a non-leaf node with k sons contains k-1 keys, $m/2 < k < m$.

It has been shown in [22] that the search time of a B-tree of order m is bounded by $1 + \log_b [(N+1)/2]$, $b=\lceil m/2 \rceil$ and the average amount of work per insertion is bounded by $1+(N-1)/(b-1)$.

Another method for storing and retrieving a record is to do some arithmetic calculation on the primary key and obtain an address at which the record is stored; this is known as hashing. Unlike the search techniques described earlier, hashing is not applicable to disk storage unless the following conditions are met:

1) the records are grouped into buckets and the addresses of the buckets are computed using hash functions;

2) more time is spent in computing the addresses since a careless computation might result in a collision causing a whole bucket to be brought in and examined before a decision is made.

Hash coding seems immensely suitable for several applications. Given S storage locations, we can define a hash function as a mapping $H : K \rightarrow \{1,2,\ldots,S\}$ where K is the set of keys in the file. The hash function $\hat{H}$ must be easily computable and preferably a random function of the input, i.e. each key is mapped onto a different storage location. Unfortunately such a function hardly exists. Easy methods such as chaining enable several records to be stored in a single list (bucket). A record R with key K will be stored in list i if H(K)=i, $1\leq i\leq S$, where S is the number of buckets. To search for a given key K we compute H(K) and examine all the records from that bucket. Such an examination can be done serially if the length of the list is small on the average. For a good hash function this is the case.

So far we have considered file designs and search methods based only on the primary key. When the queries specify values for several attributes, more sophisticated filing schemes and search methods are needed to achieve reasonable retrieval time for queries. For example, suppose there are k attributes A_1, ... A_k in a record and the set of queries allowed in a system are those that specify values for two attributes, A_1 and A_2. It is possible to organize two index sequential files F_1 and F_2 where F_1 is based on the values of the attribute A_1 and F_2 is based on the values of A_2. In retrieving the records pertinent to a query $q=(A_1=v_1, A_2=v_2)$ the file F_1 is searched first to locate the set S_1 of records corresponding to the value v_1 and the file F_2 is searched for locating the set S_2 of records corresponding to v_2. The intersection of these two sets S_1 and S_2 is the set of records pertinent to the given query q. Thus for this filing scheme two files F_1 and F_2 have been searched and an intersection needs to be performed. Such a method which keeps several indexes usually stores redundant information.

It is possible to speed up the retrieval time by combining hashing and index sequential organizations. The idea is to keep n distinct hash functions $H_1, H_2, \ldots, H_n$ and n sets of lists $L_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq s$, where s is the number of buckets in each list. A record R is stored in n lists L_{i,p_i}, $p_i = H_i(R_i)$, where R_i is the i-th attribute in record R. Although this method is an efficient solution to searching on secondary keys, both storage and update requirements grow as the file grows.

The two methods mentioned above are known as inverted list techniques since a separate list is maintained for all the records having a particular value for a particular attribute.

For illustration of inverted files, suppose the file given is an employee file, with attributes:

EMPLOYEE # (E#), OCCUPATION, SALARY, SEX.

The following table represents a file of 8 employee records.

ADDRESS	E#	OCCUPATION	SALARY	SEX
1	256	ANALYST	$15,000	F
2	359	PROGRAMMER	$12,000	M
3	676	OPERATOR	$10,000	M
4	739	ANALYST	$17,000	M
5	757	OPERATOR	$13,000	M
6	999	PROGRAMMER	$14,000	F
7	1029	ANALYST	$20,000	M
8	1256	KEYPUNCHER	$10,000	F

An index is kept for each of the attributes of the records:

OCCUPATION INDEX		SALARY INDEX		SEX INDEX	
ANALYST	1,4,7	$10,000	3,8	F	1,8
PROGRAMMER	2,6	$12,000	2	M	2,3,4,5,6,7
OPERATOR	3,5	$13,000	5		
KEYPUNCHER	8	$14,000	6		
		$15,000	1		
		$17,000	4		
		$20,000	7		

Note that in this case it is not necessary to keep an index for E# since this is the primary key.

A boolean query q=[(OCCUPATION=ANALYST)∧(SEX=F)] will search the index for OCCUPATION and extract the list (1,4,7).

Then it searches the index for SEX and extracts the list (1,8). The intersection of these two lists, address 1, constitutes the response to this query.

Inverted lists may be represented by bit strings. The length of the bit string is equal to the number of records stored in the file. For a particular attribute value in an index, the bits corresponding to the records which have that attribute value are set to 1, while the others are 0. Through careful programming and space allocation, it is possible to reduce greatly the time required to answer a boolean query for a file stored on disk with the use of an inverted file of bit string inverted lists.

In several instances it is possible to save the storage required for the inverted file by keeping a sparse index. In a sparse indexing scheme not all the key values of an attribute would be stored, but rather a selected set of values which specify range information. Such an organization may prove to be quite advantageous for range queries in addition to being useful for general boolean queries.

For an information retrieval system oriented mainly towards range queries or boolean combinations of range queries, tree structures seem to be more suitable. Queries of the type $[(v_1 \leqslant A \leqslant v_2)\ (u_1 \leqslant A \leqslant u_2)]$ where v_1, v_2, u_1 and u_2 are values of A_1 and A_2, are called orthogonal range queries.

There are several methods which can be used for systems which allow such queries exclusively. In one method the set of all possible values for A_1 and A_2 is partitioned coarsely and inverted lists are created for pairs of A_1 and A_2 values. This is in a sense a tree structure with two-dimensional branching at each internal node.

A multi-dimensional binary search tree has been suggested by Bentley [5], in which an internal node on an odd level compares attribute A_1 and an internal node on an even level compares attribute A_2. The average path length of such a tree is the same as that of a one-dimensional search tree.

A tree-structured approach is suitable for distance-oriented queries also, such as "what is the nearest city to Sudbury". In a binary search tree corresponding to this problem, a node stores 2 values, i.e. city and test radius. The left subtree of this node contains cities within the radius, while those in the right subtree are outside the radius. The radius diminishes uniformly from one level to the next, up to some tolerance limit δ.

In some instances it is possible to combine two or more attributes into one super attribute and construct an inverted file for each of the super attributes. In this way a boolean query may be satisfied by taking the union of several shorter lists rather than the intersection of larger lists.

Another technique for combining attributes is to obtain a super attribute for each permutation of the set of attributes, along with ordered inverted lists for each super attribute. This organization makes it possible to satisfy queries based on any combination of the attributes which were used in the permutations.

From our discussion in this chapter it is clear that each file design has a trade-off between access time and redundant storage space. In a purely inverted file design there is a great amount of redundancy of storage, whereas search time can be minimized.

There are many techniques for reducing storage overhead, for example secondary indices can be created which would reduce redundancy in the actual file, but this approach inherently requires storage for these secondary indices. Unless the query set is well defined and the pattern of querying remains stable, it seems difficult to construct a file design that guarantees minimum retrieval time. Even under certain probabilistic assumptions, the selection of attributes to form an index seems a hard problem (see Schkolnick [28]). This is precisely why we want to restrict the qGery set in addition to imposing probabilistic assumptions on the transactions of a file and examine retrieval algorithms.

CHAPTER 4

PARTIAL MATCH RETRIEVAL - REVIEW

Knuth [22] is one of the earliest to describe partial match file designs in an exhaustive manner. Although many of the file designs that we described in chapter 3 can handle partially specified queries, they are not efficient. The first efficient file design for partial match retrieval was proposed by Gustafson [16]; he was soon followed by Burkhard [7,8,9] and Rivest [25]. In this chapter we give a review of the various methods and discuss their relative efficiency.

In an inverted list organization a separate list is maintained of all the records having a particular key value. This technique is appropriate for retrieval for key based queries or when the number of secondary attributes which are specified is small. The reason for this restriction will become clear when we see how queries are processed in an inverted file system.

For the discussion below, the following notations apply:

A -denotes the set of all the attributes in a record, $A=\{A_1, A_2, \ldots, A_k\}$;

A' -denotes a subset of these attributes for which there exist inverted files, $A'=\{A_1', \ldots, A_j'\}$, $A' \subset A$, $j \leq k$;

B -is the set of attributes which are specified in a query, $B=\{B_1, B_2, .., B_s\}$, $B_i \in A$, $1 \leq i < s$, $s \leq k$;

B'-is a subset of B, containing those attributes for which there are inverted files, $B' = \{B'_1, B'_2, .., B'_t\}$, $t \leq j$, $B' = B \cap A'$.

A query with s specified attributes will be processed as follows:

1) for each attribute $B'_i \in B'$, $1 \leq i \leq t$, and a value v_i in the domain of definition of B'_i, construct a list $L(B'_i, v_i)$. This list contains the addresses of all the records in which the attribute denoted by B'_i has the value v_i;

2) for each list $L(B'_i, v_i)$, $1 \leq i \leq t$, form the intersection list $L = \bigcap_{i=1}^{t} L(B'_i, v_i)$. Note that this list contains the addresses of all the records in which the attributes in the set B' have the same values as the corresponding attributes specified in the query. The records in L match the query as far as the attributes in B' are concerned, but nothing can be said about the remaining attributes in B, for which there are no inverted files.

3) bring into memory the records from secondary storage whose addresses are given in L. Compare the values of the remaining specified attributes for which there is no inverted index, with the corresponding attribute values in the records (the set B-B'). This can be done, say, by sequentially searching the actual

records.

Since the number of lists formed in step 1 would increase as s increases, the amount of work to read these inverted lists into memory and form the intersection list L in step 2 also increases. However the size of L (the expected number of records) would decrease as s increases. Thus one has to do more work to obtain fewer records. This seems to be the main drawback in any inverted file system.

It would be reasonable to consider algorithms that would do an amount of work proportional to the size of the response (the number of records pertinent to the query). The first attempt to find an algorithm in this direction was made by Wong and Chiang [30]. However this algorithm is not strictly linear and moreover it uses an exorbitant amount of storage. Their basic strategy was to use a large number of short lists and construct the response to a query by taking the union of many such short lists. This is opposed to the phylosophy of the inverted index, where a small number of large lists are usually set up, and each list corresponds to a value of some attribute. In the technique of Wong and Chiang each short list corresponds to records of a partially specified query. But in order to accomodate all partially specified queries, this system would invariably require a large number of lists, proportional to the cardinality of the cartesian product of all the attribute domains.

To explain this better, assume the notation given above. If there are k attribute domains, and each attribute domain A_i has cardinality a_i, $1 \leq i \leq k$, then there are a maximum of $\Pi(1+a_i)$ different attribute combinations, and hence lists. Thus we can see why this technique is not practical. Either the number of possible partial match queries must be restricted or a compromise must be made on storage demands.

Setting an upper bound T on the number of specified attributes, one can limit the number of lists required. This is achieved by reserving one bucket for each possible combination of T attributes. The number of buckets that are needed is $\binom{k}{T}V^T$ where V is the maximum of the cardinalities of the domains. It should be noted that each record must be stored in $\binom{k}{T}$ lists . For any query in which $s \leq T$ attributes are specified, the response will consist of the union of some or all of the existing lists. This file design known as "algebraic filing scheme" was first introduced by Ghosh and Abraham [15].

A class of file designs that avoid redundancy of records to a great extent and are efficient will be known as superimposed coding. This technique is quite similar to hashing. The basic idea is to map each attribute into an m-bit code and superimpose the codes of each attribute to obtain an m-bit code for the whole record. Harrison [17] has given a superimposed coding technique to speed up text searching. The first efficient adaptation of superimposed coding to partial match retrieval was developed by Richard A. Gustafson [16]. We will describe his method in the following simple situation.

Consider a document retrieval system where each record is a technical article and each attribute is a keyword describing the article. Suppose there are 8 attributes $A_1, \ldots, A_8$. Let H be a hash function which maps each attribute into a number between 1 and 16. In Gustafson's method a 16-bit address $b_1 b_2 \ldots b_{16}$ is computed for each record where $b_i = 1$ iff $H(a_j) = i$ for some $1 \leqslant j \leqslant 8$, $a_j \in A_j$, $1 \leqslant i \leqslant 16$. Let p be the number of bits in the 16-bit string which have the value 1. If $p < 8$, then assign the value 1 to 8-p positions chosen at random from among the remaining 16-p bits which are 0. The aim is to have a code with a weight of 8 (i.e. 8 bits have the value 1). Since there are $\binom{16}{8}$ such bit codes, there will be $\binom{16}{8}$ buckets. The address of each bucket will have exactly 8 1's in it. On the average $N/\binom{16}{8}$ records will be mapped into each bucket, after computing the bucket address for each record (N is the total number of records in the file).

A query which specifies, say, 3 attributes, A_1, A_3 and A_5 will be processed as follows:

1) compute $H(a_1)$, $H(a_3)$ and $H(a_5)$, where $a_1 \in A_1$, $a_3 \in A_3$, $a_5 \in A_5$;
2) if these values are distinct, we need to examine $\binom{16-3}{8-3} = \binom{13}{5}$ buckets, which represents a proportion of the file of $\binom{13}{5} / \binom{16}{8}$.

In general, if b is the number of bits in the code and T is the number of 1's generated in the bit pattern, for a query that specifies s attributes out of T we need to examine $\binom{b-s}{T-s}$ out of $\binom{b}{T}$ buckets. Thus the amount of work decreases as s increases. In addition to this, each record is stored only once. But the union of the lists (buckets) formed in response to a query may contain undesirable records, thus forcing a sequential pass over these lists in order to extract the relevant records. However we note that all the relevant records will be retrieved in this method.

A hashing method differs from superimposed coding only in the way the address of a bucket is computed. If 1,2,...,b denote the addresses of buckets and F is the file of records, a hash function H maps F into (1,2,...,b). In the case of superimposed coding the bucket address was computed from up to k different hash codes, where k is the number of attributes. In a pure hashing scheme there is only one function that is made use of. However, what is common to both methods is the mapping of a set of records onto a bucket so that relatively few buckets need to be retrieved for a query. Below we review some hashing methods which are useful in partial match retrieval.

Assume that each record consists of non-binary values. Let k be the number of attributes and assume that each attribute is a character. If there are B buckets such that

$$k \leq w = \log B$$

then it can be shown [25] that the hash function

$$H(r)=H(c_1c_2...c_k)=h(c_1)h(c_2)...h(c_k)$$

is a "good function", where r is the record and $h(c_i)$ is a hash function mapping the i-th character onto w/k bits. The bits obtained from hashing each character are concatenated such that H(r) maps a record r onto k(w/k)=w bits, which give the address of the bucket where r is stored.

When a partially specified query is posed, with s characters specified out of k, s<k, then several bucket addresses are computed for the query. The number of buckets which have to be fetched depends on how many of the characters are unspecified. For each unspecified character c_j, $h(c_j)$ represents a set of $2^{w/k}$ bit strings of length w/k. Thus for a query with s characters specified, there are k-s which are unspecified, which requires

$$(2^{w/k})^{k-s} = 2^{w(1-s/k)} = B^{1-s/k} \quad \text{buckets}$$

to be fetched.

The importance of this result can be seen through the following example:

Suppose that the file contains 1,000,000 records and k=9. Let there be 2^{18} buckets, (that is 262,144), such that w=18. Then, according to the scheme described above, each record will be mapped into an 18-bit address, and each character (attribute) into a 2-bit address.

The query A*AL**I*G has 4 unspecified characters. The number of buckets which need to be examined is:

$$B^{1-s/k} = 2^{18(1-4/9)} = 2^{10} = 1024.$$

Since there are approximately 4 records per bucket, this means that only about 4000 records will be fetched.

Rivest [25] shows that the hashing scheme given above is optimal in the average case for non-binary attributed records. For binary records, when $k \geqslant \log_2 B$, Rivest shows that an optimal hash function is one which merely extracts the first $\log_2 B$ bits of each record for the list index. Sacerdoti [27] has also suggested this approach for partial match retrieval.

Although these hashing methods are optimal on the average, it is important to consider hash functions which are not only optimal on the average, but which have good worst-case performance as well.

Rivest [25] has developed a hashing method in which a query with s specified attributes corresponds approximately to $N^{1-s/k}$ hash addresses, where k is the number of attributes and N the total number of records in the file. This method requires no inverted lists and, for large values of N, is faster than the previously discussed methods.

The underlying idea of this partial match file design has been to place in each bucket records contained in a "subcube" of R_k, where $R_k=A_1 X A_2 X....X A_k$ (cartesian product of attribute domains). Below we briefly review Rivest's method. See [25] for details.

A partial match file design (PMF) is a table with k columns and $b=2^w$ rows with entries over {0,1,*}such that:

1) each row contains exactly w digits and k-w "*"s;
2) given any two rows there exists at least one column in which the two rows have different values.

Such a PMF design is called PMF(k,w).

As an example of a (4,3) PMF assume the following table:

1	*000
2.	*111
3	0*01
4	1*10
5	01*0
6	10*1
7	110*
8	001*

Each row in the table corresponds to a partial match query with 4 attributes of which 3 are specified. It is also the address of a bucket in which at most 2 records will be hashed. For instance, in bucket 1 records 0000 and 1000 will be stored (if they are in the file). This shows that a 4-bit record is mapped into a 3-bit address.

In order to obtain the response to the query *0*1 it is necessary to examine the buckets 3, 6 and 8.

Burkhard [9] remarks that the PMF design of Rivest [25] is restrictive and proposes tries as a data structure for indexing and hence partitioning the file into buckets.

A trie is a tree such that:

a) each leaf node corresponds exactly to one record and conversely;

b) each internal node i specifies an attribute position j uniquely on the path from the root to node i;

c) the node i in which attribute position j is specified has one subtrie for each value of attribute j.

This definition of tries was first given by de la Briandais [12] and Fredkin [13].

A generalization of the notion of tries as applied by Burkhard [9] to binary attributed records is a (k,w) PMF trie and it differs from the one above in the following points:

a) the trie is a full binary trie with $2^{w+1}-1$ nodes;

b) each leaf node corresponds to one bucket;

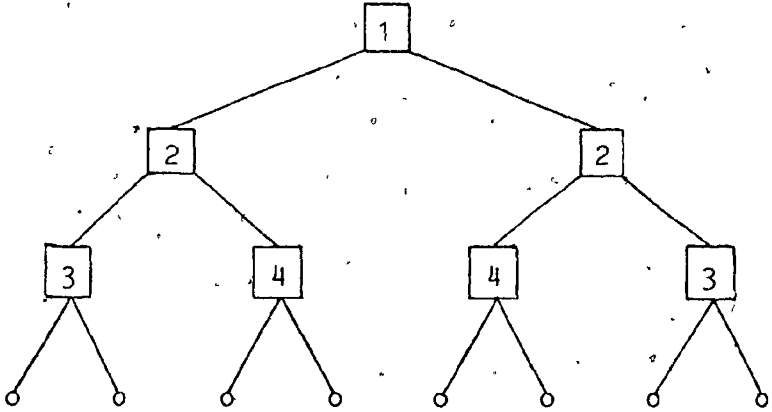
c) the left subtrie of a node i which specifies bit position j contains records having 0 in bit j and the right subtrie contains records with a 1 in bit j, $1 \leqslant j \leqslant k$, $k \geqslant w$.

This implies that on any path from the root to a leaf node (bucket) exactly w attributes are tested out of the k that each record has.

This definition can easily be extended to non-binary PMF tries.

A (k,w) PMF trie corresponds to a (k,w) PMF design. The converse is not always true.

The following is an example of a PMF trie and the corresponding PMF design.



1	000*
2	001*
3	01*0
4	01*1
5	10*0
6	10*1
7	110*
8	111*

FIGURE 5.1

To obtain the response to a query with s specified attributes, the trie is traversed as follows. The attribute which is specified at the root is tested with the corresponding attribute in the query. If it is 0, the left branch is taken, if it is 1, the right branch. When the attribute is * (undefined), both left and right subtries are traversed. This is repeated at each level.

Various classes of PMF tries can be defined, notably the family of (2n+1,n+1) PMF tries. The outstanding features of such tries are:

1) all the nodes specifying bit position j, $1 \leq j \leq 2n+1$, are in the same level of the trie;

2) the direct descendants of any node specify different attribute positions.

EXAMPLE: n=2

(5,3) PMF tries

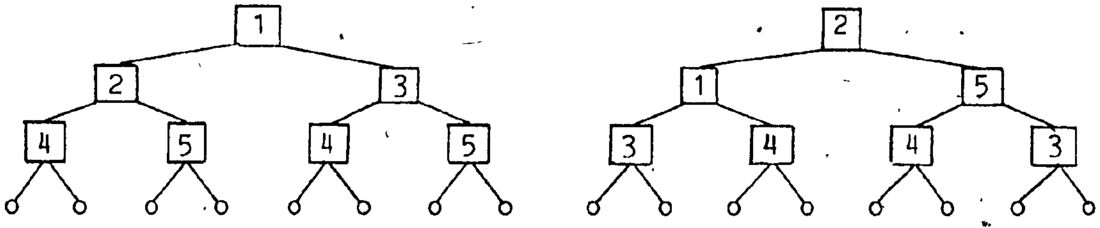


FIGURE 5.2

The average number of buckets which have to be examined for a query with s specified attributes is the same as for (k,w) PMF designs, i.e. $2^{w(1-s/k)}$ where $2^{w} = b =$ the number of buckets, in the case of uniform queries.

Rivest [25] also mentions the use of tries and derives a formula for the average cost. The study of Burkhard has been essentially restricted to tries. Burkhard shows that every PMF design corresponds to a trie in the class he considers and shows the generality of tries. Moreover the class of tries considered by Burkhard [9] has better worst-case performance.

To summarize, both Rivest and Burkhard have given efficient solutions to partial match retrieval. Both have considered tries as suitable data structures. Under the assumption of uniform querying(i.e. all partially specified queries are equally likely to occur), Rivest has shown:

1) the existence of a hashing scheme in which a) the average cost of answering a query with $s<k$ specified attributes is close to but greater than $(2-s/k)^{w}$, where 2^{w} is the number of buckets and b) the maximum cost is bounded by $2^{w-\lceil ws/k\rceil}$;

2) the existence of a trie for which the average cost of searching is $N^{\log_2(2-s/k)}$ where N is the number of records in a file;

3) finally he has conjectured that the expected number of records which have to be examined to answer a query with $s<k$ specified attributes has a greatest lower bound of $N^{1-s/k}$, where N is the number of records.

Burkhard's remarks are similar except that his tries have a much better worst-case performance.

In practice it does not seem natural to assume uniformity for queries. For example, in the case of a telephone directory, most of the time a query will specify the name of a person whose telephone number is requested. Thus it would not be realistic to assume that the attribute NAME has the same probability of unspecification as any other attribute.

Therefore it seems more desirable to consider file designs that are efficient in handling non-uniform queries. Rivest [25] has posed this as an open problem. The methods given in this thesis answer this problem in the following sense:

a) tries are good file designs even when the queries are non-uniform;

b) the shape of tries and hence the search time should be made dependent on the query distribution as well as the record distribution

and

c) there exists a method of constructing a trie for any given file, whose average performance seems to be optimal (in the sense conjectured by Rivest).

CHAPTER 5

PROBABILISTIC CONSIDERATIONS AND COMPLEXITY MEASURES

In this chapter we will discuss probabilistic notions that affect the design of an information retrieval system. It is evident from the work of Schkolnick [28] and Comer and Sethi [11] that the selection of a subset of attributes to form an index of a file so as to answer all boolean queries in minimum amount of time is "NP-hard"; that is this problem is intrinsically difficult and belongs to the class of NP-complete problems. It appears likely that none of these problems can be solved in polynomial time. Several researchers, especially in combinatorial investigations, have examined polynomial time algorithms that are approximately optimal to solve an NP-hard problem. However it is reported by Karp [20] that in several problems such approximation methods fail. As such probabilistic approaches to solutions of difficult problems have been attempted. Several problems and their probabilistic solutions have been discussed most notably by Karp [20] and Rabin [24].

Simplistically a probabilistic approach will assume a suitable "probability measure" associated with instances of a specific problem type and attempt a solution that is a possible "best solution" . We will make precise the notions of "probability measure" and "best solution".

In analysing an algorithm it has been a tradition to consider average case analysis and worst case analysis. In attempting average case analysis it is usual to assume uniform distribution on the space of instances of a given problem. For example the average time of a sorting algorithm is usually studied under the assumption that all possible N! instances of the input sequences are equally likely. Similarly the average search time of an item on a binary search tree is usually studied under the assumption that all keys are equally likely to appear in any input order. Such an assumption usually amounts to uniform labelling of the nodes in a tree subject to the property that the label of every node in the left subtree of the root is smaller than the label at the root and the label of every node in the right subtree of the root is larger than the label at the root; moreover this holds for the label of every node in the tree. (See [1] and [22] for the analysis of several problems based on this notion.)

From a practical point of view an algorithm which takes an inordinate amount of time on a small number of instances and remains fast on most other instances can be considered useful and hence an average case analysis is a proper measure of the performance of the algorithm. Although the assumption of uniformity that is usually made in the analysis gives some insight into the algorithm, it may be misleading, especially when the actual problem instances are statistically biased.

As an example of a probabilistic method in computer science, consider the problem of constructing a tree on which a given set of keys are to be searched. Assuming probabilities $p_1,...p_n$ and $q_0,...q_n$, where p_i is the probability that the given search key is K_i and q_i is the probability that the given search key lies between K_{i-1} and K_i; Knuth [22] and Hu and Tucker [19] have given "optimal" algorithms for constructing a binary search tree storing the keys. Here "optimal" means that the cost of searching the tree remains smaller than or equal to the cost that would be incurred on any other binary search tree which stores the same key values. This particular problem is an example that illustrates the strength as well as the weakness of a probabilistic approach.

The strength of this method lies in the fact that the tree constructed in this way guarantees minimum retrieval time as long as the relative frequency of the keys remains stable, that is the key set as well as the underlying probabilistic assumptions remain unchanged.

The weakness of this method is that the assumption about the distribution of queries (frequency of search keys) overlooks the possibility that the problem (i.e. the keys) may be changing in time in an unpredictable manner. Hence such methods are quite useful for a static file for which the assumed probability distribution is also static.

For dynamically changing situations it is important to conceive a randomization process in the algorithm itself which changes according to the problem instances. See the article by Rabin [24] for an excellent description of problems where probabilistic algorithms of this kind are most useful and effective.

In this thesis we give methods that have a strong bearing on the characteristics of the records in a file and that assume a known probability distribution of the query patterns. We assume the relative frequency of instances of query patterns in the query space. The sample of query instances actually appearing over a period of time might be estimated in an unbiased way and these are incorporated in the design and evaluation of our methods.

For example, if our data base is a telephone directory, we may assume that a particular attribute that is specified in a query depends on its significance as seen by the user of the data base. Let us assume that each subscriber record in the data base consists of five attributes: last name, first name, middle name, street address and telephone number. A query might specify the last and first name, or perhaps only the street address. The answer is a listing of all subscribers who match the attributes specified in the query.

Assume the following probabilities of attribute specification:

ATTRIBUTE	PROBABILITY
last name	.9
first name	.8
middle name	.1
street address	.2

The probability that a particular field is specified depends on the field but is independent of whether other fields are specified as well. The probability of any query (type) can be obtained by multiplying the appropriate probabilities of the associated fields. Thus a query that specifies the last and first name only has a probability of:

$$(.9)(.8)(1-.1)(1-.2)=.5184$$

The query with no specified fields (list all subscribers) has a probability of occurrence of:

$$(1-.9)(1-.8)(1-.1)(1-.2)=.0144$$

Although it is unrealistic to assume the possibility of such a query, it does not affect the design too much; yet it would somewhat influence it.

Assume a fixed number of buckets (2^{16}) and that there are 2^{20} records. Each bucket contains 16 records. According to Rivest's optimality criterion, a fixed set of bits must be selected from each record in order to determine the address of the bucket where it may be stored. In a query either all or none of the bits of a field are specified. Thus we need a way of determining how many bits to extract from each field to make up the bucket address. Intuitively the fields with the highest probability of specification should provide most of the bits.

In one approach, let all the bits belong to the field with the highest probability of specification (i.e. the last name). If there are are 2^{20} records in 2^{16} buckets, the address requires 16 bits. Choosing these 16 bits from the last name, 90% of the time exactly one bucket needs to be examined. The remaining 10% of the time all the buckets have to be fetched. So, on the average, a query will examine:

$$(.9)(1)+(.1)(2^{16})=6555 \text{ buckets.}$$

In another approach 8 bits may be chosen from the last name and 8 bits from the first name (the two most probable fields). Thus 72% of the time one bucket is examined, 18% of the time 2^{8} buckets, 8% of the time 2^{8} also, and 2% of the time all the buckets have to be examined, which gives an average of:

$$(.72)(1)+(.18)(2^{8})+(.08)(2^{8})+(.02)(2^{16})=1378 \text{ buckets.}$$

As illustrated by the previous examples, for a given file design one can always find a partially specified query for which the cost of search is maximum. However, if this query occurs more often than others in practice, the file design would be far from optimal even in the average case: Keeping this in mind one must design the file suitably so that the cost of retrieval would remain as small as possible for "almost all" query instances. This notion of "almost all" has been made clear in the paper by Alagar and Soochan [4].

Suppose B_s, A_s and W_s denote the optimal, average and worst case costs (on-line measure) for answering a query with s specified attributes. Let Q_1 contain the set of queries for each of which the cost of searching is either close to B_s or A_s, and Q_2 contain those queries for each of which the cost of searching lies between A_s and W_s. We say that the performance of the file design is near optimal almost everywhere (in probabilistic sense) if the following conditions are met:

1. The absolute value of $\lceil A_s \rceil - B_s$ remains small for all s=0,...,k. Here $\lceil \; \rceil$ represents the ceiling function.

2. $\sum_{q \in Q_2}$ Pr[q] remains small.

3. $\sum_{q \in Q_1}$ Pr[q] is large.

The significance of the proposed measures is to imply the average case behaviour as prevailing "almost everywhere" over the query space and to imply that the probability measure of query instances demanding an inordinate amount of retrieval time is small. As we remarked earlier our probabilistic notion is meaningful only within the context of our assumptions, however it remains independent of the file structure.

Burkhard [8] has discussed non-uniform partial match file designs; he achieves non-uniformity of labels within the tries that he constructs. One obvious accomplishment of this non-uniform distribution of labelling has been to obtain analytic solutions of certain reccurrence relations. Our concept of non-uniformity is different from Burkhard's in the sense that our probability space is the query space and not the record space. However the tries constructed by our methods might cause non-uniform distribution of labels within the constructed tries.

In chapter 6 we give four methods for the construction of tries. These methods assume that the records have binary attributes and are well suited to the problem instances:

a) non-uniform data and uniform query pattern;

b) uniform data and non-uniform query pattern;

and

c) non-uniform data and non-uniform query pattern.

In chapter 7 we consider data bases over non-binary alphabets. We employ superimposed coding and concatenation techniques to obtain a binary coded data base. The methods of chapter 6 can be applied to this coded data base and, with suitable decoding procedures, querying in the original data base can be handled. In such instances the coded data base must be processed in addition to the original data base. The amount of overhead involved in searching a coded data base depends on its size. This necessitates the introduction of "bit complexity" (explained in chapter 7) in addition to the complexity measures we have defined earlier.

Based upon these notions of probability and complexity measures we can briefly summarize the performance of our methods as follows:

The average case performance is improved for uniform query patterns and A_s satisfies the following inequality for binary attributed records and queries with s specified attributes:

$$N^{1-s/k} < A_s < N^{\log_2(2-s/k)} \qquad \text{for } 0<s<k$$

The right side of the inequality is the best average cost reported and the left side is the conjectured minimum cost. Our empirical studies show that for large values of k, and N close to 2^k, A_s approaches $N^{1-s/k}$. This only reaffirms the conjecture on the lower bound. In the case of non-uniform querying we have observed in several cases A_s lower than $N^{1-s/k}$; in general A_s oscillates on either side of this value. However, according to the "almost everywhere" concept, the average case behaviour prevails almost everywhere.

The results cited above are for a large file kept in secondary storage. In the case of a small file kept in core, the predominant factor in the cost is the "bit complexity", i.e. the number of bit comparisons which have to be performed in order to answer a query. For a trie structure, if s out of k attributes (bits) are specified in a query, the bit complexity of answering this query is:

$$B_q = C_q + k(\sum_{j \in L_q} N_j)$$

where

C = the number of internal node comparisons on the trie (that indexes the file) for a query q;

k= the number of bits in a record;

s= the number of specified bits in the query;

L_q= the set of lists examined;

$\sum_{j \in L_q} N_j$= the number of records in all the L lists.

Thus the average cost would be

$$[\sum_{q \in Q_s} B_q \Pr(q)] / [\sum_{q \in Q_s} \Pr(q)].$$

We shall comment on the significance of this measure in chapter 7 where we discuss superimposed coding.

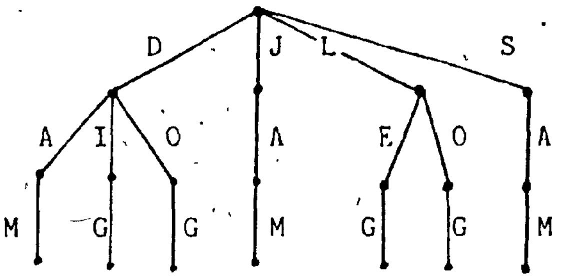
CHAPTER 6

ALGORITHMS FOR TRIE CONSTRUCTION

An efficient indexing mechanism to handle queries based on secondary keys was first introduced by de la Briandais [12] and the structure was named "trie" (tree in information retrieval). Fredkin [13] has given hardware implementation for tries. Rivest [25] briefly discusses a trie-based method for partial match retrieval. Burkhard [7] sets up a correspondance between a generalized partial match file (PMF) design and a trie structure.

If Σ is an alphabet of size k, a "full" trie can be constructed to discriminate between the strings that can be formed over Σ. Such a full trie is a k-ary tree whose leaves (terminal nodes) correspond to buckets and each internal node is a vector with k components, one for each element in Σ. Each node on level j specifies the set of all strings that begin with a sequence of at most k-j characters and the node specifies a k-way branch depending on the (j+1)-st character (or attribute).

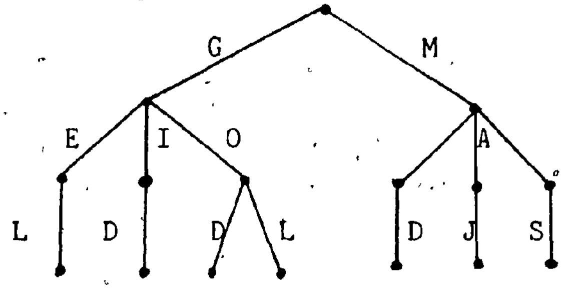
For example, the set of 3-letter words DOG, LOG, DAM, SAM, JAM, LEG, DIG, can be represented by the following trie structure:



level	character tested
1	1
2	2
3	3

Figure 6.1

By following a path from the root to a leaf node (terminal node), the word can be reconstituted. In each level a different character position is tested. In this case characters are tested from left to right and the resulting trie has 12 internal nodes. If the characters were tested from right to left, the trie would have 7 internal nodes, as shown below:



level	character tested
1	3
2	2
3	1

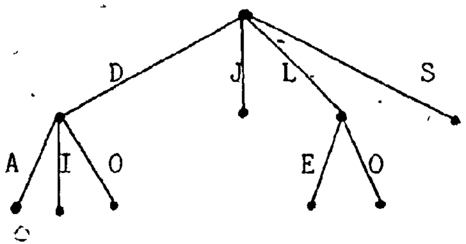
Figure 6.2

We can see that the order of testing the characters influences the size (number of internal nodes) of the trie.

It should be noted that each leaf node in a full trie stores exactly one string, each internal node specifies a character position such that it has not been specified on any level along the path from the root to this node.

If at any level j there is a node n_j below which there is only one path leading to a leaf node n_ℓ, we say that the path from n_j to the leaf node n_ℓ is a "leaf chain". This means that all the attributes (characters) tested below n_j are not discriminants and thus the leaf chain can be deleted and the leaf node n_ℓ can be attached in place of n_j. Such an operation is called "pruning". By pruning all the leaf chains we obtain a "pruned trie" that has a smaller number of internal nodes than the original full trie.

EXAMPLE 1: The pruned trie obtained from figure 6.1 is:

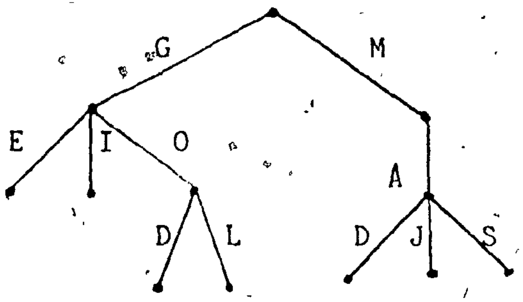


level	character tested
1	1
2	2

Figure 6.3

The trie in figure 6.3 has 4 internal nodes. Note that only sufficient characters are tested to differentiate the given words.

EXAMPLE 2: The pruned trie for figure 6.2 is



level	character tested
1	3
2	2
3	1

Figure 6.4

with 5 internal nodes.

A sequence $n_1 .. n_p$ of internal nodes such that n_i is the only direct descendant of n_{i-1}, $i=2..p$, will be called an "internal chain". All the attributes (characters) tested in the nodes n_1 to n_{p-1} are not discriminants, therefore they can be deleted. By removing all internal and leaf chains the number of internal nodes necessary for proper discrimination of the strings can be further reduced. Such an operation will be called "collapsing" the trie. The resulting trie will be called a "collapsed" trie.

EXAMPLE 3: The figure below represents the collapsed trie obtained from figure 6.2:

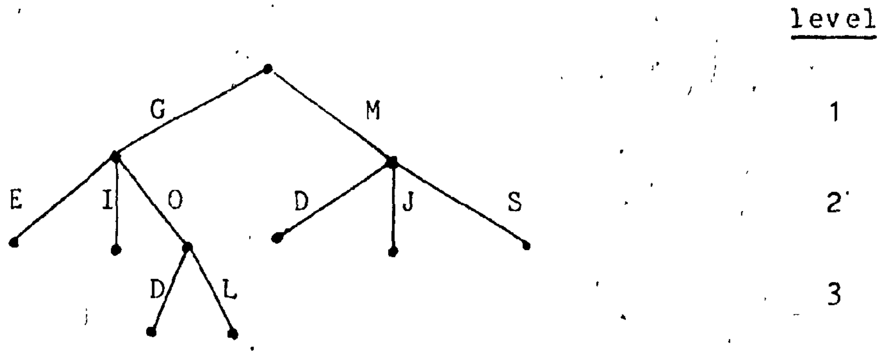


Figure 6.5

This trie has 4 internal nodes.

The two operations of deleting both internal and leaf chains from a full trie lead to a compact representation, but the position and the order of testing the characters in various levels are no longer explicit. It becomes therefore necessary to store in each internal node the attribute (character) position tested in addition to the k-vector. We call this structure an "order containing collapsed" trie (O-collapsed trie). The O-collapsed trie obtained from figure 6.5 is shown in figure 6.6.

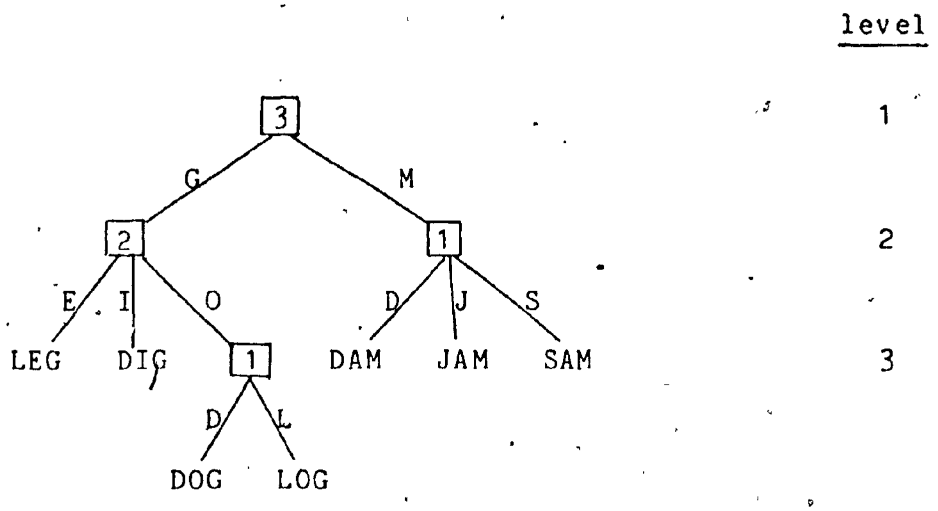


Figure 6.6

For the trie shown in figure 6.1, searching for a word involves starting from the root and taking the appropriate branch for each character position. If at some internal node there is no branch for a given character in the word to be searched, then the word is not present. Thus searching for the word DOG is successful, while the word DIP is not found as there is no branch for the character P on the path for the first 2 characters DI. If the same word DIP was searched on the trie in figure 6.2, right at the root it could be determined that it is not present in the file. The search time for the second trie in this case is smaller than for the first one. It is not difficult to see that tries are useful data structures for searching on partially specified queries. What is required is only a small variation on the search strategy, namely, if a certain character is not specified in a certain position and this position is tested in an internal node of the trie, then we need to search all the subtries of this internal node. For example, if D*G is a partially specified string and it is required to search and output all the strings with D and G as the first and third characters respectively, then the strings DIG and DOG are retrieved. If the trie in figure 6.1 is constructed, then the search will be performed as shown in figure 6.7 (arrows indicate the paths followed):

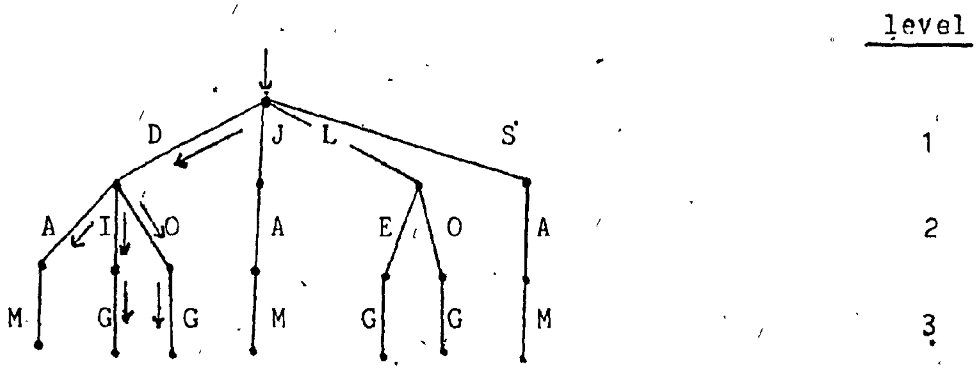


Figure 6.7

However, if the O-collapsed trie in figure 6.6 is constructed and searched for the same string, the search paths would be as shown below in figure 6.8

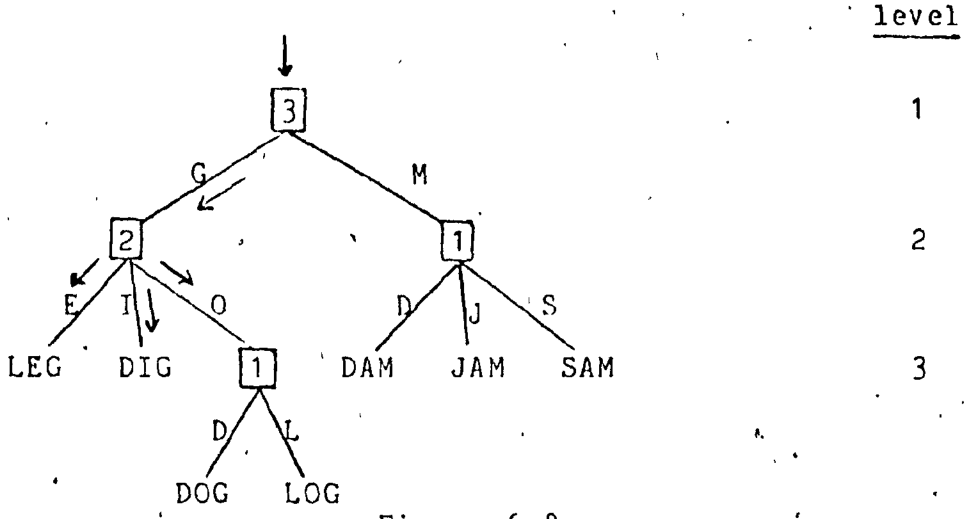


Figure 6.8

From these examples we can infer that the order of testing the attributes and the collapsing of the trie influence the size, height, shape and balance of the trie, and hence the search time.

In the case of the Latin alphabet which has 26 letters, a straightforward implementation of a trie is to represent each internal node by a 26-element array with an entry for each letter of the alphabet. Such a tabular implementation would allow the test at a node to be made in a constant amount of time. However many of the table locations would be wasted.

In order to save space any trie can be implemented in the form of a binary tree. The technique is to link a node with the first descendant (leftmost son) and to link all the sons of a node among themselves. Although such a binary representation of a general trie will save storage, we note that the time spent in selecting the appropriate descendant of a node is no longer a constant: in the worst case all the descendants may have to be examined. Severance [29] and Yao [31] have examined the trade-offs between the storage and time requirements and have suggested some heuristics that combine tabular and linked list representations. More precisely the first few levels in the trie will use tabular representation, thus allowing fast directed search, and linked list representation at the rest of the levels, which would reduce storage.

In this thesis we are not directly concerned with the storage requirements of a trie for two reasons:

1) we mainly consider binary attributed files and thus the trie is always binary;

2) any non-binary attributed file will be suitably coded before a trie is constructed.

As we mentioned in chapter 4 our interest in this thesis has been to consider file designs that minimize access time for any partially specified query.

When the file is small and is kept in core, every leaf node of an 0-collapsed trie will actually contain one or more records. In this case we have to minimize the average time taken to access one record. This average time we define as follows:

The access time of a leaf node in a trie is given by the depth (level) of the leaf. The access time of the trie is the sum of the access times of all the leaves. The average acess time of a leaf in the trie is the access time of the trie divided by the number of leaves under the assumption that all leaf nodes are visited with the same probability. In the case when the probabilities of the requests are non-uniform, the average access time is simply the weighted sum of the individual leaf access times, i.e proportional to the weighted sum of the number of accesses.

When the file is large and kept on disk, the leaf nodes in the trie are pointers to buckets. Assuming that the trie itself is small and can be kept in core, the access time of a bucket influences the overall performance of the trie. Hence the average access time for a query would be the weighted sum of the access times of the buckets.

Comer and Sethi [11] have shown that determinig minimal size tries is an NP-complete problem for several classes of tries. In particular they have shown that constructing an O-collapsed trie for a given file that has a minimal average access time is also NP-complete.

These results have a direct bearing on the trie and the file when the file is small. Yet, for large files, these results are not so significant since the dominating factor is the number of external accesses during retrieval.

EXAMPLE 4: Let k=3 and the file F be as follows:

001, 011, 100, 101, 111

The query q =(0*1) requests all the records with the first and third attributes (bits) equal to 0 and 1 respectively, without any regard to the value of the second attribute.

Let $q_2(F)$=[001, 011] be the response to the query q_2. The set of all possible queries for s=2 is Q_2:

queries in Q_2	records examined
01*	011
10*	101,101
11*	111
0*0	nil
0*1	001,011
1*0	100
1*1	101,111
*00	100
*01	001,101
*10	nil
*11	011,111
00*	001

One can verify that the total number of records which are retrieved from F to answer all the queries in Q_2 is 15. If we assume that the queries from Q_2 occur with equal probability, then the average number of records examined to answer any query from Q_2 is 1.25.

The records in the file F from this example may be stored as shown in figures 6.9 and 6.10. A record may be stored in a leaf node of the trie as soon as it is found that it is the only record in its subtrie.

The tries in figures 6.9 and 6.10 store the same file. Note the difference in shape and balance of the two structures. Both these tries are 0-collapsed.

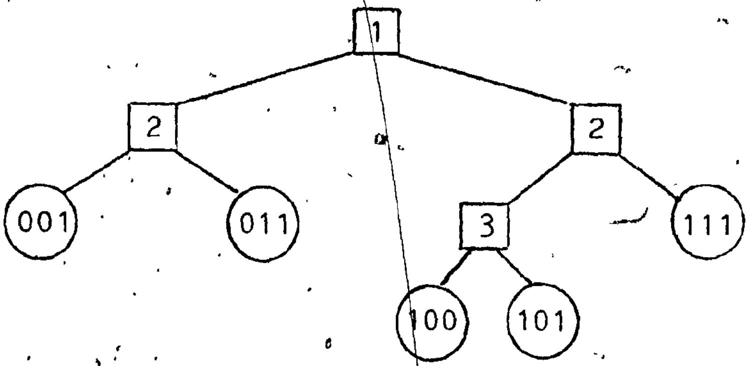


Figure 6.9

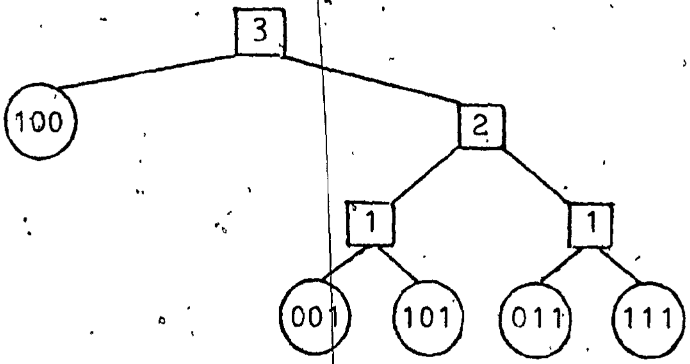


Figure 6.10

Assume the queries from the set Q' come more often than queries from the set Q".

$$Q' = \begin{bmatrix} 1*0 \\ **0 \\ *10 \end{bmatrix} \qquad Q'' = \begin{bmatrix} 0*1 \\ **1 \\ *0* \end{bmatrix}$$

In this case fewer comparisons need be made in the trie of figure 6.9 than in the one of figure 6.10. Moreover the number of records examined in the latter trie will be smaller than the number of records examined in the former in order to answer a single query from the set Q'.

In the example discussed above, each leaf node contains exactly one record. In a more general case a leaf node may contain several records (it is then called a 'bucket'). If more than one record is stored per bucket, the size of the trie is reduced. Often it may be necessary to meet certain constraints of space by fixing the number of buckets which can store the records. For instance, in the file F given in the previous example, there are 5 records. If we fix the number of buckets to 4, there are several possible resulting structures, some of which are illustrated in figures 6.11a to 6.11d.

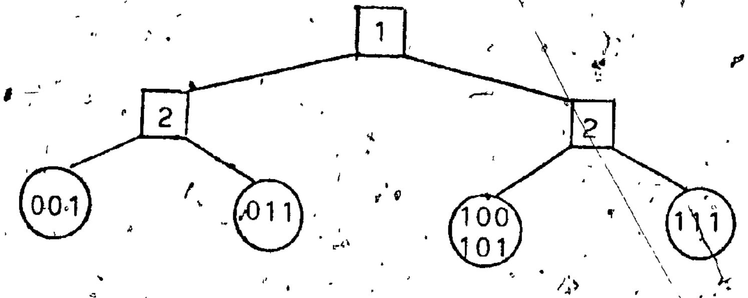


Figure 6.11a

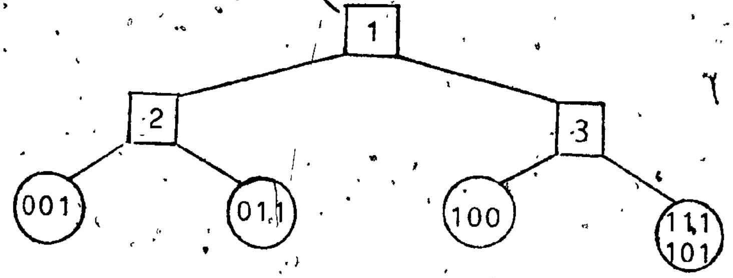


Figure 6.11b

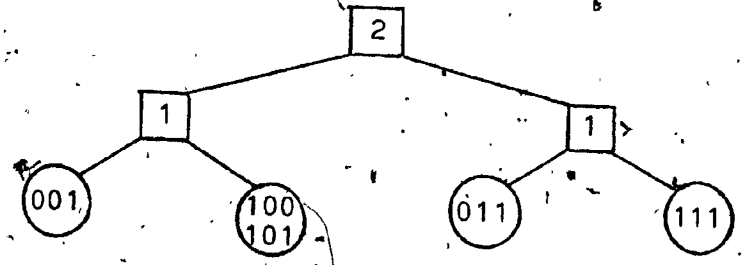


Figure 6.11c

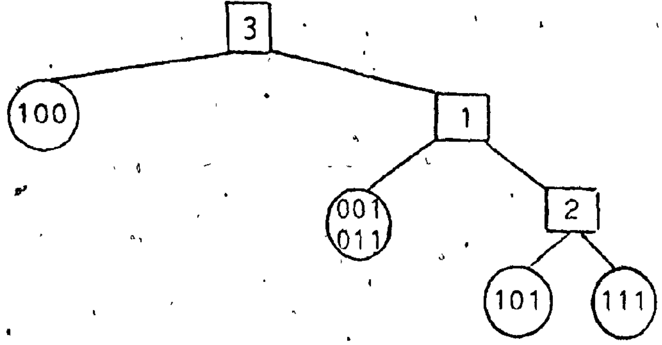


Figure 6.11d

For the set Q_2 of all queries with 2 specified attributes and a set of probabilities P_2 corresponding to the queries in Q_2, the number of buckets examined by each query in each trie from figure 6.11a to figure 6.11d is shown in table 6-1.

THE NUMBER OF BUCKETS EXAMINED PER QUERY
IN FIGURES 6.11a to 6.11d

query	probability	6,11a	6.11b	6.11c	6.11d
*00	1/30	2	2	2	1
*01	1/12	2	1	2	2
*10	1/24	2	1	2	1
*11	5/72	2	1	2	2
0*0	3/80	2	2	2	1
0*1	1/16	2	2	2	1
1*0	1/80	2	1	2	1
1*1	1/48	2	1	2	2
00*	1/30	1	1	1	2
01*	1/24	1	1	1	2
10*	1/90	1	2	1	2
11*	1/72	1	2	1	2
total	83/180				
average (uniform)		1.67	1.58	1.67	1.58
average (non-uniform)		1.77	1.58	1.78	1.59

TABLE 6-1

The results of this table indicate that the trie in figure 6.11b has the smallest average access time for the given set of probabilities. If we assume that all queries are equally likely, however, then the tries in figures 6.11b and 6.11d are the best, both having the same average access time.

SUMMARY

In general fixing the size of the trie fixes the number of buckets and conversely. But the shape and height of the tries vary.

The methods that are given below will choose the bits and impose an ordering for testing them, thus splitting the file into buckets minimizing the access time of queries. All the buckets examined by a search will contain all the records in the data base that satisfy the given partially specified query. However, all the records in the buckets examined are not necessarily relevant to the given query. This would happen especially when a certain specified attribute in the query is not chosen for testing anywhere along the search path in the trie.

6.1 METHOD A

It is observed in [25] that unbalanced tries have good average case performance. Assuming that all partial match queries are equally likely, the probability that a query will examine a node at level ℓ in the trie is $(2/3)^{\ell}$; this is because there are $2^{\ell}3^{k-\ell}$ queries that will visit a node at level ℓ and there are 3^{k} partially specified queries in all. This implies that a query will less frequently visit the node that is farthest from the root. This observation suggests that the trie should be constructed so as to maximize the skewness at every level, hoping to achieve global skewness.

The notion of skewness or unbalance at a node can be defined in more than one way. Recall that a bit position will be tested at every level and hence the measure of unbalance at a node is the same as the measure of unbalance of a bit position among the given records. We propose two measures of unbalance.

Consider the file as a table of N rows and k columns. Compute $C_1, C_2, \ldots, C_k$, where C_j is the j-th column sum. Then the ratio $C_j/(N-C_j)$ is a measure of unbalance for bit position j. If this ratio is 0 then all the records have 0 in j-th bit position, alternately, if the ratio is infinity, all the records have 1 in the j-th bit position. Thus such a measure will accomodate all ratios in $[0,\infty)$. If necessary we can redefine the above ratio as $(1+C_j)/(N-C_j+1)$ which would force a finite range for the ratios.

Another measure for the same bit position j would, be the absolute value of $(N-2C_j)/N$. The significance of this measure is that it reflects the relative difference between the two groups of records that are partitioned by the bit position j. Moreover the range of this ratio will be [0,1]; the ratio is 0 when the subfiles partitioned on the j-th bit have the same number of records. The ratio is 1 when all the records have the same bit value in position j.

Whatever be the choice of measure that is consistent with the notion of unbalance, the main idea is to choose that bit position for a level in the trie which maximizes the skewness at that level as well as at every level in that subtrie. It seems advantageous to look ahead several levels down in the trie in choosing a bit position to be tested at a certain level. Such a look-ahead scheme attempts to avoid those choices of bit positions that would optimize locally but not globally.

Informally we describe the method of trie construction as follows:

For definiteness assume the measure $C_j/(N-C_j)$ for every bit position that has not been selected yet for testing in the trie. For the full file of N records we compute the k ratios $C_j/(N-C_j)$, and select j, $1 \leq j \leq k$, for which $C_j/(N-C_j)$ is maximum. However if the maximum ratio is infinity for this choice, (which happens when all the records have the j-th bit 1 and in this case we need not test this bit position on the trie), we select the maximal ratio less than infinity and that is the bit position (say j) to be tested at the root of the trie. Subdivide the file into 2 parts so that the left subtrie will have records which have 0 in attribute position j and the right subtrie will have records with 1 in attribute position j. We recursively construct the subtries for the partitioned files. The constructed trie is an 0-collapsed trie, having N-1 internal nodes and the property that the unbalance at every level is maximum.

EXAMPLE 6-A: The file is F=(a,b,c,d,e,f) where

a=1011, b=0001, c=0110, d=1101, e=1010, f=1111.

The trie constructed by method A for this file is:

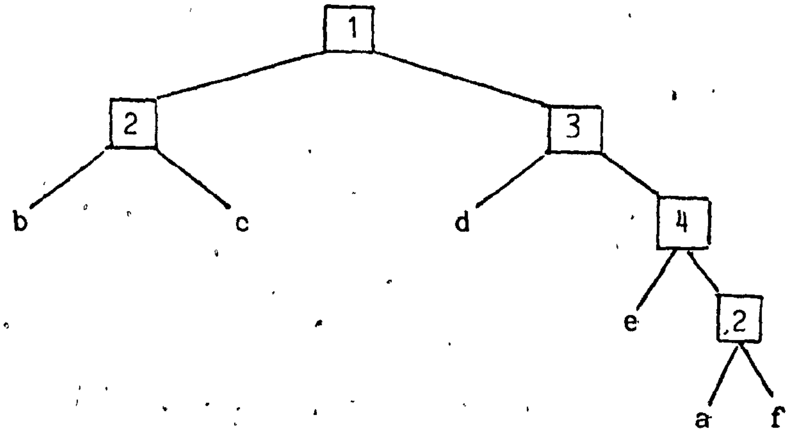


Figure 6.12

The trie shown in figure 6.12 was constructed without using the look-ahead scheme. It is easy to check that the ordering of attributes to be tested at the root is [1,3,4],2, where we have listed within [ ] the attributes that have the same ratio. If we choose 1 to be tested at the root and insist that the attribute tested at the root must have an unbalance ratio at least as high as the parent node, and recursively for every level, then we obtain the trie in figure 6.13.

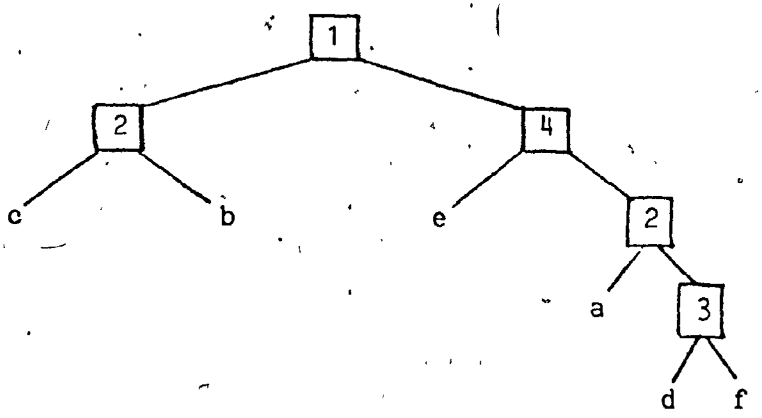


Figure 6.13

Alternately if we choose 3 and use the look-ahead scheme in successive levels, we will get the trie in Figure 6.14.

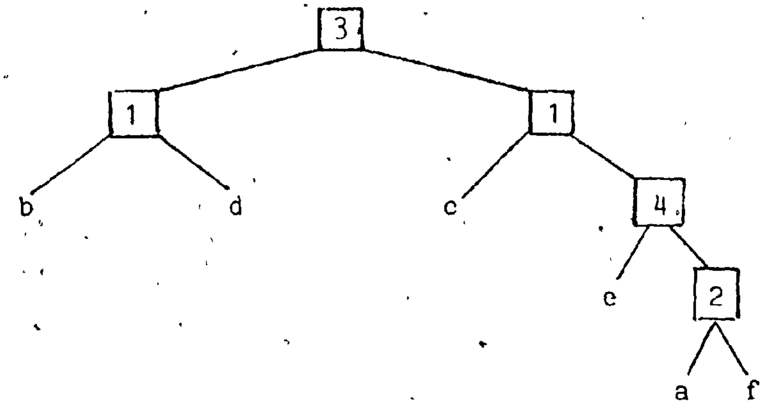


Figure 6.14

We observe that for this particular choice of file the look-ahead has not improved the structure of the trie for the reason that each subtrie below the root is split as skewed as possible in figure 6.12.

When we have a large file we must take into account the number of available buckets and the maximum size of each bucket in pruning the trie to the desired level. This can be enforced by controlling the splitting process of the file; i.e. we stop splitting as soon as its size is no larger than the maximum size of the bucket. There are some consequences of this decision in the deletion and insertion of records. We shall comment on this later.

COST OF CONSTRUCTING THE TRIE

There is no cost if only one record is given. In order to compute the ratios at any node in level j, we have to make one pass over the entire file of N records and another pass to split the file. Let us aggree that accessing one record is equal to one unit of work. Let C(N) denote the cost of constructing any trie for N given records. Then

$$C(N)=2N+C(N1)+C(N2)$$

where $N1+N2=N$, $N\geq2$, $C(1)=0$.

If the records are such that at each level the balance ratio is the same, we get a completely balanced trie. The cost of constructing a balanced trie is given by:

$$C(N)=2N+2C(N/2),\ N\geq2,\ C(1)=0$$

Clearly $C(N)=2N \log_2 N$.

At the other extreme, if the records are such that a trie of the class of the most skewed tries ("one and the rest" splitting at each level) is obtained, then we have:

$$\begin{aligned} C(N) &= 2N+C(N-1)+C(1) \\ &= 2N+C(N-1) \\ &= 2N+2(N-1)+C(N-2) \\ &\ldots\ldots\ldots\ldots\ldots\ldots \\ &= 2[N+(N-1)+\ldots+2] \\ &= 2[N(N-1)/2 +1] \\ &= N^2-N-2 \end{aligned}$$

This simple analysis shows that most skewed tries are most expensive to construct. However this is only the worst case. Let us define the class of (p,q) tries, $0<p<q<1$, $p+q=1$, as those tries for each of which the property "the measure of unbalance of each node lies between p and q" is true. The records for which such class of tries can be constructed would have the property that for every bit position j the ratio of the number of records having this bit as 1 to the number of records having this position as 0 will be at most p/q. Thus the cost C(N) will satisfy the equation:

$$C(N)=2N+C(pN)+C(qN),\ p+q=1$$

It can be shown that $C(N)=2N \log_{\alpha}N$, where $\alpha = p^{-p} q^{-q}$. We verify this as follows:

Substitute in the right hand side of $C(N)=2N+C(pN)+C(qN)$.

$$\begin{aligned}
&2N+2pN \log_{\alpha}pN+2qN \log_{\alpha}qN \\
&= 2N(1+p \log_{\alpha}N+q \log_{\alpha}qN) \\
&= 2N \log_{\alpha}(\alpha\, p^{p} N^{p} q^{q} N^{q}) \\
&= 2N \log (p^{-p} q^{-q} p^{p} q^{q} N^{p} N^{q}) \\
&= 2N \log_{\alpha}N \\
&= C(N)
\end{aligned}$$

Thus it is moderately expensive to construct the most skewed tries. However we remark that only skewed tries are very effective for partial match success. Although we do not have a nice analytic result for the cost of searching a trie constructed by this method, (i.e. the most skewed trie possible), our results obtained through systematic and exhaustive construction and search reveal that the average cost of searching for uniform queries is very low. In fact we have computed the average A_s, the average cost for answering a query with s specified attributes, which satisfies the inequality:

$$N^{1-s/k} < A_s < N^{\log_2(2-s/k)} , \quad 0 < s < k$$

where the left hand side is the conjectured minimum cost for any algorithm and the right hand side is the best average cost reported for a trie [25]. In some instances A_s is actually equal to the lower bound. One conclusion is that this method improves the average case performance. However there seem to be only fewer number of queries for which the cost of search is far away from the average. Hence, according to the "almost everywhere" concept introduced in chapter 5, the average case behaviour prevails almost everywhere. See the table at the end of this chapter as well as the remarks following.

When new records are inserted or some of the existing records are deleted, a trie constructed for a file using this method seems to require a total reorganization most of the time. In order to insert a new record a straightforward approach would be to search the trie and insert the record as a leaf node. For example consider the trie shown in figure 6.12. Suppose record g=(0011) is to be inserted. The node Y below which this record is to be inserted is found and a new internal node is created to test an attribute which distinguishes g from the record in the leaf node which is the left son of Y. This new node is attached as the left son of Y and the two records are the left and right leaves of Y.

Figure 6.15 below shows the new trie obtained in this way.

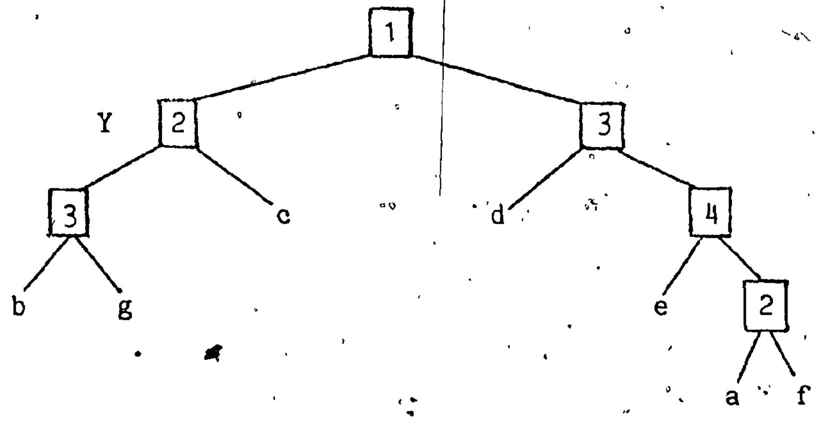


Figure 6.15

Clearly the trie shown above does not have the property of maximum unbalance at every node. This is obvious if we were to rebuild the trie completely, including the new record g. See figure 6.16 for one of the tries which could be constructed using method A.

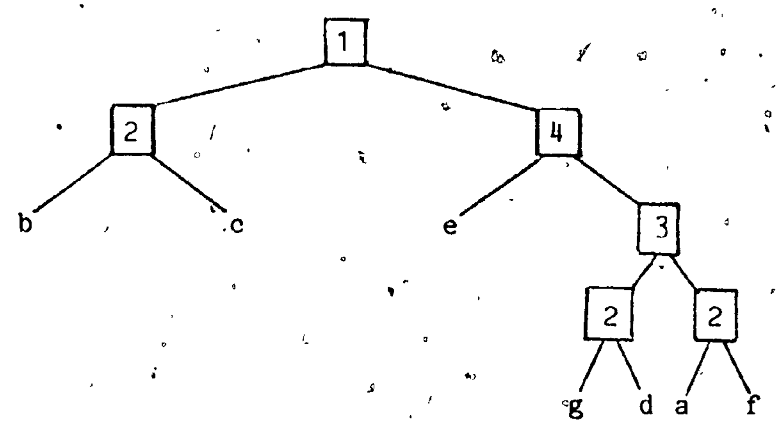


Figure 6.16

It is obvious that the straightforward insertion of a record in an unbalanced trie tends to increase the balance of a node at a certain level. The best that can be done is to reorganize the subtrie whose root has gotten more balanced in the most recent insertion. Similar remarks apply to deletion of a record. Thus method A seems more suitable to situations where there are frequent retrievals and infrequent updates. As we have already remarked, under the assumption of uniform querying the average number of records examined is very close to $N^{1-s/k}$ for a query with s specified bits.

If the queries are assumed to have a non-uniform distribution, our empirical studies reveal that the expected number of records examined for a query with s specified bits oscillates on either side of $N^{1-s/k}$. This observation leads us to conclude that:

1) the conjectured minimum $N^{1-s/k}$ seems to hold good only for uniform querying.
2) the optimality of the number of records examined for non-uniform querying is still an open problem.

Further we describe a simple method of trie construction to handle non-uniform partially specified queries.

6.2 METHOD B

We assume that P is a given probability matrix having k rows (one for each attribute) and three columns (for the values *,0,1). The entry P_{ij} is the probability that the i-th attribute will have the value j, $1 \le i \le k$, j=*,0,1. The probability associated with any query can be computed from P. In real situations the most probable queries are concentrated in a small subset of the query space and the matrix P is assumed to reflect such a situation.

This method is mainly concerned with P and one may assume that the record space is uniform, i.e. the N records are equally likely to be chosen from 2^k records. Since a search on the trie will visit both subtries of a node at level ℓ if the attribute tested there is unspecified in the query, and the probability of unspecification of that attribute is available in the matrix P, it is important to test that attribute in the lowest possible level in the trie; for this would decrease the number of buckets examined. Towards this we choose those attributes with high probability of unspecification and test them lower in the trie and choose those with low probability of being unspecified to be tested higher in the trie. Hence, we order the probabilities that the attributes may be unspecified. If $(j_1,\ldots,j_k)$ is the ordering of the attributes corresponding to the increasing order of the probabilities of unspecification, then this is the order in which the attributes would be tested along any path from the root to a leaf node. Without any regard to the record space, the previous ordering establishes a full binary trie skeleton. At the root the attribute j_1 is tested; in level 2 the attribute j_2 is tested and so on. The records are stored as leaf nodes in this trie and finally the trie is collapsed to minimize the internal path length.

In this method the order of testing the attributes is independent of the record space. This trie will retain the same relative order of testing of the attributes, i.e. j_i is tested higher in the trie than j_ℓ if and only if j_i precedes j_ℓ in the ordering obtained from P. Note that in the collapsed trie not all the attributes will be tested along every path from the root to a leaf node. We have reason to believe from our results that the branching decisions made on the untested bit positions are favorable both in the average and worst case analysis. The trie constructed by this method for the file given in example 6-A is given in figure 6.17 along with an assumed query probability matrix. We call any trie constructed by this method an "order containing and order preserving trie", or "OP-trie". The order preserving property of a trie has a great impact on insertion and deletion. That is, insertion or deletion of a record will not change the skeleton structure of the trie. Moreover our empirical evidence indicates that tries constructed by method B for any arbitrary file for various query distributions peform far superior to tries that can be constructed by any other method. Although the evidence makes us believe that method B constructs optimal tries for a given distribution, we have not been able to establish this analytically.

EXAMPLE 6-B:

$$P=\begin{array}{|ccc|} * & 0 & 1 \\ \hline 1/3 & 1/2 & 1/6 \\ 1/4 & 1/3 & 5/12 \\ 1/5 & 3/10 & 1/2 \\ 1/2 & 1/5 & 3/10 \\ \hline \end{array}$$

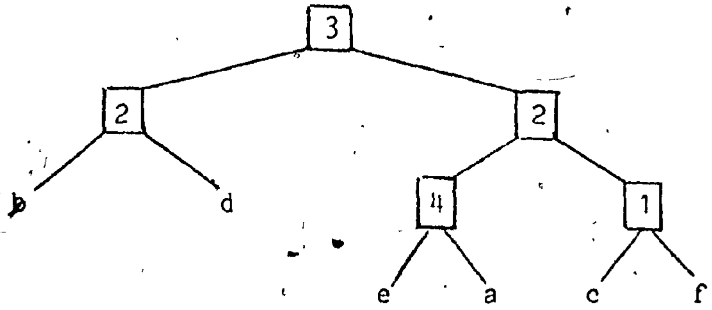


Figure 6.17

When constructing a trie by this method, no ratios need be computed. Since the probabilities of unspecification have to be sorted, the cost for this step would be $O(k \log_2 k)$.

If $2^{k+1}-1$ is small, the skeleton trie can be kept in core and the records can be inserted in the proper leaf nodes. This requires one pass over the file and k comparisons for each record. Usually k is too large to maintain a skeleton trie at any one time, but it will be shown that this is not necessary.

Starting at the root node at most one pass per attribute is necessary to subdivide the file at every node. If one of the subfiles is empty for a particular attribute, then the attribute is not tested and the next attribute in the pre-established ordering will be chosen for testing. Hence collapsing is enforced automatically. Moreover at most k attributes are tested along any path and at most one pass is made for every attribute. Thus the cost of constructing a collapsed OP-trie is $O(k \log_2 k)+O(N)$.

In practice, when the size of a subfile becomes smaller than or equal to the size of a bucket, no more splitting and hence no pass is necessary. Hence the actual cost will be smaller than the one indicated above.

Whereas method A ignores query distributions, method B ignores the characteristics of the record space. Method C that we propose below attempts to exploit both the non-uniformity of the record space and query space. In some sense we correlate the structure of most probable queries with their most relevant records and choose bit positions to test at various levels of a trie.

6.3 METHOD C

From the file F considered as a (0,1) matrix and the query distribution matrix P, we form the product F*P, a matrix with N rows and 3 columns (N is the number of records in F). The entries in this matrix can be looked upon as "similarity measure" between the record space and query space. The selection of attributes and their order of testing is done as explained below:

Find the minimum in the first column of F*P and the maximums in columns 2 and 3. Consider the subset of F indicated by those row numbers wherein these extremums occur. Note that these extremums may not be unique. Apply method A to this subset of F and select that attribute number that maximizes the unbalance at the root. Partition the file so that all records with 0 in the selected attribute position will be in the left subtrie and those with 1 will be in the right subtrie. The partitioned files and the remaining attributes are used to compute the similarity measure matrix for the subtries and the procedure is repeated. The significance of the bit selected at each node is that it has the least probability of unspecification and maximum probability of finding matching records in the file.

EXAMPLE 6-C: Take the same file F and probability matrix P from Example 6-B used for illustration of method B. The product F*P is shown below:

$$F*P = \begin{bmatrix} 7/6 & 1 & 29/30 \\ 1/2 & 1/5 & 3/10 \\ 7/12 & 19/30 & 11/12 \\ 13/12 & 31/30 & 52/60 \\ 2/3 & 4/5 & 2/3 \\ 17/12 & 4/3 & 83/60 \end{bmatrix}$$

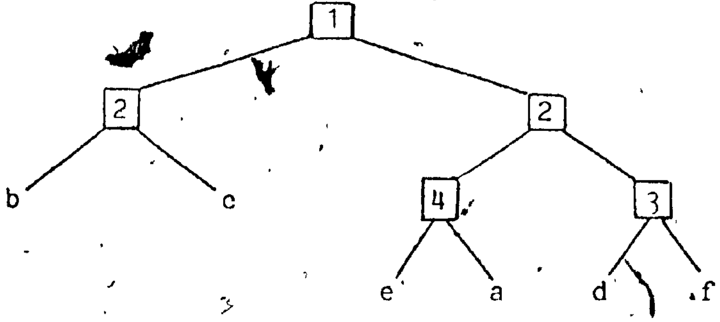


Figure 6.18

Note that the structure of the trie of figure 6.18 is similar to that of the trie constructed by method B. Since method C uses method A for selection of attribute positions to be tested, let us employ the look-ahead scheme discussed in conjunction with method A. The trie obtained is that of figure 6.19.

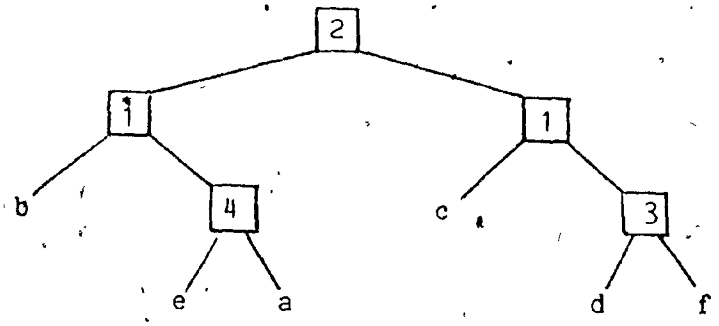


Figure 6.19

6.4 METHOD D

Assuming that there is one record per bucket we will construct a trie in a top-down manner and choose the attribute at a node that minimizes the average number of buckets examined at that level. We assume a given probability matrix that gives the query distribution. At the level of the root let j-th attribute be tested. Let L_j and R_j denote the number of records that have 0 in the j-th position and the number of records that have 1 in the j-th position, respectively. Regardless of which other attributes are specified in a query the expected value of the maximum number of records examined at the root level is:

$$A_j = P_{j0} L_j + P_{j1} R_j + P_{j*} (L_j + R_j)$$

We want to minimize A_j. Let $N_j = L_j + R_j$. Then we have $\alpha_j = L_j/R_j$.

$$A_j = N_j[P_{j*} + (\alpha_j P_{j0} + P_{j1})/(1 + \alpha_j)]$$

Since $P_{j*} + P_{j0} + P_{j1} = 1$, we substitute for P_{j*} in A_j to get

$$A_j = N_j[1-P_{j0}-P_{j1}+(\alpha_j P_{j0} + P_{j1})/(1+\alpha_j)]$$

$$= N_j(1 + \alpha_j - P_{j0} - \alpha_j P_{j1})/(1 + \alpha_j)$$

$$= N_j[1 - (\alpha_j P_{j1} + P_{j0})/(1 + \alpha_j)]$$

Thus in order to minimize it is enough to minimize $1 - \beta_j$, where

$$\beta_j = (\alpha_j P_{j1} + P_{j0})/(1 + \alpha_j) .$$

This is equivalent to finding the attribute position j that maximizes β_j.

Initially N_j=N, the number of records in the file. The ratio α_j , j=1,...,k is obtained for the whole file. By maximizing β_j for this α_j (simply finding the maximum of the β's), we get j, the attribute to be tested at the root level of the trie. The file is split on the j-th attribute and the subtries are constructed recursively. The example given below illustrates this method.

EXAMPLE 6-D:

Again we use the same file F of 6 records and matrix P from section 6.2 and example 6-C. In order to compute the ratios β_j, $j=1,..,4$, the j-th column sum has to be calculated. For the first pass over the file, the following results are obtained:

$$\alpha_1 = 1/2 \quad \alpha_2 = 1 \quad \alpha_3 = 1/2 \quad \alpha_4 = 1/2$$

$$\beta_1 = 1/3 \quad \beta_2 = 3/8 \quad \beta_3 = 11/30 \quad \beta_4 = 7/60$$

$$\max(\beta_j) = \beta_2$$

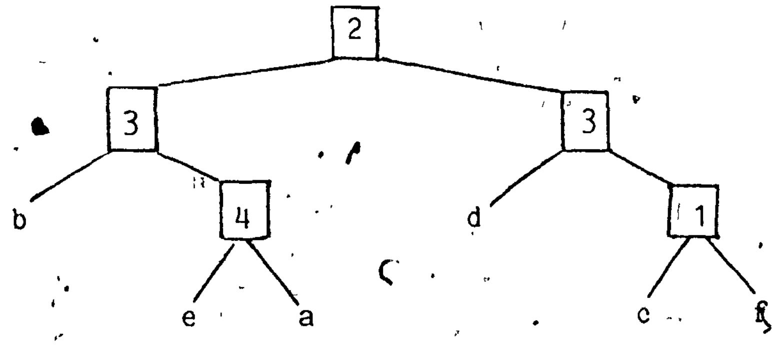


Figure 6.20

The results obtained for these methods for a sample data are shown in the table attached at the end of this chapter. We have done several experiments and the results lead us to believe that all the methods have better performance than reported in [25]. Moreover, tries constructed on probabilistic assumptions have good performance almost everywhere over the query space.

NOTATION FOR TABLES 6-2 TO 6-5

k = number of bits in a record

s = number of specified bits in a query

B = best case cost

W = worst case cost

Tb = number of queries that achieve B

Tw = number of queries that achieve W

A = average case cost

Ta = number of queries that exceed the rounded value of A

Note that the values of B, W, A, Tb, Tw and Ta have been averaged over all tries with N leaf nodes, where each leaf node is a bucket containing exactly one record. For a fixed value of k, for each value of s all the queries from Q_s have been tested.

$$P = \begin{bmatrix} 1/3 & 1/2 & 1/6 \\ 1/4 & 1/3 & 5/12 \\ 1/5 & 3/10 & 1/2 \\ 2/5 & 1/3 & 4/15 \\ 1/2 & 1/5 & 3/10 \end{bmatrix}$$

UNIFORM QUERY DISTRIBUTION - METHOD A

	s	B	W	Tb	Tw	A	Ta	$N^{1-s/k}$
k=4 N=9	3	1.0	2.0	7.81	24.19	1.77	0.0	1.73
	2	1.33	4.0	2.72	7.78	3.06	7.78	3.0
	1	3.39	6.81	1.41	2.29	5.28	3.54	5.2
N=12	3	1.0	2.0	4.16	27.84	1.87	0.0	1.0
	2	1.87	4.0	2.08	13.68	3.48	13.68	3.46
	1	4.9	7.94	1.63	1.5	6.48	4.08	6.45
k=5 N=17	4	1.0	2.0	17.14	62.86	1.78	0.0	1.76
	3	1.07	4.0	3.19	31.16	3.17	31.16	3.11
	2	2.57	7.9	1.94	3.25	5.59	21.6	5.47
	1	6.85	12.54	1.30	1.46	9.79	3.42	9.64
N=24	4	1.0	2.0	8.3	71.7	1.90	0.0	1.89
	3	1.74	4.0	3.31	51.79	3.59	0.0	3.57
	2	4.31	8.0	2.4	10.71	6.78	10.71	6.73
	1	10.5	15.08	1.8	1.65	12.77	3.09	12.71

TABLE 6-2

Note that the last column in this table represents the conjectured lower bound.

NON-UNIFORM QUERY DISTRIBUTION - METHOD B

	s	B	W	Tb	Tw	A	Ta
k=4	3	1.0	2.0	7.74	24.26	1.54	0.0
N=9	2	1.32	4.0	2.96	8.52	2.51	8.52
	1	3.5	7.0	1.68	2.08	4.6	3.18
	3	1.0	2.0	4.10	27.9	1.75	0.0
N=12	2	1.9	4.0	2.12	14.08	3.19	14.08
	1	4.92	7.8	1.7	2.5	6.13	4.2
k=5	4	1.0	2.0	17.28	67.72	1.63	0.0
N=17	3	1.02	4.0	2.68	33.88	2.74	33.88
	2	4.18	8.0	2.74	13.4	6.27	13.4
	1	10.38	15.22	1.62	2.44	12.14	3.16
	4	1.0	2.0	8.26	71.74	1.80	0.0
N=24	3	1.74	4.0	4.78	53.44	3.33	0.0
	2	4.18	8.0	2.74	13.4	6.27	13.4
	1	10.38	15.22	1.62	2.44	12.14	3.16

TABLE 6-3

NON-UNIFORM QUERY DISTRIBUTION - METHOD C

	s	B	W	Tb	Tw	A	Ta
k=4	3	1.0	2.0	7.79	23.96	1.51	0.0
N=9	2	1.28	4.0	3.11	8.39	2.48	8.39
	1	3.49	6.98	1.71	2.01	4.53	3.03
	3	1.0	2.0	4.15	27.73	1.71	0.0
N=12	2	1.83	3.94	2.18	13.98	3.17	13.98
	1	4.78	7.62	1.66	2.42	6.08	4.07
k=5	4	1.0	2.0	17.65	67.46	1.61	0.0
N=17	3	1.01	3.97	2.73	33.42	2.69	33.42
	2	4.08	7.96	2.78	13.21	6.2	13.21
	1	10.27	15.03	1.68	2.37	12.11	3.09
	4	1.0	2.0	8.37	71.59	1.78	0.0
N=24	3	1.63	3.98	4.84	53.26	3.32	0.0
	2	4.11	8.0	2.79	12.97	6.19	12.97
	1	10.24	15.09	1.71	2.37	12.09	3.04

TABLE 6-4

NON-UNIFORM QUERY DISTRIBUTION - METHOD D

	s	B	W	Tb	Tw	A	Ta
k=4	3	1.0	2.0	7.83	23.91	1.48	0.0
N=9	2	1.26	3.97	3.14	8.29	2.41	8.29
	1	3.32	6.91	1.73	1.98	4.46	3.01
	3	1.0	1.97	4.19	27.57	1.69	0.0
N=12	2	1.81	3.87	2.23	13.92	3.07	13.92
	1	4.75	7.59	1.79	2.37	6.04	4.05
k=5	4	1.0	2.0	17.69	67.42	1.59	0.0
N=17	3	1.0	3.91	2.77	33.28	2.67	33.28
	2	4.05	7.86	2.81	13.18	6.12	13.18
	1	10.18	14.83	1.72	2.31	12.09	3.1
	4	1.0	2.0	8.41	71.42	1.74	0.0
N=24	3	1.6	3.98	4.87	53.25	3.21	0.0
	2	4.11	7.88	2.82	12.89	6.14	13.89
	1	10.09	14.97	1.74	2.33	12.05	3.02

TABLE 6-5

CHAPTER 7

CODING TECHNIQUES

This chapter discusses coding techniques useful in partial match retrieval. These compaction methods can be looked upon as superimposition and partitioning of hash functions. One important advantage of the method of superimposed codes seems to be that it is well suited to an implementation in special purpose digital hardware and partial match queries can be efficiently processed either in on-line or in batch mode.

When implemented in software, the method of superimposed coding is competitive with other alternatives; yet it has some disadvantages as well. The major portion of the method of superimposed coding can be viewed as a preselection algorithm that narrows down the region of records pertinent to a query. Hence the method of superimposed codes will achieve substantial increase in execution speed over the straight sequential search. The disadvantage of this method is that the binary coded file must be kept in addition to the actual file.

When the coding is done towards compressing and hence saving storage, only the compressed data base and the dictionary are essential for any transaction. Algorithms for performing partial match searches on a compressed file stored on a direct access device when the elements in a query are fragments chosen for compression are discussed by Alagar [3]. These methods can handle queries that use either whole words or word fragments operating on a compressed data base through the dictionary of fragments.

Of the three methods discussed in [3], it has been shown that the method which exploits the interdependence of fragments as well as the relevance of fragments to the records in the file has the maximum design cost and least retrieval cost.

A partitioned hash function obtains a code for a record and the code is interpreted as a bucket address. The hash code of a record can be obtained as a concatenation of the hash codes of the values in the various fields of the records. The advantage of this method is that concatenation of codes has more information content than the superimposed code of a record and the coded file is no longer required.

In the method of superimposed coding the coded file must be searched first to find the records which might satisfy a query. This search can be either sequential or hierarchical. We have found that a trie constructed for the coded data base is more efficient than sequential search (see section 7.3 for details). Similarly in the case of partitioned hash functions the hash codes of the records are used to construct tries. When the query probabilities are assumed, they are used in computing the codes of records.

The following sections will discuss these methods and possible refinements.

7.1 SUPERIMPOSED CODING

So far in our discussions we have considered binary attributed records, each record consisting of the same number of binary attributes. For any given file F consisting of N records $R_1 \ldots B_N$ the number of keys per record need not be the same and the values of the keys need not be binary. In such a situation a binary valued record is obtainable through a suitable mapping. Although several mappings are suggestive, we develop a "random" mapping through a module called "Record Mask Generator".

During the design phase of the system every record in the file is processed by the Record Mask Generator and coded so that all coded records are of the same length. See diagram in figure 7.1.1.

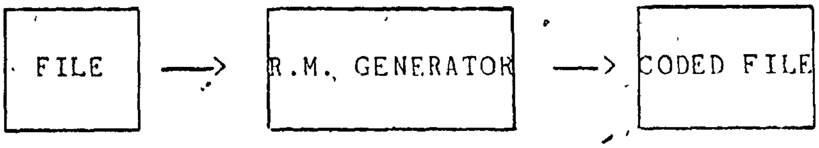


Figure 7.1.1

An overall block diagram for a retrieval system using superimposed codes is shown below in figure 7.1.2.

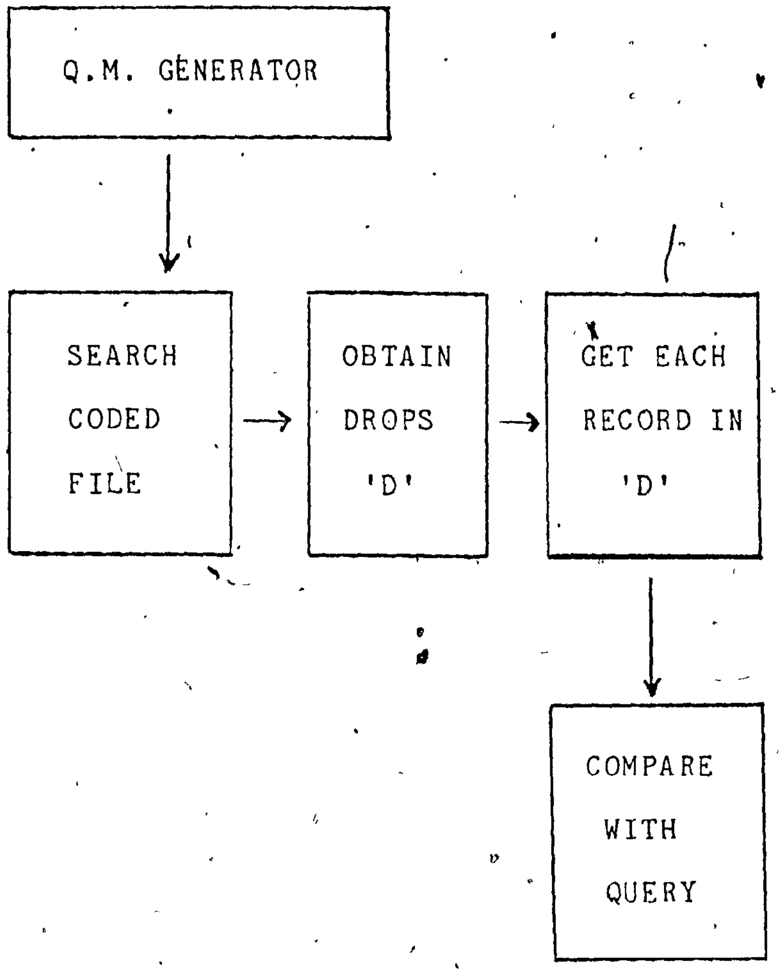


Figure 7.1.2

Let $K_{i_1}, K_{i_2}, \dots K_{i_j}$ be the keys in record R_i.

Let H be a mapping such that $H: K_{i_t} => (s_1 \dots s_b)$ where each $s_i = 0$ or 1.

The vector $(s_1, s_2, \dots s_b)$ is the binary coded word corresponding to K_{i_t} and b is the length of the code.

We define the superimposed code word (SCW) of R_i as

$$C_i = \underset{t}{\vee} H(K_{i_t}) = (c_1, c_2, \dots c_b), \text{ where } c_j = 0 \text{ or } 1,\ j=1,..,b. \quad [7.1]$$

The set $S = \{C_i | C_i = SCW(R_i),\ i=1,N\}$ is the coded file.

Although the superimposed code method will work no matter how H is defined, the performance achieved by an implementation of this technique will critically depend on the choice of H. Moreover the length b of each binary coded word as well as the weight w (defined to be the number of 1's in each code) will affect the performance.

In the context of partial match retrieval using superimposed codes it seems advantageous to choose H satisfying the following conditions:

1) The computation of $H(K_{i_t})$ must be accomplished rapidly.
2) The number of different (or substantially different) keys having the same binary coded word should be minimized.

Clearly there are many candidates for H and any one function cannot be expected to be optimal for all systems.

Moreover the value of b critically affects the size of the ancillary file of superimposed code words, that is the cost of extra memory is directly proportional to b.

However only for large values of b the initial search of the coded file for a given query would produce fewer record identifiers. This implies that the number of different accesses to external storage to retrieve records from the original file F is inversely proportional to b.

There is one more factor that affects the performance, namely the number of bits in a binary coded word that have the value 1. We call this the weight of the code. This is due to the fact that a partial match query is transformed by the Query Mask Generator into a b-bit binary value $(q_1, \ldots q_b)$, $q_j=0$ or 1. The Query Mask Generator is identical to the Record Mask Generator (see details later on). The relationship between the query mask and the file S can be stated as follows:

PROPERTY D: If a superimposed code word $S_i \in S$ contains 0 in any bit position where the query mask computed from a query q contains a 1, then the corresponding record R_i is not pertinent to the query q. The truth of this statement will be obvious when we describe the mask generation process.

The above property is used in searching the coded file to obtain record identifiers (we call these "drops"), with the purpose of narrowing down the number of possibly pertinent records in F. The weight w of a binary coded word of length b influences the number of "false drops", that is those record identifiers selected from S that do not satisfy the query. Our experiments reveal that when w is roughly 40-50% of b the smallest number of false drops is obtained.

We will fix b and w and generate the binary coded word for a key using the algorithm below:

```
ALGORITHM A
Given a key value K, find the corresponding binary coded word
(bcw) S of length b and weight w.

Step 1. [ initialization ].
        Set $S_j$<- 0 for all j=1,..,b
        Set  w'<- 0.
Step 2. [ use the key K to obtain a numeric equivalent R]
        Let NUM be a function which returns a numeric value for
        the key K.
        Set  R <- NUM(K).
Step 3. [ use R to initialize random number generator]
        Set  x <- RANDOM(R); ($0 \leq x \leq 1$, x is first random number).
Step 4. [ get a bit location]
        Set  j <- $\lfloor xb \rfloor$ + 1; (1<j<b).
        Set x <- NEXT(x).
Step 5. [ is j-th bit 1?]
        If $S_j$= 1 go to Step 4.
Step 6. [set j-th bit to 1]
        Set $S_j$<- 1, w'<- w'+1.
Step 7. [ is w'=w ?]
        If w'< w go to step 3
        else RETURN (S).
END of Algorithm A. (The vector S is the bcw of the key K).
```

For each record using algorithm A and the relation [7.1] the superimposed code word is generated. For any given query that specifies s attributes we use the same algorithm to generate the binary code words and hence the query mask; the unspecified bits are just ignored.

For a given query the search strategy is described below:

```
ALGORITHM B
Step 1. [generate query mask];
        Use algorithm A and the relation [7.1] to produce the
        query mask $q=(q_1 \ldots q_b)$ where $q_i=0$ or 1, $i=1,b$.
Step 2. [find drops];
        Set $D \leftarrow \{S_i \mid S_i \in S$ and for $S_i=(s_1 \ldots s_b)$ $q_i=1 \Rightarrow$
        $s_i=1$, $i=1,..,b\}$.
Step 3. [retrieval of records];
        If $D=(S_{i_1},\ldots S_{i_j})$ where $i_1$ to $i_j$ are record
        identifiers, the records $D'=(R_{i_1},..,R_{i_j})$ are retrieved;
Step 4. [pertinent records];
        Compare the given query with each record in D' and
        output those that match the specified attributes.
END of Algorithm B.
```

There are several remarks we wish to make about this algorithm:

If the size of the coded file is small enough to fit in core the cost of step 2 is negligible compared to ~~the~~ cost of step 3. Moreover the internal processing of step 2 can be speeded up by constructing a trie of coded records. However the cost of step 3 would remain the same.

If sequential search is used in step 2 then all the time all the coded records must be compared with the query mask. Even if the coded file is sorted the relative ordering of the records cannot be used to effectively isolate the drops. An alternative scheme seems to be to build a trie for the coded file. The trie construction is similar to the methods that we described earlier. However, in searching the trie, a 0 in a bit position in the query mask must be treated as a *. This is due to property D and consequently seems to increase the search time on the trie that can be constructed for the coded data. The weight w and the distribution of 1's in the query mask influence the search time also; yet the search is faster than a pure sequential search. The drops produced in a trie search would still have to be examined sequentially to produce the set D. In order to compare the sequential and trie based search, we introduce bit complexity, that is the number of bit comparisons necessary to be done on the file S to produce the set D.

In our analysis below we consider the set Σ of all superimposed code words of length b and weight w generated by algorithm A. $|\Sigma|$ is the cardinality of Σ. Clearly this quantity is $\binom{b}{w}$. For simplicity of the analysis we also assume that all records have the same number of fields; the analysis for the case of variable number of fields is difficult even under very simplistic assumptions on the distributin of the superimposed code words.

ANALYSIS

Let $R=(K_1,K_2,\ldots,K_t)$ be a record having t fields. In algorithm A each K_i is mapped onto a binary vector of length b and weight w and the superimposed code C is obtained by taking logical OR on these vectors. If S1 denotes the set of all binary coded words generated independently on the t fields, then

$$S1 = [\binom{b}{w}]^t \, . \qquad [7.2]$$

Since the binary code words are generated using a uniform random number generator, we may assume that all codes of length b and weight w are equally likely to occur independently accross the fields. Although in practice the data values belonging to the various fields are less likely to be random and hence the binary code words are not necessarily uniform, the analysis based on the assumption of uniformity gives an approximate validity of the actual results obtained. Moreover we pin our faith on the random number generator which we hope will achieve random codes from non-random data.

The main purpose of the analysis is to determine:

1) The probability $p(b,w,t;r)$ that $r\leqslant w$ specified bits of a superimposed code generated from a record R will have the value 1.
2) The probability $q(b,w,t,s)$ that a record R is a false drop for a query Q with s specified fields. In other

words this is the probability that R is not pertinent to Q, yet it passes the criterion stated in property D.

3) The average weight (the number of 1's) w in a query mask.

4) The relationship between b and w and, in particular, for a fixed b the value of w that minimizes q(b,w,t,s).

We first determine p(b,w,t,r). Let S2 denote the set of all binary code words that have at least one 0 among the chosen r bit positions. Clearly

$$p(b,w,t,r) = 1 - |S2| / |S1| \qquad [7.3]$$

Using the principle of inclusion and exclusion and simple combinatorial arguments, we can show that

$$|S2| = \sum_{1}^{r} (-1)^{i-1} \binom{r}{i} \left[\binom{b-i}{w}\right]^{t} \qquad [7.4]$$

Substituting [7.2] and [7.4] in [7.3] we get

$$p(b,w,t,r) = 1 - \left[\frac{\sum_{1}^{r} (-1)^{i-1} \binom{r}{i} \left[\binom{b-i}{w}\right]^{t}}{\left[\binom{b}{w}\right]^{t}} \right]$$

$$= 1 - \sum_{1}^{r} (-1)^{i-1} \binom{r}{i} [\alpha(b,w,i)]^{t} \qquad [7.5]$$

where $\alpha(b,w,i) = \dfrac{\binom{b-i}{w}}{\binom{b}{w}}$

We can rewrite [7.3] as

$$p(b,w,t,r) = \sum_{0}^{r} (-1)^{i} \binom{r}{i} [\alpha(b,w,i)]^{t} \qquad [7.6]$$

For the particular case r=1 we have

$$p(b,w,t,1) = 1 - [\alpha(b,w,1)]^{t}$$

$$= 1 - (1-w/b)^{t} \qquad [7.7]$$

This is the probability that in a binary coded word generated by the algorithm A there is a 1 in any prescribed position. We can write $\alpha(b,w,i)$ as

$$\alpha(b,w,i) = \prod_{0}^{w-1} \left(\frac{b-i-j}{b-j} \right).$$

So we have

$$\alpha(b,w,0) = 1$$

and for i>1 this quantity is equal to

$$\alpha(b,w,i) = \prod_{0}^{i-1} \left(\frac{b-w-j}{b-j} \right)$$

$$= (1-w/b)^{i} \prod_{0}^{i-1} \left[\frac{b(b-w-j)}{(b-w)(b-j)} \right]$$

$$= (1-w/b)^{i} \prod_{0}^{i-1} \left[1 - \frac{jw}{(b-w)(b-j)} \right] \qquad [7.8]$$

Let $\beta = w/(b-w)$. If $w < b/2$ then $\beta < 1$. So we have

$$\alpha(b,w,i) = \frac{w^i}{b^i \beta^i} \prod_0^{i-1} \left[1 - \frac{j\beta}{b-j} \right]$$

$$= \frac{w^i}{b^i \beta^i} \sum_0^{i-1} (-1)^j d_j \beta^j \qquad [7.9]$$

where

$$d_0 = 1$$

$$d_1 = \sum_1^{i-1} \frac{j}{b-j} = \sum_1^{i-1} \left(\frac{b}{b-j} - 1 \right)$$

$$= b \left(\sum_1^{i-1} \frac{1}{b-j} \right) - i + 1$$

$$= b \left[H_b - H_{b-i} \right] - i + 1 \qquad [7.10]$$

where $H_n = 1+1/2+1/3+....+1/n$.

So

$$\alpha(b,w,i) = \frac{w^i}{b^i \beta^i} (1 - d_1\beta + d_2\beta^2 - ....)$$

$$= (1-w/b)^i (1 - d_1\beta + d_2\beta^2 - ....)$$

Since β remains smaller than 1 and the coefficients $d_1 ... d_{i-1}$ remain bounded, we can ignore all powers of β and as a first approximation we take

$$\alpha(b,w,i) = (1-w/b)^i \qquad [7.11]$$

Such a simplistic approximation seems necessary for a tractable analysis. However the actual expression for $\alpha(b,w,i)$ will be quite useful for computing the probability more exactly. Now subsitute [7.11] in [7.6] to get

$$p(b,w,t,r) = \sum_0^r (-1)^i \binom{r}{i} \left[(1-w/b)^t \right]^i$$

And using the binomial theorem we obtain

$$p(b,w,t,r) = [\ 1-(1-w/b)^t\ ]^r \qquad [7.12]$$

This gives a first approximation to the probability that any prespecified set of r out of w<b bits in a superimposed code word will have value 1.

Next we find the expected weight of a binary code generated by algorithm A on a query with s fields. In [7.12] if we set r=1 and t=s we get

$$p(b,w,1,s) = 1 - (1-w/b)^s .$$

This is the probability that any prespecified bit position is 1 in a binary coded word generated from a query with s specified fields. Since all bits will have the same probability of being a 1, the expected value $\overline{w}$ of the weight (number of 1's) in a query mask is given by

$$\begin{aligned} \overline{w} &= b\ p(b,w,1,s) \\ &= b\ [\ 1 - (1-w/b)^s\ ] \end{aligned} \qquad [7.13]$$

Now we want to find the probability q(b,w,t,s) that a b-bit record mask generated from a record with t fields is a false drop for a query mask when s fields are specified in the query. This is the same as $p(b,w,t,\overline{w})$ since $\overline{w}$ is the expected weight of the query mask. Therefore

$$q(b,w,t,s) = [\ 1 - (1-w/b)^t\ ]^{\overline{w}} \qquad [7.14]$$

where $\overline{w}$ is given in [7.13].

In the discussion below we will simply denote q(b,w,t,s) by q and our aim is to find a relation between w, b and t which minimizes q.

Taking logarithms on both sides in [7.14] we have

$$\ln q = \overline{w} \ln [\ 1 - (1-w/b)^{t}\ ]$$
$$= b\ [\ 1 - (1-w/b)^{s}\ ] \ln [\ 1 - (1-w/b)^{t}\ ]$$

Let $x=(1-w/b)^{t}$, then

$$\ln q = b\ [\ 1 - \exp(s/t \ln x)] \ln (1-x) \qquad [7.15]$$

For fixed b, t and s we can differentiate [7.15] with respect to w and establish that there exists a minimum value for the function ln q. Therefore there exists an optimum value of w that minimizes the false drop probability. Once again we use an approximation since 0 < s/t < 1 for a partial match query.

For the case s=t we find the optimum value of w as given by

$$w = b\ [\ 1 - \exp(1/t \ln (1-1/e))]$$
$$= b\ [\ 1 - (1-1/e)^{1/t}\ ] \qquad [7.16]$$

For 0<s<t a closed form expression for w is rather difficult to obtain. By taking a first approximation to the expression in [7.15] we get

$$\ln q = -bs/t \ln x \ln (1-x) \qquad [7.17]$$

Formally differentiating this we find that the minimum value occurs at x=1/2. Therefore we conclude that the optimal w is one which causes 0 and 1 to occur with equal probability in the code words and this is independent of s.

Substituting x=1/2 in [7.17] we have

$$\ln q \simeq -bs/t\,(\ln 2)^2 \qquad [7.18]$$

The optimal value of w is given by

$$\begin{aligned} w &\simeq b\,[\,1 - \exp(-(\ln 2)/t)] \\ &= b\,[\,1 - 2^{-1/t}\,] \end{aligned} \qquad [7.19]$$

The expression in [7.19] is a crude approximation. A much better value for optimum w can be obtained from [7.15] as follows.

Differentiating [7.15] with respect to w and equating the result to zero we get

$$tx\,[\,1 - (1-w/b)^s\,] + s(1-x)(1-w/b)^s \ln(1-x) = 0 \qquad [7.20]$$

Observing that $0 < w/b < 1$, we expand the left hand side in power series and ignore terms of $(w/b)^j$, $j \geq 2$. Thus we get

$$(1 - tw/b) + (2tw/b - 3/2)(1 - sw/b) = 0$$

Let $\lambda = w/b$. Then

$$2ts\lambda^2 - \lambda(t + 3s/2) + 1/2 = 0$$

This has real roots given by

$$\lambda_s = \frac{(t + 3s/2) \pm (t^2 - ts + 9s^2/4)^{1/2}}{4ts} \qquad [7.21]$$

For all values of 1<s<t there are two positive roots of which the one that makes the second derivative of [7.15] positive will minimize q(b,w,t,s). Let us call this value λ (=w/b). Let $y=(1-\lambda)^t$. Substituting these values in [7.15] we have

$$\ln q = b\,(1 - y^{s/t})\ln(1-y) \qquad [7.22]$$

This result is a much better approximation than [7.18] and in particular note that it is dependent on s, while [7.18] is not. It is more realistic to expect the weight to vary and depend on s; yet in a practical information retrieval system it is desirable to determine w which is good on the average. Below we discuss this choice in some detail.

This analysis suggests that design parameters be chosen as explained below:

1) the average number of false drops $\overline{D}_f$ tolerated in the system must be decided beforehand, that is $\overline{D}_f/N=q$. Substituting this quantity in [7.22] we can find b as a function of s.

2) the value of λ as given in [7.21] is a function of s, i.e. the number of specified fields in a query. In the overall design of the system we must have a value

independent of s, unless the system is restricted to allow only queries in which any s fields can be specified. In many real life situations this may be desirable. However, for the general situation, the ratio can be computed as

$$\lambda = \sum_{1}^{t-1} (\sum P_{i_1} \ldots P_{i_s}) \lambda_s \qquad [7.23]$$

where λ_s is given in [7.21], the inner summation is over all s combinations of t fields and the P_i 's are the probabilities with which the fields are specified in the queries. Here we recall that the probability P_i that the i-th field is specified in a query is independent of whether other fields are specified or not in the query. If the queries are uniform, then

$$P_{i_1} P_{i_2} \ldots P_{i_s} = \frac{|D_{i_1}||D_{i_2}| \ldots |D_{i_s}|}{\prod_{1 \le i \le k} (D_i + 1)} \qquad [7.24]$$

where $|D_j|$ is the cardinality of the j-th field domain.

3) Thus $w = \lambda b$ can be computed.

Although this method is simple, the analysis beyond the second approximation seems mathematically complex. If more accuracy is desired, known methods in numerical analysis may be employed to obtain the optimal value of λ from [7.15]. Our experiments seem to indicate that neither the assumption of uniformity for code words, nor the second approximation cause serious errors. See the table of results [7-1] for the performance of this method. Finally we remark that these results remain independent of the actual size of the data base; it is sufficient to presuppose the false drop probability tolerated in the system and the query probabilities.

7.2 SUPERIMPOSED CODING AND TRIE INDEXING

We can improve the superimposed coding method if sequential search of the coded data is replaced by a trie based method. Assuming that a fixed amount of storage is available for holding the trie in memory, we construct a binary trie for the coded data base. We use the methods of chapter 6 for such a construction. However we have to take into account the fact that a 0 in a query mask has to be interpreted as a * because of property D (see section 7.1). Hence every query mask should be treated as an inclusive query. Thus during trie construction we would minimize the number of left branches.

Since trie construction and search have been explained previously, it is enough to show here the superiority of the trie approach over the sequential search of the coded data base. Intuitively it is clear that a trie based search requires sequential search of a subset of the coded data base. In order to establish its efficiency, we introduce "bit complexity" which is defined to be the number of bit comparisons in a search.

If N denotes the number of distinct code words, L denotes the length in bits of a record, N equals the number of records in a bucket and D is the number of drops in a séquential search of the entire coded data base, then the bit complexity for sequential search of a query is

$$B_s = Nb + L \sum_{j \in D} N_j \qquad [7.25]$$

For the same query searched on the trie this would be

$$B_t = I + (D + E)\ b + L \sum_{j \in D} N_j \qquad [7.26]$$

The term I represents the number of internal bit comparisons and the middle term reflects the possibility that there are more drops resulting from a trie search; this is true in general because not all the bits in the query mask wil be tested along a path from the root of the trie to a leaf node. The first term is the number of bit comparisons on the trie. We remark that the above expression can be computed for each query and averaged to obtain the expected value of the bit complexities. See table [7-1] for our experimental results. The results indicate that the value of B_t remains smaller than B_s; in fact B_t is quite small compared to B_s when (near) optimal w has been chosen. From [7.25] and [7.26] we have

$$B_t < B_s \quad \text{if} \quad I + (D + E)\ b < Nb \qquad [7.27]$$

The code words generated by the Record Mask Generator are random and thus the bits that get selected to be tested on the trie are also randomly chosen (although restricted to maximizing the unbalance at each node at each level in the trie). In [22] it is shown that in a random trie the average number of bit inspections is close to $\log_2 N$. Thus approximately

$$I = \log_2 N$$

Substituting in [7.27] we have

$$B_t < B_s \quad \text{if} \quad \log_2 N + (|D| + E)\, b < Nb$$

Asssuming that $N = 2^b - 2^w$, (note that each code has weight at least w), we have approximately

$$B_t < B_s \quad \text{if} \quad |D| + E < N - 1 + \frac{1}{b\, 2^{b-w}}$$

i.e. the number of trie drops is less than (N-1).

However this is true for almost all queries as can be seen from the statements of chapter 6. A much stronger statement based on this result is that on the average the number of trie drops is $N^{1-\lambda}$, where λ is given in [7.23]. Although the results obtained prompt this statement, exact analysis is required in proving or disproving this claim. Hence trie based search is superior to sequential search of the coded data base; however we require storage for the trie in addition to the storage required for the coded data base. When the amount of space is limited, the trie constructed for the coded file can be pruned to the desired level without affecting the performance.

7.3 CONCATENATION OF CODES

The superimposed coding method requires maintaining a separate file for the code words and this seems to be a disadvantage. However searches are much faster than with associative search and trie based search is preferable at an extra overhead of storage. A definite disadvantage of the superimposed coding method is that updates require changing the coded file and the master file as well as the trie index if there is one. To a large extent these would be eliminated if we concatenate instead of superimpose the individual codes. We outline two methods below.

METHOD 1

This method is due to Aho and Ullman [2]. The basic idea here is to consider all fields in a record and hash the full record onto a bucket wherein it can be stored. By suitable choice of the hash function, the number of buckets to be examined in a search for a partially specified query can be minimized.

Assume that each record has L bits and the file contains $M<<2^L$ records. A query specifies some subset of the L bits. Let there be 2^B buckets, each bucket with the same capacity, i.e. the number of records that can be stored in a bucket is a constant. Let there be k fields in each record and P_i be the probability that the i-th field is specified in a query. Denote by $Q=(i_1,\ldots i_s)$ a query where $1 \leqslant i_j \leqslant k$ indicates specified fields. The essential feature of this method is to estimate the number of bits to be extracted from each field so that their concatenation produces a bucket address. Certainly the number of bits $b_1,\ldots,b_k$ to be extracted from each field are functions of the P_i's. The b_i's are found in [2] using a simple maxima, minima principle.

Observe that the average number of buckets to be examined to answer a query is

$$A = \sum_{Q} \left(\prod_{j \in Q} P_j \right) \left(\prod_{i \notin Q} (1-P_i)\, 2^{b_i} \right) \qquad [7.28]$$

where Q may range over all subsets of (1,2,..,k). This is easy to verify if we argue that for each specified field all the bits are known and for each unspecified field all bits are not known; in the latter case 2^{b_i} buckets are to be searched. Since we must have

$$B = b_1 + b_2 + \ldots + b_k \qquad [7.29]$$

we want to minimize A subject to [7.29]. It is shown in [2] that the optimal number of bits to allocate to field i is

$$b_i = B/k + \log \left(\frac{P_i}{1-P_i} \right) - 1/k \sum_{1}^{k} \log \left(\frac{P_j}{1-P_j} \right) \qquad [7.30]$$

Since these values are real, one has to obtain optimal integer solutions. These are also given in [2]. For a sample data, this method was applied and the results are shown in table [7-2].

The primary strength of this method is the ease with which new records can be inserted into or deleted from the file; the disadvantage is the relatively higher number of accesses needed to answer some queries. Moreover this method gives the b_i's but does not suggest as to which b_i bits are to be selected from the i-th field. The following example illustrates this method.

EXAMPLE. Suppose we have records with 3 fields

LAST_NAME PHONE_NUMBER ADDRESS

and we wish to store the records in $2^{10}=1024$ buckets. Suppose 60% of the queries specify LAST_NAME, 10% specify PHONE_NUMBER and 30% specify ADDRESS. We have $P_1=0.6$, $P_2=0.1$, $P_3=0.3$, $B=10$ and $k=3$. The b_i's can be computed using [7.30]:

$$b_1=4.41, \quad b_2=2.12, \quad b_3=3.47$$

The optimal integer solutions are obtained by rounding up the maximum b_i, rounding down the minimum b_i, and recursively for the rest of them, and if there is one b_i left at the end it will be assigned the remaining bits in B. Thus for the values given above, the optimal integer solutions are

$$b_1=5, \quad b_2=2, \quad b_3=3$$

We should mention that in order to code each field i of a record into b_i bits, we have used a hash function which computes the value

$$z_i = \text{(numeric value of field i) mod. } 2^{b_i}$$

which is a number that occupies b_i bits.

METHOD 2.

It is relatively easy to write down the expression for A, the average number of buckets examined to answer a query, for the case when the bucket address is computed directly from the bit values; however when the buckets are stored in the leaf nodes of a trie, it seems difficult to explicitly write down the bucket address in terms of the bit patterns of the search paths. In principle therefore it remains an open problem to show the optimality of tries constructed by the methods in chapter 6.

We consider a concatenation scheme and build a trie for concatenated codes and compare the results of Aho and Ullman. Whereas in [2] the number of bits to be extracted are known but the actual bits are not known, in our scheme the situation is reversed, i.e. the bits to be extracted are determined (say by methods in chapter 6); however we are unable to show theoretically the result that the number of bits thus selected for testing is optimal. Simulation experiments seem to indicate optimality.

For our method we need the probabilities $P_1 \ldots P_k$ and d_i, i=1,..k that are the cardinalitites of the domains. Set $B=\sum_1^k \log_2 d_i$. Clearly a record needs at most B bits for the code. A value from the i-th field is mapped onto $b_i = \log_2 d_i$ bits. We attach a weight w_i to the code generated from the i-th field. As remarked earlier w_i is the number of 1's included in the b_i bits. The weights $w_1 \ldots w_k$ are chosen proportional to $P_1 \ldots P_k$ so that a field of higher probability will have a larger weight than others. The Record Mask Generator is used to generate binary codes of the fields and these are concatenated. Using method A of chapter 6, the trie is built and pruned to have a prescribed number of buckets. Clearly the address of a bucket is a function of the search path and the bits are selected by the trie construction algorithm.

This method was applied to the same sample data used for method 1. The results are given in table [7-3]. These results indicate that the trie based method is superior to partitioned hashing in computation of the bucket address. Although we have achieved the same kind of results for various data, we are again unable to show the superiority of the trie based method in a theoretical way. The obvious disadvantages of this method are

1) more overhead than in the partitioned hashing method;
2) insertions and deletions are more difficult to handle without a total collapsing and rebuilding of the trie (for preservation of the optimal search time).

Thus we conclude that there can be no single method that can be applicable uniformly to all situations; however we have shown in this chapter that trie based methods outperform many of the other competitive methods as long as the data base remains static over a period of time, or insertions and deletions do not adversely affect the unbalance and shape, of the constructed tries.

NOTATION FOR TABLES 7-1 TO 7-3

N = number of records

Nb = number of buckets

k = number of attributes in each record

d_i = cardinality of i-th attribute domain

s = number of specified attributes in a query

B = total length of a coded record

b_i = length of i-th field code

w = weight of a coded word

w_i = weight of i-th field code

I = number of internal bit comparisons (on the trie)

TD = number of codes in the trie drops

ES = number of codes eliminated in sequential search of the coded file

M = number of records that are pertinent to a query

Min.TC/Max.TC/Av.TC = maximum/minimum/average bit complexity for trie search

Min.SC/Max.SC/Av.SC = maximum/minimum/average bit complexity for sequential search of the entire coded file

Av.I = average number of internal bit comparisons

Av.TD = average number of codes in the trie drops.

The following information applies to all the tables.

N = 100, k = 4

The same file and query set have been used in all the experiments.

For each value of s the query set consists of 100 queries, for each of which there exists at least one pertinent record in the file.

The values for I, TD, ES and M represent the sum of the results for the 100 queries tested in each case.

The cardinalities of the attribute domains are:

$$d_1 = 57, \ d_2 = 14, \ d_3 = 31, \ d_4 = 38$$

The probabilities of specification for each attribute are:

$$P_1 = .35, \ P_2 = .20, \ P_3 = .50, \ P_4 = .75$$

Each record in the master file is of length 40 bits.

SUPERIMPOSED CODING

B = 30

	w = 6 Nb= 100			w = 9 Nb= 100		
S	3	2	1	3	2	1
I	2657	3683	5352	2344	3082	5006
TD	1578	2518	4408	1054	1763	3632
ES.	1474	2346	3482	934	1500	2398
M	100	115	544	100	115	544
Min.TC	270	342	654	293	316	911
Min.SC	840	1080	2100	940	1140	1520
Max.TC	1042	1874	7252	1244	2466	6606
Max.SC	3080	3260	7720	2780	3680	7260
Av.TC	536.31	856.89	3135.49	468.98	926.76	3553.23
Av.SC	2070.60	2147.80	4178.40	1898.00	2212.20	4487.60

TABLE 7-1 A

SUPERIMPOSED CODING

B = 30

	w = 12 Nb= 98			w = 15 Nb= 71		
S	3	2	1	3	2	1
I	2832	3661	5302	2836	3316	4479
TD	868	1586	3439	524	963	2439
ES	525	924	1491	67	146	358
M	100	115	544	100	115	544
Min.TC	736	835	3030	2430	3170	7678
Min.SC	420	1000	3780	2940	3420	8300
Max.TC	2342	4260	8210	6303	8184	12021
Max.SC	3960	4580	8800	6520	9120	12360
Av.TC	1174.62	2016.77	5413.19	3932.19	5381.29	9757.47
Av.SC	2013.60	2689.80	6053.00	4464.00	5846.20	10368.80

TABLE 7-1 B

SUPERIMPOSED CODING

B = 30						
	w = 18 Nb= 39			w = 21 Nb= 16		
S	3	2	1	3	2	1
I	2085	2295	2828	1500	1500	1500
TD	293	522	1298	164	245	589
ES	3	7	18	0	0	0
M	100	115	544	100	115	544
Min.TC	10350	10350	12834	19950	19950	20727
Min.SC	11520	11520	14020	21420	21420	22440
Max.TC	14238	16038	18251	20946	21207	22271
Max.SC	14860	17140	19200	22580	22920	24280
Av.TC	11447.92	12455.16	15559.40	20157.88	20381.75	21389.91
Av.SC	12503.20	13501.40	16722.80	21680.80	21970.60	23245.80

TABLE 7-1 C

CONCATENATION CODING - METHOD 1

B = 8

Nb= 66

S	3	2	1
I	1340	2571	3958
TD	615	1793	3345
ES	79	145	151
M	100	115	544
Min.TC	248	250	250
Min.SC	540	540	540
Max.TC	7711	13894	24065
Max.SC	5660	9260	15840
Av.TC	1759.34	2566.43	5117.34
Av.SC	2712.93	2833.10	4099.72
av.TD	4.10	7.01	14.67
Av.I	19.52	18.79	23.00

TABLE 7-2

$b_1 = 1$, $b_2 = 0$, $b_3 = 1$, $b_4 = 6$

CONCATENATION CODING - METHOD 2

B = 21 $\bar{w}$ = 11

	w = 6 Nb= 96			w = 11 Nb= 98		
S	3	2	1	3	2	1
I	1571	2837	5003	1424	2902	5233
TD	443	1457	3620	465	1540	3751
ES	290	995	1701	349	1222	2183
M	100	115	544	100	115	544
Min.TC	244	262	406	246	246	253
Min.SC	440	440	780	540	540	440
Max.TC	1636	4178	13414	1120	3690	13349
Max.SC	3140	5260	13200	2720	5440	13200
Av.TC	907.09	1204.66	2668.53	759.84	792.72	2018.89
Av.SC	3411.58	3133.72	3571.43	2766.12	2112.30	2627.87
av.TD	6.00	14.21	29.50	6.18	12.25	26.92
Av.I	30.28	35.17	43.41	24.65	25.62	35.90

TABLE 7-3 A

CONCATENATION CODING - METHOD 2

B= 21 $\bar{w}$ = 11

	w = 15
	Nb= 95		

S	3	2	1
I	1232	2646	4969
TD	358	1289	8332
ES	226	803	1368
M	100	115	544
Min.TC	252	253	823
Min.SC	640	540	1020
Max.TC	988	4081	12766
Max.SC	2820	5360	12020
Av.TC	843.25	1132.40	3114.98
Av.SC	3041.32	3010.49	4046.16
av.TD	6.51	15.36	31.06
Av.I	27.22	36.06	46.94

TABLE 7-3 B

CHAPTER 8

CONCLUSION

This thesis has been intended to show the suitability of tries for partial match retrieval. We have restricted ourselves to partial match queries since exact match queries are well understood and the more general intersection queries demand searching the whole file almost always.

Four methods of constructing tries have been proposed. Superimposed coding with sequential search, superimposed coding with trie search, concatenation of codes with direct access and concatenation of codes with trie indexing have also been investigated. Previous research in partial match retrieval suggested trie as a suitable partial match file design and posed the question of constructing tries for non-uniform partial match queries as an open problem.

The investigation in this thesis has shown convincingly that tries are well suited for this problem in the sense that average retrieval cost per query is small compared to other methods.

We have measured the cost of retrieval in terms of the most "expensive" operation performed which dominates the overall cost. In the case of large files stored on secondary storage devices this has been essentially the number of accesses (read commands). For small files kept entirely in core the cost is measured in terms of bit comparisons. Other factors which are installation dependent such as latency time and transfer rate have not been taken into account as they would be unavoidable and constant in a given environment. The cost of updating a file has not been discussed at length.

The main contribution of this thesis has been to exhibit explicitly how the query distribution and the structure of the file may be combined to build a trie whose performance, we believe, is close to optimal.

In the absence of query distributions the algebraic design of a trie essentially depends upon the structure of the file which is reflected in the statistics of the bit patterns; this characteristic influences the selection of bits to be tested at the various levels in the trie.

Previous research has shown that skewed tries have better performance than balanced tries. We have postulated the measure of skewness in more than one way. When the queries are assumed equally likely (and hence do not influence the construction of the trie), we have found that the best measure of skewness is the absolute value of the difference between the number of 0's and the number of 1's (normalized when divided by the number of records). This is because, when bits are chosen to optimize (maximize) this measure, the average number of buckets examined to answer a simple query is minimum. In fact our results show that this average is very close to the optimum conjectured by Rivest [25].

In most real-life situations, however, queries are not equally likely. Different fields in a query will have different probabilities of being specified. Thus the choice of bits for testing in the trie has to be made in accordance with the distribution of query patterns. Constructing the most skewed trie may not be a good approach since the bits that cause most unbalance may be those that are highly probable to be unspecified in queries that are more frequent than others. Consequently the average cost of searching such a skewed trie will be less than optimal. Hence those bits with high probability of being unspecified in queries must be tested lower in the trie. Method B is based on this idea. Here the bits to be extracted are determined entirely from the query distribution and the file structure influences only the resulting collapsed trie.

This method has a definite advantage in its conceptual simplicity and, above all, in its ability to handle dynamically changing files, since the amount of restructuring of the trie is minimal and straightforward.

For a given file, the average cost of answering a non-uniform query when searching the trie built by method B is lower than the cost incurred by method A. This result shows that tries constructed by method A are near optimal only when the query patterns are assumed to be uniform, whereas method B constructs near optimal tries based only on the query distributions.

Method C combines the strategies of methods A and B to construct a trie correllating the structure of the file with the query space. It is assured in this method that the bits to be tested on the trie will have both least probability of unspecification and highest probability of isolating the pertinent records as we progress in a top-down manner.

For a given query distribution and a file the empirical results obtained show that the cost in method C is much lower than the cost per query in method B. If we add to this the fact that for uniform queries in effect this method reduces to method A, we are lead to believe that it is indeed a good approach.

A final attempt to improve trie performance has been made in method D. We have sought to minimize the number of buckets to be examined by suitably choosing the bit positions to be tested at each level in the trie. This incorporates two highly desirable features of a trie structure: "maximum unbalance and minimum probability of unspecification" at each level in the trie. In effect we have tried to minimize the worst case behaviour, since we assume at each level the "worst" type of query - that which has to examine all the records below that level in a subtrie. Here again we have come across better results than those obtained previously.

Naturally, as the criteria for choosing the bits to be tested become more complex, the tries constructed are more costly to update, though the cost of searching diminishes.

Of the four methods described in this thesis, only method B, which is quite independent of the file structure guarantees a trie which is relatively easy to maintain. This is a classical situation of a trade-off, i.e. method B is recommended when approximately optimal performance is tolerated and frequent updates are essential.

In the last chapter of this thesis we have attempted to narrow the gap between the conceptual aspect of the class of binary tries that we have discussed previously and the implementation aspect. These ideas can be directly applied to almost any file if a record is considered as a long string of bits. In general such a file would be a very small subset of all the possible bit strings. It has not been our aim to study compaction techniques per se, as these rarely cause no loss of information. But rather we have tried to use compaction techniques as an auxiliary tool for implementation of our tries, in order to reduce the cost of searching.

Recall that a query may retrieve irrelevant records by virtue of the fact that usually not all the specified bits are actually tested during trie search. Thus it makes sense to reduce as much as possible the number of records to be retrieved for comparison with the query. We may say that the cost associated with a query is "worse" than it should be only when the query retrieves more records than are pertinent, i.e. when the precision factor is low. Since the recall factor is always 1 for trie indexing, we must try to improve the precision factor in order to minimize search time.

We have thus studied two methods of coding a file. In the method of superimposed coding an ancillary coded file is obtained which can be kept in core. We have applied the techniques of trie construction of chapter 6 and have compared the cost of searching a trie indexed file with the cost of sequential searching on the coded file. We have found that trie search on the coded file is a remarkable improvement over the sequential method advocated in [26]. We believe that the extra overhead introduced by building and searching a trie index is well worth this improvement in the results..

The major disadvantage of the superimposed coding method is that the ancillary file of coded words must be maintained in addition to the master file. This poses some problems for updates as well.

A seemingly better method of coding a file is offered by concatenation of the codes of the individual fields of a record. An obvious advantage of this technique is that a code thus obtained preserves more information content than a superimposed code. We have studied two methods of concatenation coding.

In the first method [2] a certain (optimal) number of bits are extracted from each field of a record and these bits are concatenated so as to produce a bucket address where the record is stored. The number of bits to be extracted is determined, but there seems to be no way of deciding the best choice of bits. A file coded in this manner is easy to maintain, but the cost of answering certain queries may be quite high due to an increased number of buckets which have to be examined; however the average is optimal.

Finally a simpler approach is outlined to determine both the number of bits and the choice of bits necessary to code each field in a record. The codes of the various fields are obtained independently and concatenated to produce a coded record. The weights attached to the codes of different fields are determined from the probabilities associated with the query patterns and the code itself is generated randomly. The various bits to be extracted for determining bucket addresses are automatically decided by trie construction.

Although our experiments do indicate better results than the one reported in [2], we feel that at least three points must be clarified before we can convincingly show the superiority of this method: 1) the set of bits extracted during trie construction is an optimal choice; 2) the relation between the bit patterns along the various paths on the trie and their associated buckets addresses; 3) the number of distinct bits tested on the trie is optimal.

The main conclusion to be drawn from this research is that the methods we have outlined construct tries that are good almost everywhere over the query space. As this statement rests almost exclusively on the empirical results we have obtained through (most of the time) exhaustive simulation, a theoretical analysis remains open for future research.

REFERENCES

[1] AHO, A.V., HOPCROFT, J.E. and ULLMAN, J.D [1974]
"The Design and Analysis of Computer Algorithms", Addison-Wesley, Reading, Mass.

[2] AHO, A.V. and ULLMAN, J.D [1979]
"Optimal Partial-Match Retrieval when Fields are Independently Specified", ACM Trans. on Database Systems 4:2, 168-179.

[3] ALAGAR, V.S. [1980]
"Algorithms for Processing Partial Match Queries Using Word Fragments", to appear in Information Systems.

[4] ALAGAR, V.S. and SOOCHAN, C. [1979]
"Partial Match Retrieval for Non-uniform Query Distributions", Proceedings of the National Computer Conference, New York, AFIPS Press, 775-780.

[5] BENTLEY, J.L. [1975]
"Multidimensional Binary Search Trees Used for Associative Searching", CACM, 509-517.

[6] BENTLEY, J.L. and BURKHARD, W.A. [1976]
"Heuristics for Partial Match Retrieval Data Base Designs", IPL, 132-135.

[7] BURKHARD, W.A. [1976]
"Hashing and Trie Algorithms for Partial Match Retrieval", ACM Transactions on Data Base Systems, 176-187.

[8] BURKHARD, W.A. [1977]
"Non-uniform Partial Match Retrieval File Designs", Theoretical Computer Science, 5, 1-23.

[9] BURKHARD, W.A. [1976]
"Partial Match Retrieval", BIT 16, 13-31.

[10] COMER, D. [1978]
"The Difficulty of Optimum Index Selection", ACM Trans. On Database Systems, 4, 440-445.

[11] COMER, D. and SETHI, R. [1977]
"The Complexity of Trie Index Construction", JACM, 3, 428-440.

[12] de la BRIANDAIS [1959]
"File Searching Using Variable Length Keys", Proceedings of the Western Joint Computing Conference, IRE., n.y., 295-298.

[13] FREDKIN, E. [1960]
"Trie Memory", CACM, 490-499.

[14] GHOSH, S.P. [1977]
"Data Base Organization for Data Management", Academic Press, New York.

[15] GHOSH, S.P. and ABRAHAM, C.T. [1968]
"Application of Finite Geometry in File Organization for Records with Multiple Valued Attributes", IBM Journal of Res. Develop. , 12, 180-187.

[16] GUSTAFSON, R.A. [1969]
"A Randomized Combinatorial File Structure for Storage and Retrieval Systems", Ph.D. Thesis, University of South Carolina, Columbia.

[17] HARRISON, M.C. [1971]
"Implementation of the Substring Test by Hashing", CACM 14, 12, 777-779.

[18] HSIAO, D. and HARARY, F. [1970]
"A Formal System for Information Retrieval from Files", CACM 13, 67-73.

[19] HU, T.C. and TUCKER, A.C [1971]
"Optimal Computer Search Trees and Variable-length Alphabetical Codes", SIAM Journal of Applied Mathematics, 21, 514-532.

[20] KARP,R.M. [1976]
"The Probabilistic Analysis of some Combinatorial Search Algorithms", Algorithms and Complexity- New Directions and Recent Results, Edt. J.F. Traub, Academic Press, New York.

[21] KNUTH, D.E. [1973]
"The Art of Computer Programming", Volume 1: "Fundamental Algorithms", Addison-Wesley, Reading, Mass.

[22] KNUTH, D.E. [1973]
"The Art of Computer Programming", Volume 3: "Sorting and Searching", Addison-Wesley, Reading, Mass.

[23] MINKER, J. [1971]
"An Overview of Associative or Content Addressable Memory and a KWIK Index to the Literature", Technical report TR-157, University of Maryland Computing Center, College Park.

[24] RABIN, M.O. [1976]
"Probabilistic Algorithms", Algorithms and Complexity-

New Directions and Recent Results, Edt. J.f. Traub, Academic Press, New York.

[25] RIVEST, R.L. [1976]
"Partial-match Retrieval Algorithms", SIAM Journal on Computing, 1, 19-50.

[26] ROBERTS, C.S. [1977]
"Partial Match Retrieval via the Method of Superimposed Codes", Bell Laboratories, Computing Science, T.R. #64.

[27] SACERDOTI, E.D. [1973]
"Combinatorial Mincing", unpublished manuscript.

[28] SCHKOLNICK, M. [1975]
"Secondary Index Optimization", ACM-SIGMOD International Conference on Management of Data, San Jose, California, 186-192.

[29] SEVERANCE, D.G. [1974]
"Identifier Mechanisms: A Survey and Generalized Model", Computing Surveys, 6, 175-194.

[30] WONG, E. and CHIANG, T. [1971]
"Canonical Structure in Attribute Based File Organization", CACM 14, 593-597.

[31] YAO, S.B. [1975]
"Tree Structures Construction Using Key Densities", ACM Annual Conference, Minneapolis, Minn., 337-340.

BIBLIOGRAPHY

FILES, J.R. and HUSKEY, H.D. [1969]
"An Information Retrieval System Based on Superimposed Coding", Fall Joint Computer Conference, AFIPS, 35, 423-432.

LEE, D.T. and WONG, C.K. [1977]
"Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees", Acta Informatica 9, 23-29.

MASAHIRO MIYAKAWA, TOSHITSUGU YUBA, YOSHIO SUGITO and MAMORU HOSHI [1977]
"Optimum Sequence Trees", SIAM J. Comput., 6,2, 201-234.