PREFIX RECODING: A FRONT-END COMPRESSION TECHNIQUE FOR SIMPLE PREFIX B-TREES

By

JOHN PATRICK JAGOE

Bachelor of Science

Rhodes University

Grahamstown, R.S.A.

1974

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirement for
the Degree of
MASTER OF SCIENCE
May, 1984

PREFIX RECODING: A FRONT-END COMPRESSION TECHNIQUE FOR SIMPLE PREFIX B-TREES

Thesis Approved:

Michael J. Folk
Thesis Adviser

J P Chandler

M D Grace

Norman N Durham
Dean of the Graduate College

PREFACE

This study examines the effect of recoding common prefixes of shortest separators and thus extending the alphabet and compressing both the sequence set and the simple prefix B-tree index. The purpose of the study is to investigate the effect on a simple prefix B-tree of recoding prefixes with a shorter symbol that maintains collating sequence order.

I would like to thank my major adviser Dr. Michael Folk and the members of my committee Dr. John Chandler and Dr. Donald Grace for their guidance and instruction in this study. I am particularly indebted to Dr. Chandler for his friendship and advice throughout my coursework.

In addition to the members of the committee, I would like to thank Dr. James van Doren for the interest he took in the thesis and for the encouragement he gave me as a student. Special thanks are due to Dr. Donald Fisher who, as department head, made it possible for me to participate in the program and who was a continual source of encouragement and support.

Lastly, I wish to thank Leeanna Jackson who met numerous deadlines, always with a smile.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
II.	KEY COMPRESSION IN TREE INDICES	13
III.	PREFIX RECODED B-TREES	27
	Sequence Set Compression	34
	Alphabet Cardinality	34
	Index Compression	35
	Minimum Storage Limit	41
	Maximum Length of Prefixes	41
	The Choice of Prefixes	42
IV.	EMPIRICAL MEASUREMENTS OF THE EFFECT OF RECODING PREFIXES	49
	The Representational Model	49
	Test Cases	53
	The Results of Empirical Testing	54
V.	SUMMARY, CONCLUSIONS, AND SUGGESTED FUTURE RESEARCH	65
	Summary	65
	Conclusions	66
	Suggestions for Further Work	67
SELECTED BIBLIOGRAPHY		70
APPENDIX		72

LIST OF TABLES

Table		Page
I.	Expected Length of Separators in the Index of a Simple Prefix B-tree	30
II.	Length of Partial Separators in a Prefix B-tree	31
III.	The Effect Prefix Recoding on the Base File	55
IV.	The Effect on the Sequence Set	56
V.	The Effect on the Number of Separators	57
VI.	The Effect on the Separators Generated	58
VII.	The Effect on the First Index Level	59
VIII.	Tree Structure: the Number of Nodes at Each Level (Node Size=64 Bytes)	61
IX.	Tree Structure: the Number of Nodes at Each Level (Node Size=128 Bytes)	61
X.	Tree Structure: the Number of Nodes at Each Level (Node Size=256 Bytes)	62
XI.	Tree Structure: the Number of Nodes at Each Level (Node Size=512 Bytes)	62
XII.	Tree Structure: the Number of Nodes at Each Level (Node Size=1024 Bytes)	63
XIII.	Tree Structure: the Number of Nodes at Each Level (Node Size=2048 Bytes)	63

LIST OF FIGURES

Figure		Page
1.	B-tree Index.	7
2.	B-tree Index Node	7
3.	B^+-tree Index and Sequence Set.	9
4.	A Sequence Set Node Boundary.	21
5.	A Prefix B-tree Index Node.	23
6.	Hypothetical File	36
7.	Symbol Requirements without Compression	43
8.	Symbol Requirements with co Compression	44
9.	Symbol Requirements with co and con Compressed.	44
10.	Symbol Requirements for Non-adjacent Nested Groups.	45
11.	Symbol Requirements for Adjacent Nested Groups.	46
12.	Index Node Structure.	50
13.	Sequence Set Node Structure	51
14.	Internal Node Organization.	52

CHAPTER I

INTRODUCTION

The rapid increase in the size of internal memories has not moderated the need to discover more efficient methods of transferring data from external to internal memory. As internal memories have grown, so have the uses to which we put computers. Today, typical database applications are such that we need to access efficiently huge volumes of data on external storage devices.

Information stored externally is usually organized in a file, a collection of records of similar structure each of which has a unique primary key. A file, or group of files making up a database, may be accessed in two ways: sequentially or randomly by the primary key. To access a file sequentially, one starts at the beginning of the file and accesses the records in the logical key sequence in which the file is organized, called the key sequence order of the file. Random access, also known as direct access, refers to the retrieval of records by key independently of each other. The choice between these two methods depends, of course, on the requirements of a particular application; frequently, applications will require that files may be

accessed both sequentially and randomly. It is for these applications that the approaches discussed and the method described in this thesis will be of most interest. Applications that require either sequential access or random access alone are more easily and efficiently implemented by means other than tree structured indices. If files are only accessed sequentially, sequential files on tape drives will suffice, and if random access alone is the required means, then hash organized files (key-to-address transformation) are superior to tree indices, especially where files are static. Even in applications where files are dynamic, hashing may be chosen, at the expense of re-hashing when performance degrades, in preference to tree structured indices because retrieval from a hash file can be achieved in constant time as opposed to the logorithmic retrieval times provided by tree structued indices. A point worth noting here is that, when the nature of the key set is not known in advance, designing an efficient key-to-address transformation algorithm may be difficult or impossible. Sequential access is of course impossible in a hash organized file.

Index structures in general direct the search for a particular record to a relatively small section of the file thus circumventing the need to begin a search at the beginning of a file. For a given file size, a logically small index will tend to direct the search to a relatively large interval in the file, while a logically large index

will facilitate the direction of the search to a relatively small interval. For this reason, indices to large files tend to be large, and it may be necessary to store the index itself as a file on an external storage device. Tree structured indices--multilevel indices in which the first index built on a file is itself indexed by a higher level index, which in turn may also be indexed, and so on--evolved in answer to the need to index very large files using large indices stored on external devices.

There are many data structures that may be used to implement an index: binary trees, height balanced trees, and tries, to name but a few; however, the evolution of tree indexing structures has developed towards B-trees and the many varieties of this structure. Today, in the words of Douglas Comer, "the B-tree is, de facto, the standard organization for indexes in a database system" (7).

The evolution toward B-trees with large numbers of keys per node and away from tree structures with nodes containing fewer keys (binary, AVL etc.) has been influenced by the nature of external storage devices used in applications that require random access. Disk drives are the primary device used in these applications though drum and, more recently, laser disks and bubble memory can and have been used. The latter three suffer from the same disadvantages in seek time as disk drives, and the details will largely be ignored here.

Accessing information on an external device such as a

disk drive is extremely slow relative to internal memory accessing times. Typical disk drives have access times on the order of milliseconds and data transfer rates of the order of megabytes per second. Disk drives may be divided into two classes: those with movable read/write heads (one head per surface) and those with fixed read/write heads (one head per track). Movable head drives are slower than fixed head drives, due to the additional delay, called seek time, of moving the head to the required track. Using either type of drive, the access time of external storage is a significant bottleneck. For large indices stored externally, the major measure of efficiency is the number of external accesses required to complete a search. Whatever the chosen index system, database designers select parameters such as node size so that the physical characteristics of the particular device or system are utilized efficiently. Node size is frequently chosen to match the track dimensions or, in a virtual paging system, to match the virtual page size. This strategy leads to a generally accepted measure of efficiency for tree structured indices: node visit cost. This measure assumes that each node visited represents a new random access to external storage and that the degree to which a particular index structure solves the problems of the external access bottleneck is given by the length of the path from root to leaf in a tree structured index: the shallower the tree, the fewer are the external accesses. The index is thus

deemed more efficient if the tree is shallow.

The tree structured indices common to large database applications are multiway branching structures and can be divided into two classes: static directories, also known as index sequential structures, and uniform depth, dynamically restructuring trees, commonly called B-trees. These two classes will be discussed below.

Static directories are created so that they are, in the initial state, balanced, thus giving the desired effect of a uniform search length for a particular key. The weakness of this structure becomes apparent only in a dynamic environment where insertions cause performance to degrade. Static directories have a constant number of levels during the period between restructuring. Insertion into the underlying file create the need to insert keys in the index. When an index node overflows, these index insertions are not placed in the logical sequence of an index node but are chained into overflow areas. The imbalance introduced by insertions leads to performance degradation as well as the loss of collating sequence order between primary pages and overflow pages (loss of order may be corrected by sorting primary and overflow pages on insertion). Periodic restructuring eliminates the inefficiency introduced but the structure requires careful monitoring to determine when performance is approaching unacceptable limits.

B-trees, and the many variants of B-trees, differ from the static directories discussed above in an important

regard: insertions and deletions result in local reorganization which is performed incrementally at update time, which maintains the structure's balance, and which can be achieved inexpensively and at known cost. The depth of a B-tree is guaranteed to be uniform, by rule 5 below, and the cost of updating and dynamic restructuring is at worst $O(\log_m n)$ where m is the branching factor and n the file cardinality. The following is a set of rules that define the traditional B-tree:

1. The root, unless it is also the only leaf node, will have at least two subtrees.

2. The order of a B-tree is said to be m, where m is the maximum branching factor of a node. A node has at most m subtrees.

3. All internal nodes (nodes other than the root and leaf nodes) have at least $\lceil m/2 \rceil$ subtrees.

4. All internal nodes have one more subtree than keys.

5. All leaves are on the same level (the tree grows by splitting the root into two nodes and propagating a single key up into the new root).

It can be seen from the definition above that the B-tree indexing structure is the result of a trade-off. Optimal storage utilization is traded for guaranteed retrieval times. Nodes are allowed to remain only partly filled (at least half full) in order that the uniform depth characteristic may be guaranteed. Figure 1 below depicts a

simple B-tree.

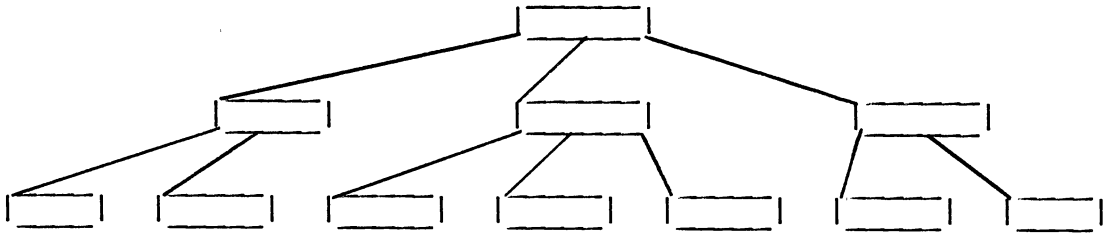

Figure 1. B-tree Index.

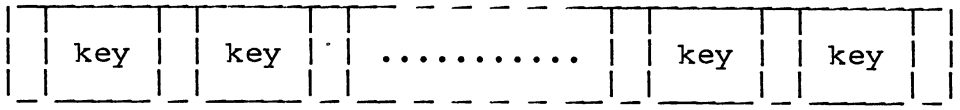

Figure 2. B-tree Index Node.

In fact, the cost in terms of decreased storage utilization is not, in practice, as severe as might be expected from rule 3. Empirical studies have demonstrated that nodes approach, on average, 70% storage utilization after random insertions and deletions (22). In addition, some variants of B-trees have more stringent rules for underflows (rule 3) which require that nodes be at least 2/3 full, for example B*-trees.

Overflow conditions created by insertion into a full node are resolved by various combinations of overflow

sharing schemes (keys are passed to underfull adjacent sibling nodes when possible) or by node splitting when necessary (a node is split into two and a key propagated up to the next level, this process possibly cascading all the way to the root where a split produces a new root and increases the level of the tree by one).

Underflow conditions created by deletion from a node such that rule 3 is contravened are resolved by various combinations of underflow sharing schemes (keys are passed from adjacent sibling nodes when these siblings have more than sufficient keys to remain legal) or by node concatenation when necessary (adjacent sibling nodes are coalesced into a single node and a key is removed from the level above, this process possibly cascading all the way to the root, causing the tree to shrink by a level).

These updating strategies are the basis of the incremental, dynamic, logical reorganization that underlies the uniform depth advantage of B-trees, the uniform node visit cost, to use the term current in index evaluation. It is this structure's avoidance of imbalance and thus performance degradation which recommends it so strongly over the static directory index.

A fact that recommends B-trees over hash indexing is that, even in the traditional B-tree, the logical collating sequence order of the keys is maintained albeit at the expense of costly symmetric order traversals of the index (full records and keys are stored in the index of a

traditional B-tree, unlike some of the later variants discussed below).

Before B-tree variants are discussed, it is necessary to define the terms B*-tree and B^+-tree. Following the nomenclature proposed by Douglas Comer (7), the term B*-tree will be reserved for a "B-tree in which each node is at least 2/3 full" (instead of just 1/2 full). Again following Comer, the term B^+-tree will be used to refer to the B-tree variant in which the tree is organized in two distinct parts: the index part, which is a B-tree of search keys but contains no other information other than pointers, and the sequence set, a linked list of leaves in which the full record or the key and a pointer to the full record is stored. Figure 3 below depicts a B^+-tree.

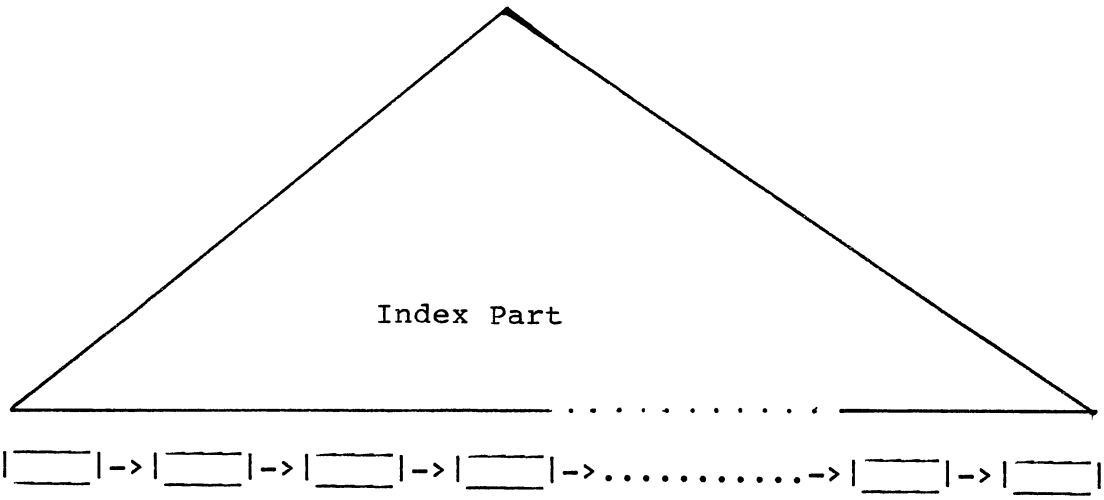

Figure 3. B^+-tree Index.

Noteworthy here is the fact that this sequence set is the equivalent of the leaf level in the traditional B-tree and that, in the B^+-tree, a sequence set node split propagates upwards only the key part of the entry. Part of the confusion of terminology in the literature results from the fact that trees are frequently hybrids: a particular tree may, for instance be both a B*-tree and a B^+-tree (rule 3 defines underflow at the 2/3 full level and all full records are stored at the leaf level, in the sequence set).

The B*-tree requirement that nodes are at least 2/3 full increases storage utilization at the expense of slightly more complicated balance maintaining algorithms and the associated increase in processing complexity. Overflow sharing is required until two adjacent nodes are full; these two nodes are then split into three nodes, each 2/3 full. Deletions result in underflow sharing until three adjacent siblings can be coalesced into two full nodes.

The major contribution of the B^+-tree variant is that, by maintaining a sequence set of full records and an index of keys alone, all the full records are at the bottom level--in the sequence set. This fact, in addition to the customary practice of linking the sequence set with horizontal pointers, ensures that sequential access is trivially implemented. Not only is sequential processing from the beginning of the underlying file easily implemented, but also, a "next" operation after a random

access can be achieved in at most one additional access to external storage (to fetch the next sequence set node in logical collating sequence order).

Thus far, this description has been concerned with the design of indices. Before further discussion is possible, it is necessary to make a distinction between two broad classes of record type: fixed length records and variable length records. In the following discussion, a record may mean either the full record containing all information or a "record" from the index point of view, a field containing only a key and a pointer to the full record. Applications requiring fixed length records abound; however, we shall be concerned here with applications that necessitate the use of variable length records and keys. More specifically, this thesis and the B-tree variants simple prefix B-trees and prefix B-trees deal with applications in which the search key is a variable length word constructed from some alphabet. The empirical studies presented in Chapter IV use a 24,000 word dictionary of English language words.

The programs written to support this study were written as though for a simple dictionary application; however, this specialized type of applicaton is becoming increasingly important with the advent of document data base retrieval systems. These systems enable a user to access variable length unformatted records, namely documents in a document collection, by a variable length key, namely words from the document which are extracted and designated as indexing

terms or keys based on analysis of the frequency of use in the various documents making up the collection. The records in the sequence set are usually made up of a key index term and an inverted list of document numbers which identify the documents containing the term with some required frequency.

Simple prefix B-trees and prefix B-trees were developed by Bayer and Unternaur as a method of increasing the efficiency of B^+-trees (called by them B*-trees) in an environment that necessitated the use of variable length keys. This thesis will compare empirically simple prefix B-trees and an implementation of an indexing structure that is based on Bayer and Unterauer's prefix B-tree and on their observations about the effect of the alphabet size on the average expected length of separators. This investigation is limited to the measurement of the effect of this technique on static dense indices (packed newly recreated dynamic B-trees and static B-trees), and in addition, to the testing of the method on indices built on words in the English language. These indices find important applications in document data base retrieval systems as well as in dictionary data bases.

CHAPTER II

KEY COMPRESSION IN TREE INDICES

Text compression is of interest beyond the area of data base indexing; however, the ideas developed in other areas influence the approaches taken by those interested in index compression. There are, in the specialized field of index key compression, some additional constraints not encountered when one attempts to compress data, for example, to ensure efficient transmission. These constraints arise from the purpose of the strings we would like to compress, namely that the meaning of the terms is derived solely by virtue of the ability to direct a search through an index. Thus, the properties that distinguish various areas in the collating sequence of the underlying data base cannot be destroyed without making the index keys worthless. On the other hand, the keys can be transformed in any convenient way without concern for recovery as long as this ability is not lost. In addition to the constraints indicated above, the very nature of index keys gives rise to opportunities not available in general text compression, namely what is referred to as the sorting induced redundancy inherent to indices. It is the removal of this sorting induced

redundancy that the front-end compression techniques discussed below will attempt to achieve. This chapter will discuss the rationale for index compression and key compression, present a broad overview of general data compression techniques and the applicability to the task at hand, and then briefly describe some proposed solutions to the problem of compressing indices.

The justification of index compression is reasonably obvious. Very large databases require large indices to direct the search to a relatively small areas in the data set. Given the size of modern indices, the alternatives to storing the index on external devices are few, and therefore, a decrease in the physical size of an index while maintaining its logical size is advantageous. Although pointer compression is a means of contributing to this decrease, it is key compression that concerns us here. In a multilayered tree structured index, compression of the keys defining the search path increases the fanout at any particular level in the index and may reduce the number of levels and thus the disk accesses required during traversal of a search path. In B-trees and B-tree variants this increased fanout is referred to as an increase in the branching factor or the order of the B-tree.

There are various ingenious methods of compressing textual data. Some of the general principles are discussed here as well as the reasons why some of these schemes are not readily applicable to the coding of keys in an index.

A common approach is to use bit strings of just sufficient length to encode the required set of characters. For example, data consisting of the decimal digits, upper and lower case alphabetic characters, and any two special characters may be coded in bit strings of length 6, thus saving 25% of the space normally required. This seemingly attractive approach would require, in the context of index keys, either a decoding step for each character for each key comparison or the adaptation of the comparison process so that the standard unit of comparison becomes 6 bits. The approach presented later in this thesis takes the opposite view. The alphabet is extended so that all 8 bits are used. This will take advantage of the savings in index space observed by Bayer and Unterauer and discussed at the end of this chapter.

Another standard method of compression is to recode a substring chosen on the basis of its length and frequency in the text and replace it with a symbol or a number that is an index into a dictionary of compressed substrings. This is very similar to the prefix recoding technique presented in Chapter III. In standard text compression, substrings are compressed throughout the entire text, whereas, with the prefix recoding technique, only prefix frequency is used to choose substrings for recoding and then only prefixes and initial characters are recoded. Common prefixes of length greater than one are recoded to save storage while all other initial letters are recoded to maintain a key's capacity to

direct a search and to extend the alphabet.

Huffman coding, a minimum redundancy code for single symbol encoding, produces variable length bit strings which represent symbols. The idea here is that the number of bits used to represent a symbol is inversely proportional to the logarithm of that symbol's frequency of occurrence in the text. Frequent symbols are given the shortest codes and longer bit string codes are constructed for less frequent symbols such that the short codes already assigned do not appear as initial bit sequences in these longer bit strings. This eliminates the need for bit string demarcation. Not only does the variable length of bit strings produce problems in the context of index key comparison, but also, the assignment of codes does not maintain collating sequence order between keys.

Numerical encoding compresses text by using a symbol's position (P_i) in a dictionary of symbols and the total number of symbols in the dictionary (B) in conjunction with some unit of grouping (N) to compress N symbols by creating a unique number from the following expression:

$$P_1*B^{N-1}+P_2*B^{N-2}+\ldots+P_{N-1}*B+P_N$$

The original text can then be recreated because B and N are known, P can be derived, and the dictionary used to expand the compressed text. This method does preserve collating sequence order, but is not applicable to keys of variable length unless N is chosen equal to the maximum key

length or, in the case where rear end compression has already been performed, equal to the maximum length of the shortest separator. This restriction introduces difficulty. While it is easy to establish the maximum length key or shortest separator length when an index is created, there is no way of knowing how this maximum will change as insertions are made in the database. The savings achieved would vary with the amount by which the average key or separator is shorter than the maximum key or separator. This technique is interesting, but not the subject of this study.

The last standard data compression technique discussed here is that of squeezing out long sequences of identical characters such as leading or trailing blanks. This technique is obviously of little value to index compression where these long sequences do not typically occur.

The problem of compressing indices has been approached in various ways. The temptation to compress keys is frequently resisted in environments where keys are of fixed length since compression techniques frequently yield variable length compressed keys (not the case for numerical encoding) which necessitates additional administrative space in the node, either in the form of intra-node pointers indicating the starting position of keys or a length field attached to each key. Another objection to compression is that variable length keys cannot be searched efficiently within a node (15). The second objection is just not valid since using intra-node pointers permits intra-node searching

in time of O(log n) by means of a traditional binary search with an extra level of indirection via the intra-node pointers. The first objection is moot in the applications under consideration here since compressing keys that are already of variable length does not add to the cost of intra-node operations nor to the intra-node storage requirements: intra-node pointers are already required.

There are several approaches to the compression of indices: binary compression (22), prefix B-trees (3), simple prefix B-trees (3), and a front-end compression technique developed by Clarke et al (5 and 6). These approaches are discussed below.

The technique of binary compression of index keys produces a binary tree of variable search path length (22). The creation of the binary compressed index is achieved by virtue of the placement of a pair of index keys in two index positions based on the most significant single bit difference. The position of the difference bit and either a pointer to another position in the index or a data address are retained in the index. The fact that the nodes, being binary, are not easily tied to the read capacity of an external device makes this type of index an unlikely design choice for applications with very large indices stored externally. In addition, the indexing technique does not necessarily produce a uniform length search path. Although this can be forced during the initial building of the index, the tree may become unbalanced during insertion or deletion,

thus making it subject to performance degradation, an undesirable possibility. The ability to access the sequence set sequentially can only be obtained by creating the index pointers as offsets into a list of sequence set pointers. This addition represents additional index overhead.

The remaining techniques are termed character compression. These methods take advantage of two aspects of indices, namely that the sorted order of an index of keys ensures that the set has inherent to it some degree of prefix redundancy and that the least significant bits of a pair of keys are not needed in the determination of a search path (the first difference bit or character suffices). These approaches have led to what is termed front-end compression in the case of prefix redundancy and rear-end compression in the case of least significant character truncation. Of course, the degree of sorting induced prefix redundancy depends heavily on the size of the alphabet making up the keys relative to the size of the index. Considering the two extremes makes this observation intuitive. Given an alphabet of 26 characters and a file of 26 keys, it is possible, though not inevitable, to have no prefix redundancy at all. Given an alphabet of 1 character and a file of any length, it is inevitable that there is total prefix redunancy. This idea is important to the method tested in this thesis and will be discussed further in the next chapter.

There is an important difference between front and

rear-end compression. Since, in the B^+-tree variant, the index serves only to guide a search of the sequence set where all full records are stored (or pointers to full records), keys in the index may be altered at will as long as their ability to direct the search is maintained. This assertion is not true of pure B-trees where the search may terminate above the leaf level and alteration of a key would create ambiguity as to whether a key existed, because full records are stored in the index. The truncation of least significant characters may be performed in a B^+-tree application without concern for key reconstruction. However, if we compress a prefix, we must either be able to reconstruct it or have replaced it with a symbol that assumes the prefix's function of maintaining collating sequence order. The methods discussed here are based on the ability to reconstruct the prefix or construe its value from adjacent keys. The method that is proposed in this thesis is based on the replacement of a prefix by a shorter symbol that assumes the function of the prefix. The weakness of the prefix reconstruction approach is that the complexity of index operations is greatly increased; the cost of the replacement technique is that the replacement byte is required and that less storage is saved.

Three approaches to index compression will now be discussed: simple prefix B-trees, prefix B-trees (3), and an unnamed character compression technique developed at IBM by Clarke et al (5 and 6) and discussed by Chang (4) and

Wagner (22). For convenience, this method will be referred to here as the Clarke method. The order of this discussion is based on the following. Simple prefix B-trees produce rear-end compression and are of proven value. The other two methods, prefix B-trees and the Clarke method, compress both front and rear portions of a key and overlap conceptually to such a degree that the adjacent discussion simplifies the explanation.

A simple prefix B-tree is a B^+-tree in which the variable length, shortest separator between two keys bridging a node boundary at the sequence set level is propagated up into the index thus saving space in the index and decreasing the number of disk accesses.

For example, a node division at the sequence set level may appear as shown in figure 4.

Figure 4. A Seqence Set Node Boundary

In this case, the shortest separator needed to guide a search in the immediately preceding level is the unique prefix of the second key, cond . The rear end compression of "emn" saves space. There may be many possible separators

of optimal length; there are certainly many of less than optimal length (conde would function as well as cond). However, since the object is to save storage, Bayer and Unterauer (3) define the selection of a shortest separator as follows:

> Let x and y be any two keys such that x<y. Then there is a unique prefix $\bar{y}$ of y such that (a) $\bar{y}$ is a separator between x and y, and (b) no other separator between x and y is shorter than $\bar{y}$ (p.12).

It should be noted that the separator may be either less than or equal to the collating sequence value of y and that it is always greater than x by virtue of y being greater than x. Equal key values are not permitted (or useful) in an index. The search algorithm for a simple prefix B-tree is based on this relationship between separator and pairs of index keys used to generate the separator: in the event that a search key is equal to a separator over the entire length of a separator, the search path associated with that separator is taken. It should also be noted that, in a multilevel index structure, rear-end compression only takes place with the initial creation of index keys. Further rear-end compression at higher index levels would create search path ambiguity.

Bayer and Unterauer (3) found the following by experimentally comparing simple prefix B-trees with B^+-trees (called B*-trees in the article):

1. Time complexity - index operations required time almost identical to the same operations in B^+-trees.

2. External disk accesses - (1) no decrease in trees of fewer than 200 nodes. (2) 20-25% decrease in trees with between 200 and 400 nodes (p. 24).

A prefix B-tree takes the idea a step further and stores only once, in the index nodes or preceding parts of the subtree, the common prefix of the shortest separators, thus further reducing the storage requirements and external accesses (reducing the height of the index).

For example, if at some level of the index all the shortest separators in a node share the common prefix con , the keys may be stored as shown in figure 5.

Figure 5. A Prefix B-tree Index Node

The compression is achieved by avoiding multiple storage of the common prefix con .

The common prefix can be reconstructed either from the node itself or from the node and the node's ancestors. An alternative here that saves multiple reconstruction of index

keys, one for each key comparison during an intra-node search, is to adjust the search key by deleting prefix characters from the search key prior to an intra-node search. It is worth noting that savings from front-end compression are likely to be greater at lower levels in the index. This increased compression is likely by virtue of the fact that adjacent index keys at the lower levels are certain to be closer together in terms of collating sequence distance than are index keys at higher levels in the tree. In other words, keys nearer the root direct the search to wider and more distant parts of the sequence set, and the search is narrowed as we drop down from level to level. Unlike rear-end compression where a one time truncation take place and then saves space at all levels of the index, front-end compression is most effective at the lower levels and then less effective with each higher level, if it takes place at all (the least likely place for keys to share a common prefix is at the root level).

This type of key compression does increase the branching factor and decrease the size of the index; however, these advantages are achieved at the cost of greatly increased time needed for index operations. This is partly due to the cost of reconstructing keys during a search, but also due to the necessity of reconstructing and recompressing keys during a page split and even during the insertion of a new key in a node with sufficient space. There is also a potential instability: if a new key is

inserted, either at the logical beginning or end of a node and if this insertion changes the common prefix of that node, then the compressed keys have to be expanded and the possibility exists that the expanded keys cannot be handled by a simple page split. While it is true that the compression prior to the insertion would have to have been very significant, this potential for instability exists and makes questionable the idea that dynamic node restructuring takes place in order of $\log_m n$ where m is the branching factor and n the index size. In addition to these above mentioned drawbacks, Bayer and Unterauer (3) have established experimentally that prefix B-trees reduce external accesses very little relative to simple prefix B-trees (a 2% decrease) and that normal indexing operation require 50-100% more time.

The compression method termed here the Clarke method is very similar to Bayer and Unterauer's prefix B-tree in terms of its goals, that is front-end compression of sorting induced redundancy and rear-end truncation of characters that are functionally redundant for search path definition. Compressed keys are generated by comparing adjacent index keys. Characters preceding the first difference byte of the larger key are not stored but are later construed from the preceding keys using two count fields kept with each compressed key: the length of the compressed key and the length of the prefix that was compressed. The underlying idea is very close to that of the prefix B-tree; the

realization differs in an important respect: an intra node search must take place sequentially, an expensive operation for large numbers of keys. Recall that the prefix B-tree node could be scanned using a binary search via intra-node pointers once the compressed prefix was removed from the search key. The Clarke method has some advantages over prefix B-trees in that the operation for inserting a key requires only a local update of the preceding and succeeding keys. The cost of serial intra-node searching can be reduced by creating, internal to the node, a two or more level index , which in turn will make the compression rules, search, and splitting algorithms more complex. The performance of the Clarke method is unknown; however, based on the rules of compression, it appears that more storage is saved than prefix B-trees save (a prefix does not have to be common to all keys, just to adjacent keys) but that this savings is achieved at the expense of much greater intra-node traversal time and at the expense of the count fields required for each key (if both numbers stored in the count fields are less than 16, both may be represented in 8 bits).

CHAPTER III

PREFIX RECODED B-TREES

The two front-end compression techniques described in chapter II rely on the reconstruction of the prefix to define the search path. In an attempt to avoid the additional time complexity involved in these approaches, prefix recoded B-trees are proposed here. This method will trade some of the storage that could be saved by either the prefix B-tree or the Clarke approach for greatly decreased time complexity during indexing operations. It will avoid the necessity to reconstruct keys by replacing compressed prefixes and all initial characters with symbols that maintain collating sequence order. Part of the motivation for this approach is Bayer and Unterauer's (3) observations (analytically arrived at and experimentally confirmed) about the effect of the alphabet cardinality on separator size in a randomly generated key set.

Bayer and Unterauer's analysis summarized here is only an approximate analysis. Their purpose is to arrive at a means of obtaining a theoretical approximation of a simple prefix B-tree, an index having the least significant characters truncated. The results are only vaguely related

to what is attempted here: the recoding of prefixes as a means of saving storage. The expected length of their shortest separator is somewhat related to prefix recoding. Although their purpose is not to suggest the extension of the cardinality of the alphabet, the results they present suggest that such an extension would save much storage.

Several points with respect to their analysis should be noted here. The analysis assumes fixed length keys. The authors see no reason why variable length keys should produce significantly different results; however, since prefix recoding is a strategy proposed to deal with variable length keys, it should be borne in mind that Bayer and Unterauer's assumption may make a significant difference. For this reason, their analysis is presented here only as a recommendation of recoding and not as a theoretical underpinning of the method. In addition to key length, Bayer and Unterauer assume that the keys are randomly distributed over the possible maximum cardinality of the file (given by a^k where a is the alphabet and k the fixed length of keys). In the English language, however, keys tend to cluster in certain areas. For example in the dictionary used as a key set to test prefix recoding, 95Ø of the 24,ØØØ words start with the prefix co . There are 676 possible 2 letter combinations for an alphabet of 26 letters (26^2). The fact that 4% of the keys fall into 1 of 676 slots (Ø.ØØ15%) illustrates this clustering. The greater the degree of this clustering, the greater is the chance

that longer separators will be needed for a given alphabet. But in addition to this observation, it seems that the longer the generated separators are, the more room there is for prefix compression.

The drawback to the theoretical approach to this problem is that it is impossible to draw general conclusions unless one assumes that the key set is of a random nature. Index terms extracted from a natural language are unlikely to be random and thus the closest we can come to predicting the results of this type of compression is to make preliminary measurements of a particular key set. An analysis of the key set used for this study indicated that of there were 703 variable length combinations with a maximum length of 2. This number was determined as follows:

1. There are no duplicate keys.
2. There are 26 letters in the alphabet (26^2=676)
3. There are keys with a trailing blank (giving 26 additional possibilities.
4. The string 'bb' was allowed (1 additional possibility).

By scanning the actual key set it was found that 19,543 of the 24,000 keys (81.43%) were in 98 of the 703 slots (13.94%). This fact seems to indicate that there is considerable clustering in the key set and that recoding common prefixes may be profitable. It should be noted that, when Bayer and Unterauer extend the cardinality of the alphabet, this extension takes effect over the entire key,

whereas prefix recoded B-trees extend the cardinality of the alphabet used to code prefixes and initial characters but leave the remaining characters in a key consisting of the original alphabet (having cardinality of 26).

Tables I and II below summarize Bayer and Unterauer's observations.

TABLE I

EXPECTED LENGTH OF SEPARATORS IN THE INDEX OF A SIMPLE PREFIX B-TREE*

	n			
a	10^3	10^4	10^5	10^6
26	2.483	3.104	3.842	4.615
256	1.774	1.976	2.517	2.936

* a=alphabet cardinality; n=file cardinality

Source: R. Bayer and K. Unterauer, "Prefix B-Trees," ACM Trans. on Database Syst. Vol. 2, No. 1, March, 1977.

The average length of separators determined experimentally was 0.35 lower than the theoretical results above.

TABLE II

LENGTH OF PARTIAL SEPARATORS IN A PREFIX B-TREE

	n				
k	10^3	10^4	10^5	10^6	
10	1.73	1.80	1.91	1.80	
100	2.48	2.35	2.54	2.69	a=26
100	2.48	3.10	3.09	3.32	
10	1.77	1.23	1.55	1.60	
100	1.77	1.98	1.77	1.97	a=256
1000	1.77	1.98	2.52	2.19	

Before discussing prefix recoded B-trees and comparing these with prefix B-trees and the Clarke method, it should be noted that all three methods will employ rear-end compression, a proven method of index compression and the subject of Bayer and Unterauer's compression technique: simple prefix B-trees.

Prefix recoding will attempt a slightly less ambitious compression of the sorting induced redundant prefixes inherent in indices. Each initial letter will be replaced by a new ASCII character and, where it is found to produce nearly maximum saving, not only the initial character but also an entire prefix (of variable length) will be replaced by one of the 255 ASCII characters. Using 255 of the symbols provided by 8 bits is, in a sense, an extension of

the alphabet, although the extension is restricted to the initial parts of index keys. This fact and the fact that rear-end compression truncates all characters after the first difference character motivates the proposal for prefix recoded B-trees made here.

The details of this proposal and of the "storage versus intra-node operation efficiency" trade-off are as follows.

The 26 letters of the alphabet are usually stored one character per byte (upper case letters can be converted to lower case since storing words in alphabetic order requires that upper and lower case letters be interspersed). This usage, of course, is wasteful since it takes only 5 bits to code 32 and thus 26 patterns. We can make use of the additional bits by defining a 255 letter alphabet (one pattern being reserved as an end-of-word delimiter) which is used to represent words by replacing common prefixes in the sequence set with patterns that are chosen so as to maintain collating sequence order and compress prefixes. The approach is to choose the prefixes for compression so that the storage saved is maximized of nearly maximized. This prefix recoding appears to have considerable advantage over the original method of front-end compression proposed by Bayer and Unterauer with respect to computational complexity during index operations such as inserting and deleting. Whereas Bayer and Unterauer propose a compression technique that requires dynamic updating during these operations the technique proposed here requires a once-only analysis and

recoding of the sequence set prior to building the packed index for static trees as well as the relatively much simpler task of converting each search key to the new scheme and decoding entries in the sequence set for display. This advantage is achieved at the expense of the additional character that replaces the prefix during recoding and of limited compression (the number of prefixes that can be compressed is limited by the cardinality of the new alphabet). However, prefix recoding takes place in the sequence set and the savings in storage is felt at this level, as opposed to the prefix B-tree approach where full keys are stored at the sequence-set level. This extra savings at the sequence set will provide some small additional savings but is not exected to be significant where the node size is large and the number of index keys per node is high.

As was outlined in the previous chapter, the Clarke method enjoys the advantage of greater prefix compression than the prefix B-tree (due to the fact that prefixes do not have to be common to a large number of keys before these prefixes are compressed), but this additional compression is achieved at the expense of requiring a sequential intra-node search during index operations or a complex and expensive intra-node structure consisting of multiple levels. Prefix recoding will allow efficient intra-node operations ($\log_2$ m where m is the branching factor of a node) while achieving compression that, although not as great as

the Clarke method, is expected to be significant in terms of index size reduction and decrease in node visit cost. The optimization of performance within a node is said by Lomet (13) to be important, especially in a multi-user environment where the release of a node may be awaited by a second process.

The effect of prefix recoding has three different aspects. These are discussed in increasing order of expected importance below.

Sequence Set Compression

There will be a decrease in the size of the sequence set due to prefix compression at this level. This decrease may not be enough to save even one node at the lowest index level (the level above the sequence set) when nodes are large and relatively many keys per node are stored, but as node size decreases, and thus the number of keys stored in a node decreases, the effect may become significant.

Alphabet Cardinality

By increasing the cardinality of the alphabet, the method will decrease the size of shortest separators. A simple illustration is as follows: if every possible key exists in a file of keys where the alphabet size is three,

then there are only three shortest separators of length one. Given the fixed maximum key length, increasing the alphabet size to 1Ø creates many more possible keys, but, if the original keyset can somehow be translated into the new alphabet, there would then be 1Ø shortest separators of length one, thus reducing the average shortest separator length. A similar assertion can be made for the increase in the number of shortest separators of length two and greater.

Index Compression

Common prefixes are compressed in the index. For example, since the prefix con is common in the sequence set, it is reasonable to expect it to be more frequently encountered in the index than a prefix occuring less frequently in the sequence set. Hence, the compression of con in the sequence set saves storage in the index.

There follows a discussion of how prefixes are chosen for compression. Common prefixes occur where the keys in a set are relatively long and relatively close in terms of collating sequence distance. What we mean by this is as follows. Given the length of the largest word in a key set and the cardinality of the alphabet making up the keys, the maximum number of keys (called slots here) in a hypothetical file is given by the following formula.

$$\text{maximum} = \sum_{i=0}^{n} a^{i} \qquad (1)$$

In the above formula, a represents the alphabet cardinality and n the length of the largest word. Keys are unique and so the closest the keys can be in terms of collating sequence distance is that the keys occupy two adjacent slots in the hypothetical file. Figure 6 below is used as a simple example. The underscore character '_' represents a blank.

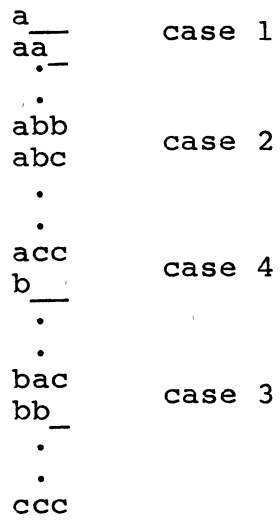

Figure 6. Hypothetical File.

Consider figure 6 above for purposes of illustration. The alphabet consists of three characters (a,b, and c) and the maximum key length is three. The adjacent pairs of keys exhibit one of the following characteristics:

1. The two keys have in common all characters except that the second key has an additional character that is the lowest character in the alphabet (aa_ and aaa, b_ and ba).

2. The two keys are of the maximum length and differ only in that the second key has a least significant character that is greater by one than the least significant character of the first key (abb and abc, cbb and cbc).

3. The two keys have in common all characters up to some point between the first and the last character, after which the first key has the highest character in the alphabet in all subsequent positions and the second key has blanks in all positions following the difference.

4. The first key is the highest possible key in the subset defined by the initial character and the second key is the lowest key possible in the subset defined by the character higher by one than the initial character of the first key. (acc and b__, bcc and c__).

From the above description it can be seen that, in case 1, there is room for some prefix compression, the degree of which depends on how many characters the keys share in common. In case 2, there is room for much compression: all but the least significant character is a candidate for compression. Case 3 provides some room for compression.

The extent of this compression depends on how many characters are common to both keys. Case 4 provides no room for compression; the shortest separator will be of length 1 (the initial character of the second key). However, after extending the alphabet, the number of times this case occurs in the hypothetical file is increased. It should be noted that this case occurs infrequently: as many times as there are characters in the alphabet.

In this thesis, prefixes are chosen so as to save storage. An attempt is made, within two arbitrarily set limits, to closely approximate maximization of storage saved. Another approach is to attempt to spread the new symbols as evenly as possible over the sequence set, so that groups of words sharing a common recoded prefix are of nearly equal size. This would avoid the possibility that a recoding symbol is used on a prefix that is long, but common to few words and thus susceptable to being 'buried' in the middle of a sequence set node and not participate in decreasing the size of shortest separators. This second alternative is not considered here although one of the parameters that is arbitrarily set (the length of the prefixes considered for compression) has the effect of influencing how evenly the new symbols will be spread.

When we consider real key sets, especially if these consist of index keys from a natural language like English, the number of slots defined by the alphabet and maximum word length is far greater than the number actually filled by

existing keys. Also, the keys are not randomly interspersed among the slots, but, to some extent, tend to cluster. It is in the areas where clustering occurs that prefix compression by recoding will be most beneficial.

There are two dimensions that affect the amount of storage saved by prefix compression. The first is the number of occurrences of a particular prefix. The second is the length of that prefix.

In this thesis, prefixes of variable length are considered. The shortest prefixes considered are two characters long. Since the two byte prefix is replaced by a single-byte ASCII symbol, the saving of storage is exactly one byte per occurrence of the prefix in the sequence set and, likewise, one byte per occurrence in the index. The storage saved by prefixes of greater length--three and four character prefixes--is given by the following formula:

$$\text{Storage saved} = (\text{len} * \text{num}) - \text{num} \qquad (2)$$

(where len is the prefix length and num is the number of occurrences of the prefix). Since, by definition, a sequence of keys having in common an n character prefix have in common an n-1 character prefix, we can speak of prefix groups, candidates for compression, as being contained within other prefix groups. For example, the prefix group con is contained in the group co . Both the groups cons and cont are contaned in con which is of course

contained in the group co . This nesting introduces some complexity in the choice of prefixes for compression. To consider all possibilities--prefix groups ranging from groups of length two to groups equal in length to the largest key as well as all possibilities for number of occurrences, ranging from a pair of keys sharing a common prefix to the largest group of two character prefixes--is impossible for a large sequence set on the equipment available for this study and within reasonable time constraints. For these reasons, two arbitrarily set limits (described below) constrain the choice of prefix groups for compression.

It should be borne in mind that there are only 255 symbols available in the present proposal. Also, most prefixes chosen for compression require the use of two of these new symbols. If we choose the group co for compression we need three symbols as opposed to the single symbol previously used to encode the letter c . These symbols are used as follows:

1. a symbol to encode all initial c's preceding co
2. a symbol to encode all prefix co 's
3. a symbol to encode all initial c's following co

There are exceptions to this "two extra symbol" rule; these will be discussed later.

The arbitrarily set limits are as follows.

Minimum Storage Limit

A minimum is set on the storage saved by any prefix group under consideration. This limit gives prefix groups of length greater than two the opportunity to compete by virtue of the greater compression offered. It should be noted, however, that combinations of nested prefix groups are only considered after this pruning of less profitable possibilities and it cannot be claimed that the method will produce an optimum saving, though it is thought that the procedure produces a good solution.

Maximum Length of Prefixes

A maximum is set on the length of prefixes that are considered. This limit restricts the depth to which prefix groups are allowed to be nested, but not the number of groups that can exist at any particular depth. This maximum is set as a means of controlling the complexity of considering all possible combinations of nested prefixes. Since the decision to include a candidate nested group in a combination is a "yes-no" decision, it is obvious that the number of combinations in a nested group is 2^j where j is the number of groups nested within the two character prefix group. In the sequence set used in this study the groups contained within co numbered 17 when the first

arbitrary limit was set at 90 bytes saved (arrived at by trial and error) and the second to four characters. Considering the resulting 131,072 (2^{17}) possible combinations of nested prefixes within this one group is a task that approaches the limits of reasonableness.

The Choice of Prefixes

Step 1

A key is read and the two, three,and four character prefix is extracted and stored for comparison with later keys.

Step 2

Subsequent keys are read and a count kept for the number of occurrences of common prefixes of lengths two, three, and four.

Step 3

When a prefix group changes--the new prefix replaces the stored prefix for later comparison and the number of bytes saved by the just completed group is calculated. If the bytes saved exceeds the first arbitrarily set limit discussed above, then the prefix and some administrative

information is stored for later consideration. If the just completed group is a two character prefix, a check is made to see whether the group contained nested groups. In the event that this is so, all of the mutually exclusive combinations are generated and the best (most storage saved) combination for each possible number of symbols required is retained. Combinations of prefixes may require from two symbols in the simplest case to twice the number of groups nested within the two character prefix (described below). Most prefixes considered for compression require two symbols in addition to the symbol already used to encode the initial characters.

This requirement can be seen in figures 7 through 9:

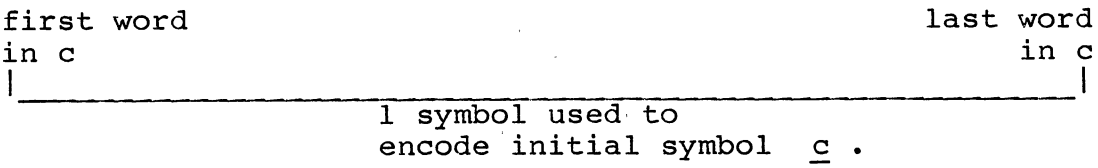

Figure 7. Symbol Requirements Without Compression.
Total symbols used: 1

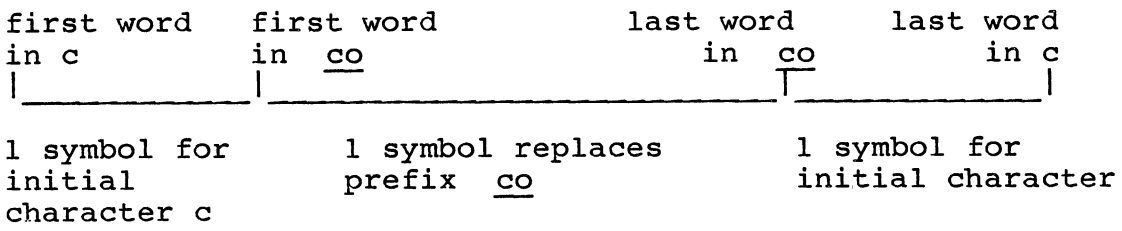

Figure 8. Symbol Requirements with co Compressed.
Total symbols used: 3

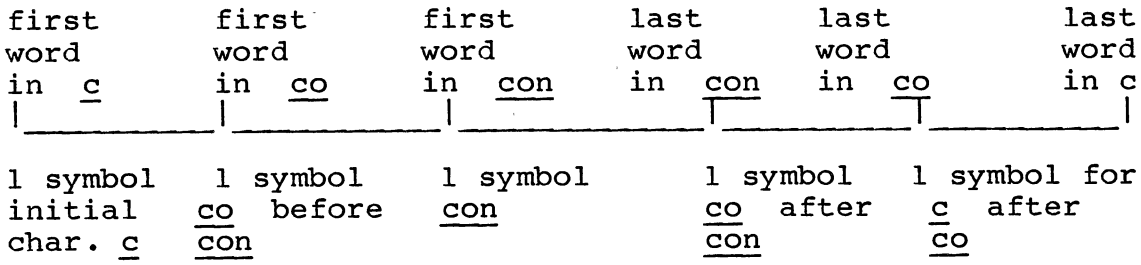

Figure 9. Symbol Requirements with co and con Compressed.
Total symbols used: 5

It can be seen that, in the examples, each additional prefix, whether it is nested at a deeper level or at the same level, requires two additional symbols. This is not always the case. Consider figures 1Ø and 11:

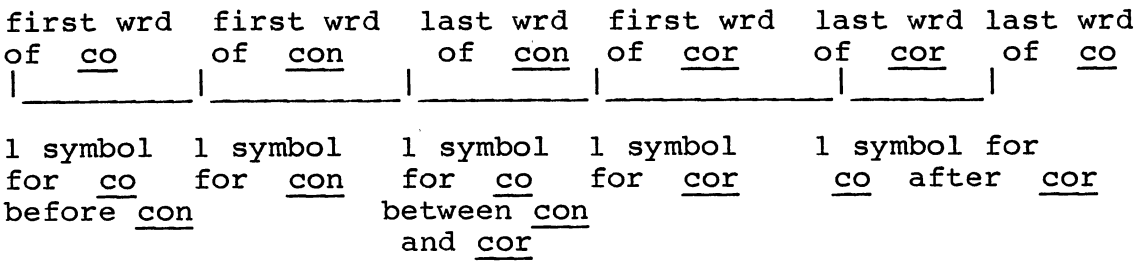

Figure 1Ø. Symbol Requirements for the Non-adjacent Nested Groups.
Total symbols used: 5.

Here each nested prefix group requires an additional two symbols, but in the example below, the nested groups com and con are adjacent in the sense that no symbol is required to represent co between these two groups because no slot exists over that section that does not fall within either com or con . This fact is obvious when we consider that the prefix groups are of the same length and the least significant character of the second (n) is greater by 1 in terms of collating sequence distance than the least significant character (m) in the preceding group com .

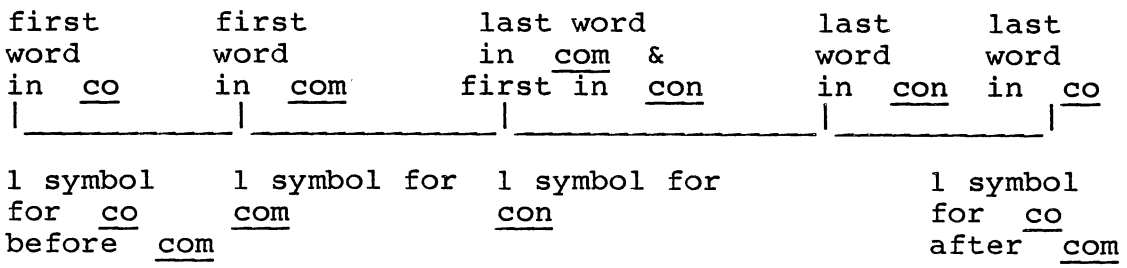

Figure 11. Symbol Requirements for Adjacent Nested Groups.
Total symbols: 4

In the above figure, only one extra symbol is required to add con to the nested group co and com . This reduced requirement is taken into consideration in the evaluation of combinations of nested prefixes.

The choice among combinations of nested groups is made as follows. The single group yielding the largest marginal saving in storage is chosen as the initial position (marginal savings being defined here as the number of bytes saved per additional symbol used). This possibility is saved for comparison with other prefix groups outside the nested group being processed. All of the other possibilities are adjusted to reflect the marginal savings in relation to this first choice. Recall that much pruning has already been done; only the best combination requiring n symbols is considered here, all others having been

eliminated earlier by virtue of their inferiority relative to this best choice for numbers of symbols required. After the adjustment of marginal savings, negative marginal savings are discarded and the best choice is retained for comparison with other groups outside the nested group. This new combination is marked as being mutually exclusive with the first group retained for later comparison. The remaining combinations are adjusted for comparison with this second retained possible combination and the process continues with ever decreasing marginal savngs until a point is reached where the marginal savings does not exceed the first arbitrarily set limit. At this point the process stops since there will be other groups preferable to any further combination or the arbitrary limit will be lowered, to yield in the next run further possibilities during the above discussed process (evaluation of nested combinations) and during the overall process (the evaluation of prefixes in general, simple and nested).

Step 4

The final step in the process of choosing prefixes is taken when the entire sequence set has been used to generate all of the simple candidate prefixes and all of the mutually exclusive nested combinations subject to the two arbitrarily set limits. The last step requires only that the best prefixes or groups of nested prefixes are chosen based on

the number of bytes saved per symbol required. During this step, the move from initial choices within mutually exclusive groups to subsequent positions within the current group is made based on a comparison of the benefits of the move relative to other possibilities, provided either by simple groups or other nested groups.

After the prefixes are chosen for compression, the relatively simple task of translating the sequence set remains. This translation is achieved during a single pass over the set. The prefixes chosen for compression and all initial characters are replaced by one of the 255 ASCII symbols provided by 8 bits (recall that one pattern is reserved for an end-of-word delimiter). After this encoding process, packed prefix recoded B-trees can be created and a table of symbols for prefixes and initial characters created for use during indexing operations.

CHAPTER IV

EMPIRICAL MEASUREMENTS OF THE EFFECT OF RECODING PREFIXES

The Representational Model

Two types of trees were created and compared in this study: simple prefix B-trees and prefix recoded B-trees. For the purpose of comparison, various node sizes of identical structure were created and the trees built. There follows here, first, a description of the tree structure and then a description of the sequence set node structure and index node structure.

Both types of tree are implemented as B^+-tree structures: the full records are kept at the sequence set level and compressed keys are propagated up into the index, rear-end compressed shortest separators in the case of the simple prefix B-tree and prefix recoded shortest separators in the case of prefix recoded B-trees. The sequence set is a doubly linked list of nodes which facilitates traversal of the keyset in both directions, in sequential order or the reverse, after either an initial probe to the beginning of the file or a random access probe to any point in the file.

The node size of the index is the same as the node size at the sequence set level and the physical structure is identical; however, the logical structure differs slightly. All nodes are implemented as an n byte buffer (n ranges from 64 bytes to 2048 bytes for the test cases) which is declared such that it can contain either n 8-bit characters or n/2 16-bit integers. (The buffer is declared to be a union of character and integer types in C, the language of implementation here).

The above physical node structure is used logically to implement the sequence-set and index nodes in slightly different ways. As can be seen in figures 12 and 13, six bytes at the right-hand edge are used for administration.

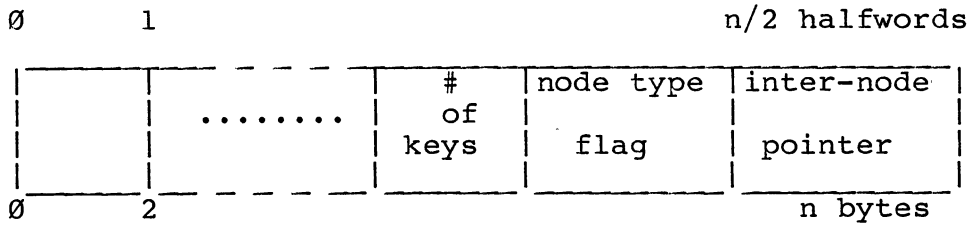

Figure 12. Index Node Structure.

In the index, this space is used for three 16-bit fields: a count of the number of keys in the node, a flag which is set to negative two if the node is the root and

negative one if the node is an internal index node, and the additional inter-node pointer required in the index (there is one more inter-node pointer than keys in an index node).

In a sequence set node, the three 16-bit fields are used as follows: a count of the number of keys in the node and two link fields used to link each node to the logically preceding and succeeding sequence set nodes, thus facilitating sequential traversal in either direction.

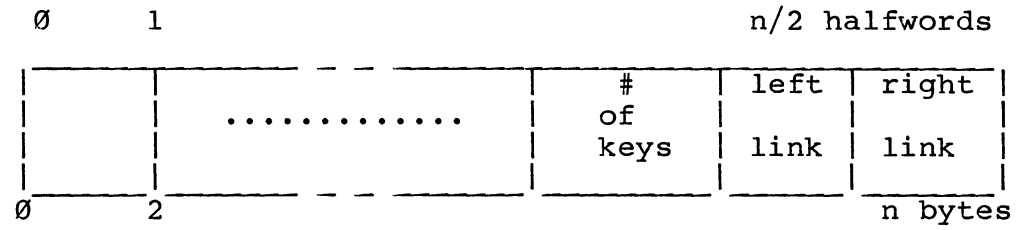

Figure 13. Sequence Set Node Structure.

Full keys are stored in the sequence set and shortest separators are stored in the index in exactly the same manner. The space in the node is used as follows. The keys are stored at the low subscript end of the node in the order of arrival: a key's physical position does not define its logical position in the sequence of keys within a node. In addition, there are no spaces between keys--these are packed together and, in the event of a deletion, all keys

physically higher are shifted down. Beginning at the other end of the node, starting from a point just below the six byte administration fields, a vector of pointer pairs is maintained and allowed to grow down toward the unordered key locations. A pair of pointers consists of an inter-node pointer which points to an index node in the following index level or to the sequence set and an intra-node pointer which points to the starting position of the key associated with the pointer pair. This organization can be seen in figure 14. It is the physical positioning of the pointer pairs that facilitates the inorder accessing of keys within a node: the intra-node pointer of the pair located immediately below the administration fields points to the lowest key in the node; the intra-node pointer below that gives the location of the next key in order, and so on.

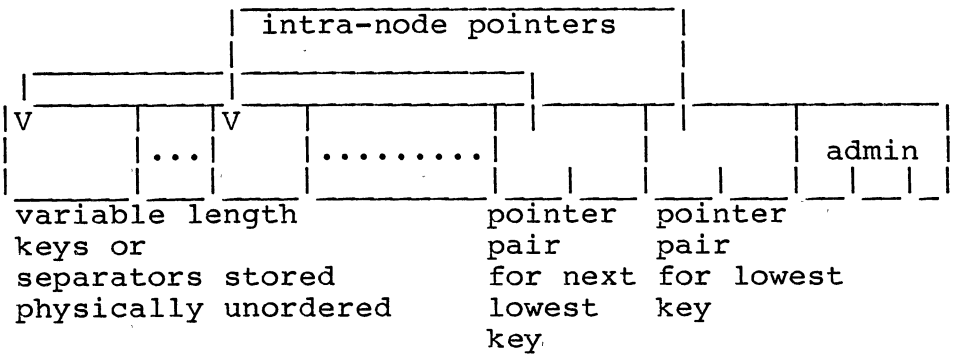

Figure 14. Internal Node Organization.

B-tree nodes are subject to some rule that defines an underflow condition (for example, nodes are not allowed to be less than half full). In the case of variable length keys, such a rule may be defined based either on the number of keys, or, more likely, on the number of bytes used. For the purpose of measuring storage utilization in this study, this underflow rule is ignored. Nodes are packed with as much information as possible as an index level is built and the fact that a final node on any particular level may violate this rule is ignored. If the trees built here were going to be used as an index, it might then be necessary to underflow share keys so that all nodes obeyed whatever underflow rule was defined.

Test Cases

A total of twelve trees was built for the purpose of comparing simple prefix B-trees and prefix recoded B-trees. For both of these tree types, trees of six different node sizes were generated. The node sizes chosen were 64 bytes, 128 bytes, 256 bytes, 512 bytes, 1Ø24 bytes, and 2Ø48 bytes. It was decided not to take this progression to smaller node sizes because nodes might then not be sufficiently large to contain a very large single key and because the range at the low end was considered to cover adequately all reasonable choices of node size. At the high end, the node size of

2048 bytes produces a B-tree index consisting of a single node, the root. This fact made it obvious that considerations of larger nodes for the keyset under study would yield no further information other than that these node-sizes would also require a single node B-tree index consisting of a partially filled root node.

Results of Empirical Testing

Twelve trees, six node sizes for each tree type, were generated from two base files (one containing the full words and the other containing the same words with common prefixes recoded) to produce the results presented in this section as empirical evidence of the effect of recoding prefixes. The 24,000 words used in the base file came from the UNIX dictionary facility.

Prefix Compression and the Effect on the Base File

Common prefixes in the original UNIX words file were identified and evaluated as described in Chapter III. A table of the prefixes chosen for compression is given in Appendix A. The base file used to generate the six bench-work simple prefix trees was processed and the designated prefixes recoded and compressed. The results of this compression are given in Table III below. The figures represent the compression achieved in the creation of the

unstructured recoded base file. Later tables will give figures which represent the savings once the information is loaded into sequence set nodes.

TABLE III

THE EFFECT OF PREFIX RECODING ON THE BASE FILE
(UNIX WORDS FILE CONTAINS 24,000 WORDS)

WORDS FILE (BYTES)	RECODED FILE (BYTES)	% COMPRESSED
196,476	174,319	11.28

The Effect on the Sequence Set

The base files discussed above were loaded into sequence set records as the first step in the creation of the two types of tree. Trees for each of the node sizes were generated and the sequence set files measured to determine the effect of prefix compression on the sequence set. It should be noted that these files contain all of the administrative information (count fields, pointers, and links) associated with nodes at this level. Table IV below gives the space requirements for the sequence sets of the different trees.

TABLE IV

THE EFFECT ON THE SEQUENCE SET

NODE SIZE	SIMPLE PREFIX B-TREE (BYTES)	PREFIX RECODED B-TREE (BYTES)	% COMPRESSED
64	299,520	269,888	9.89
128	269,440	244,096	9.41
256	256,512	232,960	9.18
512	250,880	227,840	9.18
1024	248,832	226,304	9.05
2048	249,856	227,328	9.02
2048	246,856	227,328	9.02

From the above table, it is evident that the decrease in the sequence set is, in all test cases, between 9% and 10%. The significance of this decrease becomes apparent when we consider that the decrease in size means that more records are packed into a prefix recoded B-tree node of given size, thus requiring fewer sequence set nodes and thus fewer separators in the index. The separators themselves are compressed and the compounded effect of fewer, shorter separators can be seen in subsequent tables.

The Effect on the Separators Generated

The effect on the separators generated by creation of the sequence set is evident in the tables below. Table V

gives the effect on the number of separators generated. It should be recalled that a separator is generated between every pair of nodes at the sequence set level. There is thus one fewer separator at all levels of the index than the number of nodes in the sequence set.

TABLE V

THE EFFECT ON THE NUMBER OF SEPARATORS GENERATED

NODE	SIMPLE PREFIX B-TREE	PREFIX RECODED B-TREE	% CHANGE
64	4,678	4,215	9.99
128	2,103	1,905	9.42
256	1,000	908	9.20
512	488	443	9.22
1024	241	219	9.13
2048	120	109	9.17

Table VI below gives the compounded effect of recoded prefixes. The figures represent the savings for the unformatted shortest separator and prefix recoded separator sets: the separators are measured before insertion into index nodes and thus the figures overstate the savings in the index.

TABLE VI

THE EFFECT ON THE SEPARATORS GENERATED

NODE SIZE	SIMPLE PREFIX B-TREES (BYTES)	PREFIX RECODED B-TREES (BYTES)	% CHANGE
64	30,026	22,982	23.46
128	13,456	10,316	23.23
256	6,328	4,973	21.41
512	3,123	2,378	23.86
1024	1,529	1,192	22.04
2048	757	607	19.81

Table VII below gives a more realistic idea of the savings in the index at the first level. The figures give the compression of the formatted separators at the first level: all of the administrative overhead (count fields, flags, and pointers) are taken into account. Table VI above may be viewed as an upper bound on the savings possible in each test case.

TABLE VII

THE EFFECT ON THE FIRST INDEX LEVEL

NODE	SIMPLE PREFIX B-TREE (BYTES)	PREFIX RECODED B-TREE (BYTES)	% CHANGE
64	48,960	40,320	17.65
128	22,016	18,176	17.44
256	10,752	8,960	16.67
512	6,144	5,632	8.33
1024	5,120	5,120	0.00
2048	6,144	6,144	0.00

A fact worth noting that is not evident from the above table is that at the larger node sizes, 1024 bytes for instance, the saving in storage is concealed by the fact that the same number of nodes are required to contain the index at this level (see the following table giving node numbers). Within the larger nodes, there is a significant difference in the storage available for subsequent insertions. In the case of the 1024 byte node size, at the first level in the index, there are 1,002 bytes available for future insertion in the prefix recoded tree compared with only 572 bytes in the simple prefix B-tree. This additional space means that the prefix recoded tree, while it requires the same index storage in the current state, can withstand far more insertions without requiring more storage

for this level (without splitting). It is thus not true to say that there is no advantage to recoding prefixes for this node size with this key set. The above discussion is applicable to the information given for larger node sizes in the following section.

The Effect on Tree Structure

Tables VIII through XIII give information about the tree structures generated by the six test cases. The figures represent the numbers of nodes at the various levels in the trees. The number of separators propagated to the next higher level can be calculated since, if at level k there are n nodes, there must be n-1 separators propagated up to level k+1. Some of these will, in turn, be propagated to higher levels if these levels exist.

TABLE VIII

TREE STRUCTURE: THE NUMBER OF NODES AT EACH LEVEL (NODE SIZE = 64 BYTES)

LEVEL	SIMPLE PREFIX B-TREE	PREFIX RECODED B-TREE	% CHANGE
5	1 (ROOT)	1 (ROOT)	--
4	4	3	25.Ø
3	22	16	27.3
2	128	97	24.2
1	764	629	17.7
Ø	4679	4216	9.9

Again the compounded effect (fewer, shorter separators) is evident at the higher index levels.

TABLE IX

TREE STRUCTURE: THE NUMBER OF NODES AT EACH LEVEL (NODE SIZE = 128 BYTES)

LEVEL	SIMPLE PREFIX B-TREE	PREFIX RECODED B-TREE	% CHANGE
4	1 (ROOT)	--	--
3	2	1 (ROOT)	--
2	15	11	26.7
1	171	141	17.5
Ø	21Ø4	19Ø6	9.4

TABLE X

TREE STRUCTURE: THE NUMBER OF NODES AT EACH LEVEL (NODE SIZE = 256 BYTES)

LEVEL	SIMPLE PREFIX B-TREE	PREFIX RECODED B-TREE	% CHANGE
3	1 (ROOT)	1 (ROOT)	--
2	2	2	--
1	41	34	17.1
Ø	1ØØ1	9Ø9	9.2

TABLE XI

TREE STRUCTURE: THE NUMBER OF NODES AT EACH LEVEL (NODE SIZE = 512 BYTES)

LEVEL	SIMPLE PREFIX B-TREE	PREFIX RECODED B-TREE	% CHANGE
2	1 (ROOT	1 (ROOT)	--
1	1Ø	9	1Ø.Ø
Ø	489	444	9.2

TABLE XII

TREE STRUCTURE: THE NUMBER OF NODES AT EACH LEVEL (NODE SIZE = 1Ø24)

LEVEL	SIMPLE PREFIX B-TREE	PREFIX RECODED B-TREE	% CHANGE
2	1 (ROOT	1 (ROOT)	--
1	3	3	--
Ø	242	22Ø	9.1

TABLE XIII

TREE STRUCTURE: THE NUMBER OF NODES EACH LEVEL (NODESIZE = 2Ø48)

LEVEL	SIMPLE PREFIX B-TREE	PREFIX RECODED B-TREE	% CHANGE
1	1 (ROOT)	1 (ROOT)	--
Ø	121	11Ø	9.1

It can be seen that the savings in the number of index nodes is negligible for the trees built with the two largest node sizes. However, it should be noted that the benefits of prefix compression are concealed by the node size and shallowness of the index. If the tree were to grow with

insertion the positive effects would become evident, as these effects are in the deeper trees. It is also worth noting that the prefix recoded trees are more resilient to splitting on insertion by virtue of the fact that, though the node count is not significantly less, the nodes contain fewer keys.

The results of the tests indicate that the effect of prefix recoding is beneficial for the base file tested. This observation is only an indication that the same may be true of other bases. The extent to which we can extrapolate from the results presented here depends upon the extent to which any other base is similar in terms of prefix clustering or sorting induced prefix redundancy. It is uncertain that this method would be as useful for bases consisting of random keys where clustering might occur less often than it does in keys extracted from a natural language.

CHAPTER V

SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FURTHER WORK

Summary

Prefix recoding is an approach to improving the efficiency of an index to a large file of variable length keys. The method approaches front-end compression by replacing the common prefixes at the sequence set level and all the initial characters with symbols that maintain collating sequence order by assuming the function of the prefixes compressed. The method does not achieve as much key compression as might be expected from the other front-end key compression techniques discussed in Chapter II. However, the method proposed here has the advantage that indexing operations can be performed at very little cost during the time that the index is in use. It is true, of course, that the process of choosing prefixes can be a very expensive operation; however, this process is done off-line and thus does not adversely affect indexing efficiency. In addition, the expensive recoding process requires a once only analysis for each recoding. Subsequent

recodings are only necessary if the system is subject to significant insertion or deletion and then only if these operations significantly change the nature of the key set. During indexing operations the technique requires only an inexpensive translation, a table look up, when the search key is entered and when keys are displayed. The avoidance of the computational complexity inherent in both the Clarke and the prefix B-tree approaches is a factor that strongly recommends prefix recoding over these other techniques. A comparison of the three techniques in terms of storage efficiency would be interesting.

Conclusions

It was hoped that this study would demonstrate significant improvements in node visit cost for the prefix recoded B-tree. This improvement is not evident from the results in Chapter IV. However, it is clear that, if the nodes in the test cases are subject to identical insertion, the keys inserted into the prefix recoded tree being recoded, the simple prefix B-tree will require additional storage at all levels before the same additional storage is required by the prefix recoded tree. This additional storage required will, eventually in both trees, cascade up to the root and increase the number of levels. However, this increase will occur in the simple prefix B-tree before it occurs in the prefix recoded tree. This fact is evident

from the fact that there are always, in all test cases, either fewer nodes at a particular level, or more space in the prefix recoded B-tree nodes. The prefix recoded trees are clearly less likely to split on insertion, and on the average prefix recoding can be expected to reduce the node visit cost where the recoding produces significant compression of the base file.

The application of prefix recoding depends on the base file considered for recoding. It is apparent that the degree of success depends on the degree of prefix clustering in a file. An initial analysis of prefix redundancy in a sorted file is a relatively easy and inexpensive task, and decisions pertaining to recoding ought to be preceded by such an analysis.

Suggestions for Further Work

Several interesting questions were raised by this study. These are discussed below.

The test cases for this study examined the two types of tree in a static environment. Although many applications such as dictionary or document database systems require only minimal insertion and deletion, it would be interesting to measure the effect of prefix recoding in a dynamic environment. Dynamic test cases could be run. These tests would involve building trees by repeated insertion and then subjecting the trees to random insertion and deletion. An

extension of this investigation might entail designing efficient methods to evaluate the effect of insertion and deletion on the keyset so that recoding during periodic backups might be achieved easily.

Another interesting direction of investigation is that of testing the effect of B-tree compaction on prefix recoded B-trees. This compaction would involve the rearrangement of separators in nodes at all levels such that the nodes nearest the root had increased and perhaps near maximal fanout. This compaction would change the shape of the trees from what Rosenberg and Snyder (17) call scrawny to what they call bushy. The effect would be to make the trees' node-visit cost miminal. A comparison of the two types of tree in this state would yield valuable information about the potential of prefix recoding for decreasing node-visit cost. This information is especially interesting for data bases in a relatively static environment: bushy trees are sensitive to insertion and split relatively soon on insertion.

The focus of this study has been on the effect of prefix compression. There is much work to be done on the methods of choosing prefixes. No attempt was made to choose an optimal prefix set, and the effect of prefix recoding may be enhanced by the design of an optimal algorithm for the choice of prefixes. In addition, it might be interesting to know what effect evenly distributing symbols over the keyset would have on the average length of shortest separators.

Lastly, numerical encoding is a technique suggested by Hahn (9) for general text compression applications. It appears that this technique may find useful application in the compression of fixed length keys in indices. The technique maintains collating sequence order and produces fixed length compressed keys. An investigation of this technique in the database environment would be worthwhile.

A SELECTED BIBLIOGRAPHY

(1) Aho, A. Hopcroft, J., and Ullman, J. The Design and Analysis of Computer Algorithms , Reading, Massachusetts: Addison-Wesley Publishing Company, 1974.

(2) Bayer, R., and McCreight, E. "Organization and Maintenance of Large Ordered Indices". Acta Informatica , Vol. 1, (1972), pp. 173-189.

(3) Bayer, R., and Unterauer, K. "Prefix B-trees". ACM Transactions on Database Systems , Vol. 2 (March, 1977), pp. 11-16.

(4) Chang, H. K. "Compressed Indexing Method". IBM Technical Disclosure Bulletin . Vol. 11, No. 11, (April, 1969), p. 1400.

(5) Clark, W. A. IV, Davies, C. T., Salmond, K. A., and Stafford, T. S. "High-Level Index-Factoring System". United States Patent Number 3,646,524 (February 29, 1972).

(6) Clark, W. A., Salmond, K. A., and Stafford, T. S., "Methods and Means for Generating Compressed Keys". United States Patent Number 3,593,309 (January 2, 1969).

(7) Comer, D. "The Ubiquitous B-tree". Computing Surveys , Vol. 11, No. 2 (June, 1979), pp. 121-137.

(8) Feng, A. L. "A Study of Two Computing Index Mechanisms: Prefix B+-Tree and Trie Structures". (Unpublished M.S. Thesis, Oklahoma State University, 1982).

(9) Hahn, Bruce "A New Technique for Compression and Storage of Data". Communications of the ACM , Vol. 17, No. 8, (August, 1974), pp. 434-436.

(10) Held, G., and Stonebraker, M. "B-trees Re-examined". Communications of the ACM , Vol. 21 (February 1978), pp. 139-143.

(11) Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes". IRE 40 (September, 1952), p. 1098.

(12) Knuth, D. E. The Art of Computer Programming Vol. 3: Sorting and Searching. Reading Massachusetts: Addison Wesley Publishing Company, 1973.

(13) Lomet, D. B. "Multi-Table Search for B-Tree Files". IBM Technical Disclosure Bulletin , Vol. 22, No. 6, (November, 1979), pp. 2565-2570.

(14) Marroin, B. A. and deMaine, P. A. D. "Automatic Data Compression". Communications of the ACM , Vol. 10, No. 11, (March 1967), pp. 711-715.

(15) Maruyama, K. and Smith S. "Analysis of Design Alternatives for Virtual Memory Indexes". Communications of the ACM , Vol. 20, No. 4, (April, 1977), pp. 245-254.

(16) Mattson, R., Gecsei, J. Stutz, D., and Traiger, I. "Evaluation Techniques for Storage Hierarchies". IBM Systems Journal , Vol. 9, No. 2, (1970), pp. 78-117.

(17) Rosenberg, A., and Snyder, L. "Time- and Space-Optimality in B-trees". ACM Transactions on Database Systems , Vol. 6, No. 1 (March, 1981), pp. 174-193.

(18) Rubin, R., "Experiments in Text File Compression". Communications of the ACM , Vol. 19, No. 11 (November, 1976), pp. 617-623.

(19) Salton, G., and McGill, M. J., Introduction to Modern Information Retrieval New York: McGraw Hill Book Company, 1983.

(20) Saltern, G. Automatic Information Retrieval . New York: McGraw-Hill Book Company, 1968.

(21) Snyderman, M., and Hunt, B. "The Myriad Virtues of Text Compaction". Datamation Vol. 16, No. 12 (December, 1970), pp. 36-40.

(22) Wagner, R. A., "Indexing Design Consideration". IBM Systems Journal , Vol. 12, No. 4 (1973), pp. 351-367.

(23) Webster, R. E. "B^+-tree". (unpublished M.S. report, Oklahoma state University, 1980).

APPENDIX

THE PREFIXES COMPRESSED FOR THE TEST CASES

PREFIXES COMPRESSED	SINGLE CHARACTERS RECODED	STARTWORD	ENDWORD
	a	1	8
ab		9	118
ac		119	238
ad		239	362
	a	363	518
al		519	729
am		730	830
an		831	1056
	a	1057	1057
ap		1058	1158
	a	1159	1166
ar		1167	1343
as		1344	1467
at		1468	1532
au		1533	1614
	a	1615	1684
	b	1685	1686
ba		1687	2006

	b	Ø	Ø
be		2ØØ7	2318
	b	2319	232Ø
bi		2321	2453
	b	Ø	Ø
bl		2454	259Ø
	b	2591	2591
bo		2592	2811
	b	2812	2812
br		2813	3Ø62
	b	3Ø63	3Ø64
bu		3Ø65	3262
	b	3263	3282
	c	3283	3284
ca		3285	3749
	c	375Ø	3751
ce		3752	3837
	c	Ø	Ø
ch		3838	4162
	c	4163	423Ø
cl		4231	4384
	c	Ø	Ø
co		4385	4495
col		4496	4569
com		457Ø	4723
con		4724	5Ø69
co		5Ø7Ø	5334

	c	5335	5335
cr		5336	554Ø
	c	Ø	Ø
cu		5541	5648
	c	5649	5682
	d	5683	5687
da		5688	58Ø8
	d	58Ø9	58Ø9
de		581Ø	629Ø
	d	6291	6292
di		6293	6587
	d	6588	659Ø
do		6591	6769
	d	Ø	Ø
dr		677Ø	6869
	d	Ø	Ø
du		687Ø	6959
	d	696Ø	6986
	e	6987	7161
el		7162	7268
em		7269	7359
en		736Ø	7455
	e	7456	7733
ex		7734	7963
	e	7964	7977
	f	7978	7979
fa		798Ø	8121

	f	8122	8124
fe		8125	8228
	f	Ø	Ø
fi		8229	8381
	f	8382	8382
fl		8383	8539
	f	854Ø	8541
fo		8542	8715
	f	8716	8716
fr		8717	8871
	f	8872	8872
fu		8873	8945
	f	Ø	Ø
	g	8946	8947
ga		8948	9116
	g	Ø	Ø
ge		9117	9224
	g	9225	9234
gi		9235	93Ø5
	g	Ø	Ø
gl		93Ø6	9393
	g	9394	94Ø6
go		9497	95Ø9
	g	951Ø	951Ø
gr		9511	97Ø9
	g	971Ø	971Ø
gu		9711	98Ø1

	g	98Ø2	9817
	h	9818	9819
ha		982Ø	1ØØ93
	h	Ø	Ø
he		1ØØ94	1Ø335
	h	Ø	Ø
hi		1Ø336	1Ø442
	h	Ø	Ø
ho		1Ø443	1Ø681
	h	1Ø682	1Ø682
hu		1Ø863	1Ø775
	h	1Ø776	1Ø781
hydr		1Ø782	1Ø811
	h	1Ø812	1Ø847
	i	1Ø848	1Ø939
im		1Ø94Ø	11Ø86
in		11Ø87	11717
	i	11718	11732
ir		11733	11796
	i	11797	11859
	j	1186Ø	11861
ja		11862	11931
	j	11932	12ØØ1
jo		12ØØ2	12Ø78
	j	12Ø79	12Ø79
ju		12Ø8Ø	12156
	j	Ø	Ø

	k	12157	12278
ki		12279	12348
	k	12349	12448
	l	12449	12451
la		12452	12662
	l	Ø	Ø
le		12663	12828
	l	Ø	Ø
li		12829	13Ø14
	l	13Ø15	13Ø15
lo		13Ø16	13174
	l	13175	13176
lu		13177	13269
	l	1327Ø	13289
	m	1329Ø	13291
ma		13292	13757
	m	13758	13759
mc		1376Ø	13818
	m	Ø	Ø
me		13819	14Ø49
	m	Ø	Ø
mi		14Ø5Ø	14265
	m	14266	14266
mo		14267	14523
	m	14524	14527
mu		14528	14654

	m	14655	14686
	n	14687	14688
na		14689	14774
	n	14775	14781
ne		14782	14925
	n	14926	14927
ni		14928	15ØØ9
	n	15Ø1Ø	15Ø11
no		15Ø12	15133
	n	15134	15193
	o	15194	15498
or		15499	15582
	o	15583	15673
	p	15674	15675
pa		15676	16Ø27
	p	16Ø28	16Ø28
pe		16Ø29	16323
	p	16324	16325
ph		16326	16414
pi		16415	16565
	p	Ø	Ø
pl		16566	16Ø2
	p	16Ø3	16Ø5
po		16Ø6	16957
	p	16958	16979
pre		16980	17Ø8
	p	Ø	Ø

pri		17109	17154
	p	Ø	Ø
pro		17155	17367
	p	17368	17410
pu		17411	17521
	p	17522	17545
	q	17546	17549
qu		17550	17673
	r	17674	17676
ra		17677	17845
	r	17846	17846
re		17847	18306
	r	18307	18337
ri		18338	18429
	r	Ø	Ø
ro		18430	18582
	r	18583	18584
ru		18585	18670
	r	18671	18675
	s	18676	18677
sa		18678	18931
	s	Ø	Ø
sc		18932	19139
	s	19140	19140
se		19141	19395
	s	19396	19396
sh		19397	19638

si		19639	19817
	s	19818	19866
sl		19867	19959
	s	19960	20001
sn		20002	20065
so		20066	20251
sp		20252	20469
	s	20470	20506
st		20507	20914
su		20915	21147
	s	21148	21301
	t	21302	21303
ta		21304	21469
	t	0	0
te		21470	21666
	t	0	0
th		21667	21864
ti		21865	21953
	t	21954	21954
to		21955	22104
	t	0	0
tra		22105	22237
	t	22238	22272
tri		22273	22351
	t	22352	22410
tu		22411	22494
	t	22495	22558

	u	22559	22744
	v	22745	22746
va		22747	22829
	v	Ø	Ø
ve		2283Ø	22952
	v	Ø	Ø
vi		22953	23Ø84
	v	23Ø85	23147
	w	23148	23149
wa		2315Ø	23331
	w	Ø	Ø
we		23332	23421
	w	Ø	Ø
wh		23422	23534
wi		23535	23681
	w	Ø	Ø
wo		23682	23777
	w	23778	23821
	x	23822	23833
	y	23834	23938
	z	23939	23992

VITA 2

John Patrick Jagoe

Candidate for the Degree of

Master of Science

Thesis: PREFIX RECODING: A FRONT-END COMPRESSION TECHNIQUE FOR SIMPLE PREFIX B-TREES

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in the Transvaal, Republic of South Africa, May 6, 1952, the son of Charles Malcolm Jagoe and Elizabeth Barbara Jardine.

Education: Graduated from Selborne College, East London, Republic of South Africa in December, 1969; received Bachelor of Arts degree in Economics and Political Philosophy from Rhodes University, Grahamstown, Republic of South Africa in December 1974; completed requirements for the Master of Science degree at Oklahoma State University, Stillwater, Oklahoma in May, 1984.

Professional Experience: Lecturer, Oklahoma State University, Computing and Information Sciences Department, 1982-1984.