Department of Computer Science

VLC Tries

Karl W. Glander and Karl P. Durre

Technical Report CS-94-102

February 8, 1994

Colorado State University

VLC Tries

Karl W. Glander
glander@cs.colostate.edu

Karl P. Dürre
durre@cs.colostate.edu

Computer Science Department
Colorado State University
Fort Collins, CO 80523, U.S.A
Phone: (303)-491-7017
FAX: (303)-491-6639

Abstract

The basic trie structure has the desirable feature of O(| W |) access time to determine if key word W is present but tries also have the undesirable feature of an excessive amount of wasted storage. A variation of the trie, called a VLC trie, addresses the issue of wasted storage when storing large static data sets while maintaining the O(| W |) access time. The VLC trie applies a divide and conquer approach to minimize storage requirements through the efficient storage of the different components of the basic trie structure.

Keywords

Information Retrieval; Indexing; File Organization; Tries; Trees; Data Structures.

Introduction

Given a set S of n distinct keys from a key space of L = { m | m $\in$ Σ^*} over alphabet Σ, a frequently asked question is "Is x $\in$ S?" for some x $\in$ L. Instances of this question arise in compiler construction ("Is x a valid token?" "Is X a desirable code optimization?"), information retrieval ("Is x spelled correctly?" "Is x a valid entry into file F?"), data compression ("Is X a substring that should be compressed?"), as well as many other computer science and information processing fields. Two desirable features for any structure implemented to answer this question, in any of its instances, is fast access time while requiring minimal storage space. One particular tree data structure, called a trie [De La Brandais 59, and Fredkin 60], stores the set S and can answer the question "Is x $\in$ S" with an access time of O(| x |). A primary disadvantage of tries is their low utilization of storage space. To address this problem, several variations of tries have been proposed by De La Brandais in [De La Brandais 59], Fredkin in [Fredkin 60], Comer in [Comer 76, Comer 76b, Comer 79], Comer and Sethi in [Comer 77, Comer 85], Maly in [Maly 76], Knuth in [Knuth 73], Al-Suwaiyel in [Al-Suwaiyel 79], Al-Suwaiyel and Horowitz in [Al-Suwaiyel 84], Dürre in [Dürre 84] and Purdin in [Purdin 90]. While some of the variations have maintained the ability to dynamically change the contents of S, the variations that most effectively reduce

the storage requirement require a non-trivial amount of time for constructing the trie and thus impose the condition that S be static. Even with this imposition on S, the storage requirement of the trie variants is not substantially better than some of other data structures that are commonly used especially when the construction time for the trie variants is considered. A new approach to minimizing the storage requirement of the trie structure, when storing a static data set, is presented in this paper. The new approach employs a divide and conquer approach to reduce the storage requirement of the whole trie while maintaining the O(| x |) access time of the trie data structure.

The paper begins with a brief review of the basic trie structure and an explanation of the Single-Linked Compression technique developed by Dürre. In the second section, an explanation of the transformation from the basic trie structure to the VLC trie structure is presented. In the third, and final section, the performance of the VLC trie structure is examined when given static test data sets from two large databases.

Trie Basics

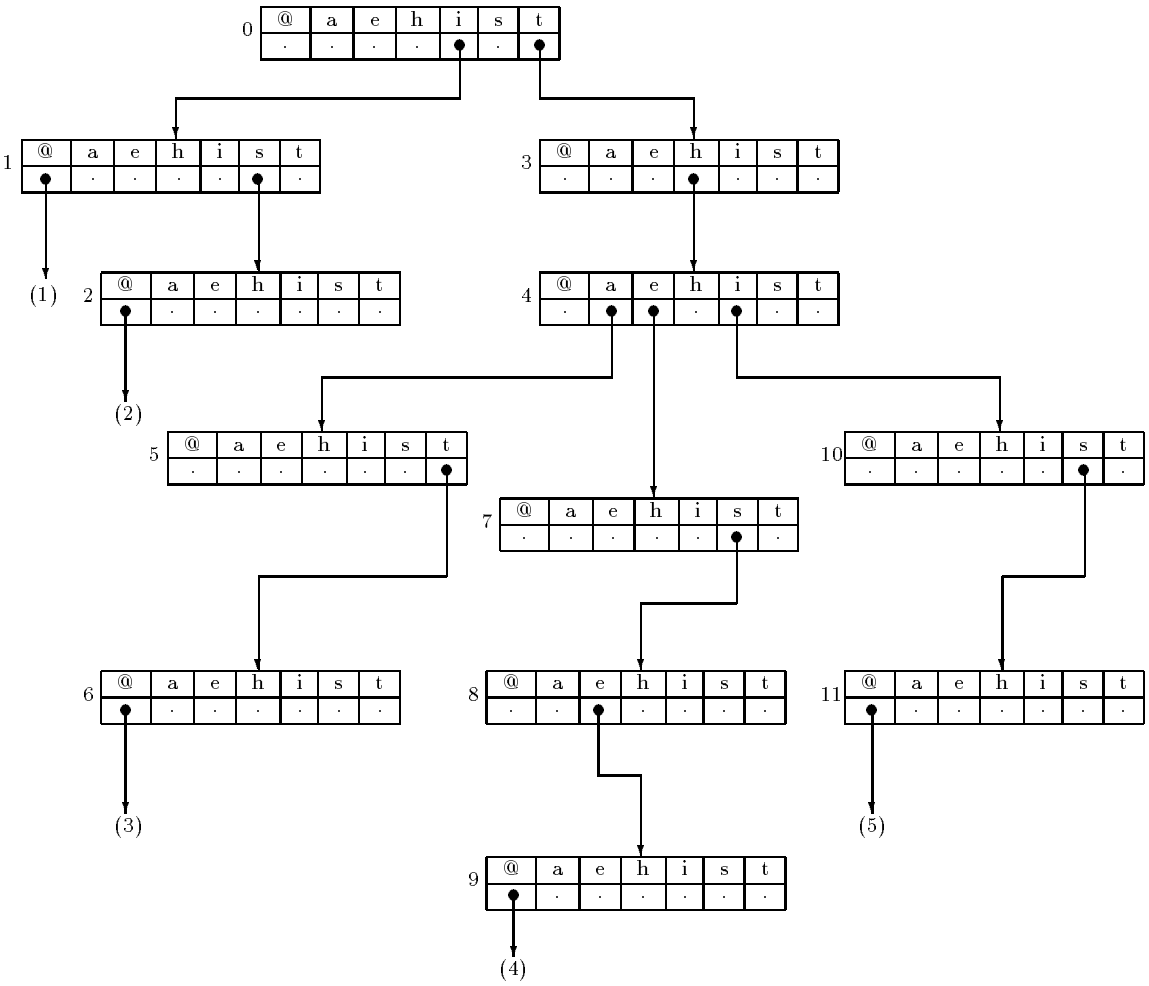
A trie is an M-ary tree where each node consists of an M-positional vector of pointers that correspond to the symbols in the trie alphabet. The trie is constructed such that the M-way branch in a node at level i is dependent on the symbol at the (i + 1)st position of the key word. The trie alphabet is an ordered, finite set of symbols where one of the symbols is a special symbol serving as the end-of-string symbol. A trie may either be represented as a tree or as a matrix as shown in figure 1. Both tries in figure 1 contain the words {i, is, that, these, this} and the trie alphabet for these tries contains { @, a, e, h, i, s, t} with the '@' symbol being defined as the end-of-string symbol. A key word is contained in a trie as a unique path from the root to a leaf. The pointers between trie nodes within the tree representation are implicit while within the matrix representation the pointers are explicit.[1]

A primary feature of the trie structure is that if S is the set of all key words contained in the trie, then a node p at level i partitions S by defining the subsets of all key words that have the same prefix of length i defined by the path from the root to node p. For example, node four in figure 1 defines the subset of key words that begin with the prefix "th-". An important effect of this feature is that the trie stores the characters of common prefixes once. Thus instead of six characters to store the three words with the "th-" prefix, the trie uses two characters. This means that if (1) the data set stored in the trie is large enough, (2) there is a high percentage of common prefixes, and (3) most null pointers are removed from the structure then the number of bytes needed to store the trie structure will be less than the number of bytes needed to store the original data set. In other words, indexing the data set may *reduce* the amount of memory needed to store a data set.

Toward the goal of exploiting this feature of tries, all trie variations have focused on the various techniques to remove null pointers from the structure. One particular technique, proposed by Dürre and later investigated by Purdin, is called

[1] Within the tree representation node 4 may be stored anywhere in memory; there is no explicit link between node number and storage address. Within the matrix representation trie node 4 will be located at row 4 of the matrix.

(a) tree structure



(b) matrix representation

node	@	a	e	h	i	s	t
0	–	–	–	–	1	–	3
1	(1)	–	–	–	–	2	–
2	(2)	–	–	–	–	–	–
3	–	–	–	4	–	–	–
4	–	5	7	–	10	–	–
5	–	–	–	–	–	–	6
6	(3)	–	–	–	–	–	–
7	–	–	–	–	–	8	–
8	–	–	9	–	–	–	–
9	(4)	–	–	–	–	–	–
10	–	–	–	–	–	11	–
11	(5)	–	–	–	–	–	–

Figure 1: Two trie representations containing words {i, is, that, these, this} with Σ = { @, a, e, h, i, s, t}

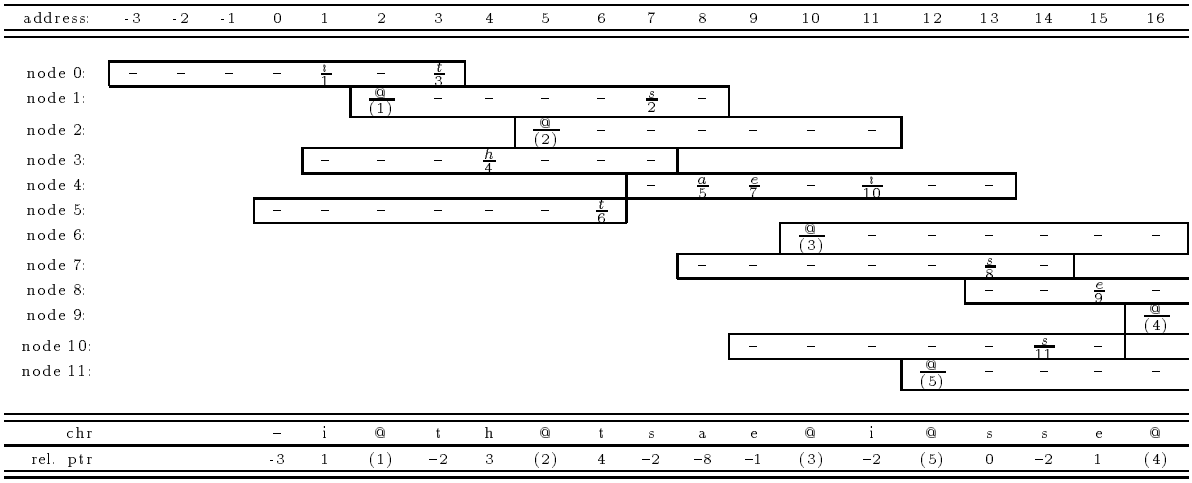


Figure 2: Construction of a Single-Linked Compressed Trie.

Trie Compression. Trie Compression removes null pointers by merging the nodes of the trie into a two-dimensional array. Figure 2 shows the compression of the trie given in figure 1. The first dimension of the Single-Linked Compressed (SLC) trie contains the character associated with each non-null pointer. The second dimension implements the pointers between trie nodes by providing a relative pointer from the current location in the SLC array to the location where the desired node has been placed in the SLC array. For example, node 0 contains a pointer to node 3. This pointer is placed in cell 3 of the SLC array. Node 3 is merged into the SLC array at cell 1. The relative pointer for cell 3 is thus -2. The compressed trie has the access relationship:

$$\text{chr}[\text{i} + \text{ptr}[\text{i}] + \text{ord}(\text{KEY}[\text{i}])] = \text{KEY}[\text{i}].$$

This compression technique works well to remove the null pointers from the trie structure, but the size of the relative pointer when storing large data sets becomes a major factor in the storage requirements of the SLC trie. In general, each relative pointer must be able to point from the first cell to the last cell in the SLC array. Although restricting the order in which nodes are merged into the SLC array can have an effect on the size of the relative pointer by localizing the node of a subtrie to a certain area, results obtained from this technique will be very specific to the particular data set. Realizing that only a relatively small number of SLC cells contain the excessively large relative pointers and that these cells were primarily found in the node at the top of the trie, a technique that separated out the nodes in the top of the trie and individually compressed the remaining subtries was proposed and

investigated. A structure called the VLC trie resulted from these investigations.

VLC Trie

The VLC trie is constructed from a trie by establishing a partition of the trie edge set and a decomposition of the trie node set. In particular, trie T is decomposed into $\{ \tau, t_1, t_2, \cdots, t_n\}$ such that τ contains the top k levels of T and each t_i is a subtrie defined by the edges between the k and (k + 1) levels of the trie. To complete the construction, each t_i is compressed using the smallest cell size possible (cell size is allowed to vary from one subtrie to the next) and each node in τ is converted to a linked list of chained records.

The process of converting the trie nodes in τ into the linked list of chained records in the VLC trie is straight forward. Each non-null pointer in the nodes in τ is represented in a sequential list of records with the first non-null pointer of the root node being placed in the first cell of the list. Non-nulls from the same trie node are chained together by a one bit flag that is set to indicate when the next record in the list is from the same node as the current record. A pointer from one trie node to another is represented by a pointer to the first record of the chained list of non-nulls. Included in the list record is a field that contains the character associated with each non-null pointer[2] and a field that indicates the number of bytes used in a compression of a subtrie.

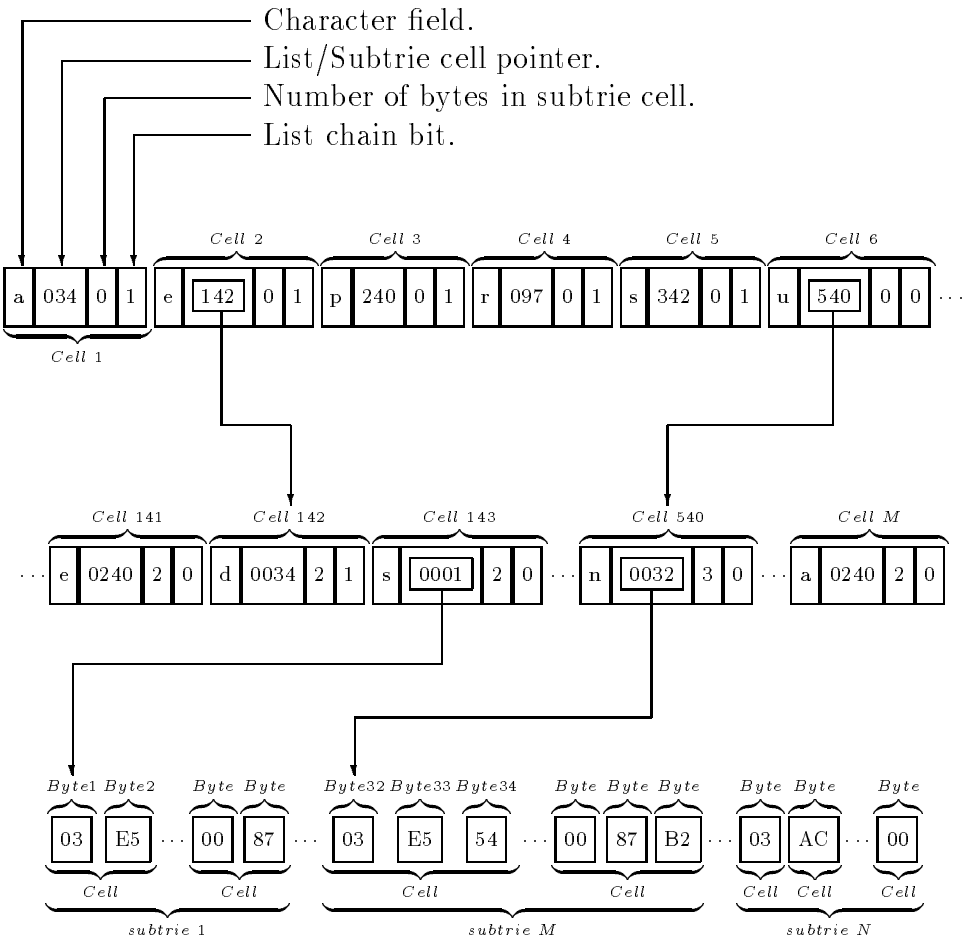
Figure 3 graphically shows the VLC trie structure when the top two levels of the trie are stored in the τ list. The chain bits in the top row of cells indicate that cells 1 through 6 contain all the non-null pointers that were contained in the root node of the trie and that cells 142 and 143 constitute all of the non-null pointers in a secondary trie node. Two paths are drawn in the figure, the left most path represents the path for the prefix "es-" and the right most path represents the prefix "un-". Notice that a linear search of the chained records is needed to determine if a given character is present. The search would terminate in one of three states: (1) the search character is found, (2) an unset chain bit encountered before the search character is found, or (3) character that is larger than the search character is encountered. A search terminated in either state two or three means that the search character is not present and subsequent search is not needed.

At the bottom of figure 3 three compressed subtries are identified. The purpose in identifying these subtries is to stress the fact that each subtrie is compressed with the smallest cell size possible and that the number of bytes per compressed subtrie cell is stored in the linked list structure. List cell 540 indicates that byte 32 in the compressed trie byte stream is the first byte in a compressed subtrie and that each cell is three bytes in length for this particular compressed subtrie.

As a final note about the VLC trie structure, notice that the pointer in the linked list record is either a list cell address or a subtrie byte address and that no explicit flag is given to indicate which type of pointer it is. A flag indicating the type of pointer is not needed since the list structure contains the top k levels of the trie.

[2]This explicit representation of the character is omitted in the standard trie representation since the character can be determined by the placement of the non-null pointer in the array of pointers. The process of removing all null pointers makes the explicit statement of the character necessary.

Tau list structure containing a character field, a list/subtrie cell pointer field, subtrie cell size mask, and list chain bit.



Compressed trie byte stream with variable subtrie cell size.

Figure 3: VLC Trie Structure.

An implementation of the VLC trie could record k separately from the structure and thus know that the first k − 1 pointers will be list cell pointers. Alternatively, the subtrie byte address pointers may be identified without knowing k by realizing that a non-zero subtrie cell size only occurs when the pointer field contains a subtrie byte address.

The access time for determining if word $W = c_1 \cdots c_k c_{k+1} \cdots c_w$ is stored in the VLC trie is composed of two parts.[3] The first part corresponds to accessing the list structure with the prefix $c_1 \cdots c_k$. Traversing the chained linked list requires between

[3]Note: By convention, character c_w of W is required to be the end-of-string character.

k and σk comparisons where σ is the size of the trie alphabet. The second part of the access time corresponds to accessing the compressed subtrie with the suffix $c_{k+1} \cdots c_w$. Since compressing the subtrie does not affect the access time of the subtrie, the access time will be proportional to the number of characters in the suffix which is $w - k$. The number of comparisons needed in the best case situation will be proportional to $k + w - k = w$ which means the best case access time is O(| W |) access time. For the worst case situation, there will be $\sigma k + w - k$ comparisons made, but realizing that both k and σ are constants one sees that the worst case access time for the VLC trie is O(| W |). This means that converting a trie to a VLC trie does not unduly affect the access time of a standard trie.

VLC Trie Performance

The performance of the VLC trie, in terms of calculated storage requirements, were derived using data sets generated from two databases. Both databases contained 351,644 key words. The first database, called WORDS, contained English words, names and abbreviations; while the second database, called NUMBERS, contained unique nine-digit numbers randomly selected from the range 000,000,000 to 999,999,999. Test sets ranged in size from 10,000 to 351,644 key words and were generated from each database by partitioning the database into as many sets as possible for a given test set size. For example, there were 35 test sets containing 10,000 key words while there were only three test sets containing 100,000. Partitioning of the databases into test sets was chosen to ensure that results were not unduly affected by nearly identical data sets. In addition to the data sets generated from one partition of the data base, multiple partitions were used to increase the number of data sets. For data sets containing less than 100,000 key words, ten partitions were conducted[4] while only three partitions were used for the data sets containing 100,000 key words and above.

To calculate the storage requirement for a given data set, a trie was constructed from the data set and for each level of the trie an estimate of the number of bytes to store the top levels of the trie was added to a byte estimate for storing the subtries as compressed tries[5] and the level containing the minimal storage requirement was selected as the result of the test set. To normalize the estimates each minimal estimate is divided by the original size of the data set to get the *cost* of storing the data set as a VLC trie. A cost of one means that the VLC trie needs exactly the same number of characters as the original data set.[6] A cost less than or greater than one means respectively that less or more storage than the original data set was needed.

The average performance of the VLC trie on data sets from the WORDS data base is given in figure 4 while figure 5 shows the results given data sets from the NUMBERS data base. Included in these figures are an upper and lower bound cost estimates of the VLC trie. The difference between the upper and lower bound estimates lies in how the storage requirement for the compressed subtries were calculated.

[4]Thus for the 10,000 key word data set results were collected for 10 partitions each of which contain 35 test for a total of 350 test results.

[5]See [Dürre 93] for process of estimating storage requirements of compressed tries.

[6]Note: The end-of-string character is assumed to be part of each string in the original data set.

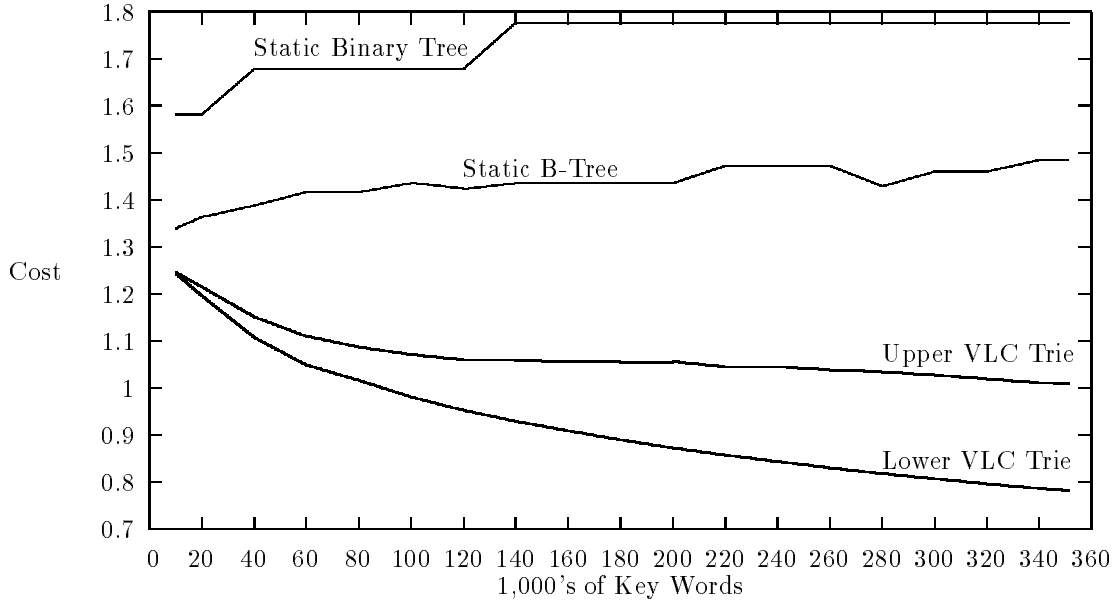


Figure 4: Cost Performance of VLC Trie Structure on Subsets of WORDS.

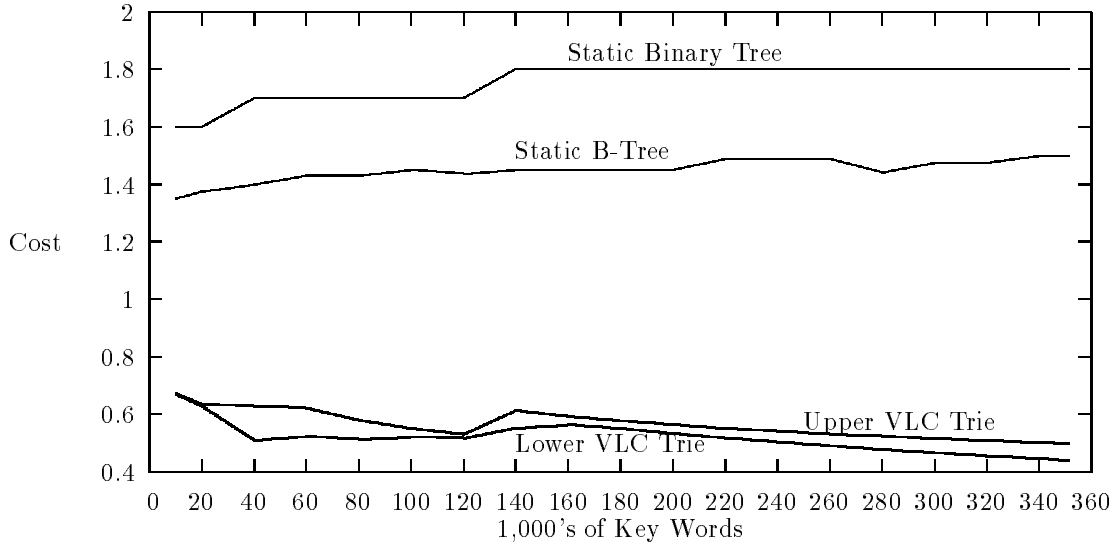


Figure 5: Cost Performance of VLC Trie Structure on Subsets of NUMBERS.

The lower bound VLC trie costs are obtained by assuming that all subtries can be optimally compressed. The upper bound VLC trie costs are obtained by calculating a guaranteed compression size for each subtrie that uses the fact that tries with a specific characteristic can be identified as being optimally compressible. The size used for subtries with this characteristic is optimal while the size of subtries without this characteristic is calculated by assuming a very basic compression that results in a generous compression size. The subtrie compression routine used for the non-optimally compressible subtrie is such that an implementation of the VLC trie that compresses the non-optimal subtries using any aggressive compression scheme will have a cost that is between the upper and lower VLC costs.

The figures include cost calculations for storing the data sets in a static binary tree and a static B-Tree. The storage requirements for these data structures were minimized for the comparison. Minimization for these structures was accomplished primarily by requiring that node records contain an integer number of bytes while fields within the records are bit sequences using the smallest number of bits possible. This means that if the fields of a record need a total of 11 bits then the record size will be 2 bytes. The storage requirements for the B-tree were additionally minimized by searching the different orders and choosing the order which resulted in the smallest storage requirement. These structures are identified as being static because in both cases the number of bits assigned to pointers within these structures is limited.

The performance of the VLC trie is clearly better than that of both the static binary tree and static B-tree. The VLC trie costs exhibit a decreasing cost to increasing set size relationship in contrast to the increasing cost to increasing set size relationship exhibited by the binary tree and B-tree structures. Specifically the upper bound VLC trie costs, when dealing with English word data sets, decrease from 1.25 to 1.01 while for the 9-digit number data sets, the costs decrease from 0.67 to 0.50. The lower bound VLC trie costs for English word data sets decrease from 1.24 to 0.78 while for the 9-digit number sets, the costs decrease from 0.67 to 0.44.

Conclusions

The VLC trie structure addresses the problem of excessive storage requirements of the basic trie structure while maintaining the O($\mid W \mid$) access time for key word W. The VLC trie, when dealing with static data sets, substantially reduces the storage requirements of a standard trie as well as out performing the best performance of binary trees and B-trees. The trade off for this performance is in the static nature of the VLC trie. The VLC trie can index a data set with nearly the same number of bytes as contained in the original data set and in some cases the indexing can be accomplished with less than the number of bytes in the original data set.

References

Al–Suwaiyel, Mohammed Ibrahim, *Algorithms for Trie Compaction*, Ph.D. Thesis at University of Southern California. June 1979.

Al–Suwaiyel, M. and E. Horowitz, *Algorithms for Trie Compaction*, **ACM Transactions on Database Systems**, v.9, n.2, (June) 1984, p.243–263.

Comer, Douglas E., *Trie Structured Index Minimization*, Ph.D. dissertationat Pennsylvania State University, 1976.

Comer, Douglas, *Analysis of a Heuristic for Full Trie Minimization*, **CSD–TR 217 Purdue University**, December 1976.

Comer, Douglas, and Ravi Sethi, *The Complexity of Trie Index Construction*, **Journal of the ACM**, v.24, n.3, (July) 1977, p.428–440.

Comer, Douglas, *The Ubiquitous B–Tree*, **Computing Surveys**, v.11, n.2, (June) 1979, p.121–136.

Comer, Douglas, *Heuristic for Trie Index Minimization*, **ACM Transaction of Database Systems**, v.4, n.3, (Sept.) 1979, p.383–395.

Comer, Douglas, and Ravi Sethi, *Complexity of Trie Index Construction*, **Proceedings of the Foundations of Data Organization, Kyoto, Japan**, May 1985, p.197–207.

de La Braindais, *File Searching Using Variable Length Keys*, **1959 Proceedings of the Western Joint Computer Conference**, p.295–98.

Dürre, Karl P., *Storing Static Tries*, **10th International Workshop WG 84 on Graphtheoretic Concepts in Computer Science**, 13–15 June 1984, Berlin, Germany, p. 125–135.

Glander, Karl W. and Dürre, Karl P., *Estimating Bounds on the Size of Compressed Tries*, (Submitted for publication.).

Fredkin, Edward, *Trie Memory*, **Comm. of the ACM**, v.3, n.9, (Sept.) 1960, p.490–499.

Knuth, D.E., *Sorting and Searching*, second ed., vol. 3 of **The Art of Computer Programming.** Addison-Wesley, Reading Massachusetts, 1973, pp. 481–499.

Maly, Kurt, *Compressed Tries*, **Comm. of the ACM**, v.19, n.7. (July) 1976, p.409–415.

Purdin, T.D.M., *Compressing Tries for Storing Dictionaries,* **Proceedings of the 1990 Symposium on Applied Computing,** Fayetteville, AR, USA, 5–6 April 1990, p. 336–340.