Department of

Computer Science

Estimating Bounds on the Size of Compressed Tries

Karl W. Glander and Karl P. Durre

Technical Report CS-94-114

May 19, 1994

Colorado State University

Estimating Bounds on the Size of Compressed Tries

Karl W. Glander
glander@cs.colostate.edu

Karl P. Dürre
durre@cs.colostate.edu

Abstract

The compression of a trie reduces the storage requirements of the standard trie structure while maintaining the O(| W |), where W is a key word, access time. The problem of estimating the size of a compressed trie without taking the time to perform the compression has hindered the application of compressed tries to index large static databases. This paper addresses this problem by using results analyzed by Tarjan and Yao for storing sparse matrices to define a class of tries called "harmonic" tries. Approximate compression sizes are given for harmonic and non-harmonic tries. These compression sizes are used to show the performance of using Single-Linked Compressed tries and two hybrid data structures that use Single-Linked Compressed tries to store subsets of two large databases.

Keywords

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

Introduction

A trie [3, 5] 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-ary branch in a node at level i is dependent on the symbol located at the (i + 1)st position of the key word. The construction of a trie results in common prefixes being represented by a sequence of trie nodes beginning with the trie's root node. The access time of the trie structure is O(| W |) where W is the key word being sought in the trie. This is unlike most other data structures since the access time is not dependent on the number of key words stored in the trie. The major drawback of the trie structure, the reason tries are not frequently used, is a substantial overhead in storage requirements that arise from the fact that a large majority of the nodes do not require all M pointers.

Various methods have been investigated to reduce the storage requirements of tries. The methods of primary interest are trie compaction in [1, 2, 9, 10] and trie compression[1] in [4, 11]. Although these two methods are different in their general structure, they are similar since both are based on the process of merging nodes of the trie into an array. The process of merging the nodes of the trie takes O(mn) where m is the number of nodes in the trie and n is the number of non-null pointers in the m trie nodes. The time needed to construct a compacted/compressed trie is

[1]See Appendix A for brief review of trie compression

large enough to restrict the use of such tries to situations where the data set to be stored is static. This limitation has greatly reduced interest in the trie data structure. Additionally, interest has not been generated about the trie variations since no study has been conducted concerning the effectiveness of using the trie variations to store a wide range of data sets. Most considerations of tries have avoided the issue of estimating the size of a compressed/compacted trie since the general problem of finding a merging of nodes that results in a minimal array size is an instance of an NP-Complete problem. When storage requirements have been given in the literature, the size of the array containing the merged nodes is determined by performing the merging of the nodes. The large amount of time to merge the nodes into an array has limited the size of the data sets and the number of data sets tested.

The problem of estimating the size of compressed tries without performing the actual merging of nodes is possible. A result of Tarjan and Yao [12] derived for storing sparse compiler tables allows certain tries, labeled harmonic tries, to be identified as being optimally compressible with an O(m) algorithm. This result is presented in the first section. In the second section, an estimate is introduced that provides a compression size estimate for non-harmonic tries. The final section, presents how both compression size estimates are used to direct research efforts away from basic Single-Linked Compressed tries and towards two new trie variations.

Harmonic Tries

In many instances the matrix representation of a trie will be sparse. Tarjan and Yao analyzed the method, proposed by Ziegler in [13], of storing a sparse matrix in a one dimensional array by merging the rows of the matrix while maintaining a row displacement that contains the absolute array address of each row. Let n be the number of non-zero entries in the two dimensional array A, the function n(l), for $l \geq 0$, be the total number of non-zero entries in rows with more than l non-zeros, and the first-fit-decreasing method indicate a matrix compression algorithm whereby rows are merged at the first available position in the compressed array starting with the most dense rows and concluding with the least dense rows. Given the above definitions Tarjan and Yao proved the following theorem:

Theorem 1 *Suppose the array A has the following "harmonic decay" property:*

H: For any l, $n(l) \leq n/(l+1)$

Then every row displacement r(i) computed for A by the first-fit-decreasing method satisfies $0 \leq r(i) \leq n$.

In terms of determining the size of a compressed trie array, this theorem means that if the matrix representation of a trie T, constructed from data set D, satisfies the harmonic decay property then n, the number of non-zero entries in the matrix, will be a good estimate for the number of cells in the compressed trie array.

Theorem 1 specifies only a good estimate, as opposed to an exact value, since the compressed array constructed by the first-fit-decreasing method does not guarantee that the first cell of every node will be located at a different cell in the compressed array. The first-fit-decreasing method, however, allows nodes with the same number of non-zeros to be merged into the compressed array in any order. Should there be an attempt to merge two nodes into the compressed array with the same first cell address then there is more than likely, but not guaranteed, to be another node with the same order that can be used. This is especially true since the lower order nodes will be the nodes that will be merged into the empty spaces in the compressed array[2] and the lower order nodes will have the most alternatives. The harmonic decay property requires that at least half of the non-nulls come from nodes with one non-null.

The procedure to determine if T is harmonic is an O(m) algorithm, m is the number of nodes in the trie, since the data for function n(l) can be captured with a straight forward tree traversal algorithm.

To check the harmonic nature of tries, a database, called WORDS, that contained 351,644 English words, names and abbreviations was used to create test sets that ranged in size from 10,000 to 351,644. Several test sets were generated from WORDS 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 only three test sets containing 100,000. Partitioning of the database into test sets ensures that results were not unduly affected by nearly identical data sets. To further increase the number of test sets, multiple partitions were conducted. For test sets containing 10,000, 20,000, 40,000, 80,000, and 160,000 key words ten partitions were conducted while the remaining data set sizes were partitioned only three times.[3] Every test set generated from WORDS constructed a harmonic trie.

Non-Harmonic Tries

The harmonic nature of tries constructed from data sets[4] from a second database, called NUMBERS that contained 351,644 randomly selected nine digit numbers in the range 000,000,000 to 999,999,999, were checked. The tries constructed from data sets containing 10,000 through 80,000 key words were all harmonic, only two of the nine tries containing 100,000 key words were harmonic, and none of the tries containing 120,000 through 351,644 key words were harmonic. The tries that were non-harmonic did not have the harmonic decay property indicated in theorem 1 because of an over abundance of nodes that contained between two and six non-nulls.

Initial studies of tries constructed from subsets of the WORDS database showed that although the complete trie was harmonic, an analysis of the subtries showed that some subtries were not harmonic. For example, two subtries that were non-harmonic contained:

- 1 node containing 7 non-nulls, 1 node containing 2 non-nulls and 35 nodes containing 1 non-null;
- 3 nodes containing 2 non-nulls and 5 nodes containing 1 non-null.

In both of these subtries the distribution of non-nulls throughout the nodes was not uniform enough to satisfy the harmonic decay property. There were also subtries

[2]Higher order nodes will tend to block each other an thus be appended onto each other.

[3]The 10,000 key word data set results thus consist of $10 \times 35 = 350$ data sets.

[4]Data sets were formed in same manner as indicated for data sets from WORDS.

encountered that did not satisfy the harmonic decay property because of too many high order nodes.

To accurately give the size of a compressed trie given that the trie is non-harmonic without performing the actual compression is, in most cases, impossible since there exists no way to predict how the nodes with two or more non-nulls will fit into the compressed trie array. Nodes with a single non-null can be merged most anywhere there is an empty location in the compressed trie array. A way to provide a reasonable over-estimate of the size was sought and two different techniques for calculating the estimate were finally considered.

Both over-estimates divided the trie nodes into two parts: nodes with exactly one non-null and nodes with two or more non-nulls. The theory behind the first estimate is to initially construct the compressed trie array by appending all the nodes with two or more non-nulls onto the end of the array without trying to merge the nodes into each other. The nodes containing a single non-null are then used to fill as many of the holes as possible or if there are more nodes than holes, append the remaining nodes to the end of the list. The theory behind the second estimate is identical to the first except that instead of simply appending the nodes with two or more non-nulls, these nodes are merged together such that the left-most non-null of the node being merged into the array is placed next to the right-most non-null in the array.

The first estimate was selected for estimating the size of compressed tries. Although the first estimate will, in most cases, give a larger estimate than the second estimate, the first estimate can be calculated from the information gathered to determine if the given trie is harmonic; the second estimate requires the summation of the distances between the left-most and right-most non-null in all nodes containing two or more non-nulls. The first estimate is additionally less likely to have two nodes merged together such that the first cell of both nodes is located at the same location in the compressed trie array since only the nodes with a single non-null are merged into the compressed array.

Establishing Bounds

Realizing that the compression estimates for harmonic and non-harmonic tries are only close approximations of the expected performance, the two estimates can be combined to form approximate bounds on the size of compressed tries. This estimate can be used to determine the feasibility of continuing research into using the compressed trie structure or using a hybrid data structure in which a component of the structure uses compressed tries. If the trie or tries is/are harmonic then there will be a single value that indicates the best possible performance.[5] If the trie or tries is/are non-harmonic or any combination thereof, an upper and lower bound can be given. The upper bound would be obtained by using the appropriate estimate after determining the harmonic nature of the trie(s). The lower bound would be obtained by assuming that each trie is harmonic. For example, compression estimates were

[5]Harmonic tries are optimally compressible. This represents the case where only the non-nulls are represented in the compressed trie array. Since none of the non-nulls can be removed without loosing information this represents a fixed lower bound.

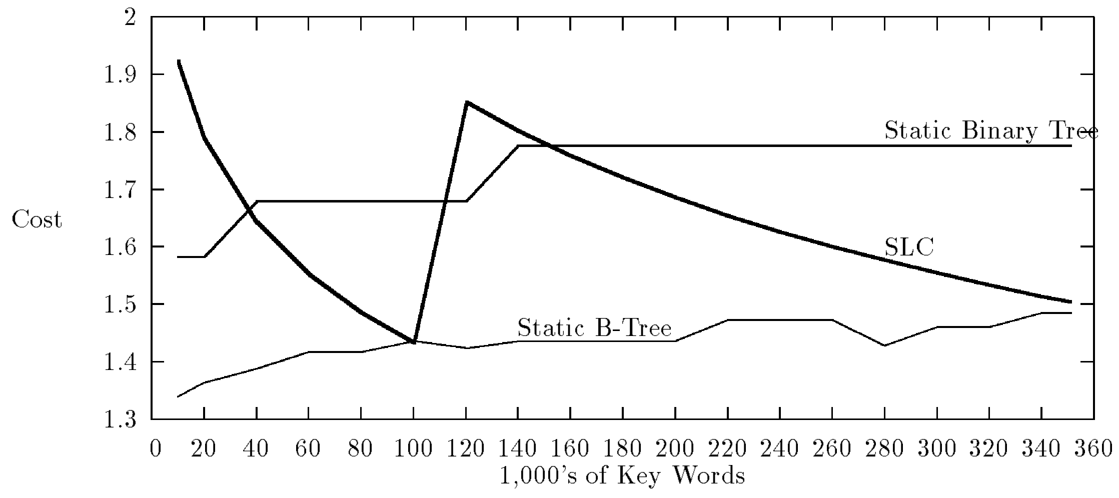


Figure 1: Average Cost of Storing Data Sets from WORDS.

generated for storing as Single-Linked Compressed (SLC) Tries the data sets from the WORDS and NUMBERS databases. The results are graphically given in figures 1 and 2 respectively.

The figures show the average cost of using an SLC trie for storage as a function of data set size. Cost is defined as the ratio of estimated storage size (in bytes) to the measured number of bytes in the original unadulterated data set.[6] To calculate the size of one cell in the SLC trie, the minimum number of bits needed to represent all characters in the trie alphabet was added to the minimum number of bits needed to indicate any of the SLC trie cells.[7] This sum was then expanded to the nearest byte. The storage requirement estimate for an SLC trie is thus calculated by multiplying the SLC trie cell byte size by the compressed trie size estimate.

Included in these figures are cost estimates for storing the data sets in a static binary tree and static B-tree. The storage requirements for these data structures were minimized for the comparison. Minimization 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 is 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.

[6]A cost of 1.0 means that the data structure requires as many bytes to store the data set as a sequential list of key words.

[7]This quantity is the compressed trie size estimate.

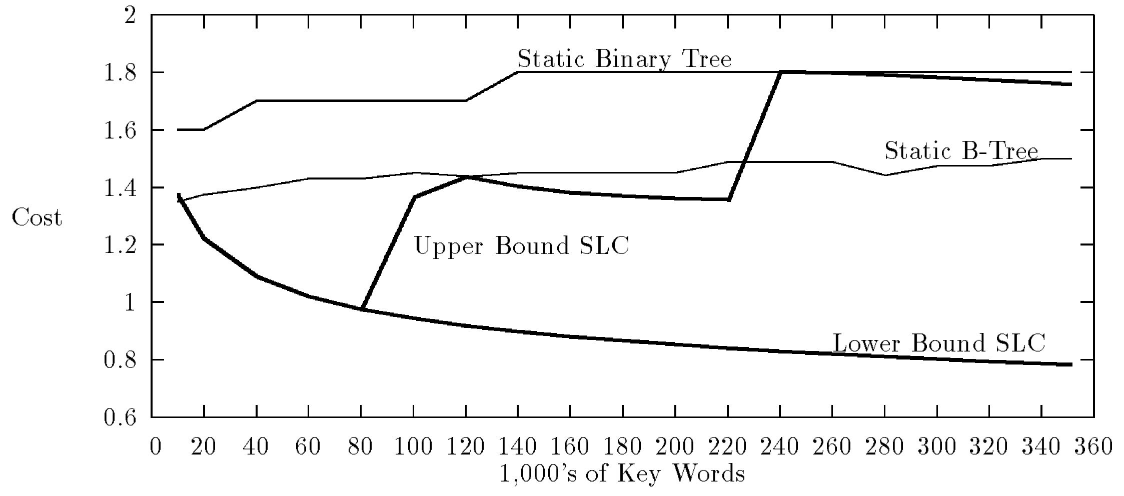


Figure 2: Average Cost of Storing Data Sets from NUMBERS.

The single estimate given in figure 1 is, once again, a result of all data sets from WORDS creating harmonic tries. The SLC trie estimate shows that further investigation into using SLC tries to store English word data sets will be unprofitable since the static B-tree will perform better.

In figure 2 the harmonic tries found in the 10,000 through 80,000 key word data sets results in a single cost estimate while the remaining data sets have a lower and upper bound. The single cost estimate shows that an SLC trie should be considered as an efficient storage structure for data sets containing 20,000 to 80,000 key words. The 10,000 key word SLC trie cost estimate is a little higher than the cost estimate for the static B-tree. Although there is great potential for data sets containing 100,000 key words and above to be stored in an SLC trie with fewer bytes than when stored in a static B-tree, the true cost of using an SLC trie will probably be closer to the upper bound than to the lower bound. This means that the static B-tree will either be slightly better or slightly worse than the SLC trie costs. When the time needed for construction is considered, the static B-tree will require far less time than that needed for the compression of the trie. Under these conditions the use of an SLC trie to store the 80,000 key words and above would not be practical.

The size estimates for harmonic and non-harmonic tries basically indicated that the sole use of SLC tries to store large data sets is impractical. The estimates, however, show that continued investigation would be advisable into two hybrid data structures in which SLC tries are a component. The trie variations called the PL trie and VLC trie apply a divide and conquer approach to minimize the storage requirements of SLC tries by storing subtries as SLC tries and storing the "top" portion of the trie in a linked list structure. Specific details about these trie variations

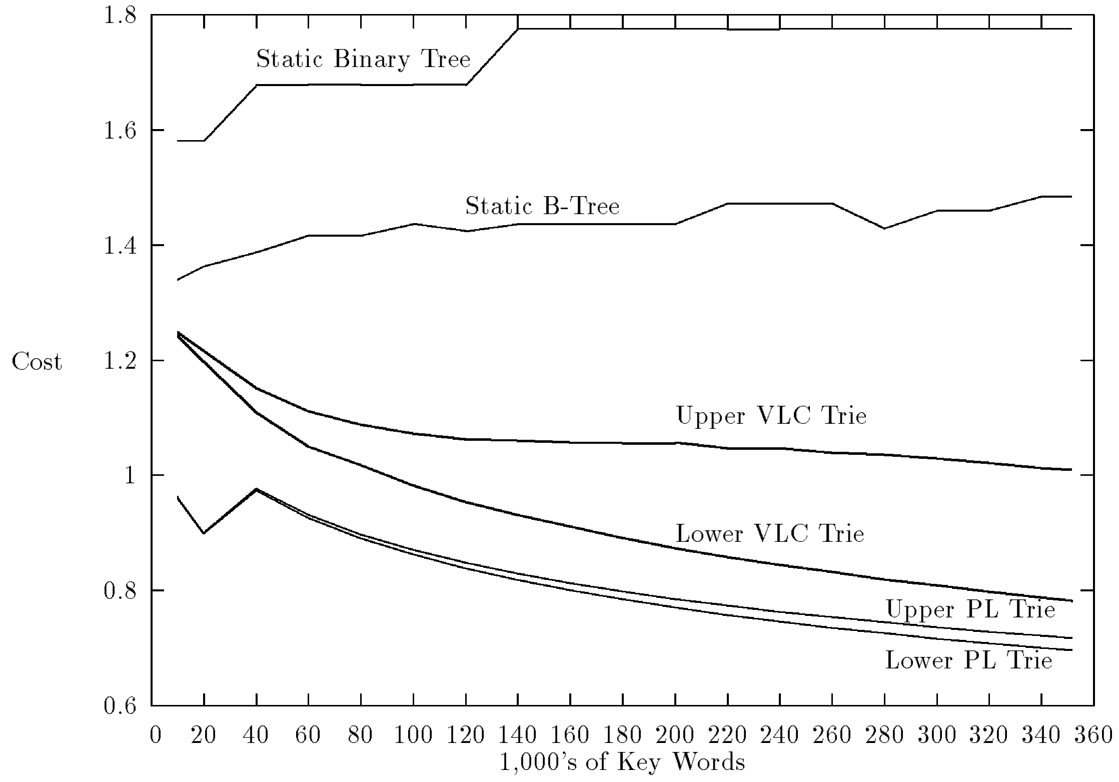


Figure 3: Cost Performance of Trie Variations on Subsets of WORDS.

can be found in [6, 7, 8].[8] Figure 3 shows the average cost performance of the trie variations in relation to the static B-tree and static Binary tree for subsets of WORDS. Figure 4 shows the average cost of the VLC trie variation in relation to the static B-tree and static binary tree for subsets of NUMBERS and figure 5 distinguishes between the average performance of the VLC trie and PL trie.

The cost estimates for the trie variations show a substantial reduction in storage requirements over both the static B-tree and static Binary tree. The approximate number of bytes saved by the trie variations can be estimated from the cost graphs and the facts that the average word length for subsets from WORDS was just over 10 and the word length for subsets from NUMBERS was exactly 10.[9] For subsets from both databases, the subset containing 180,000 key words contained approximately 1,800,000 bytes and a change of 0.1 in cost is a change of 180,000 bytes. Thus the closest the static B-tree comes to approaching the performance of either trie variation is at the 10,000 key word subset of WORDS where the difference between

[8]See Appendix B for brief review of trie variations.

[9]A nine digit number put one character for end-of-string character.

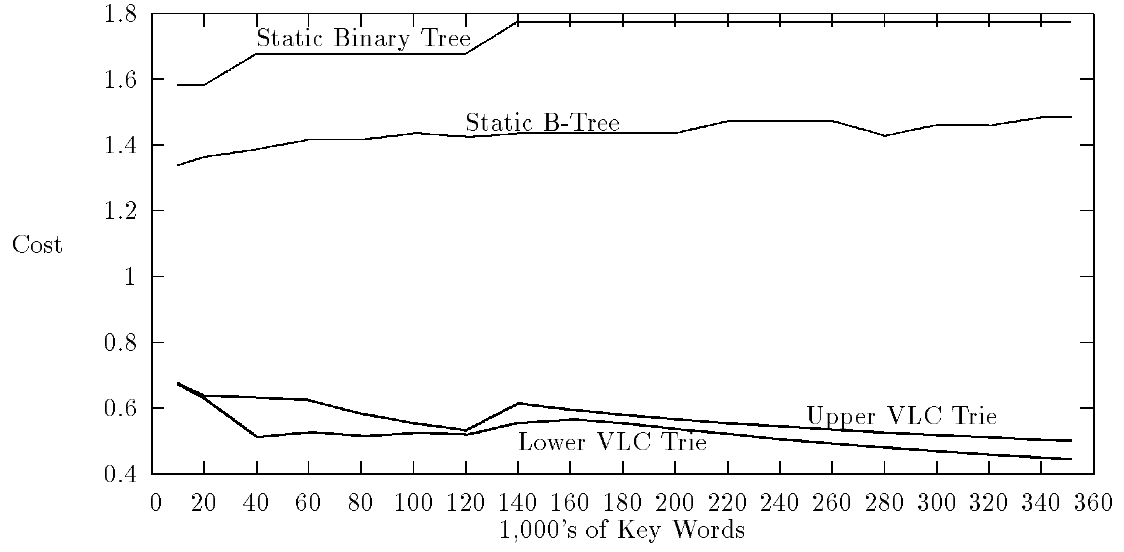


Figure 4: Cost Performance of VLC Trie Structure on Subsets of NUMBERS.

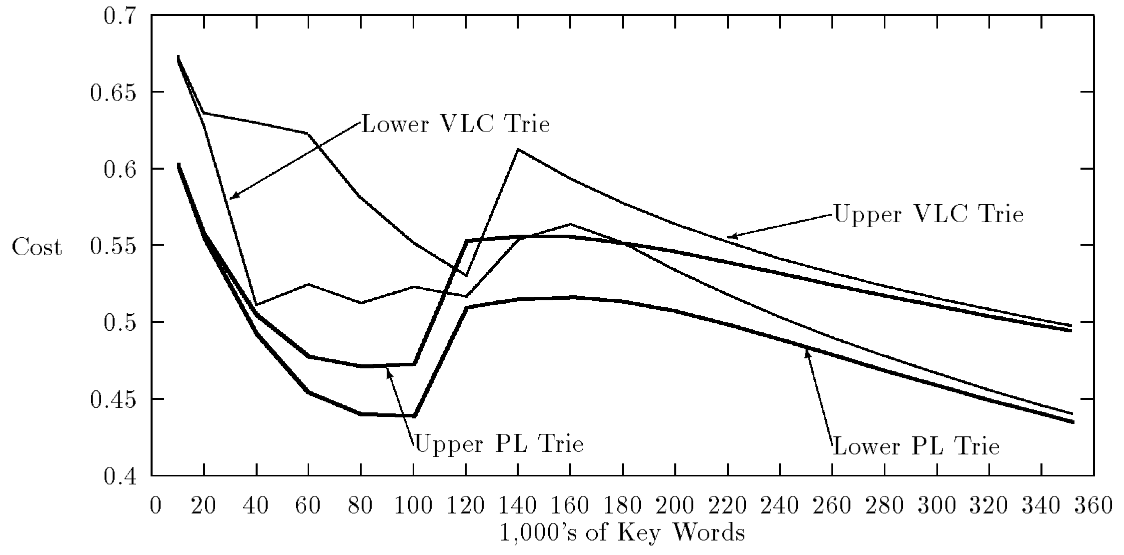


Figure 5: Cost Performance of Trie Variations on Subsets of NUMBERS.

the static B-tree cost and upper VLC trie cost estimate is 0.09 which translates into an approximate 54,000 byte difference in storage requirements. Although the error in the upper bound size estimate is unlikely to be this great, circumstances may arise where the error is in fact this large or larger.

Conclusions

The delineation of harmonic and non-harmonic tries provide the means of defining approximate bounds on the size of compressed tries. Although the estimates themselves do not guarantee specific performance, they can be used to direct research efforts. The bounds confirm the general practice of using B-trees to index large databases instead of using SLC tries. The bounds, finally, indicate that a new avenue of research be directed into indexing large static databases using hybrid data structures in which SLC tries are a component.

Appendix A: Trie Compression Review

A trie may be represented as either a tree or a matrix as shown in figure 6. Trie Compression addresses the issue of excessive storage requirements by removing null pointers through merging the nodes of the trie (as represented by either the tree nodes or the rows in the trie matrix) into a two-dimensional array without pruning any nodes from the trie. Figure 7 shows the compression of the trie given in figure 6. The first dimension of a 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 place in the SLC array.[10] 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: $\mathrm{chr}[i + \mathrm{ptr}[i] + \mathrm{ord}(KEY[i])] = KEY[i]$.

Trie Compression does not dictate the order in which nodes are merged into the compressed trie array. The method does require that no two nodes are merged into the compressed trie at the same location, i.e. the absolute address of the first cell of each node must be unique. To obtain the compression given in figure 7 the nodes of the trie in figure 6were merged into the compressed trie array in the following order: 0, 1, 3, 2, 4, 5, 7, 10,6, 8, 11, and 9. This ordering, unlike other orderings that might be used, results in a compression that is optimal.[11]

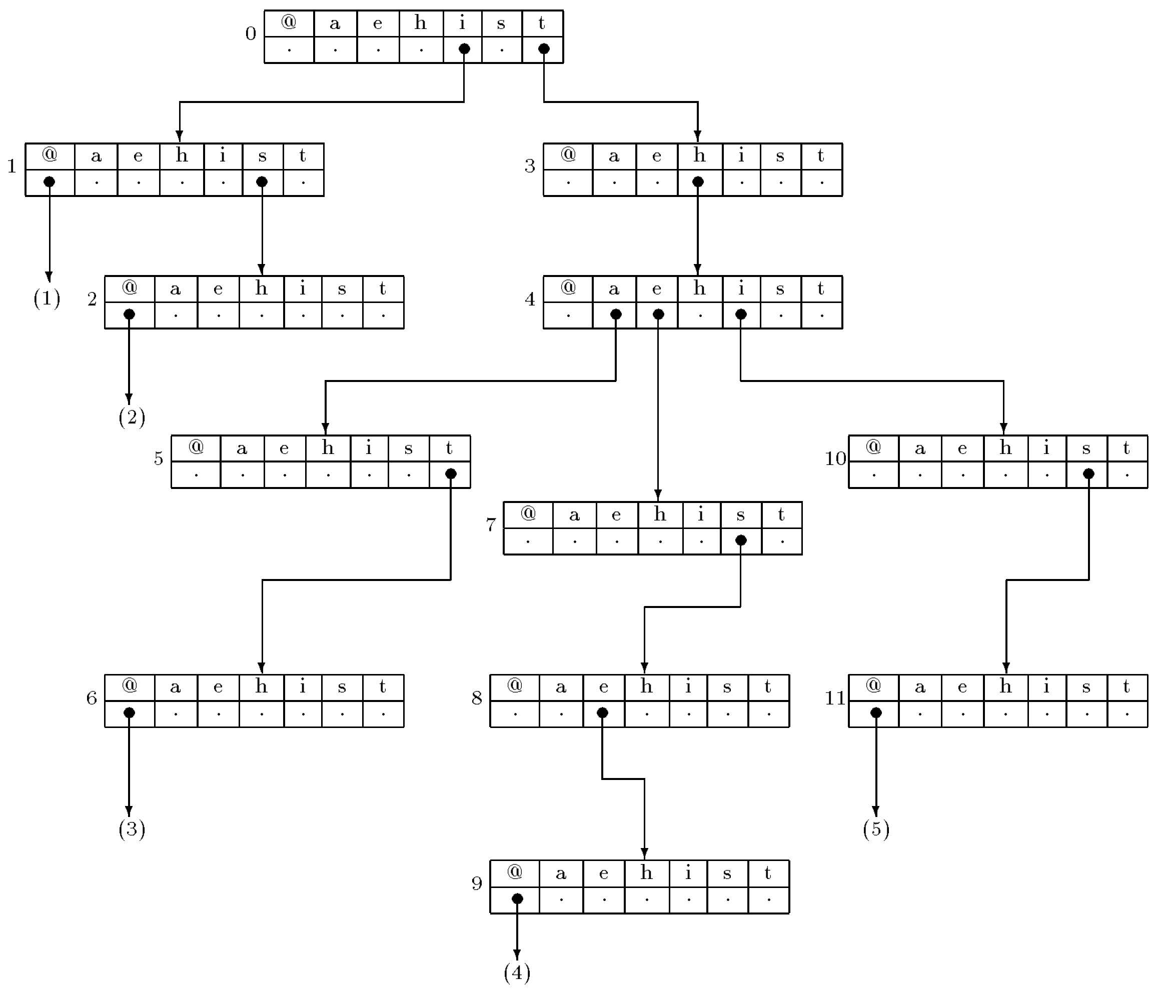
Appendix B: Trie Variation Review

Trie Compression works well to remove the null pointers from the trie structure, but even when the trie satisfies the harmonic decay property, thus guaranteeing a (near) optimal compression, the size of the pointers in the second dimension of the compressed trie array will be quite large when storing large data sets. In general, a relative pointer will have to be large enough to point from one end of the compressed

[10] An absolute pointer may be used with only a slight modification the the SLC trie structure.

[11] The relatively small number of nodes in the trie allowed an optimal compression to be found with only a small amount of work. Finding an optimal compression for larger tries in unlikely.

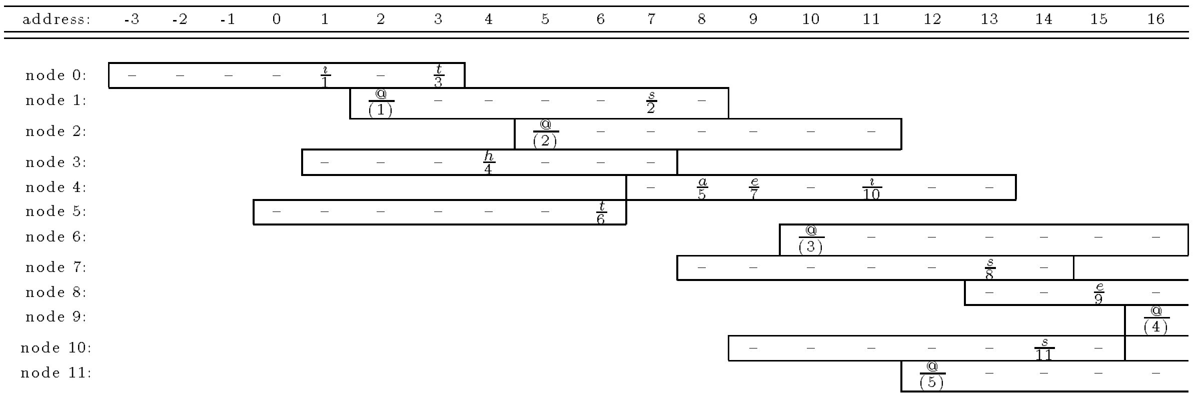
(a) tree structure



(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 6: Two trie representations containing words {i, is, that, these, this} with $\Sigma = \{$ @, a, e, h, i, s, t$\}$



address:	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
chr				–	i	@	t	h	@	t	s	a	e	@	i	@	s	s	e	@
rel. ptr				-3	1	(1)	−2	3	(2)	4	−2	−8	−1	(3)	−2	(5)	0	−2	1	(4)

Figure 7: Construction of a Single-Linked Compressed Trie.

trie array to the opposite end. With the realization that only a relatively small number of pointers in a compressed trie require the large pointers and that these pointers are primarily located in cells that originate from the nodes at the top of the trie, a technique that separates out the top nodes of the trie and individually compresses the remaining subtries was proposed and investigated.

The two trie variations are formally based on establishing a partition of the trie edge set and a decomposition of the trie node set such that trie T is decomposed into $\{ \tau, t_1, t_2, \cdots, t_n\}$ where τ contains the top nodes in the trie and each t_i is a distinct subtrie. The assignment of nodes to τ and which node in T are distinguished as the root nodes of the t_i subtries is specific to each of the two trie variations. In the first variation, called the VLC trie,[12] τ contains the top k levels of T, each t_i is a subtrie defined by the edges between the k and (k + 1) levels of the trie and each t_i is compressed using the smallest cell size[13] possible (i.e. the cell size is allowed to vary from one subtrie to the next). In the second trie variation, called the PLC trie,[14] each t_i contains the maximal number of nodes of T such that t_i is compressible into a compressed trie with a uniform cell size of c and τ is defined to contain the remaining portion of T. The fundamental difference between the VLC and PLC trie structures is that all subtries for the VLC trie structure are required to be from the same level of T while the PLC allows each T_i to be from different levels of the trie.

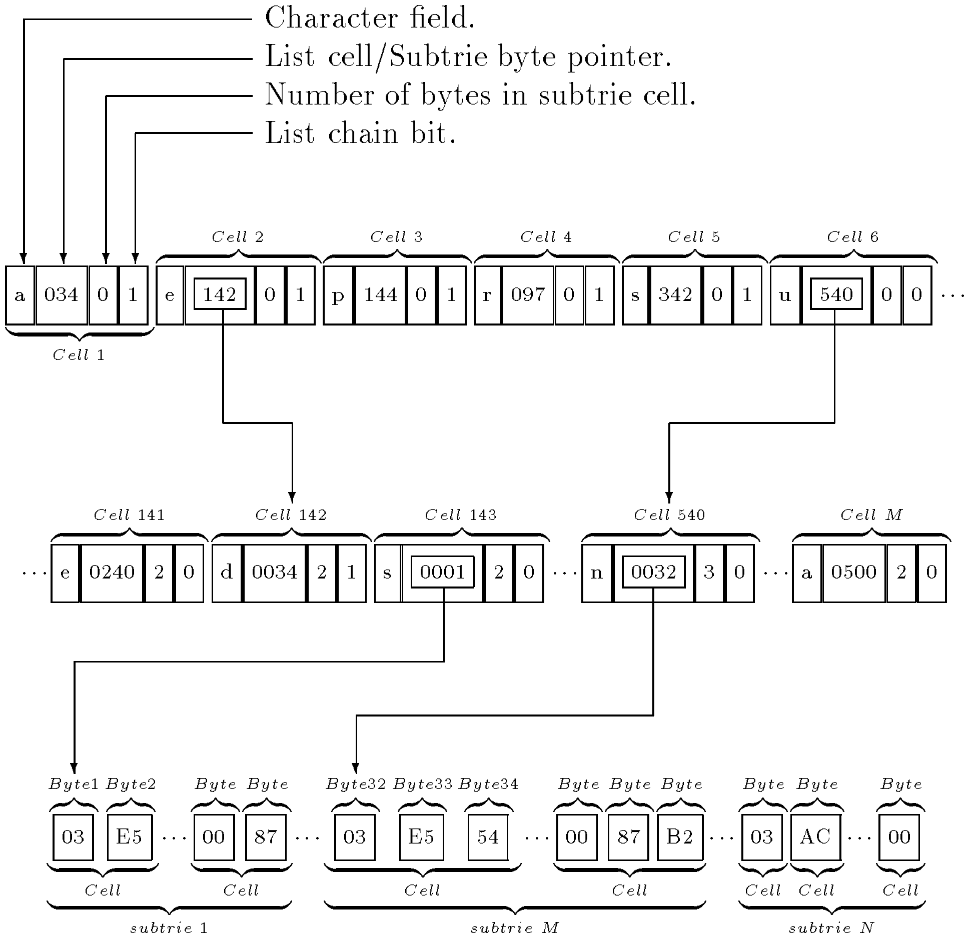
The data structure used to store the trie nodes contained in τ is a linked list of

[12]Named from its outstanding features: Variable subtrie cell size, List structured and Compressed subtries.

[13]The cell size for a compressed subtrie is composed of the minimal number of bits to represent all characters in the trie alphabet added to the minimal number of bits needed for a relative pointer all of which is expanded to the nearest byte.

[14]Name from its outstanding features: Prefix List structured Compressed trie.

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



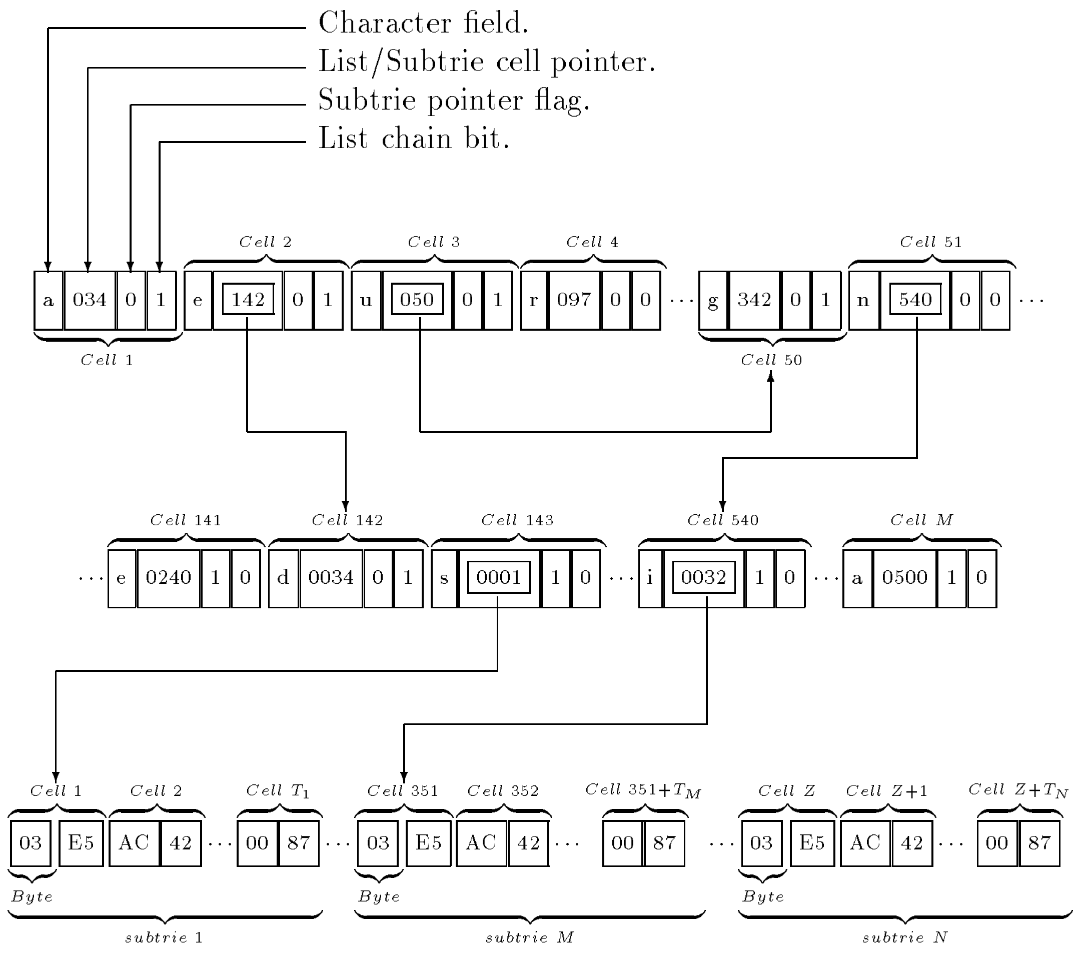
Compressed trie byte stream with variable subtrie cell size.

Figure 8: VLC Trie Structure.

chained records. 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,[15] a pointer

[15]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.

Prefix list structure containing a character field, a list/subtrie cell pointer field, subtrie pointer flag, and list chain bit.



Compressed trie byte stream with constant subtrie cell size.

Figure 9: PL Trie Structure.

field with pointer types specific to the trie variation, and an information field specific to the trie variation. For the VLC trie structure the information field indicates, by using a bit mask, the number of bytes used in storing a specific compressed subtrie while for the PLC trie the information field is a one bit flag that indicates whether or not the cell pointer in the current list cell is a list pointer of a subtrie cell pointer. The pointer field for the VLC trie is a pointer that indicates either a cell in the τ list or a byte pointer to the first byte of a compressed subtrie cell while the pointer field for the PLC trie is a cell pointer that indicates either a τ list cell or a compressed

The process of removing all null pointers makes the explicit statement of the character necessary.

subtrie cell. Subtrie cell pointers can be used in the PLC trie since the calculation of a byte address can easily be made from knowing that all of the subtrie cells are uniform in size.

References

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

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

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

4. 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.

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

6. Glander, Karl W. and Karl P. Dürre, *Minimal Storage Trie Variations*, Submitted to **Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems**, Minneapolis, MN, 24–26 May, 1994.

7. Glander, Karl W. and Karl P. Dürre, *VLC Tries (Extended Abstract)*, **Proceedings of the Data Compression Conference**, Snowbird, Utah, 29–31 March 1994.

8. Glander, Karl W. and Karl P. Dürre, *VLC Tries*, **CS-94-102 Colorado State University Technical Report**.

9. Horowitz, E., and S. Sahni, **Fundamentals of Data Structures**, Computer Science Press, Rockville, MD, 1976, p.517–525.

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

11. 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.

12. Tarjan, Robert E. and Andrew Chi–Chih Yao, *Storing a Sparse Table*, **Comm. of the ACM**, v.22, n.11, (Nov.) 1979, p.606–611.

13. Ziegler, S.F., *Smaller faster table driven parser*, **Unpublished manuscript**, Madison Academic Comptg. Ctr., U. of Wisconsin, Madison, Wisconsin, 1977.