Evaluating Alternative Structures for Prefix Trees

Feras Hanandeh, Izzat Alsmadi and Muhammad M. Kwafha

Abstract— **Prefix trees or tries are data structures that are used to store data or index of data. The goal is to be able to store and retrieve data by executing queries in quick and reliable manners. In principle, the structure of the trie depends on having letters in nodes at the different levels to point to the actual words in the leafs. However, the exact structure of the trie may vary based on several aspects. In this paper, we evaluated different structures for building tries. Using datasets of words of different sizes, we evaluated the different forms of trie structures. Results showed that some characteristics may impact significantly, positively or negatively, the size and the performance of the trie. We investigated different forms and structures for the trie. Results showed that using an array of pointers in each level to represent the different alphabet letters is the best choice.**

***Index Terms*— data structures, indexing, tree structure, trie, information retrieval.**

I. Introduction

Data structures are used to save and retrieve a large amount of aggregated data. They can vary in structure based on the nature or the purpose for having or using such data structure. Performance or the speed of storing and accessing data in those data structures are the main characteristics that can judge the quality of any data structure. Any current natural language such as: English, French, Arabic, etc. can have number of words up to one million words although dictionaries may not contain all such words especially as Languages continuously grow to add new words or borrow words from other languages. Current versions of Oxford English Dictionary may have up to half million words. As such, a software product or web application that needs or use a dictionary should have efficient data structures for effective: storage, access and expansion of data or words.

Prefix trees or tries are data structures that are usually used to store dictionaries or words. Their nature of structure can facilitate retrieving queried words quickly. In each trie, nodes form the children that can further be parents for lower nodes. Nodes contain letters that represent keys or pointers to words (or the rest of the words) at the lowest leaf levels. In principle, each node can contain the searched for word (if it is a leaf). This can dynamically change if more words are added. Finding a word in a trie depends on the size of the ree or the number of words along with its structure. The depth of the nodes that a query can go depends on the number of words in the searched for word. If the word does not exist in the tree, the longest node sequence is performed. Figure 1 show a simple example of trie structure where each node stores an array of alphabet keys or pointers.

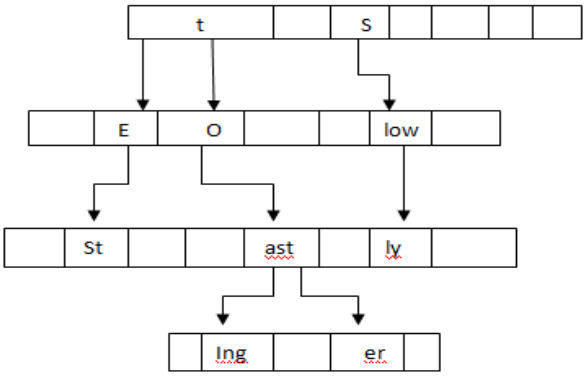

Figure 1: A trie structure for example 1

Figure 2 shows another example of a trie where each node contains the array of alphabets and an underlined letter indicates that this letter is a leaf for at least one word. As we will describe later, using array of pointers can optimize size in many cases.

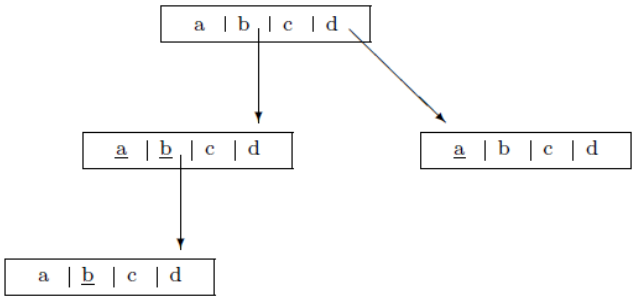

Figure 2: A trie structure for example 2

There are several factors that should be considered when designing a trie. For example:

- The memory size that each node should have affects the total trie size. The size of the whole trie is the aggregated size of all its nodes. Nodes can be empty, store one letter as

Feras Hanandeh , Dept. of Computer Information Systems, Faculty of Prince Al-Hussein Bin Abdallah II For Information Technology, Hashemite University, Zarqa, Jordan, feras@hu.edu.jo
Izzat Alsmadi , Dept. of Computer Information Systems, IT & CS Faculty, Prince Sultan University, Riyadh, KSA, ialsmadi@cis.psu.edu.sa
Muhammad M. Kwafha ,Dept. of Computer & Information Technology, Al- Balqa Applied University, Irbid, Jordan, mohkofahi@hotmail.com

pointer, or store up to certain number of letters (say 20) to represent words. A good trie structure can store the same number of words with minimum size. Of course taking a small number of words is not enough as a good trie structure. It should not be customized based on a limited number of words (for example 10 words that all start with the letters: AB). Such trie may be better in comparison with others based only on this specific set of words.

The node level specifies its location in words. For example, a node with letter E in the second level may represent all words that their second letter is E (e.g. tea, sea, key). If the nodes represent single letters, then examples of match words will only be for example: sea, set, sell, seat, etc.

While in the current large amount of available memories and disk sizes, size may not be a critical factor, however, if such trie can achieve lowering the overall size without impacting other aspects such as speed of storage or retrieval, then this can be still a major comparison factor.

- Trie structure nature: certain structures can facilitate not only quicker access and retrieval of information; they can further facilitate smooth expansion of such structures. In many cases, it is necessary for a dictionary to accept the addition of new words and hence the structure should facilitate this expansion smoothly and dynamically. This can be for either fragmentation or for allocation/reallocation. In some other cases, it maybe necessary to compress, encode or encrypt data in those tree structures.

The performance of the structure can be highly dependent on the dataset of words, the distribution of words, or the number of letters in the words. It can be also language dependent especially if the number of letters in the language alphabet is small or large. Each node of a tree store two kinds of data: user data and trie-related data. Let say alphabet has at most 256 letters, thus letter could be saved in a byte. For space optimization, it is best to store nodes in a pair form (e.g. KeyValuePair<char, TrieNode>). The first part which will take only one charachter stores the key part of the node which is one charachter size. The second part is a node, an object which further can be represented by charachters and nodes. As we will show later on, this is shown to be the best optimized approach to use for building a trie. This can be represented by a template in C++ or generic in Java and C#.

While the subject of prefix trees or tries is not new, new techniques to improve the structure of the trie showed significant improvements. This is largely due to the creation of new programming language components or libraries.

The rest of this paper is organized as follows: Section two presents' related studies to the paper subject, section three shows the proposed methodology, section four presents implementation and experimental results, Last section presents the conclusions and future work.

II. Related Work

Data structures such as trees evolved from the early 19th century. Tries evolved from trees. They were called different names such as: Radix tree, compact trie, bucket trie, prefix tree, Crit bit tree, and PATRICIA (Practical Algorithm to Retrieve Information Coded in Alphanumeric). Some current papers argue that new data structures such as Hash tables or Linked structures can be better alternatives for tries in terms of flexibility and performance.

In the current general form, it is believed that tries were first proposed by Morrison [1].Those different names may have some differences in the detail structure. For example, unlike PATRICIA tree nodes that store keys and words, with the exception of leaf nodes, nodes in the trie work merely as pointers to words [9, 10]. Before Morrison, Fredkin published a paper titled (Trie Memory) [2] describing the trie structure. Fredkin wrote in 2008 “As defined by me, nearly 50 years ago, it is properly pronounced "tree" as in the word "retrieval". At least that was my intent when I gave it the name "Trie". The idea behind the name was to combine reference to both the structure (a tree structure) and a major purpose (data storage and retrieval)”. Looking at the size issue, it is estimated that using PATRICIA trie for 100,000 words, in current typical desktop, trie size can be around 5 MBytes (assuming average size of english keywords of 5 letters.

In [3], Heinz claimed that Burst-trie version of prefix trees is the fastest. This trie tried to further reduce the number of nodes by collapsing similar nodes that share same prefixes. The Buckets or nodes were represented using linked lists. Later papers claimed that Bursts can be further reduced using caches. The main goal or enhancement of Burst trie over traditional trie is in reducing the number of required search cycles to retrieve subject or query.

Tanhermhong et al in [12] proposed a new tri-based structure called (structure-shared trie) with the goal of data compression to reduce size. The main idea is to utilize unused space within the trie structure.

Using tries for approximate string matching discussed by Shang and Merrett in [11]. The proposed approach claimed to have a search process that is independent of the subject document size. The search depends on an approximate match between query and searched for text. Such algorithms can be used for recommendations systems where the information system can suggest alternative approximate terms for users’ queries.

Askitis and Sinha proposed HAT-trie as an alternative better trie representation [4]. This is based on the previous approach: Bucket-trie where Buckets are divided using B-tree splitting. The paper tried to improve Burst-tries through caching and using Hash tables. Authors assembled several datasets of texts for comparison for their data structure with some known ones (i.e. known design structures for tries) and showed improvement in performance and memory size.

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Askitis and Sinha [13] proposed an approach to combine a trie and a Hashtable (Hat trie). The goal is to optimize in memory data usage in terms of efficiency and speed. Askitis and Zobel in [15] discussed optimizing efficient data structures such as Hash tables and Burst tries based on caching. Authors claimed that storage can be significantly reduced.

In [6], Knuth proposed enhancing performance in tries through flexible size pointers of array lists in comparison with original fixed size pointers. This requires non-root nodes to be highly occupied in order for such alternative to be competitive. Nodes can share keys with neighbors rather than split when get full.

Behdadfar and Saidi used trie search for IP routing table search optimization where they tried to scale tries to deal with large size data structures such as those of IP addresses [7]. They tried to sort nodes through encoding their addresses with numbers which can be reached faster based on encoded numbers. They found out that some methods can be optimized for search or query as methods maybe optimized for an addition or update process to the trie table. Bando and Chao tried also to use trie compression for enhancing IP lookup [8]. The proposed Flash trie data structure used compression techniques to reduce the overall size of the trie.

A recent paper, Leis et al in 2013 [14], discussed an approach to enhance Radix trees for efficient indexing in main memory. The approach is compared in terms of performance with Hash-tables. Several other recent papers such as Böhm et al in [15] discussed recent issues related to in-memory trie structures optimization in terms of size, efficiency, performance, etc.

III. Trie Structure Implementation

We tried several alternative structures for trie or prefix trees. The different alternatives are based either on using different variable or data types (e.g. strings, array of strings, characters, array of characters, templates, generics, etc.) or based on the way the trie nodes are formed. The general agreed upon structure for the trie assume nodes in the different levels with alphabet letters in those nodes as prefixes to the words. Each node can hold only one letter from each word. However, if the letter is a leaf, it can hold more than one letter if no split is required. Later on, an addition of a word may require this node of several letters to be further divided. In other words, in a complete full structure, a word of three letters will span over three nodes letter in each node. That does not prevent a single node to be an array of pointers of several different letters as it can be still seen as a single node given the ability of some data types representation to hold an array of pointers. Figure 3 shows a sample trie structure. Not all node elements should be used in a particular trie instance. This is why many recent trie structure approaches tried to optimize space through some compression techniques.

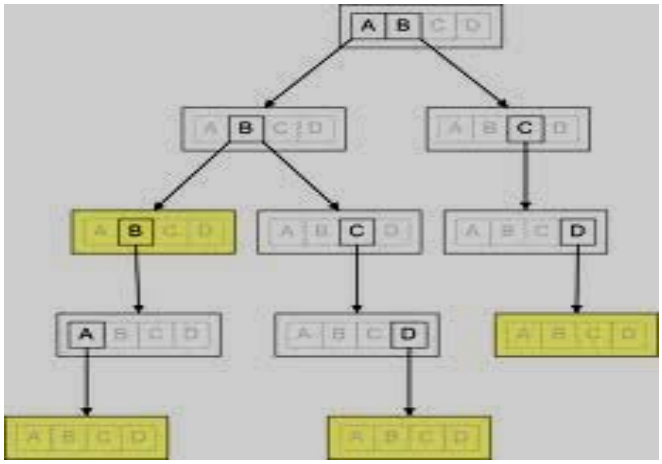

Figure 3: A trie structure

New technologies and data structures impact the usage of older data structures such as trees. As such, we tried to evaluate the impact of utilizing new programming or data type elements in the structure of the trie in terms of size or memory usage and space utilization.
In order to make a fair comparison, we implemented all different alternative implementation in one programming language (.NET C#). For each program, three classes are created, Trie, Node and Main class. In the main class, for each experiment a trie is created, experimental text files are parsed and save in a string array word by word. After that, words from the string array are inserted in the trie nodes through a loop. Size metrics are then collected at the end of the insertion process.

IV. Experiment Results

To investigate the efficiency of the different evaluated trie structures, several text files of words from English dictionary are assembled. They are selected randomly from the dictionary with only size or number of words as the main concern. The small sets are selected all from the start of the dictionary. This means that all those below a certain size (e.g. 2000 words) will be from the first English letter (A). As tries or prefix trees structure depends on the first letters of the words or prefixes, selecting all words that start from one letter may indicate different results from selecting a set of words that span over all alphabet letters. Later random selection of words from all dictionary locations will be implemented to ensure a fair distribution of words. In the largest tested size, we tried to a dictionary file of 349,900 words or 3.473 MByte memory size. Table 1 below summarizes files selected for testing, number of words and approximate memory size. Memory size is approximate since we selected each test set based on the number of words and repeat the process several times where some selections maybe slightly different from others based on the selected words.

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE 1
NO OF WORDS AND SIZE OF EVALUATED DICTIONARY FILES

No. Of words	Size (MByte)	No. of selections
100	0.001	5
500	0.006	5
2,000	0.02	5
5,000	0.051	5
10,000	0.1	5
72,858	0.65	1
349,900	3.47	1

Size of tries and nodes is measured based on debugging source code and also using memory profilers.

In the first experiment three different alternative trie structures (prog1, prog2, and prog3) were experimented on small data files. For each selected size (i.e. 100, 500, 2,000, 5,000, and 10,000 words), 5 different selections are taken and average values of those five selections is considered. The first two columns (size of created nodes, and number of nodes) were collected from memory profilers while the last two (root direct children and trie size) were collected from code debugging.

TABLE 2
SUMMARY OF RESULTS FOR THE FIRST EXPERIMENT

No. Of words (program)	Size of created Nodes (Kbyte)	No. of nodes in each level (max)	Root direct children	Trie size: children/Nodes
100(1)	4220.8	263	1	32
100(2)	4220.8	263	27	9
100(3)	5276	263	1	32
500(1)	20985.6	1311	1	85
500(2)	20985.6	1311	27	16
500(3)	26232	1311	1	85
2000(1)	84064	5254	1	365
2000(2)	84,064	5254	27	58
2000(3)	105,080	5254	1	365
5000(1)	213,843	13365	1	786
5000(2)	213,843	13365	27	115
5000(3)	267,304	13365	1	786
10000(1)	415,187	25949	1	1278
10000(2)	415,187	25949	7	174
10000(3)	518,984	25949	1	1278

Results in Table 2 shows that trie structure of program two is better in terms of size utilization for all text file sizes. Number of nodes in each level column has the same value for all program alternatives as this has to do with the general trie-node structure which is similar. However, the first column on the right (trie size) shows significantly why program two is the best in terms of size utilization. Trie size is a method we defined (Figure 4) to show the total number of actually utilized nodes. Notice that this number changes based on stored data. This value is the same for programs 1 and 3. The value of actual trie size is significantly reduced in program 2 when size of text file is large.

The key to the difference between program 2 at one side from programs 1 and 3 at another side can be seen looking the second column from the right (number of root direct children). Program two creates for each trie level nodes of the number of alphabets regardless of the size or the nature of the data. However, for programs 1 and 3 and since all selected data in Table 2 are from the letter (A) in the dictionary, those two programs have one child node from the root which is the node of the (A) letter. Program 2 created 27 nodes even if one node or letter is utilized. However, each node stores two elements: a key of type character for the key letter in the node and 27 elements of type Node. The utilization of this structure can be further seen if we selected words from all letters of the dictionary.

In the second experiment three different alternative trie structures (prog1, prog2, and prog3) were experimented on large files. We used the two large files (i.e. 72,858 and 349,900 words) to evaluate the difference between the three different alternative structures. Table 3 below shows the summary of results.

```
int getsize()
{
    int counter = 1;
    foreach (TrieNode node in root.nodes) {

    if (node !=null && node.nodes != null) {
            counter++;
       foreach (TrieNode node1 in node.nodes)  {
       if (node1 != null && node1.nodes != null) {
                    counter++;
       foreach (TrieNode node2 in node1.nodes) {
     if (node2 != null && node2.nodes != null) {
         counter++;

        }  }   }  }   }
    }
    return counter;
}
```

Figure 4: Trie-size: Number of utilized nodes

Figure 5 and 6 show snapshots from ANTS memory profiler. Two of the column variables are collected from this profiler: Nodes size and node or live instances. This is repeated for every tested file. In Figure 5 TrieNode [] is an array of nodes that was implemented in one version of the programs [program 2].

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Namespace	Class Name	Live Size (bytes)	...	Live Instances
System	UInt32[]	52		1
System.Reflection	TypeFilter	64		2
System	Type[]	124		3
ConsoleApplicatio...	**Tries**	12		1
ConsoleApplicatio...	**TrieNode[]**	3,015,000		25,125
ConsoleApplicatio...	**TrieNode**	402,000		25,125

Figure 5: A snapshot view from ANTS memory profiler: array nodes

Namespace	Class Name	Live Size (bytes)	S...	Live Instances
System	UInt...	52		1
System.Reflection	TypeFilter	64		2
System	Type[]	32		2
ConsoleApplication1	**TrieNode2**	402,000		25,125
ConsoleApplication1	**Trie2**	16		1
System.Security.Util	Tokenizer+St...	32		1
System.Security.Util	TokenBasedSet	252		7

Figure 6: A snapshot view from ANTS memory profiler: single nodes

Table 3 shows a summary of results for the second experiment with the large size text files. The experiment is conducted using the three developed versions of Trie. Table 3 shows summary of results. We focused only on code data since used memory profiler showed inconsistent results for large size data structures.

TABLE 3
SUMMARY OF RESULTS FOR THE SECOND EXPERIMENT

No. Of words (program)	Root direct children	Trie size: children/Nodes
72,858(1)	27	14,561
72,858(2)	27	3,301
72,858(3)	27	14,561
349,900(1)	27	49,394
349,900(2)	27	7,945
349,900(3)	27	7,931

Table 3 shows that all alternative programs used the 27 first level nodes due to the large size used datasets. Size in program 2 is significantly less than the size of the other two programs in most cases.

As mentioned earlier, results can vary based on the nature of data and the hardware or computer components used in the experiments or evaluation.

V. CONCLUSIONS AND FUTURE WORK

The continuous development and production of new hardware and programming components and data structures have impacts on several quality attributes related to software products. In this paper, we evaluated different alternative structures for prefix trees or tries. We used three different versions to implement the structure of tries. We then assembled several testing documents with different sizes that include words from English dictionary. Metrics related to size and usage of memory and computer resources are collected for all investigated test files. Results showed while all different evaluated versions share the same common trie structure, yet using some of the new programming components can significantly improve the trie structure and optimize its memory usage. While in this paper, we tried to focus only on one aspect related to size, several other aspects should be evaluated related to the speed of trie data insertion and update. The speed of query retrieval is also important especially for some applications such as library or web indexers. In the current large size applications and data, using a proper data structure that can optimize memory and resources usage and accelerate the speed of adding and retrieving data is always important and necessary.

REFERENCES

[1] Morrison, D. (1968) ' PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric', *Journal of the ACM (JACM)*, Vol. 15, No. 4, pp. 514-534.

[2] Fredkin, E. *(1960)* 'Trie Memory', *CACM*, Vol. 3, No. 9, pp. 490-499.

[3] Heinz, S., Zobel, J. and Williams, H. (2002) 'Burst tries: a fast, efficient data structure for string keys', *ACM Transactions on Information Systems (TOIS)*, Vol. 20, No. 2, pp. 192-223.

[4] Askitis, N. and Sinha, R. (2007) 'HAT-trie: a cache-conscious trie-based data structure for strings', *ACSC '07 Proceedings of the thirtieth Australasian conference on Computer science*, Vol. 62, pp. 97-105.

[5] Askitis,N. and Zobel,J(2011).' Redesigning the String Hash Table, Burst Trie, and BST to Exploit Cache', *ACM Journal of Experimental Algorithmics*, Vol . 15, No. 1.

[6] Knuth, D. (1998) The art of computer programming, 2nd ed., Redwood City, CA, USA.

[7] Behdadfar, M. and Saidi, H.(2008) ' The CPBT: a method for searching the prefixes using coded prefixes in B-tree' , *NETWORKING'08 Proceedings of the 7th international IFIP-TC6 networking conference on AdHoc and sensor networks, wireless networks, next generation internet*, pp. 562-573.

[8] Bando, M. and Chao, J.(2010)' FlashTrie: Hash-based Prefix-Compressed Trie for IP Route Lookup Beyond 100Gbps', *IEEE Communications Society subject matter experts for publication in the IEEE INFOCOM 2010 proceedings*, pp. 1-9.

[9] Knizhnik, K. (2008) 'Patricia Tries: A Better Index For Prefix Searches', *Dr. Dobb's Journal*.

[10] Knessl,C. and Szpankowski,W.(1999) ' Limit laws for heights in generalized tries and PATRICIA tries'.

[11] Shang, H. and Merrettal, T. (1996) 'Tries for Approximate String Matching', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 4, pp. 540-547.

[12] Tanhermhong, T., Theeramunkong, T. and Chinnan, W. (2001) 'A Structure-Shared Trie Compression Method', *Proceedings of The 15th Pacific Asia Conference*.

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

[13] Askitis, N. and Sinha, R. (2010) 'cache and space efficient tries for strings', *The VLDB Journal — The International Journal on Very Large Data Bases*, Vol. 19, No. 5, pp. 633-660.

[14] Leis, V., Kemper, A. and Neumann, T.(2013) ' The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases', *29th IEEE International Conference on Data Engineering (ICDE 2013).*, Brisbane, Australia, pp.38-49.

[15] Böhm, M., Schlegel, B., Volk, P.B., Fischer, U., Habich, D. and Lehner, W., (2011) 'Efficient in-memory indexing with generalized prefix trees', *In BTW*, pp. 227--246.

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)