Compact Radix B-trees as an Efficient Index

NEIL A. EDELMAN

2021-10-20

Abstract

A data structure based on a compact radix trees is introduced, with an index on key strings bits stored semi-implicitly. B-tree methods are used to group data in a sequential conglomeration, called a tree in a trie-forest. This increases cache-coherence, and presents an efficient ordered string set or map with prefix-matching capabilities. Tests comparing it to a hash map show the structure performs for . . .

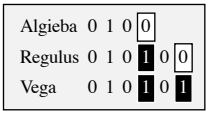
1 INTRODUCTION

A trie is, "a multiway tree structure that stores sets of strings by successively partitioning them [de la Briandais 1959; Fredkin 1960; Jacquet and Szpankowski 1991],"[1] so that, "INSTEAD OF BASING a search method on comparisons between keys, we can make use of their representation as a sequence of digits or alphabetic characters."[2] It is ordered, and allows prefix range queries.

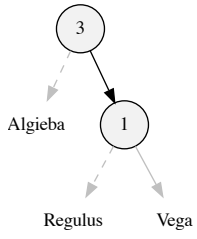
In practice, we talk about a string always terminated by a sentinel; this is an easy way to allow a string and it's prefix in the same trie[3](but not really). For proper ordering, we traverse with most-significant bit first and the sentinel should be less-than the other numeric value of all the other characters. We use null-terminated, Modified UTF-8 strings, like in Figure 1a. Keys are sorted in lexicographic order, but not by any collation algorithm. show a diagram

2 IMPLEMENTATION

A compact representation is a Patrica trie[4]: that is, a binary radix tree and skip values when bits offer no difference. In Figure 1b, the branches indicate a `do not care` for



(a) Bit view.

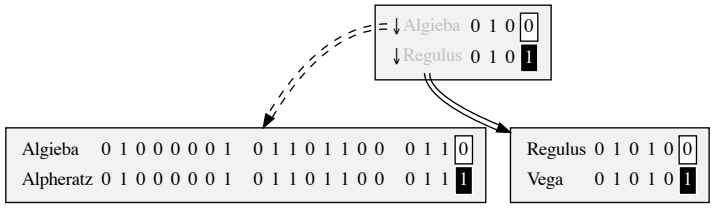


(b) Logical view.

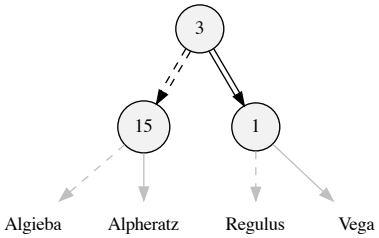
left	0	0	
skip	3	1	
leaves	Algieba	Regulus	Vega

(c) Memory view.

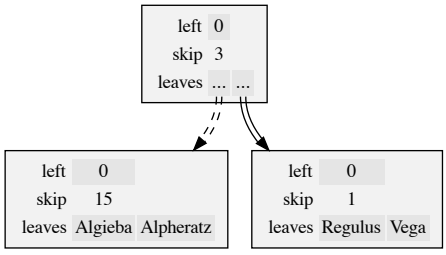
Figure 1: A full 3-trie.



(a) Bit view.



(b) Logical view.



(c) Memory view.

Figure 2: Split the 3-trie to put four items.

all the skipped bits before, corresponding to Figure 1a that offer no delineation. If a query might have a difference in the skipped values, one could also check the final result of a query for agreement with the found value.

instead of adding in one big trie, where the data type presents a limit of how big it is, we have pointers to sub-blocks. This is seen in ... This allows us to shrink the data-type significantly; for a non-empty complete binary tree that has ‘n‘ leaves (order ‘n‘), it has ‘n - 1‘ the internal nodes, or branches. The maximum that the left branch count can be is when the right node of the root is a leaf, that is ‘n - 2‘. If we set this left-branch maximum to the maximum of the data type, we can use all the range. Practically, we set it to 254 instead of 255; the branches take up 255, and the branch size number is 1, for alignment.

This is a fixed-maximum size trie within a B-tree. Because of overlapping terminology, some care must be used in describing the structure. We use the notion that a trie is a forest of trees. In B-tree terminology, nodes, are now trees, such that a complete path through the forest always visits the same number of trees. However, since it’s always the root instead of the middle item, we are not guaranteed fullness. Depending on the implementation, a complete path might be truncated; for example, `Sol` might require less than `Betelgeuse`, because it’s shorter.[ref b-tree]

2.1 Trees

Patrica trie is a full binary tree. Non-empty. Issues with having zero, one, zero, one, zero, items always needing to allocate memory. Fixme: store the order instead of the branches.

2.2 Structure in Memory

Like a node in a B-trie, a tree structure is a contiguous chunk. As an example, refer to Figure 1c. A tree whose *size*, the number of nodes, is composed of internal nodes, *branch*, and external nodes, *leaf*; $size = branch + leaf$. As a non-empty complete binary tree, $leaf = branch + 1 \to size = 2branch + 1$. We arrange the branches pre-order in an array, thus, forward read in topologically sorted order. If the number of branches remaining is initialized to $b = branch$, we can make the tree semi-implicit by storing the *left* branches and calculating $right = b - left$. When $b = 0$, the accumulator, l, is the position of the leaf. This is a succinct encoding[?]. Programme.

The above wastes half of *left*’s dynamic range, in the best case, because branches that are e from the end will have at most $left = e$ (such that the end will always be zero): a tree with all left branches.

Separate the leaves, which are pointers to data or to another tree, from the branches, which are decision bits in the key string passed to look up. Only one leaf lookup is performed per tree.

2.3 Machine Considerations

The data has been engineered for maximum effectiveness of the cache in reading and traversing. That is, the tree structure and the string decisions have been reduced to

each a byte and placed at the the top of the data structure of the tree. Always forward in memory. The size of this sub-structure should be a multiple of the cache line size, while also maximizing the dynamic range of *left*; a trie (also a B-tree) of order 256 is an obvious choice.[5]

2.4 Running Time

[6] $\log n == m$.

The 'Ø(log n)' running time is only when the trie has bounds on strings that are placed there. Worst-case $\{a, aa, aaa, aaaa, ...\}$. The case where the trie has strings that are bounded, as the tree grows, we can guarantee . . .

Tries are a fairly succinct encoding at low numbers, but at high numbers, the internal trees add up. In the limit, with bounded strings, we need $n \log_{\text{order}} n$ to store n items? However, practically, if one sets the order high enough, this is not an issue?

2.5 Limits

The skip value is limited by its range; in this case, 255 bits. For example, this trie is valid, $\{dictionary\}$, as well as, $\{dictator, dictionary, dictionaries\}$, but one ca'n't transition to, $\{dictionary, dictionaries\}$, because it is too long a skip value. There are several modifications that would allow this, but they are out of this scope. (This is not true; 8 x 8 = 64; 8 x 32 = 256.)

Insertion: Any leaf on the sub-tree queried will do; in this implementation, favours the left side.

on split, do we have to go locally and see if we can join them?

we don't need to store any data if the leaf-trees are different from the branch-trees?
$trie : root, heightbranchtree : bsize, +branchtree, branch[o-1], leaf[o]leaftree : bsize, ?, branch[o-1], is_recursive_bmp[o/8], leaf[o]orleaftree : bsize, ?, branch[o-1], leaf[o]skip, uniondata, trieorleaftree : bsize, ?, branch[o-1], leaf[o]32 : skip, 32 : height, 64 : root/64 : data$

ASCII and other encodings don't prioritize splitting the glyphs evenly between bits, therefore with a uniform distribution of strings, it's very unlikely that the trie will be balanced. On our trie, this results in wasted space.

REFERENCES

[1] N. Askitis and J. Zobel, "Redesigning the string hash table, burst trie, and bst to exploit cache," *Journal of Experimental Algorithmics (JEA)*, vol. 15, pp. 1–1, 2011.

[2] D. Knuth, "Sorting and searching. third edn. volume 3 of the art of computer programming," 1997.

[3] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[4] D. R. Morrison, "Patricia—practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM (JACM)*, vol. 15, no. 4, pp. 514–534, 1968.

[5] R. Sinha and J. Zobel, "Cache-conscious sorting of large sets of strings with dynamic tries," *Journal of Experimental Algorithmics (JEA)*, vol. 9, pp. 1–5, 2004.

[6] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.