Compact binary prefix trees

NEIL A. EDELMAN

2021-10-20

Abstract

Our prefix-tree, digital-tree, or trie is an ordered set or map with key strings. We build a dynamic index of two-bytes *per* entry, only storing differences in a compact binary radix tree. To maximize locality of reference while minimizing data range while descending the trie, these are grouped together in a forest of fix-sized trees.

1 INTRODUCTION

A trie is a tree that stores partitioned sets of strings[1, 2, 3, 4] 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 [directly].[5]" It is necessarily ordered, and allows prefix range queries.

Often, only parts of the key string are important; a radix trie (compact prefix tree) skips past the parts that are not important, as [6]. If a candidate key match is found, a full match can be made with one index from the trie.

For most applications, a 256-ary trie is space-intensive; the index contains many spaces for keys that are unused. Various compression schemes are available, such as re-using a pool of memory[1], reducing our encoding alphabet, or take smaller than 8-bit chunks[2].

We use a combination binary radix trie, described in [7] as the PATRICIA automaton. Rather than being sparse, a Patricia-tree is a packed index. Recursively, it encodes which is next distinguishing bit and how many keys are on the zero path and the one path.

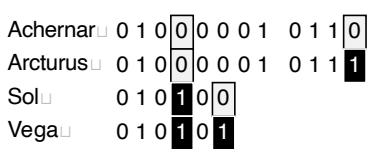
In practice, we talk about a string always terminated by a sentinel; this is an easy way to allow a string and it's prefix in the same trie[7]. In C, a NUL-terminated string automatically has this property, and is ordered correctly. Keys are sorted in lexicographic order by numerical value; `strcmp`-order, not by any collation algorithm, (although, that could be an added complication.)

2 IMPLEMENTATION

2.1 Encoding

Figure 1a is a visual example of a Patrica trie[7], that is, a binary radix tree and skip values when bits offer no difference. Note that, in ASCII and UTF-8, `A` is represented by an octet with the value of 65, binary 01000001; `c` 99, 01100011; `r` 114, 01110010; `S` 83, 01010011; `V` 86, 01010110.

We encode the branches in pre-order fashion, as in Figure 1b. Each branch has a `left` and a `skip`, corresponding to how many branches on the left, and how many

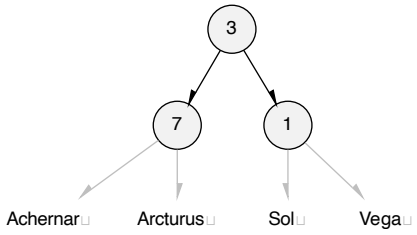


(a) Bit meaning.

Vakgimbat ∑bit=0

left	*skip*	*leaves*
1	3	Achernar
0	7	Arcturus
0	1	Sol
		Vega

(b) Memory.



(c) Tree.

Figure 1: A trie with three different views of the data.

bits we should skip before the decision bit. With the initial range set to the total number of branches, it becomes a matter of accumulating leaf values for the right branches of a key, accessing the index skip-sequentially, until the range is zero. The right values are implicit in the range. The leaves, on the other hand, are alphabetized, in-order. There will always be one less branch than leaf; that is, this is a full (strict) binary tree with $order - 1$ branches, for $order$ keys as leaves.

Figure 1c shows the conventional full binary tree view of the same data as Figure 1a and 1b. The branches indicate a `do not care` for all the skipped bits. If a query might have a difference in the skipped values, one can also check the final leaf for agreement with the found value.

2.2 Range and locality

Only when the algorithm arrives at a leaf will it go outside the `left-skip` index. This suggests that the index, and the size of the trie, be contiguous and as small as possible to fit the maximum into cache. However, in establishing a maximum `skip` value, one limits the contiguous bits that can be skipped; this has an effect on both on insertion and deletion.

The maximum `left` plus one is the maximum number of leaves in the worst-case of all-left. It also is inefficient to insert more and more leaves, the addition requiring more branches to be augmented and an array insertion of the leaf.

For these reasons, we have broken up the trie in much the same manner as [8].

Except in tries, contrary to B-trees, the data can not be rotated at will; instead, it uses a bitmap of which leaves are links to other structures. We use that a trie is a forest of non-empty complete binary trees. This tree corresponds to a B-tree node[5]; the language of a node to represent a contiguous area in memory conflicts with a node as representing a single entry, which now no longer implict.

Say how you break trees up.

3 OLD

After ‘exp(order)‘ the trie becomes saturated with ‘n-1‘ links? no, every branch could be a link.

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]

3.1 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.

3.2 Structure in Memory

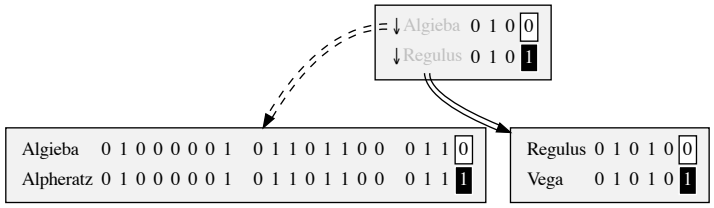
Like a node in a B-trie, a tree structure is a contiguous chunk. As an example, refer to Figure **??**. A tree whose *size*, the number of nodes, is composed of internal nodes, *branch*, and external nodes, *leaf*; $size = branch + leaf$. As a non-empty complete binary tree, $leaf = branch + 1 \to size = 2branch + 1$. 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.

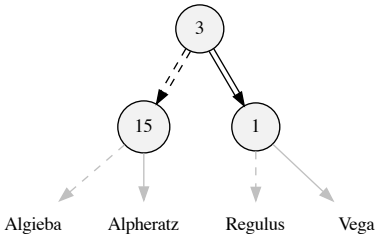
Separate the leaves, which are pointers to data or to another tree, from the branches, which are decision bits in the key string passed to look up. Only one leaf lookup is performed per tree.

3.3 Machine Considerations

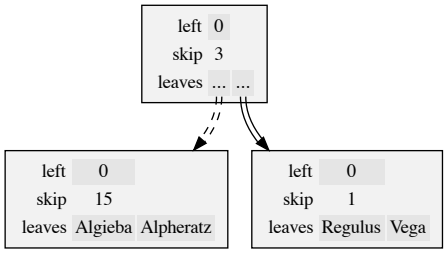
The data has been engineered for maximum effectiveness of the cache in reading and traversing. That is, the tree structure and the string decisions have been reduced to



(a) Bit view.



(b) Logical view.



(c) Memory view.

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

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.[9]

3.4 Running Time

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

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 . . .

3.5 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?

$trie : root, heightbranchtree : bsize, +branchtree, branch[o-1], leaf[o]leaftree : bsize, ?, branch[o-1], is_recursive_bmp[o/8], leaf[o]orleaftree : bsize, ?, branch[o-1], leaf[o]skip, uniondata, trieorleaftree : bsize, ?, branch[o-1], leaf[o]32 : skip, 32 : height, 64 : root/64 : data$

REFERENCES

[1] R. De La Briandais, "File searching using variable length keys," in *Papers presented at the the March 3-5, 1959, western joint computer conference*, pp. 295–298, 1959.

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

[3] P. Jacquet and W. Szpankowski, "Analysis of digital tries with markovian dependency," *IEEE Transactions on Information Theory*, vol. 37, no. 5, pp. 1470–1475, 1991.

[4] 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.

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

[6] N. Askitis and R. Sinha, "Hat-trie: a cache-conscious trie-based data structure for strings," in *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*, pp. 97–105, 2007.

[7] 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.

[8] R. Bayer and E. McCreight, "Organization and maintainence of large ordered indices," *Acta Informatica*, vol. 1, no. 3, p. 1, 1972.

[9] 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.

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