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 descending the trie and minimizing update data, 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-trie 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.

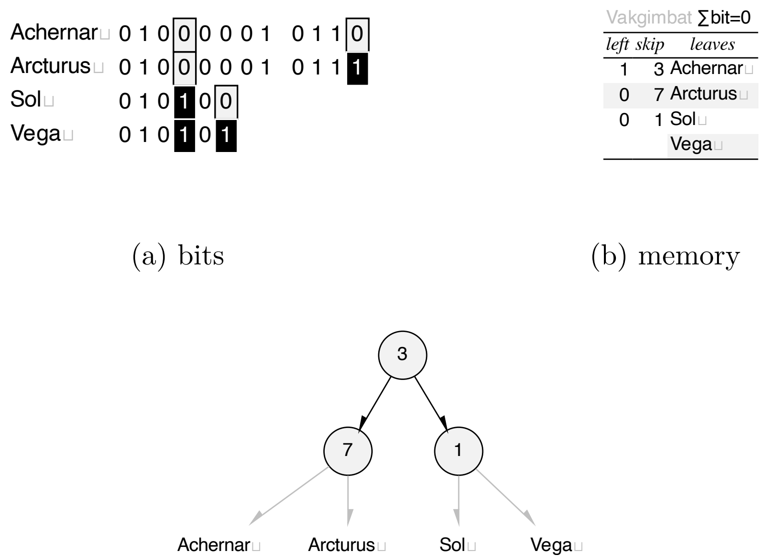
2 IMPLEMENTATION

2.1 Encoding

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.

Figure 1a is a visual example of a Patricia trie, 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 are descendants on the left, and how many bits we should skip before the decision bit. With the initial range set



(a) bits (b) memory

(c) tree

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

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`. This suggests that these be placed in a contiguous index. This index should be compact 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. One octet provides 255 bits skip, usually enough for approximately 32 bytes. More noticeably, the maximum `left` plus one is the maximum number of leaves in the worst-case of all-left. It is also inefficient to modify the trie with more and more keys; this requires more branches to be changed and an array insertion of the leaf.

To combat these two contradictory requirements, 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 relaxes the rules and instead uses a bitmap of which leaves are links to other structures, called trees. Thus a trie is a forest of non-empty full binary trees. A tree corresponds to a B-tree node[5], that is, a contiguous area in memory. This would conflict with the terminology of a key as a leaf and individual branches, which are longer implicit.

Thus, on adding to a tree in a trie that has the maximum number of keys, we must split it into two trees. We use the fact that a binary tree of $n \geq 2$ nodes can be split into two trees not exceeding $\left\lceil \frac{2n-1}{3} \right\rceil$ nodes by starting `daughter` tree at the root and choosing the subtree that is larger until the bound is achieved. The `mother` will have an extra linking leaf.

We use

3 MACHINE CONSIDERATIONS

A more complex example is given in Figure 2. This trie has 3 fixed trees of order 7 maximum leaves and 6 maximum branches with 14 keys in total.

3.1 Linked sample keys

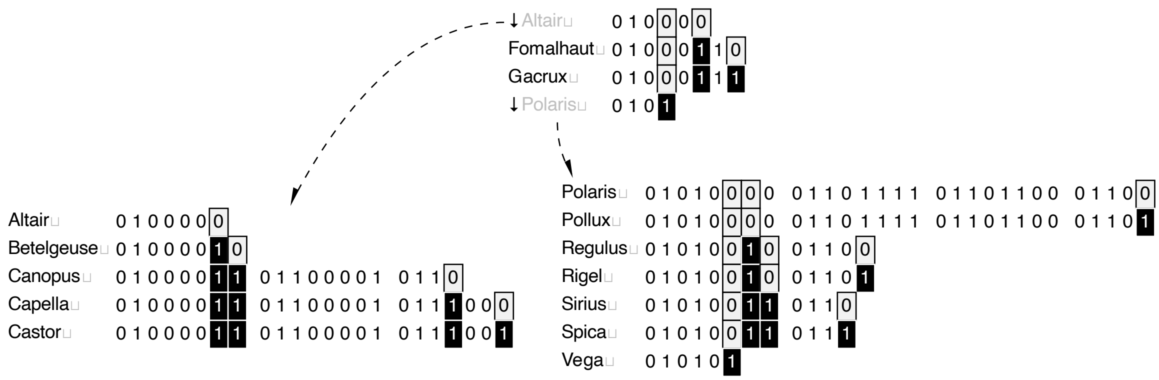
The grey `Altair` and `Polaris` in the root tree, `Vakgimbat`, in Figure 2, are samples of the the trees that are links. We could get any sample from the sub-tree, because all the bits up to bit 6 and 4, respectively, are the same in the sub-tree. Faced with this ambiguity, we arbitrarily and conveniently select the very left. This is important information to have when adding keys, because one has to test each bit of the key against something.

Picking a sample is also important when calculating asymptotic run time of adding keys. The worse case would be an engineered trie, (not with randomly distributed keys,) which with many left-links at once. On addition of a short key to the right, the algorithm must go though all the left-links to arrive at a key for comparison. This leads to worst-case performance $\mathcal{O}(|\text{trie}|)$, something we don't see in practice. We argue that the length of a key should be amortized for future samples in insertions; while there can be multiple insertions with the same trie structure, it becomes less important as the trie grows. Amortized $\mathcal{O}(|\text{key}|)$ is more what we see in practice, which is related to amortized $\mathcal{O}(\log|\text{trie}|)$[10].

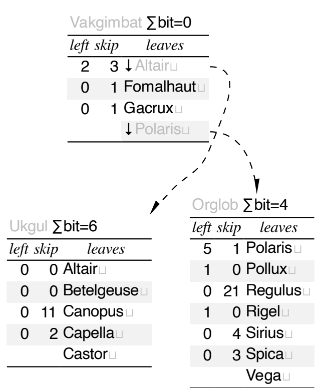
3.2 Hysteresis

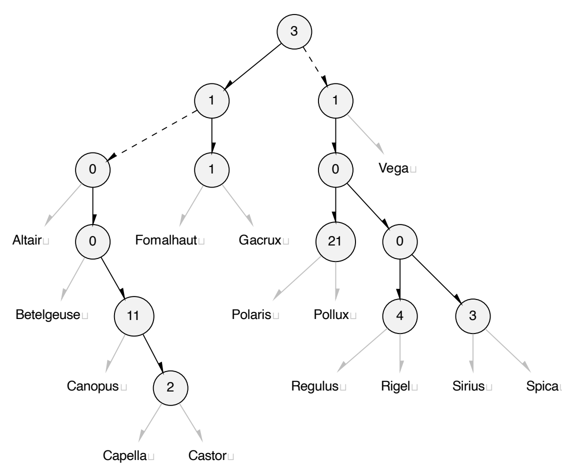
The non-empty criteria of the trees avoids the pathological case where empty trees from deletion pop-up. Further, we can always join a single leaf with its parent except the in the root.

With smaller, dynamic tries, it is more important to not free resources which could be used in the future. Anything less than greedy merging on deletion will have hysteresis. We also should have a zero-key-state with resources.



(a) bits





(b) memory

(c) tree

Figure 2: A trie as a forest of 3 trees of order-7.

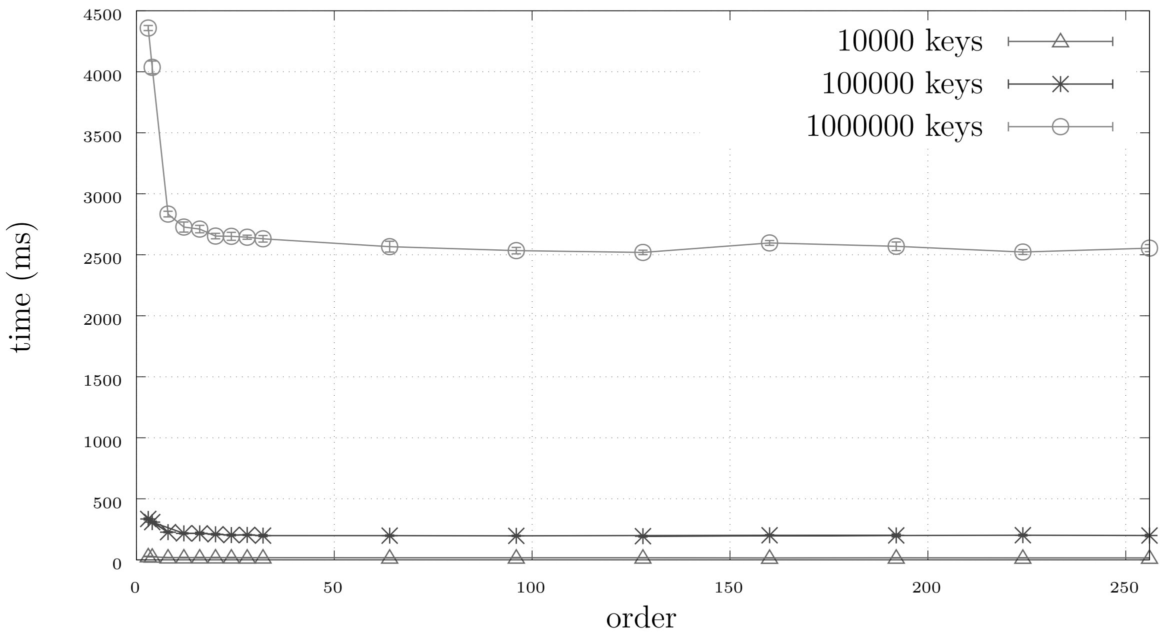


Figure 3: The effects of order on run-time.

3.3 Data size and order

We will push the index to be as small as possible, but no more. The order, or branching factor, is the number of leaves, which is bounded by max(`left`) + 2. We should have a zero-length flag on the length for empty but active. This is not onerous because the alignment supports a size, then $2^n - 1$ index entries, then 2^n leaves and bits in the bitmap.

Figure 3 shows straight insertion of different numbers of keys. It uses two-octet size for each of the branches on the index, divided evenly between `left` and `skip` . The order is how many leaves each tree holds, either keys or links.

The smaller the order, the more links; this adversely affects the performance because the contents of the next index must be fetched into cache, and the trees split more often. The larger the order, the more updates to the local tree on insertion.[9] In Figure 3, we see a very shallow maximum performance, corresponding to a minimum time. However, at low orders, the performance noticeably suffers. Specifically, when we don't fill 64 kB of our cache lines.

3.4 Comparison verses other data structures

Figure 4...

4 CONCLUSION

It's okay.

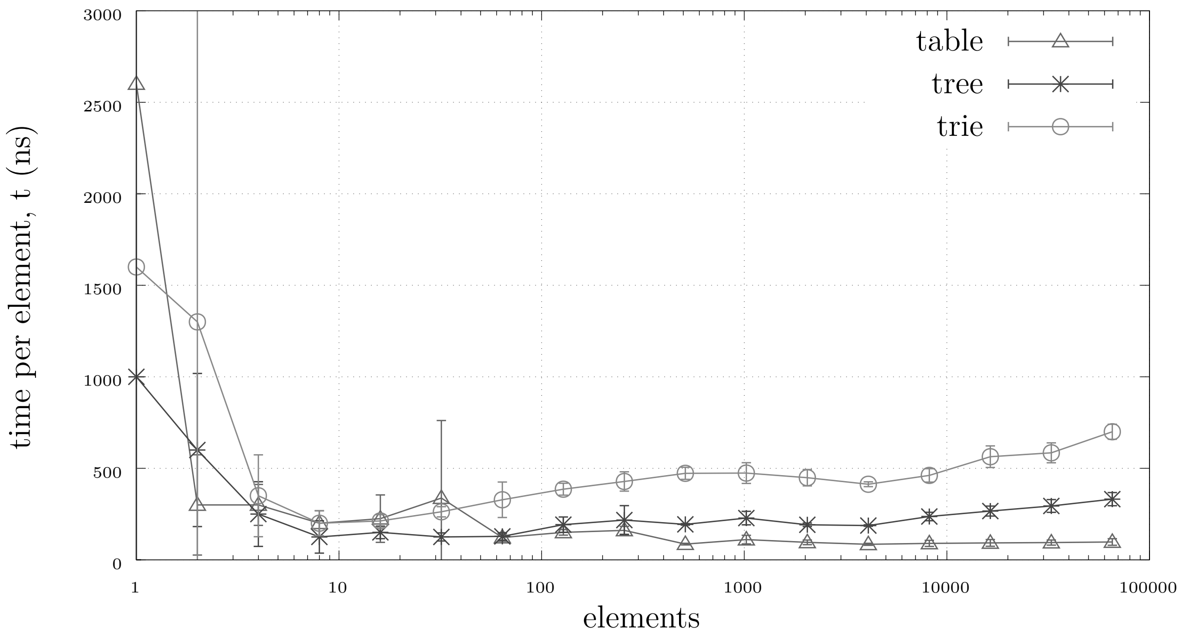


Figure 4: Compaison.

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.