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. In practice, this trie is comparable to a numerical B-tree in performance.

1 INTRODUCTION

A trie is a tree that stores ordered, 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]" This means prefix range queries are equivalent to sub-tries. In general, strings might be stored in several different, but equivalent trees—depending on the type. Tries have no such freedom—there is only one trie that lines up with the data.

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. 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].

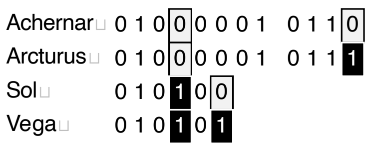
We use a combination binary radix trie, described in [7] as the PATRICIA automaton. Rather than being sparse, a Patricia-trie is a packed index. The leaves are keys. Examples of this are seen in Figure 1c and 2c.

2 IMPLEMENTATION

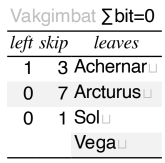
2.1 Encoding

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.

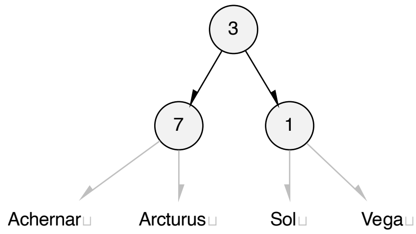
Figure 1a is a visual example of a Patricia trie, that is, a binary radix tree and skip values when bits offer no difference. Note that, in ASCII and



(a) bits



(b) memory



(c) tree

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

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 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 Optimizing data-access

In a Patrica-trie[7] path-compression, each non-decision bit is left out of the index. Therefore, in a one-pass algorithm, our key is compared to in order to a sample from the trie before the new decision bit is reached, following a binary-search pattern. This requires access to $\mathcal{O}(\log \text{size})$ keys.

We can improve on this algorithm by only checking decision bits. This can give us only a single candidate key in the list that matches the supplied key in all the decision bits. This forms an exemplar for comparison with the supplied key. In practice, the improvement only became evident only at 10 million keys.

2.3 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. 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 array with more branches to be changed owing to an array's $\mathcal{O}(n)$.

So the array maximum size and range is a trade-off between space and speed of having the data small and generality of having a larger range. One way to to deal with these conflicting requirements is due to [8], where we have fixed nodes of a given size. Except in tries, contrary to B-trees, the data can not be rotated at will; instead, our trie 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.

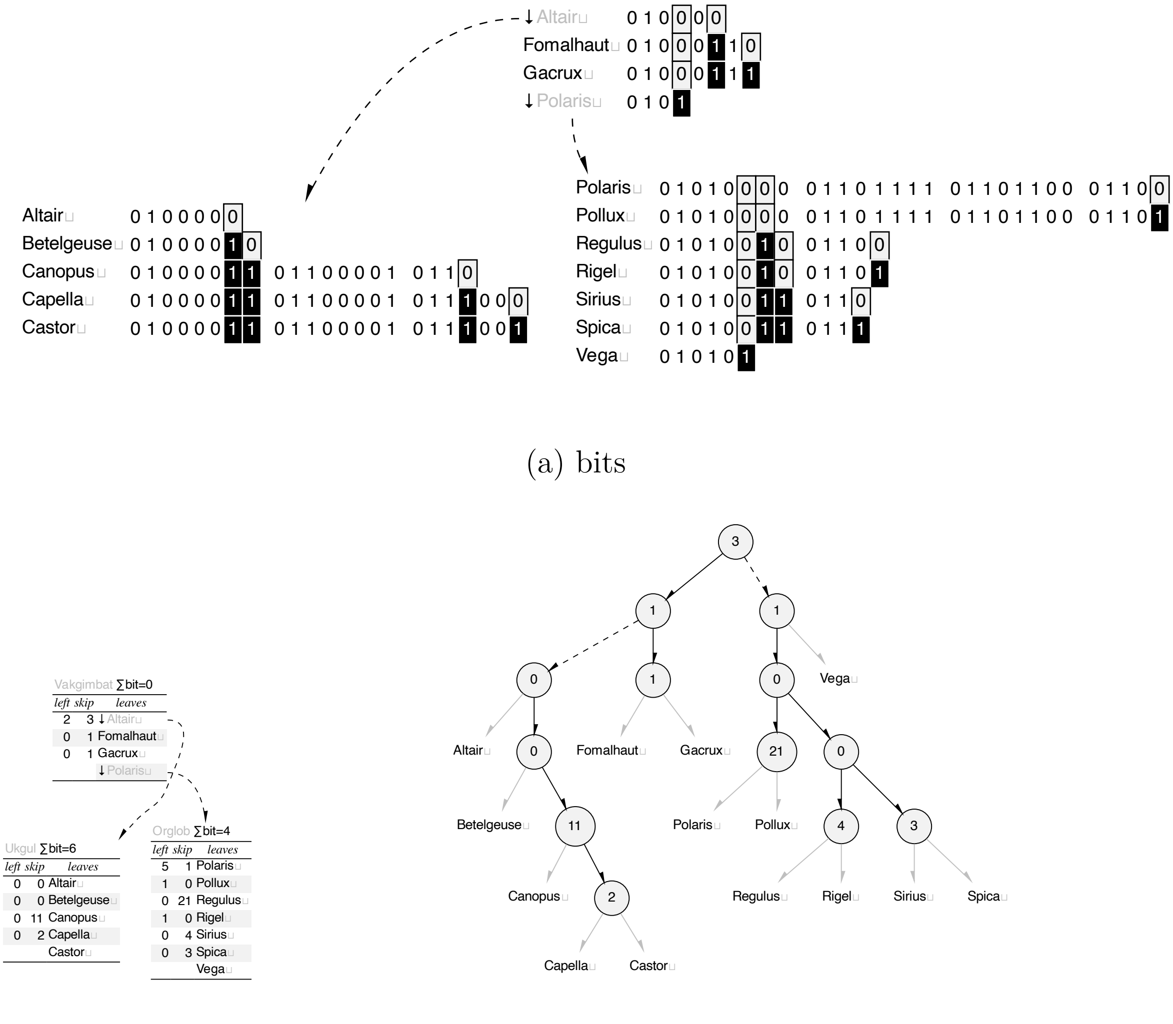
2.4 Link keys

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.

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. Any time we are faced with an ambiguity, we arbitrarily and conveniently select the very lower of the range.

2.5 Inserting and deleting keys

Then, to add a key to an existing trie, first we match the key's decision bits with the tree. If it doesn't have enough length to pick out one tree-key, we arbitrarily choose the left-most alphabetically. We call this the exemplar. The decision bit is found by comparing the new key and exemplar.



(a) bits

(b) memory

(c) tree

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

Deleting a non-leaf key involves merging `skips`. However, a key can be necessary to break up the entries in the trie into sub-maximum sized `skips`. In this implementation, one can not delete a key that causes the rest of the trie to go into an inconsistent state. For the chosen architecture, this only matters if one has longer than 32 bytes the same in adjacent keys–it is very possible for some data.

2.6 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, achieved with a flag on the tree size.

2.7 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.

3 ANALYSIS

3.1 Size of a tree

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 the performance noticeably suffers at low orders. Specifically, when we don't fill 64 kB of our cache lines. A very shallow maximum performance, however, the larger the order, the more efficient use of space.

3.2 Run-time performance

We are compressing by prefix, so generally [10] should apply. In fact, [11] shows explicitly that, in the case where the key encoding is independent and identically distributed, the height of the trie is $\mathcal{O}(\log \text{size})$. The sufficiently

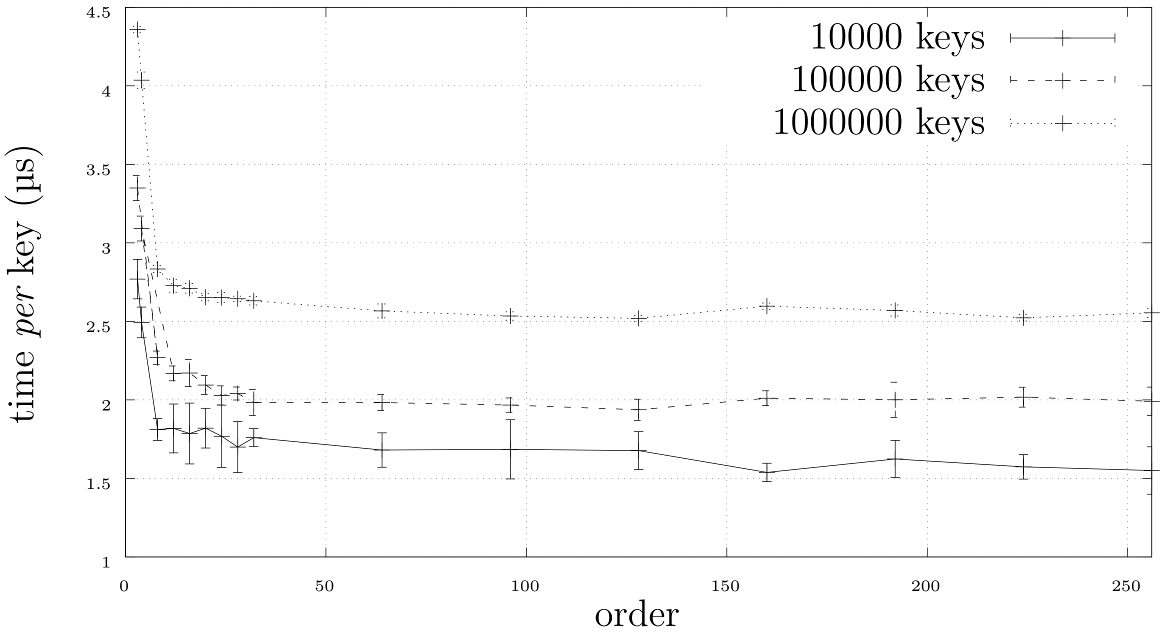


Figure 3: The effects of order on run-time.

iid encoding assumption is quite safe to make in practice because the strings are assumed to be a reasonably finite size.

Figure 4 shows a comparison of a hash table, a B-tree, and a trie. Each replica, a set of random ASCII non-sense syllables created with $\lambda = 16$ Poisson-distributed length.

For an unordered set, a hash table is hard to beat. The hash function used is *djb2*, a very simple multiplicative hash. Still, the bottleneck is almost entirely in calculating the hash. Thus, the performance will be almost entirely dependent of the average length of the string times the complexity of the hash function. Using this hash table loses all the information on the relative order.

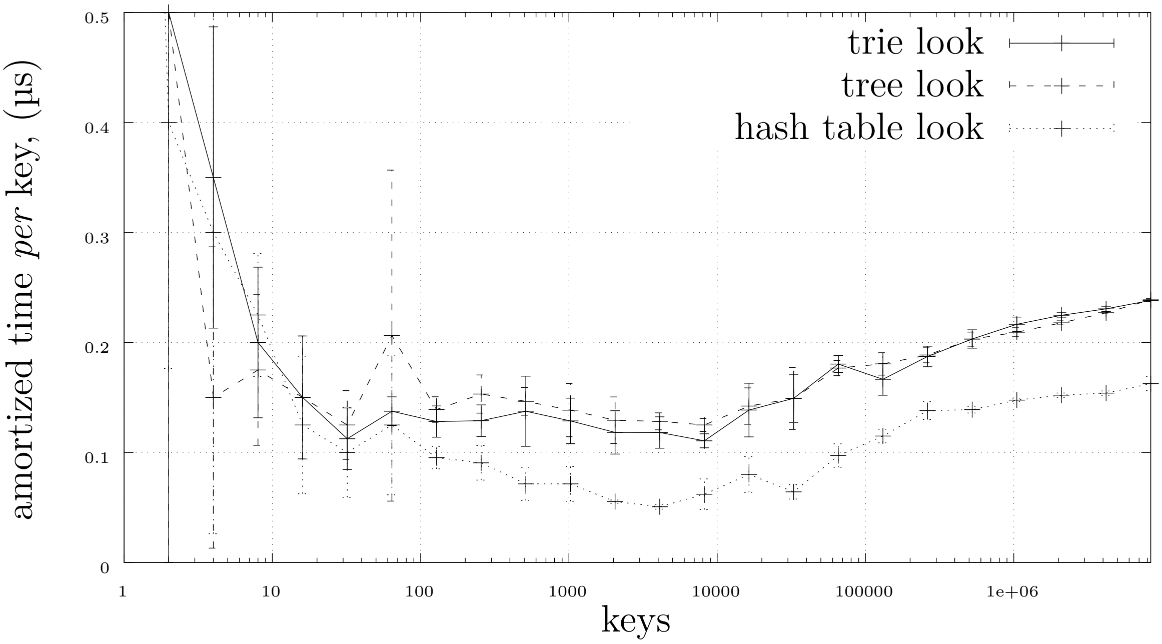
If order is needed, then a B-tree is justified. This is seen practice in, for example, `C++17` where `std::unordered_set` is commonly a hash table and `std::ordered_set` is commonly a red-black tree. Adding to the tree in Figure 4b shows $\mathcal{O}(\log \text{size})$ behaviour.

Figure 4a shows that prefix-matching is not any slower than a B-tree at look-up, in this case. In insertion, Figure 4b, it is also $\mathcal{O}(\log \text{size})$, but it appears slower by a constant amount. Possibly because we are doing multiple descents.

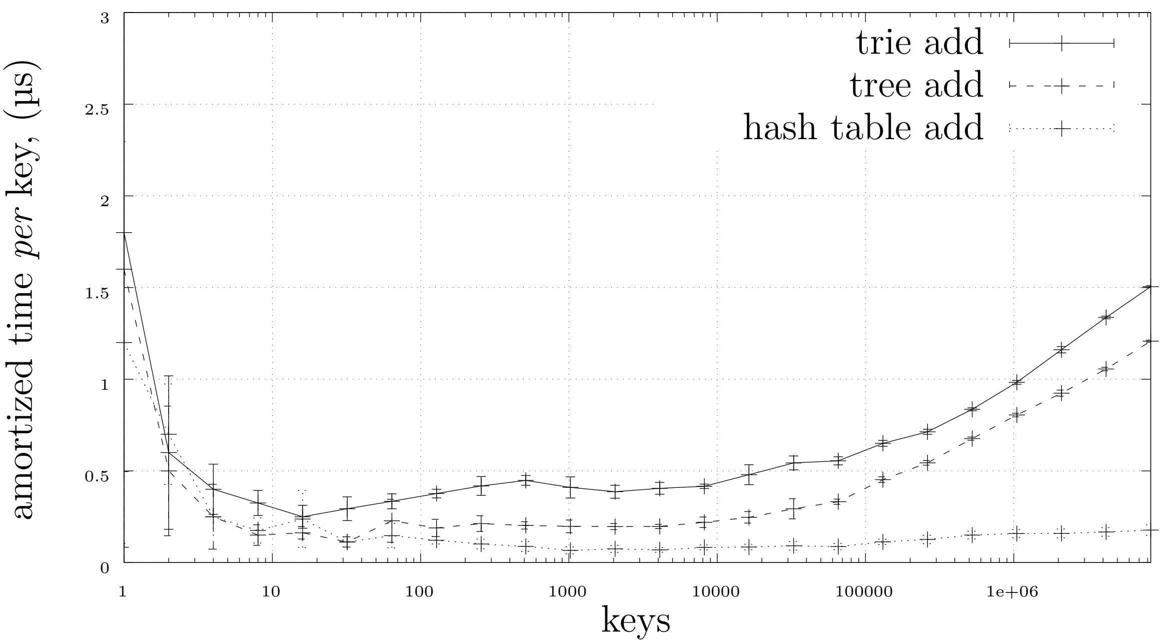
4 CONCLUSION

A Patricia trie with B-tree-like allocation is a really competitive solution in it's own right. It is especially suitable if prefix matching is convenient, as long as the strings fit within the maximum allowed similarity.

On the other hand, a B-tree could also do prefix-lookup in $\Theta(\log \text{size})$



(a) Time to lookup all keys.



(b) Time to add all keys.

Figure 4: Comparison of look-up and insertion in three different data structures.

by simulating the trie and doing two queries, one for each extremum. This is slower than the trie. Still, if one looks up a specific, small range, both the tree and the trie are bound by $\mathcal{O}(\log \text{size})$. Where the trie excels, is at looking up a wide range with a short prefix—dominated by $\mathcal{O}(\text{size of prefix})$. This is probably not useful in practice because iterating the results can take more time than getting the iterator.

At the core, a trie is really a specific ordering of a general tree. It has the same order of space requirements and usually the same performance.

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.

[11] W. Tong, R. Goebel, and G. Lin, "Smoothed heights of tries and patricia tries," *Theoretical Computer Science*, vol. 609, pp. 620–626, 2016.