Comparative case study
Patricia Tries and Hybrid Tries

Sandu Postaru
Aymeric Robini

Pierre and Marie Curie University

Table of Contents

Introduction

In this case study we will be comparing the performances of two Trie data structures : Patricia Tries, and Hybrid Tries (also known as Ternary Search Tries).

We have chosen to implement both Patricia Tries and Hybrid Tries in Java. This decision relies on reasons of performance, ease of memory management, powerful standard library and last but not least our familiarity with the language.

In order to ease the comprehension, all our functions are well documented. We have decided against copy pasting the actual code, since our case study comes with the source code and also because java code can be quite verbose. At any given time you are more thank welcomed to check the actual implementation by consulting the `src\datastructures` directory.

All of our benchmarks can be found in the `dump` directory. For a visual representation of our results, please consult the `graph` directory.

1 Patricia Trie

A compact representation of a trie in which any node that is an only child is merged with its parent.

1.1 Data Structure

For our implementation we used a hash table, the key is the first character of current prefix, the value is a Tuple containing both the prefix and the Patricia Trie representing all the words that have the same prefix.

```
public class Tuple<A, B>{
   public A prefix;
   public B child;

   public Tuple(A _prefix, B _child) {
       prefix = _prefix;
```

```
        child = _child;
    }
}

public class PatriciaTrie implements ITrie {
    Hashtable<Character, Tuple<String, PatriciaTrie>> data;

    public PatriciaTrie() {
        data = new Hashtable<>();
    }
}
```

One of our goals was to minimise the number of empty nodes while still maintaining a fast access. By using a hash table based approach and not a array, we managed to save up to 9 times in spatial complexity. While we acknowledge the fact that our implementation might be a bit slower, we found this to be a good compromise.

In Java, hash tables have a default size of 11 and a load factor of 0.75. Once we have 8 elements in our hash table we must do a complete rehash and double the size (now 22).

On average though, we don't have more than 6 - 7 words at the same level with the same prefix. By assuming that our hash function is uniform, for each node we have :

$$22 - 8 = 14 \, empty \, buckets \tag{1}$$

If we use an array indexed on ASCII codes we have :

$$128 - 8 = 120 \, empty \, buckets \tag{2}$$

On average we have :

$$\frac{120}{14} = 8.57 \, less \, empty \, nodes \tag{3}$$

We used the character # to represent the end of a word.

1.2 Implemented functions and their complexity

Unless stated otherwise, the complexity measure is the number of character comparisons. For our analysis we define :

```
m : the length of the word
A : the size of the alphabet (here 128 for the ASCII table)
N : the number of nodes stored in our Patricia Trie
N1, N2 : if two tries are needed for the same function
```

Insert Worst case complexity $O(m)$.
Assuming that we have one level for each letter of our word, in the worst case we must compare one character per level, therefore descending until reaching the last level.

Remove Worst case complexity $O(m)$.
The same strategy as for the Insert case applies.

Lookup Worst case complexity $O(m)$.
Similar to our Insert and Remove function, in the worst case we will find one letter per level, so we descend to as many levels as characters in our word.

Count Complexity measure: number of accesses (get operations) to the hash table. Complexity $O(N)$.
In order to count the number of words present in the Patricia Trie, we must access all Trie entries, we increment our counter each time # is found as a son, or a prefix contains # as a last character and no children.

For more details on the structure please check section **1.3 Graphical representation**.

ListWords Complexity measure: number of accesses (get operations) to the hash table. Complexity $O(N)$ for words unsorted or $O(N * log(N))$ for words sorted alphabetically.
In order to count the number of words present in the Patricia Trie, we must access all Trie entries, therefore the $O(N)$ complexity. Nevertheless if we want to have all the words sorted alphabetically, we can't rely on the simple access of our elements, since Java Hash tables entries are not sorted according to the natural ordering of their keys. Once the list of words obtained, we have no other choice than to sort it. We decided against using a Tree Map since that would have increased our general access from $O(1)$ to $O(log(N))$ at the cost of maintaining key order.

NbNullPointers Complexity measure: number of accesses (get operations) to the hash table. Complexity $O(N)$.
Similar to our Count function, we must access all Trie entries in order to find those who have no children.

Height Complexity measure: number of accesses (get operations) to the hash table. Complexity $O(N)$.
In order to find the height of our Trie, we must find the height of the longest branch. We must do a complete look up for all entries, for each one maintaining their current height and return as result the maximal height found.

AvgDepth Complexity measure: number of accesses (get operations) to the hash table. Complexity $O(N)$.
Similar to our Height function, we must access all Trie entries. We count at the same time the number of leafs and the sum of all leafs heights. As result we return the quotient of the two.

NbPrefixed Complexity measure: number of accesses (get operations) to the hash table. Worst case complexity $O(m+N)$ or $O(N)$ since $m \ll N$ for practical cases.
In order to find the number of words prefixed by a word, we must localize the root node that contains the prefix and count all of his children. Similar to our Lookup function we find the root node in $O(m)$ and similar to our count function we count all of his children in $O(N)$. We therefore have $O(m + N)$.

Merge Complexity measure: number of accesses (get operations) to the hash table. Worst case complexity $O(N1 + N2) = O(max(N1, N2))$.
In the worst case, our two tries are identical, therefore we access all entries for both of them.

Convert Complexity measure: number of accesses (get operations) to the hash table. Worst case complexity $O(N * A)$, since $A = 128$ is a constanct, we have $O(N)$.
In order to convert a Patricia Trie to a Hybrid Trie, we must decompose each word at character level and insert that character in the Hybrid Trie. We do this for each entry, which may contain any possible combination of characters from the alphabet.

Draw Complexity measure: number of accesses (get operations) to the hash table. Complexity $O(N)$.
We must access all Trie entries, for each word we print a new node using the DOT language.

1.3 Graphical representation

The example below is a small instance of a Patricia Trie. Larger instances could not be displayed in this report since they get quite voluminous pretty fast. The character # is used to represent the end of a word.

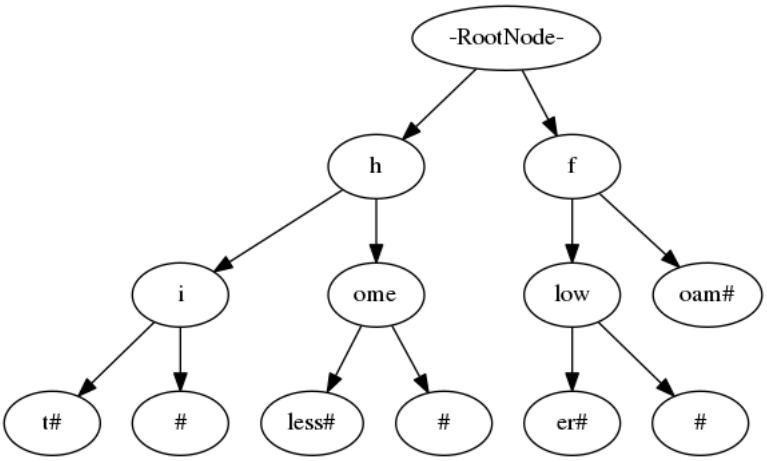

Fig. 1. The Patricia Trie containing: 'hi', 'hit', 'home', 'homeless', 'flow', 'flower', 'foam'.

We can observe on the next page the same instance of the Patricia Trie converted to a Hybrid Trie.

We can see that even for a small instance, the Patricia Trie has a much smaller height (3) than the same Hybrid Trie (9). The difference of height can reach a factor of 5, 6 on big instances. For more in depth details, please consult the `dump` directory which contains tables full of statistics including height, average depth, number of null pointers and much more, for each of the instances we analyzed.

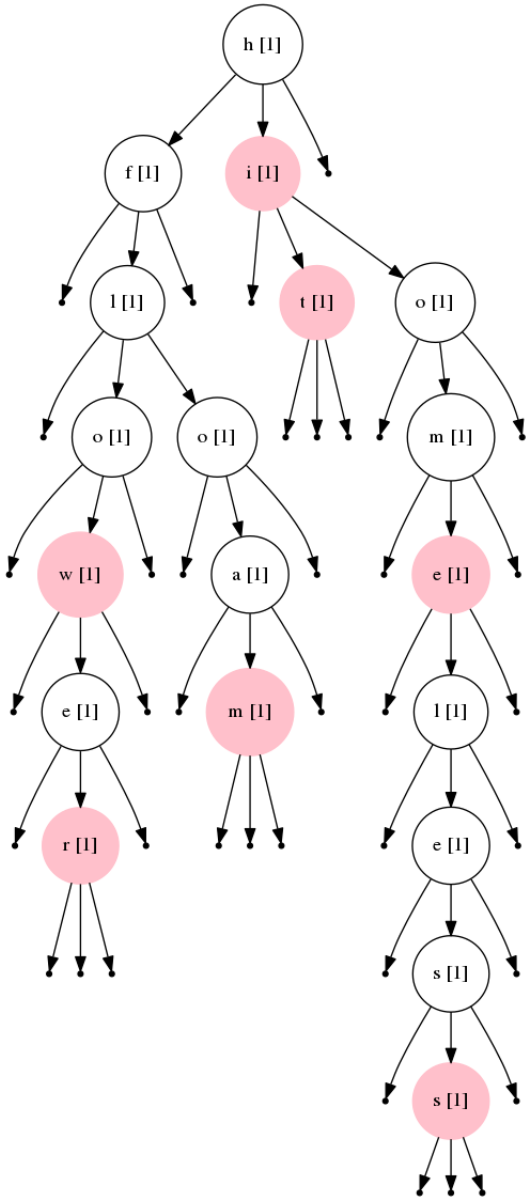

Fig. 2. The Hybrid Trie containing: 'hi', 'hit', 'home', 'homeless', 'flow', 'flower', 'foam'.

2 Hybrid Trie (Balanced)

A Hybrid Trie can be seen as a Binary Search Tree, where each node has one more child (noted as 'middle child') which contains all words prefixed by the concatenation of all parent nodes letters.

2.1 Data Structure

The implementation is quite straightforward : a Node has a letter, a boolean that indicates whether the node represents the end of a word or not, and tree Node 'left', 'middle', 'right' containing respectively words having a first letter lower than the node's one, the ones starting with the node's letter, and the ones having a first letter greater than the node's one.

For Balanced Hybrid Tries, we also need a height, which is an integer. We could have used only two bits representing the balance factor (i.e. the height difference between right and left children), but it would have complicated the code, for a hardly perceptible space optimization.

```
public class Node {
    Node left, middle, right;
    char letter;
    int height;
    boolean isEnd;
}
```

Although an efficient way of storing word sets, Patricia Tries may contain a large amount of null pointers.

By implementing Hybrid Tries, we are looking for a trade-off between space and time complexity. We can easily minimize the number of null pointers, which is at most 3 per node. Since the number of nodes can't be greater than the sum of all word sizes in our word set, we can say that memory usage is linear to the total number of characters.

Actually, this is also true for Patricia Tries, but in the worst case, a Patricia can contains up to 128 null pointers per Node (if we don't take in consideration our implementation using a hash table), while a Hybrid Trie may only contain up to 3 null pointers per Node.

2.2 Implemented functions and their complexity

As previously, the complexity measure is the number of character comparisons. As a reminder, we define :

```
m : the length of the word
A : the size of the alphabet (here 128 for the ASCII table)
N : the number of nodes stored in our trie
```

In the whole section, we will call N the number of nodes in the tree. In the worst case (when none of the words share a common prefix), it can be bounded from above by $N * averageWordLength$. In practice, it's hard to approximate since it greatly depends on our instance (especially the language).

Insert Worst case complexity $O(Am)$ for a non balanced trie, since $A = 128$ is a constant, we have $O(m)$.

Average complexity $O(log(N))$ for both balanced and not balanced tries :
Practically, we (should) never add words in a sorted order, so if we randomly insert our words, it's most likely to be approximately balanced.

Worst case complexity $O(log(N))$ for a balanced trie :
Since our trie can be seen as an AVL where nodes have an additional child, inserting consists in :
Looking where to insert the word $\rightarrow O(log(N))$ (as in AVL)
Inserting the word $\rightarrow O(m)$
Going back to the root to balance nodes if necessary (with $O(1)$ balancing) $\rightarrow O(log(N))$
Since $m \ll log(N)$, we have a $O(log(N))$ insertion.

Remove Worst case complexity $O(Am)$ if not balanced, $O(log(N))$ otherwise. The same strategy as for the Insert case applies.

As for insertion, we have to go back to the root in order to remove the useless nodes (the ones that only belonged to the removed word). Since this step is in $O(m)$, we may not take this in consideration.

Lookup Worst case complexity $O(Am)$ if not balanced, $O(log(N))$ otherwise. Finding a word in a Hybrid Trie is basically the same algorithm as for a binary search tree, hence the previous complexities.

Note that this function can easily be written in an imperative style, or benefit from tail-recursion optimization.

Count In our implementation, it has for both balanced and not balanced tries a $O(N)$ complexity.

We could have added a counter in our HybridTrie class which would have been incremented by one each time a word is inserted (and decreased for each removal). It wouldn't have changed the insertion complexity, but would have set the Count function's complexity to $O(1)$. However, for simplicity's sake, we decided to keep the previous implementation.

ListWords As for Count, it has for both tries structures a $O(N)$ complexity.

NbNullPointers Similarly, the number of null pointers is at most 3 times the number of nodes, so it is in $O(N)$.

Height There are two simple methods to get the height of the trie :

The first one is to browse the whole trie looking for the deepest branch. This has the same $O(N)$ complexity since we have no choices but to browse all nodes. This is the one we have chosen.

Another method consists in storing in each node its height, updating it for each insertion and deletion while going back to the root. Since this is in $O(1)$ for each node, it wouldn't have increased complexity neither for insertion, nor for deletion, but the Height operation would have been in $O(1)$.

AvgDepth Either we browse all the leaves in $O(N)$ worst-case complexity, or we store in our HybridTrie class the number of leaves, and the average depth ($O(1)$). When inserting a leaf,
$newLeavesNumber = leavesNumber + 1$;
$newAverage = \frac{leavesNumber * average + depthOfCurrent}{newLeavesNumber}$.

NbPrefixed Let a be the argument to the NbPrefixed function call. Worst case is when a is the empty string, and none of the words share a common prefix. This turns out to be the same thing as ListWord, so a $O(N)$ worst-case complexity.

In practice, if N' is the number of nodes which belong to words prefixed by a, then the complexity is in $O(m + N')$ (for balanced tries) since we first have to reach the tree containing all words prefixed by a in $O(m)$, then browse all the nodes in that tree ($O(N')$).

Convert To convert a Hybrid Trie to a Patricia Trie, we have to browse all the nodes ($O(N)$), and for each, add if necessary a branch in the Patricia Trie ($O(1)$). Hence a $O(N)$ complexity.

We could have implemented a naive algorithm that list all words of the trie ($O(N)$) then insert each one in the patricia trie (worst-case $O(m)$ where m is the length of the word) for a $O(Nm)$ worst-case complexity, but this option is far worse.

Draw Draw browses the tree recursively, going through all nodes, so we have a $O(N)$ complexity.

2.3 Graphical representation

Here are two hybrid trie structures, one balanced, the other non balanced. For both, blue nodes indicates that the letter is the end of the word, and dots represent null pointers. Both tries contains exactly the same words (my, mine, mein, mansion, homeless, home, hit, hi, foam, flower, flow).

As we can see, the null pointers count is acceptable (significantly less than Patricia Tries), and the number of nodes isn't that large.

For balanced tries, we can see that the trie is denser than the non balanced one, which permits a faster access to node information.

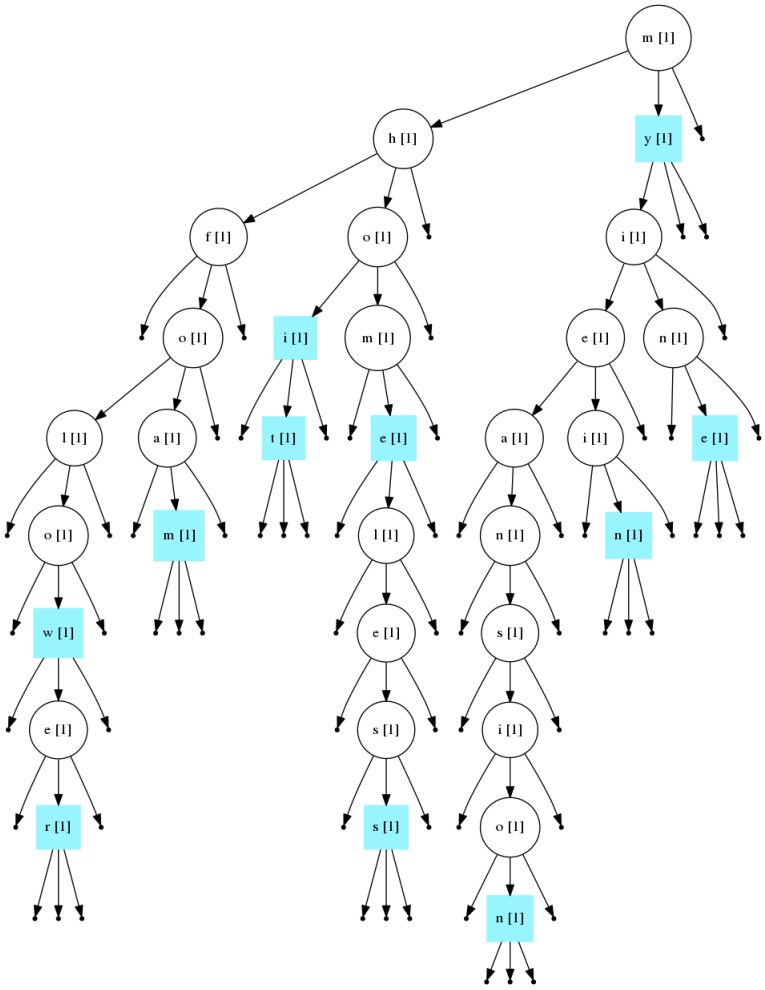

Fig. 3. A non-balanced Hybrid Trie containing the words my, mine, mein, mansion, homeless, home, hit, hi, foam, flower, flow

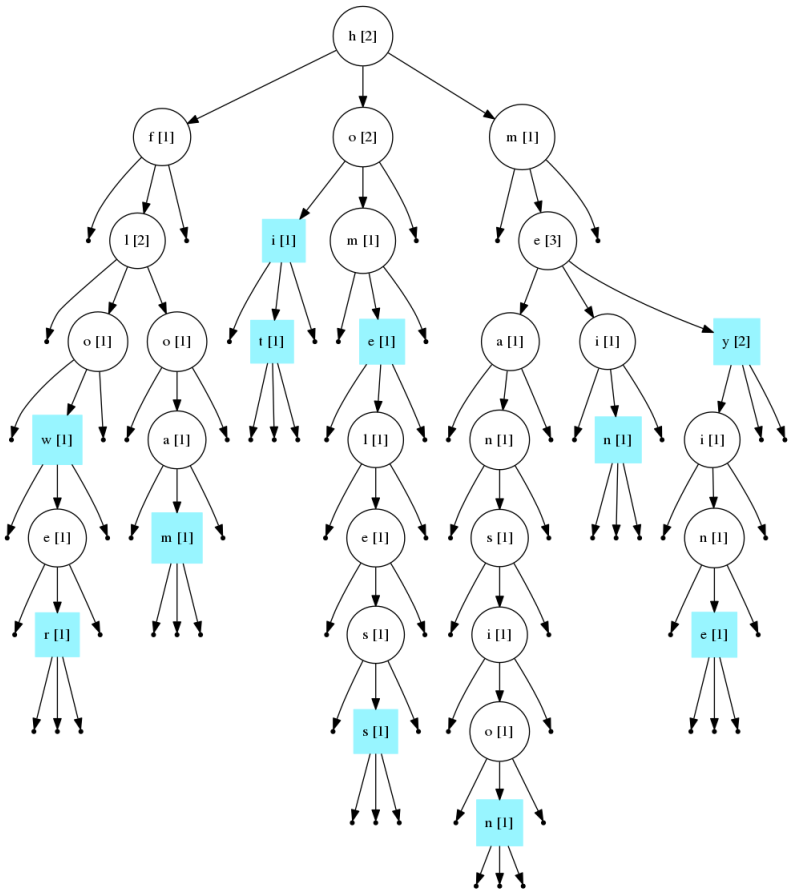

Fig. 4. A balanced Hybrid Trie containing the same words as the previous trie

3 Performance and statistics

Now that we have described the logic behind each data structure and their theoretical performance, we want to determine which structure is more efficient in real world scenarios.

Therefore we elaborated a series of tests, each one compares the tree data structures in the same fashion.

3.1 Insertion in individual tries instances

For this experiment, we took several Shakespeare's oeuvres, and measured the amount of time needed to insert, for each book, all the words in a trie (1 trie per book).

Here is a graph that shows individual insertion time for each tome, in function of the number of words in the trie.

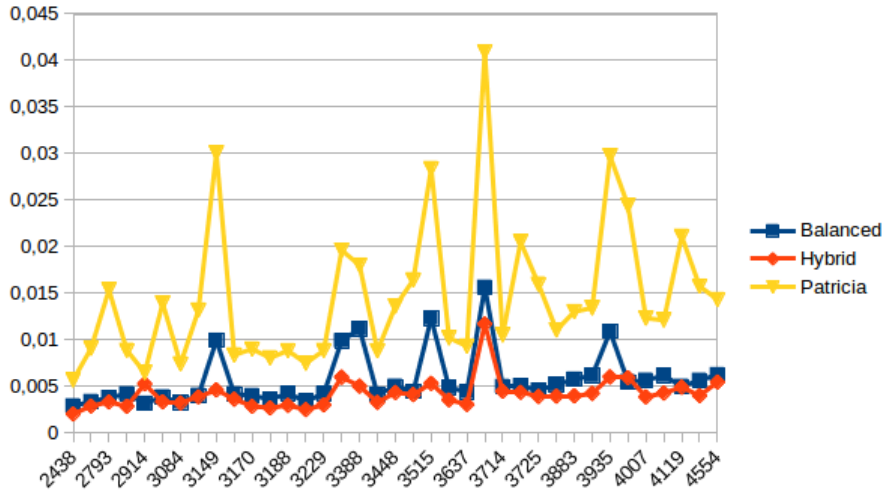

Fig. 5. Insertion time depending on the number of words to insert

As we can see, on average, Patricia Tries performs 3 times slower than its opponents. Surprisingly, insertion time in a Balanced Hybrid Trie is quite similar to the Regular Hybrid Trie, except for some cases where the number of rotations might be consequent.

3.2 Insertion in the same trie instance

In this experiment, we decided to do the same thing as in the previous one, but with one single trie for the whole set of oeuvres, and got the following results :

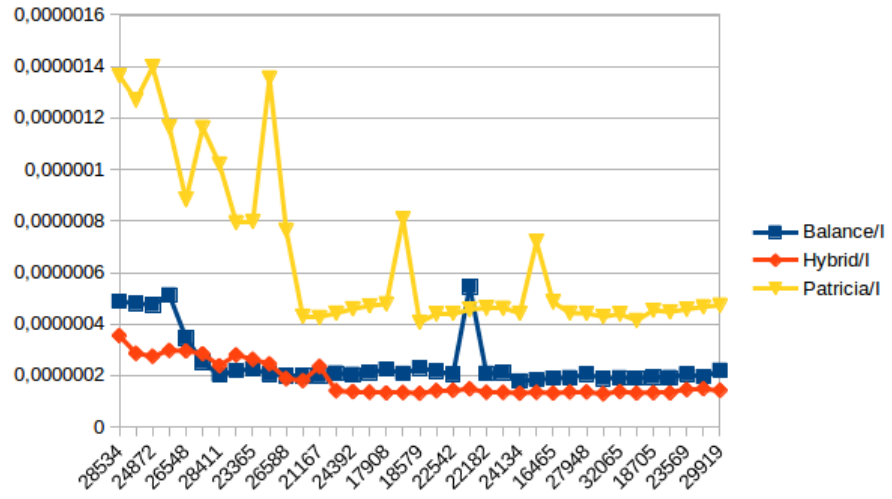

Fig. 6. Insertion time depending on the number of words to insert, with an already filled trie

With no surprises, we can see that the more the trie is filled, the cheaper insertions are. Indeed, while inserting a word which is already (or partially, with a common prefix) in the trie, the number of nodes that are created is low, and so is the number of allocations (which are quite expensive).

3.3 Checking the presence of a word set

In this experiment, we took a set of 1000 most commonly used words in the English language, and looked them up in each trie. Results are the following :

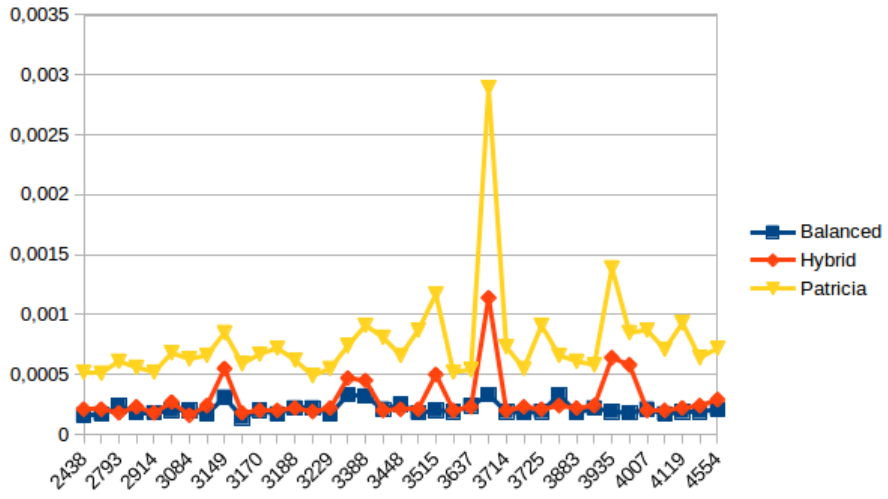

Fig. 7. Looking up time for 1000 common English words depending on the number of words to insert

As previously, Patricia still performs slower than the other tries. Although a bit slower on insertion, balanced tries show a notable improvement, especially for the cases where insertion was the slower.

3.4 Removing a word set

Since the algorithm for word removal is quite similar to the lookup algorithm, we have with no surprises similar results :

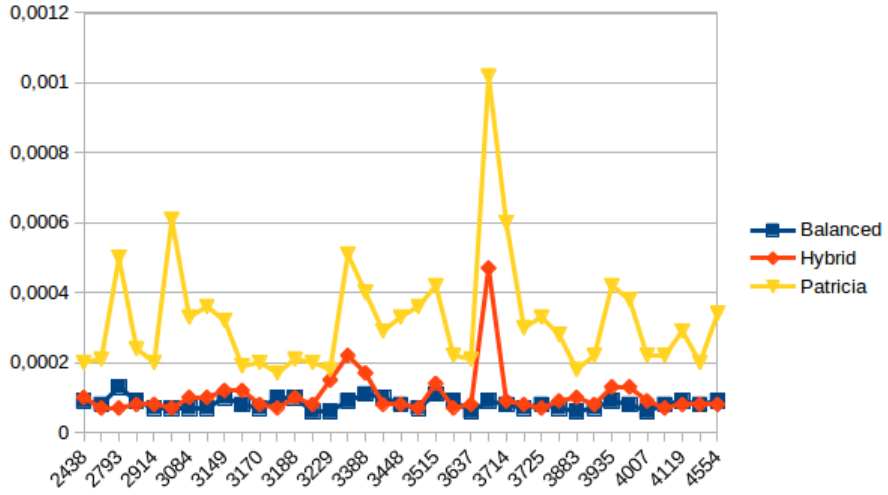

Fig. 8. Removal time for 300 common English words depending on the number of words to delete

3.5 Merging Tries

In this experiment, we wanted to compare two implementations of a merge algorithm.

Merge operations for Patricia Tries (represented in blue) are 3x times faster than a regular word listing and insertion for both Regular Hybrid Tries and Balanced Hybrid Tries. This behaviour is expected since the main advantage of merging two Patricia Tries is the fact that, once a uncommon prefix is found, the whole hierarchy is merged in $O(1)$. Meaning that the more the two tries are different in content, the more efficient the merge operation will be.

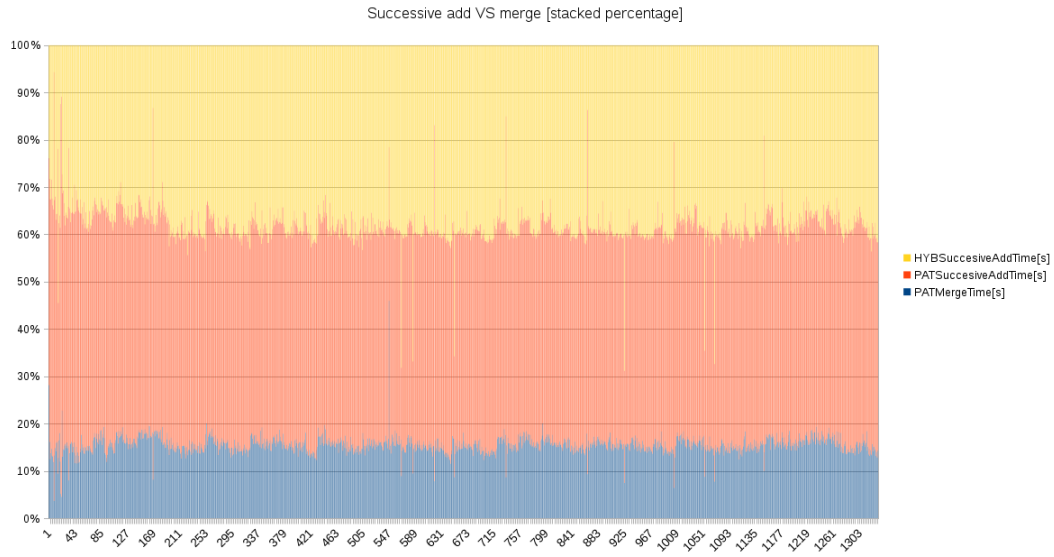

Fig. 9. Merge, naive VS optimized [stacked percentage]

4 Conclusion

In this case study we compared Patricia Trie and Hybrid Tries (both regular and a balanced version).

Even though Hybrid Tries have $4X$ times the height of the equivalent Patricia Tries, on average we observed Hybrid Tries to be $2.5X$ faster than Patricia Tries for most operations. This is mostly due to three factors :

- String comparison for Patricia Tries vs character comparison for Hybrid Tries.
- Order of insertion for Patricia Tries plays a definitive role, since depending on the common prefix, new entries must be created, their children need to be repositioned and the prefix of existing entries must be modified.
- The use of a hash table against an array, saves up to 9x the space but slows down the access to the entries.

We also observed Balanced Hybrid Tries to be slightly faster on lookup than Regular Hybrid Tries, but slightly slower on insertion. The balanced version has a reduced height so insertion is faster, but this comes at a cost, since on word insertion rotations might be made in order to preserve the balance propriety.

In cases where existing tries must be combined, we recommend the use of Patricia Tries, since the merge operations is 3x times faster than a regular word listing and insertion for both Regular Hybrid Tries and Balanced Hybrid Tries. Assuming that most of the content of the two tries is different, if a prefix is not present, Patricia Tries allow for a net gain of performance by inserting a whole hierarchy of nodes in O(1).

On terms of implementation difficulty, we consider Patricia Tries harder to implement. For most functions they contain more edge cases than Hybrid Tries and their net performance is not better than those of Hybrid Tries.