HashTrie: A Compressed Dictionary Storage Solution

Yihang Yuan
Computer Engineering
Stevens Institute of Technology
Hoboken, NJ, U.S.A
yyuan22@stevens.edu

Al Zahra
Computer Engineering
Stevens Institute of Technology
Hoboken, NJ, U.S.A
psabah@stevens.edu

Eugene Kozlakov
Computer Engineering
Stevens Institute of Technology
Hoboken, NJ, U.S.A.
ekozlako@stevens.edu

***Abstract*—Tries are frequently employed to house dictionaries of words for applications involving written text. The auto-complete feature found in most modern text-editing software is a well-known application of dictionary Tries. However, Tries can be space-inefficient with an O(n*m) space complexity, which can become a considerable drawback when dealing with large datasets. This paper introduces the HashTrie, a novel data structure designed for efficient storage and retrieval of words. HashTrie leverages the benefits of both Trie and hash table data structures to organize and search a given *dictionary*, or list of words. This innovative implementation aims to address the space inefficiency inherent in the standard Trie data structure while maintaining the majority of its operational speed. This is achieved by modifying the traditional Trie structure to incorporate hash-table mapping at its leaf-nodes. Early results indicate that the memory consumption of the HashTrie is significantly reduced, being approximately $\frac{1}{11}$th that of a conventional Trie implementation. The HashTrie hence offers a potentially effective solution for applications requiring efficient text retrieval and storage, particularly those dealing with substantial word datasets.**

I. Introduction

The HashTrie code presented here demonstrates an efficient approach for storing and searching words in a dictionary. It combines the advantages of Trie data structure, which provides fast prefix search, with the flexibility and speed of hash tables for storing and retrieving complete words. It uses a custom hashing function to generate unique hash values for words, allowing for quick lookup and retrieval. This implementation handles cases where a word is a complete unique entry or shares prefixes with other words efficiently.

The main efficiency gain is through the implementation of a Prefix Partitioming Algorithm called *breaker()* which efficiently breaks down a dictionary into prefixes and corresponding suffix lists. This algorithm ensures that the suffix lists associated with each prefix are within a specified maximum size, optimal use of memory and retrieval performance. The code in Algorithm 1 is an implementation of *prefix()* which initializes an empty HashTrie, then iterates over each word, ensuring the hash tables do not exceed the maximum size and finally stores the words efficiently.

Furthermore, the code includes memory profiling capabilities, which can provide insights into the memory usage of the HashTrie. This can be useful for evaluating the performance and memory efficiency of the code. In summary, it combines the strengths of Trie and hash table data structures, providing efficient prefix search and quick word retrieval while ensuring memory usage remains within specified limits.

Algorithm 1 Code Snippet: Intaking Dictionary

```
function WORDINTAKE
    Initiate HashTrie object
    Initiate vector<string> wordList
    Initiate maxHashTableSize
    for all $words \in$ dictionaryFile.txt do
        wordList.add($word$)
    end for
    prefixPartition(wordList)
    for all $kvPairs \in partitionedList$ do
        if in $kvPairs.suffixes.size$ == 1 then
            Map $word$ to $Trie$ in $HashTrie$
        end if
        HashTrie.addPrefix($prefix$)
        HashTrie.addToHashTable($word$)
    end for
end function
```

II. Background

The Trie, also known as a prefix tree, is the data structure traditionally used for efficient text retrieval. The key idea behind the Trie is to leverage the common prefixes of words to save storage space and facilitate quick search operations. Each node in the Trie represents a character, and each path from root to a node forms a prefix of some words. Despite its speed and efficiency in text retrieval tasks the Trie can be space-inefficient when storing a large dictionary of words. This is because each node in a Trie needs to maintain pointers to its children nodes, leading to a significant overhead when the number of children is large. For instance, in the case of English words, each node may need to maintain 26 pointers corresponding to 26 English letters, which can lead to considerable space wastage if a node has few children.

On the other hand, a hash table, another commonly used data structure, provides an efficient way to maintain a set of key-value pairs, supporting efficient insertion , deletion, search operation. However, a hash table does not maintain any ordering of keys and thus does not support prefix-based search, which is a crucial operation for many text retrieval tasks.

Given the strengths and weaknesses of Tries and hash tables, a natural question arises: Can we combine the best of both worlds to create a more space-efficient data structure that supports efficient text retrieval operations, including prefix-based search? in this paper, we introduce a hybrid data structure that combines a Trie with hash table to store a large dictionary of words, aiming to reduce space overhead while maintaining efficient retrieval operations

III. Implementation

A. Description of Dataset

The dataset used in this project is a text file sourced from the CPE593 course repository on GitHub. This file represents a comprehensive English language dictionary containing approximately 200,000 words. Each word is listed in alphabetical order, with one word per line, providing a clear and easily parsable structure for importing into our data structures.

For the purpose of testing the efficiency and effectiveness of our proposed data structure, we generate a series of smaller, subsidiary datasets from the original text file. These datasets contain varying numbers of words, allowing for a broad and robust testing environment.

In addition to the subsets of the original dictionary, we also create a series of synthetic datasets. These synthetic datasets consist of a variety of magnitudes of randomly generated words. The words within these datasets have random lengths and are not necessarily real or meaningful English words. The primary purpose of these synthetic datasets is to test the robustness and performance of our data structure under different conditions and input sizes. It helps us understand the scalability of our proposed solution and identify any potential issues or limitations that might not be evident when using only real-word dictionaries.

B. Prefix Partitioning Algorithm

Our primary challenge lies in efficiently partitioning the dictionary words based on their prefixes. More specifically, we aim to group words under any given prefix doesn't exceed a predefined threshold, for instance, 64 words.

To effectively manage these word-prefix associations, we employ an unordered map, a powerful data structure that maintains key-value pairs. In our case, the key represent the 'prefix', and the corresponding value is a 'list' holding all the words falling under that prefix.

To populate this unordered map according to our requirements, we devise an algorithm known as 'Prefix Partitioning'. The pseudo code for this depicted in the following algorithm:

Algorithm 2 Prefix Partitioning Algorithm

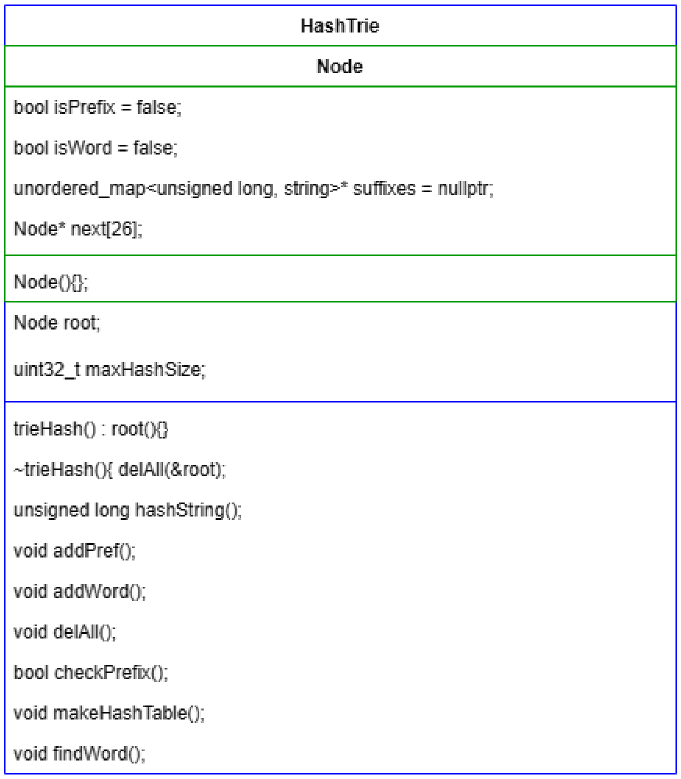
```
1: function PREFIXPARTITION($map$)
2:     while $\exists(v.length > 64)$ in $map$ do
3:         $temp\_map \leftarrow emptyDictionary$
4:         for all $(k, v) \in map$ do
5:             if $v.length > 64$ then
6:                 for $l \in \{'a', \ldots, 'z'\}$ do
7:                     $n\_k \leftarrow k + l$
8:                     $n\_v \leftarrow emptyList$
9:                     $temp\_map[n\_k] \leftarrow n\_v$
10:                end for
11:                for $w \in v$ do
12:                    $n\_k \leftarrow k + firstLetter(w)$
13:                    $map[n\_k].append(w)$
14:                end for
15:                for $l \in \{'a', \ldots, 'z'\}$ do
16:                    $v = map[n\_k]$
17:                    if $v.length = 0$ then
18:                        delete $temp\_map[n\_k]$
19:                    end if
20:                end for
21:            end if
22:        end for
23:        delete $map[k]$
24:        $map.update(temp\_map)$
25:    end while
26: end function
```

The algorithm begins with a while loop(line2). The loop continues iterating as long as any list(value i n the unordered map has a length exceeding the predefined threshold. It then identifies the prefix(key) whose associated list of words exceeds the threshold (line5). the algorithm proceeds to generate 26 new key-value pairs, where each key is a combination of the original prefix and a letter from 'a' to 'z'. Each new key is initially associated with an empty list. It then redistributes the words from the original prefix into these new key-value pairs based on their new prefixes. Some of the new prefixes might still not have any associated words. The algorithm identifies such prefixes and removes them from the unordered map. Finally, it also removes the original key-value pair, ensuring that no word is duplicated in the map.

Through several iterations of this process, we end up with an unordered map that adheres to our constraints and is ready for insertion into the Trie data structure.

C. Hashmap and Trie

The HashTrie was implemented in C++ by way of a modified form source code used to make a standard Trie implementation. This implementation included a *Node* class contained within a *Trie* class, with each node containing an a array of pointers mapping to 26 characters to account for all 26 English lowercase characters. The classic implementation also contains a boolean *isWord* to demarcate the termination point of a search, and to indicate whether the nodes traversed denote a genuine mapped (or stored) word.



```
HashTrie
Node
bool isPrefix = false;
bool isWord = false;
unordered_map<unsigned long, string>* suffixes = nullptr;
Node* next[26];
Node(){};
Node root;
uint32_t maxHashSize;
trieHash() : root(){}
~trieHash(){ delAll(&root);
unsigned long hashString();
void addPref();
void addWord();
void delAll();
bool checkPrefix();
void makeHashTable();
void findWord();
```

Fig. 1: Visual representation of the C++ class used to build the HashTrie. The individual *node* class is in green, while the actual *HashTrie* class is in blue.

The implementation for HashTrie does not stray far from this schema. As seen in 1, the *Node* class itself includes a new Boolean variable, *isPrefix* which by default is set to `null`. Additionally, in each node there is a pointer to a C++ proprietary hashmap, `unordered map`, of key `unsigned long` and values `string`.

The unordered map structured generated by the algorithm in 3, called *partitionedMap* here, is then mapped to the constructed HashTrie by way of the following pseudo-code algorithm:

Algorithm 3 Word Mapping Algorithm

```
1: function WORDMAPPING($partitionedMap$)
2:     $p \leftarrow root$
3:     for all $(prefixes, words) \in partitionedMap$ do
4:         for all $letters$ in $prefix$ do
5:             addNodeToTrie(letter)
6:             $p \leftarrow p.next$
7:         end for
8:         $p.isPrefix$ = $true$
9:         $p.wordHashMap$ = new $unorderedMap$
10:        for all $words$ at $prefix$ in $partitionedMap$ do
11:            $hashKey$ ←hashWord($word$)
12:            set $p.wordHashMap[hashKey] = word$
13:        end for
14:    end for
15: end function
```

To map each prefix, and then generate a subsequent hash-table to store every word associated with that prefix, the *partitionedMap* must be traversed in-order. The algorithm begins by accessing the first key-value pair of of *prefixes* (the key) and the list of *word*s (the value) associated with the prefix. The prefix is mapped to the Trie in the same way a word would get mapped to a Trie dictionary. However, at the leaf node of the prefix, a new *unorderedMap* object is created, and the value *word*s get mapped to the table with a key of type `unsigned long`, with the value being a `string` type. The algorithm then proceeds to the next prefix-key in the *partitionedMap*.

Following this algorithm, a compacted form of a traditional Trie is created that consumes less memory and retains performance.

IV. Benchmark and Results

Benchmark analysis was performed on the entire HashTrie implementation to analyse the time and space complexity.

To measure time complexity, we used the C++ *chrono* library which provides high resolution clock times with smallest possible tick periods. The function *now()* yields the current standard system wide time during execution of the program with flexibility to show results in seconds, milliseconds and microseconds. Every function of the HashTrie program was embedded with the *now()* function before and after the execution, results of which were re-directed to file, then tabled and graphed.

There are seven functions used by the HashTrie datastructure namely *breaker(), mapping(), prefixsearch(trietraversal), wordadd(pure trie traversal), wordsearch(pure trie traversal), wordsearch(trie traversal and hash access) and wordadd(trie traversal and hash traversal)*. The mapping function is a composite method with three methods namely *.addword, .addpref* and *.makehashtable*. The precision and accruacy of the *chrono* clocks depends on operating system and program run-time environment. The measure time may vary based on the tasks or threads run in parallel by the environment or ecosystems.

Function Name	**Avg Duration (ms)**	**Number of Runs**
breaker()	6.19	46
mapping (.addWord, .addPref, and .makeHashTable)	1.33	46
prefix search (trie traversal)	2.44	46
word add (pure trie mapping)	2.36	46
word search (pure trie traversal)	0.40	46
word search (trie traversal and hash access)	1.03	92
word add (trie traversal and hash access)	0.99	46

TABLE I: Performance Metrics: Average Duration in Microseconds and Number of Executions as Runs

The table above shows the execution time in microseconds for the functions used in the HashTrie data structure. The number of runs for the function *wordsearch(trie traversal and hash access)* is double the number of runs because the function is called twice during a single execution. Most of the execution time is front-loaded by the *breaker* function, while the other functions take less than half the execution time of *breaker* function. The *breaker* function takes approximately *6.19 microseconds*, while the rest of the functions take less than *3.0 microseconds*.

The execution times of all functions used by HashTrie data structure across different execution runs are shown in a graph below.

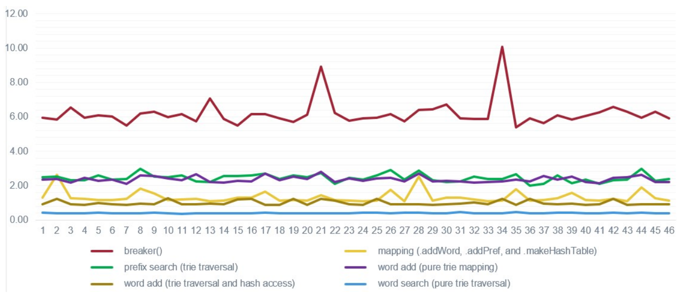


Fig. 2: Execution Time by functions

We compare the speed between generating random gibberish words (N) and the mapping speed in the HashTrie data structure. The speed of generating N gibberish words depends on the algorithm used, the length of the words, and the number of words required. Typically, generating random gibberish words is relatively fast, especially if the length and number of words are small.

The mapping speed in HashTrie refers to the process of adding words to the data structure. This involves traversing the Trie and performing hash table operations to store the words efficiently. The speed of mapping depends on the length of the word, the number of words being added, and the efficiency of the HashTrie implementation. Mapping speed in HashTrie is slower with randomly generated "gibberish" words. This is because mapping involves more complex operations like traversing the Trie and performing hash table lookups and updates.

The graph shown below shows the N (Gibberish) to mapping speed with a linear relationship [O($\sqrt{n}$)<O<O(n)], the same can be observed from the graph below.

Alternatively, we compared the speed between generated real words (N) and the mapping speed in the HashTrie data structure. Generating real words involves selecting words from a dictionary or word list. The mapping speed in HashTrie refers to the process of adding words from the real word list to the data structure. This involves traversing the Trie and performing hash table operations to store the words efficiently.

The graph below shows the N(Real Words) versus mapping speed with an approximate square-root relationship $O(n^2)$, the same can be observed from the following graph.

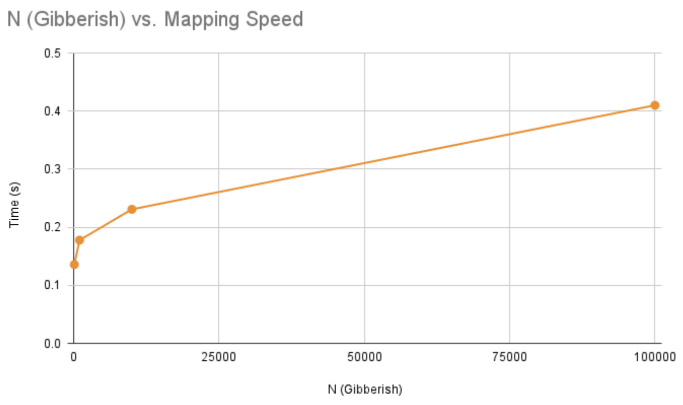


Fig. 3: Time Complexity of HashTrie when mapping words of randomized lengths and letter orders, or "Gibberish."

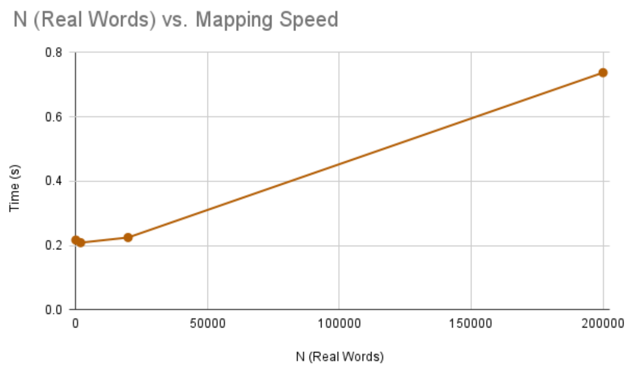


Fig. 4: Time Complexity of HashTrie functionality when mapping actual English dictionary words, or "Real Words."

It's important to note that the primary purpose of HashTrie is to provide an efficient data structure for storing and retrieving actual words, rather than storing random words. Therefore, the mapping speed in HashTrie is optimized for the efficient storage and retrieval of words, whereas storing random-character words is not the primary focus of our data structure.

Space complexity of HashTrie can be analysed from the memory footprint consumed against standard Trie implementations.

- Trie: Approximately 152 MB of space
- HashTrie: Approximately 13 MB of space

This was calculated by counting the number of hashed elements and Trie Nodes were generated during the entire mapping process, multiplying these counts by their respective sizes, and then summing the two values together. In code, this is expressed as follows:

$$T_{mem(MB)} = (n_{nodes} * size_{node} + n_{hashes} * (size_{unsignedLong} + size_{string}))/1024^2 \quad (1)$$

Hashtable size is one of the important factor in improving the performance of storage and retrieval of key-value pairs. Increasing the size of the hashtable can improve performance in several ways.

1) Reduced collisions: A large hashtable size provides more slots for storing key-value pairs. This reduces the likelihood of multiple keys being hashed to the same bucket, known as a collision. With fewer collisions, the hashtable can achieve better performance by minimizing the number of key comparisons needed to resolve collisions.
2) Improved look-up time: With large hashtable size, the average number of elements per bucket decreases. This results in a shorter linked list or fewer elements to search through when resolving collisions. Consequently, the lookup time for a specific key tends to be faster with a larger hashtable.
3) Lower load factor: The load factor of a hashtable is the ratio of the number of elements stored in the hashtable to the total number of buckets. A large hashtable size allows for a lower load factor, which means that the hashtable is less densely populated. A lower load factor reduces the chances of collisions and leads to more efficient retrieval and insertion operations.

It is important to note that there is a trade-off between hashtable size and memory consumption. Large hashtable size requires memory to store stable slots/buckets and requires a balance between the hashtable size and memory consumption.

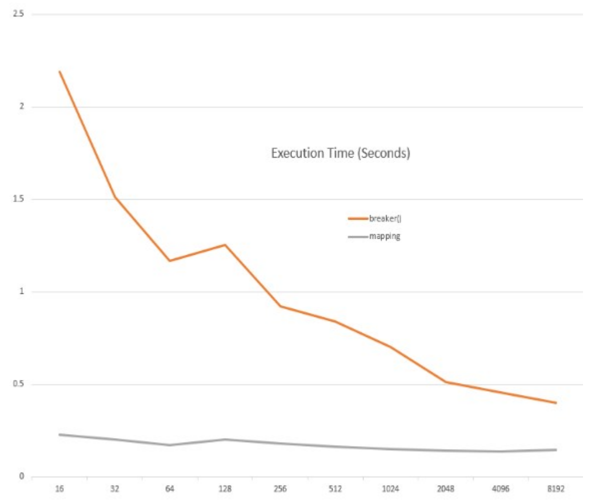


Fig. 5: Execution time for *breaker()* and *mapping()* functions with increase in hash size.

The impact of the hashtable size on the performance of each function is measured and plotted in a graph shown above. Increasing the hashtable size to 8192 has resulted in a significant improvement in the execution time of the breaker() function. The execution time has been reduced to $\frac{1}{5}$th of its original value, largest gain in the entire program.

Similarly, a graph was drawn for all other functions showing impact of performance. Increasing hashtable size do not have significant positive gain in performance. The execution time across all other functions are range bound with spike at hashtable size of 2048 as shown in the graph below.

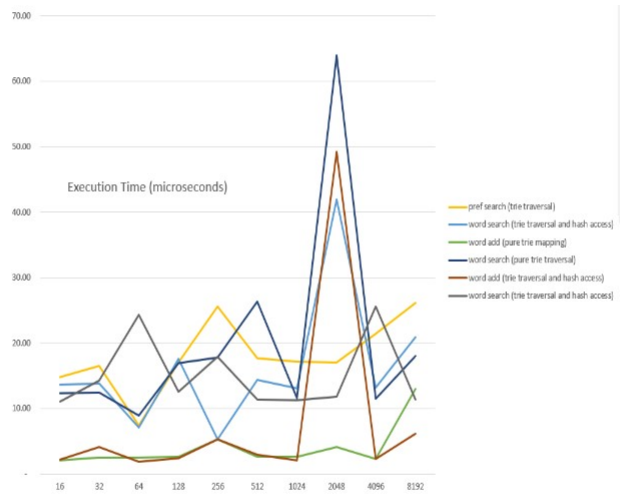


Fig. 6: Execution time for all other functions with increase in hash size.

V. Conclusion and Future Directions

The HashTrie data structure as it stands in this paper is a novel proof-of-concept of a data-structure that is more compact and just as efficient as a standard Trie. By incorporating a `unorderedMap` pointer and `isPrefix` Boolean, the HashTrie memory occupation was able to be reduced by an estimated factor of 10. If refined further, the HashTrie structure could be made into a very powerful data structure that sees commercial applications.

To further improve the HashTrie data structure, it is possible there may be ways to improve the Prefix-Partitioning Algorithm (PPA). Although the time-complexity of the PPA decreases drastically with an increase in pre-defined hashtable size, it nonetheless takes up largest amount of time and processing. This would be undesirable in implementations that cannot afford large hash-tables. Optimizing the PPA would reduce the bottleneck it creates during runtime, and therefore make it a significantly more appealing alternative to Trie for storing dictionaries.

Another potential improvement would be to optimize hashing, either by rewriting the implemented hash function for greater efficiency, using a more compact data type that works just as effectively for keys, or using the C++ proprietary hash function that is included in the ”unordered map” class. Implementing a hash function that generates more space efficient keys, or does so more quickly, can be an avenue worth exploring in improving the HashTrie structure.

References

[1] T. Zheng, Z. Zhang, and X. Cheng, “Saha: A string adaptive hash table for analytical databases,” *Applied Sciences*, vol. 10, no. 6, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/6/1915

[2] J. Lamping and E. Veach, “A fast, minimal memory, consistent hash algorithm,” 2014.

[3] K. Tsuruta, D. Köppl, S. Kanda, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda, “c-trie++: A dynamic trie tailored for fast prefix searches,” *Information and Computation*, vol. 285, p. 104794, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540121001103

[4] P. Bille, I. L. Gørtz, and T. A. Steiner, “String indexing with compressed patterns,” 2021.

[5] R. Lasch, I. Oukid, R. Dementiev, N. May, S. S. Demirsoy, and K.-U. Sattler, “Faster strong: string dictionary compression using sampling and fast vectorized decompression,” *The VLDB Journal*, vol. 29, no. 6, pp. 1263–1285, 2020. [Online]. Available: https://doi.org/10.1007/s00778-020-00620-x

Appendix

Additional images and diagrams for reference are on the following page.

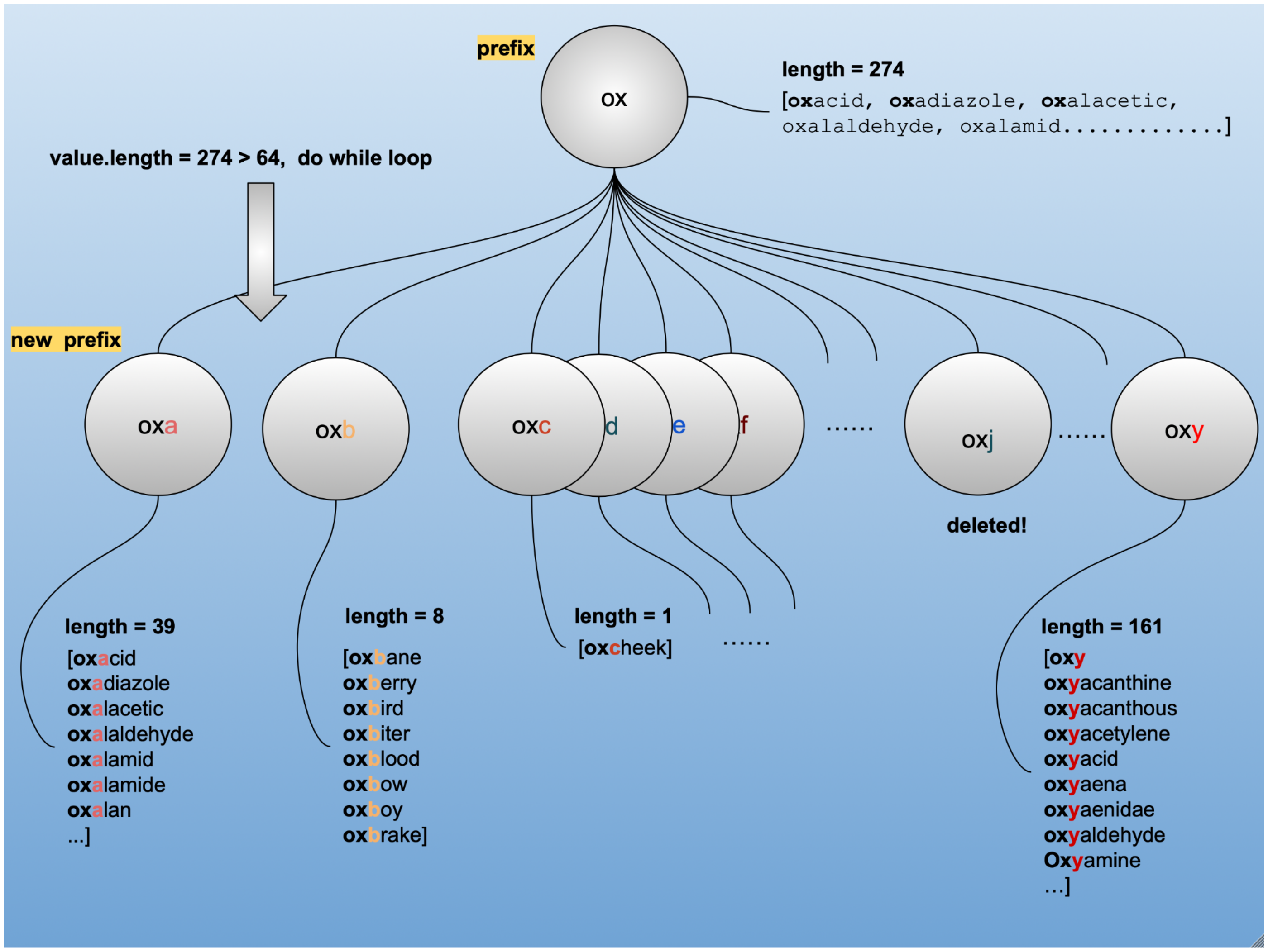


Fig. 7: Visual breakdown of the prefix partitioning algorithm.

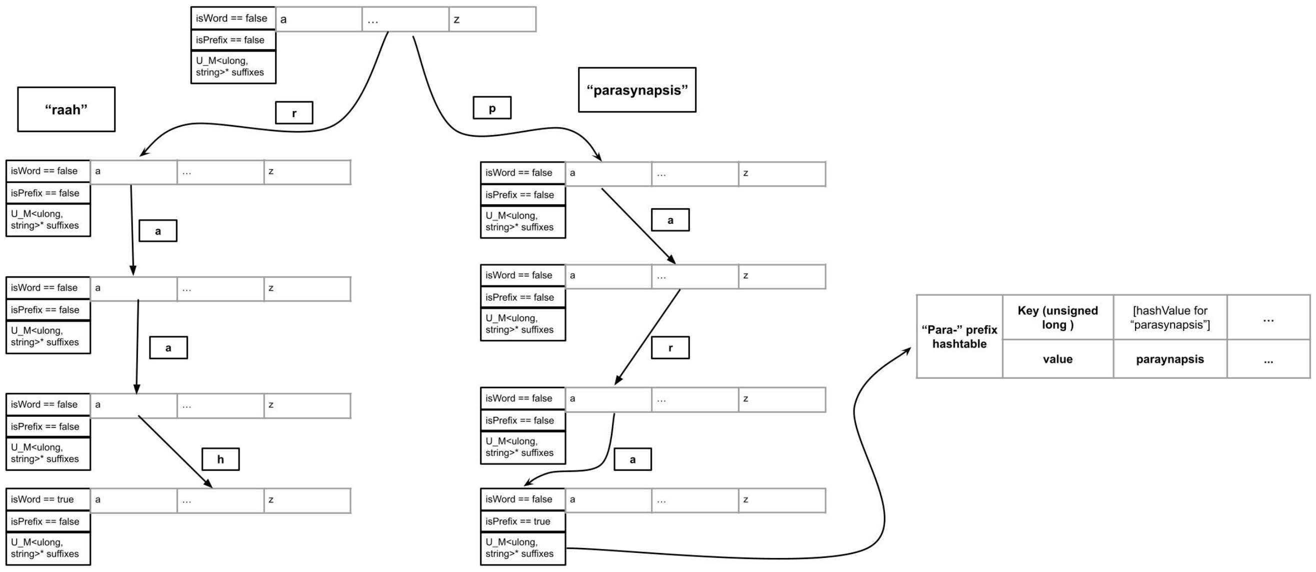


Fig. 8: Visual breakdown of TrieHash layout, with example words "raah" (unhashed) and "parasynapsis" (hashed).