****

**19CSE212-Data Structures and Algorithm**

**Trie-Based Bloom Filter**

Charan Kumar A – CB.EN.U4CSE21413​  
Dhanush S​ – CB.EN.U4CSE21416​  
Gautham Suresh​ – CB.EN.U4CSE21418​  
Joseph Jeffrey​ J – CB.EN.U4CSE21425​  
Krishna Kumar S – CB.EN.U4CSE21432​​

**Introduction:**

**Trie-Based Bloom Filter:**

A Trie-Based Bloom Filter combines the characteristics of both a Trie and a Bloom Filter. A Trie-Based Bloom Filter provides better memory efficiency and precision. This hybrid data structure is useful when dealing with large sets of data where memory optimization is desired.

**Some of the key significance of hybrid data structure is:**

- Tries are memory-efficient data structures that can store a large number of strings. By using the hybrid data structure we can reduce the memory usage. It is very efficient when dealing with large sets of strings.

- The Bloom Filter has constraints in the fixed probability of a false match which means when the data structure incorrectly indicates that an element is a member of a string, even though it is not. The Trie offers additional details to find the element's presence, improving accuracy.

- Tries provide in-built support for dynamic resizing which means the ability of a data structure to adjust its size or capacity. The Hybrid data structure is not constrained by the need for a fixed-size bit array, unlike the Bloom Filter.

- Tries are very efficient for search operations like pattern matching, substring search, and prefix matching. A Trie-Based Bloom Filter may effectively carry out such operations, delivering quicker and more accurate results.

**Projects Objective:**

The project's goal is to create and build a hybrid data structure that combines the properties of a Trie and a Bloom Filter. The Trie-Based Bloom Filter seeks to provide efficient string storage, retrieval, and string presence. The study also involves identifying practical uses for this hybrid data structure. The study will also include an analysis of the time and space complexity of the Trie-Based Bloom Filter in order to better understand its efficiency and performance characteristics. It searches for the element in a large data set and returns true if the element is present in the data set. We are implementing the Hybrid data structure in Python.

**Overview:**

**Trie Data Structure:**

The Trie is a tree data structure that is used for storing strings. The root node of a trie will represent an empty string, each node contains a single character of the string and a flag which identifies whether the string is completely traversed. These nodes are linked together and could be traversed in a depth-first fashion(which means it starts from the root node). This makes it very efficient in searching for strings. The algorithm stops searching as soon as it finds a mismatch.

**Bloom Filter Data Structure:**

The Bloom Filter is used for checking whether the element is present in the set or not. A Bloom filter uses a bit array and multiple hash functions to store elements. It is commonly used for applications like caching, spell-checking, and data filtering.

**Trie-Based Bloom Filter:**

A Trie-Based Bloom Filter combines the characteristics of both a Trie and a Bloom Filter. A Trie-Based Bloom Filter provides better memory efficiency and precision. This hybrid data structure is useful when dealing with large sets of data where memory optimization and approximate answers are desired.

**Advantages and motivations :**

- It searches for the element in a large data set and returns true if the element is present in the data set.

- We are implementing the Hybrid data structure in Python.

- It can assist reduce false positives.

- The hybrid technique efficiently provides dynamic changes to the set.

- The hybrid structure creates a balance between space efficiency and query performance

**Implementation:**

Trie Bloom-based filter is a data structure which is a combination of Trie and Bloom Filter that performs an algorithm to find the presence of the element that is being searched and also to perform operations like insertion, and deletion and to display the elements present.

During the insertion of elements, hash function is used so that there won't be any confusion between the allocation of the characters at their respective indexes with the bonds formed.

There are numerous other data structures that can be used instead of Trie based Bloom Filter such as bit array, Cuckoo Filter, Quotient Filter, etc., but the main reason to go for Trie based Bloom Filter is that it saves memory, low false positive results, time efficient, etc., hence Trie based Bloom filter is implemented in various areas like web browsers, auto spell check, auto-filling, etc.

**Code Explanation:**

**“**

from bitarray import bitarray

class TrieNode:

def \_\_init\_\_(self):

self.children = {}

self.is\_end\_of\_word = False

**“**

Bitarray are arrays that store’s Boolean values and that library package is imported. A class with the name trienode is created in order to store elements(children) and flag value is set to false.

“

class TrieBloomFilter:

def \_\_init\_\_(self, capacity):

self.root = TrieNode()

self.capacity = capacity

self.bloom\_filter = bitarray(capacity)

self.bloom\_filter.setall(False)

”

Another class is created with the name TrieBloomFilter where 8 functions are created inside it and the original capacity of the TrieBloom Filter is to be stored and the flag position is stored at the end of the word in the first function.

“

def insert(self, word):

node = self.root

for char in word:

if char not in node.children:

node.children[char] = TrieNode()

node = node.children[char]

node.is\_end\_of\_word = True

# Set the corresponding bits in the bloom filter

self.bloom\_filter[self.get\_hash(word)] = True

“

A function is created to insert data into the TrieBloomFilter where the entered value or the element is aligned into the TrieBloomFilter at an appropriate index using the hash function which helps to avoid data colliding and once the element gets inserted a flag is assigned at the end of it.Once the flag is set the element is inserted successfully.

**“**

def delete(self, word):

node = self.root

parent = None

for char in word:

if char not in node.children:

return 0# Word not found

parent = node

node = node.children[char]

if not node.is\_end\_of\_word:

return 0# Word not found

# Check if the word is really present using the bloom filter

if not self.bloom\_filter[self.get\_hash(word)]:

return 0# Word not found

# Delete the word from the trie

node.is\_end\_of\_word = False

# Remove any unused nodes from the trie

self.\_remove\_unused\_nodes(parent, word)

def \_remove\_unused\_nodes(self, parent, word):

if not parent:

return 0

if len(parent.children) == 1 and not parent.is\_end\_of\_word:

char = word[len(parent.children)]

if char in parent.children:

del parent.children[char]

self.\_remove\_unused\_nodes(parent, word)

**“**

A function to delete the elements from the TrieBloomFilter is created in which it searches for the element to be deleted in the TrieBloomFilter. Once found that the element is found the word gets deleted only when the flag is present at the end of the last character in the string. There can be two possible cases in deletion one is that the word entered to delete may not be present in the TrieBloomFilter and the another case is that the element is present. When the element isn’t present it results a false and if the word is present it deletes the element and the flag that recognizes the word that is to be deleted.When two elements for example ‘and’ and ‘ant’ is present in the data structure and you want to delete ‘ant’. In this case the ‘a’ & ’n’ of ‘ant’ gets deleted along with ‘t’ but not the ‘a’ & ‘n’ of ‘and’. And hence as a final step the unused nodes or the nodes of the element that are deleted will be deleted and further used to store the next upcoming data’s

**“**

def search(self, word):

node = self.root

for char in word:

if char not in node.children:

return False

node = node.children[char]

return node.is\_end\_of\_word

**”**

Here is the function created to search for the presence of element in the data structure where the entered element is searched character by character in the TrieBloomFilter and once it comes to the last character it see’s for the presence of flag at its end if present it results true and if not it results a false.

**“**

def display(self):

words = []

self.\_traverse(self.root, "", words)

for word in words:

print(word)

**“**

This is a function to display all the elements present in the TrieBloomFilter by traversing through it. Once traversed the word is printed on a loop format.

**“**

def \_traverse(self, node, current\_word, words):

if node.is\_end\_of\_word:

words.append(current\_word)

for char, child in node.children.items():

self.\_traverse(child, current\_word + char, words)

**”**

Traverse is a function used to display all the contents in the TrieBloomFilter. It searches for the word and ensures for the flag at the end and returns the value to the display function.

**“**

q=1 #exit condition

while(q>0):

qwe=0

x=int(input("Enter the required size of the Tri Bloom Filter : "))

trie=TrieBloomFilter(x)

n=int(input("Enter the total number of elements you want to enter inside the Trie Bloom Filter : "))

while(x>n):

k=0

while(k<n):

data=input("Enter data item to be inserted : ")

if(data.isalpha()==True):

trie.insert(data)

k=k+1

else:

print("Only strings are accepted")

k=k-1

n=n-1

p=2

while(p>1):

print("Enter 1 to insert data")

print("Enter 2 to delete data")

print("Enter 3 to search data")

print("Enter 4 to display the data in the Trie Bloom Filter")

print("Enter 5 to EXIT")

choice=int(input("Enter your choice : "))

while(choice==1 or choice==2 or choice==3 or choice==4 or choice==5):

if (choice==1):

n=int(input("Enter the number of elements you want to enter inside the Trie Bloom Filter : "))

k=0

while(k<n):

data=input("Enter data item to be inserted : ")

if(data.isalpha()==True):

trie.insert(data)

k=k+1

else:

print("Only strings are accepted")

k=k-1

n=n-1

break

elif (choice==2):

data=input("Enter data item to be deleted : ")

if(trie.search(data)==False):

print("Enter Correct data item or the item present in the data structure")

elif(trie.search(data)==True):

trie.delete(data)

print("Successfully deleted "+data+" from the data structure")

break

elif (choice==3):

data=input("Enter data item to be searched : ")

print(trie.search(data))

break

elif(choice==4):

print("The list of items in the Trie Bloom Filter is : ")

trie.display()

break

elif (choice==5):

p=0

qwe=1

break

break

else:

print("EXIT SUCCESSFUL")

if qwe==1:

break

else:

print("Enter the number of items to be inputted less than the trie bloom's capacity")

if qwe==1:

break

else:

q=q+1

“

Here is the code under the main phase where the user is expected to input the capacity and the number of elements to be inserted and then 5 choices for the user is displayed from t=which they could choose one.

The 5 choices for the user is to insert new data, delete an element, search for an element, display list of elements, exit.

The choice function is done with the help of 5 while loops where in variables like q, qwe are used to exited from the loop.

In the choice 1 the user enters new element into the data structure and in order to insert the input is checked whether it is an complete string or not if a string it accepts and if not the user gets notified and the user have to enter the data again to be inserted.

For the other choices the functions are called directly to operate according to the users choice.

The users choice is validated using if condition and gets it to the appropriate function.

This functions runs at an infinite loop and once the user enters his choice to exit the condition for the while loop is set such that it gets breaked.

**Code Repository link:**

<https://github.com/jjeffreyj/Algosaints/blob/jjeffreyj-DSA-Code/DSA.py>

**Practical Application:**

1. **Text Auto Completion:**

* Trie data structure is popularly used for text predictions and auto-completion. As many words can be stored in it, it can suggest word completion for partially entered words.
* Firstly a trie with all words has to be created, a bloom filter should be created along with it and each word inserted in the trie should be hashed with the bloom filter. When the user enters a partial word, the whole trie gets traversed, for each node traversed if it exists in the bloom filter, the word gets popped up as a candidate for word completion.

1. **Checking for Spelling Errors:**

* It can also be used for checking the spelling of a word by checking for its spelling in the dictionary of words entered in it previously.
* Firstly given an input string, the trie gets traversed based on the first letter of the word, and it moves on to further child nodes by tracing the whole word. If the current character does not exist in any node, it results in a spelling error.

1. **Dictionary:**

* As many words are entered previously and as they provide very fast and simple insertion, deletion and retrieval operations, they are best suited for dictionary purposes.
* A trie with all words has to be created, a bloom filter should be created along with it and each word inserted in the trie should be hashed with the bloom filter. To access a dictionary, the first word of the string must be entered and the next entered letters must be checked similar to that of text auto-completion and should be suggested as a word to the user, if the next entered character does not get recognized by the bloom filter it should result in an error i.e. the word is not found.

1. **Data Redundancy:**

* They are mainly used to avoid data redundancy as they can easily check for its duplicity. This can very well be used in scenarios where databases play a vital role.
* Data must be entered first, then it must be checked with bloom filter whether the data already exists in the trie or not, if data exists in the trie a search should be initiated in the trie for that exact data match. If data doesn’t exist in the trie, the data get added to it else the data can be skipped.

1. **Data Retrieval in Cache:**

* Bloom filter can be used in caching systems to identify whether an item is present in a cache or not.
* To check for cache memory, data must be entered first and it must be applied to the bloom filter to check whether the key exists in the cache. If the key is present a trie search should be initiated to find a data match and to retrieve it. If not the entered data is not previously cached in the cache memory.

1. **Web URLs:**

* Bloom filters are used in search engines to check for re-visited sites. It can also be used to identify whether a website has been modified since it has been last visited and saved in cache memory.
* Already visited URLs will usually be added to the trie and hashed using multiple hash functions to the bloom filter, so the upcoming new websites visited will further be hashed to the bloom filter and added to the trie making sure that the new URLs are added.

**Performance Analysis:**

A Trie-Based Bloom Filter, being a hybrid form of the combination of two data structures, provides to be very useful in several ways.

It has the combined characteristics of a Trie and a Bloom Filter.

It provides several benefits to the user including:

* Space Efficiency
* Easy Implementation
* Scalability
* Fast Membership Queries
* Low False Positivity Rate.

**Time Complexity:**

* **Insertion:**

The time complexity for the insertion of an element in a trie-based bloom filter is O(L) where L is the length of the element or word.

* **Searching**

The time complexity for searching an element in the trie-based bloom filter is also O(L) where L is the length of the element.

* **Deleting**

The time complexity for deleting an element in the trie-based bloom filter is also O(L) where L is the length of the element.

**Space Complexity:**

* **Insertion**

The space complexity for insertion of an element is O(n\*m) where n is the number of characters and m is the average branching factor.

* **Searching**

The space complexity for searching for an element is O(L) where L is the length of the element.

* **Deletion**

The space complexity for removing n nodes results in a space complexity of O(L+n).

* **Overall**

The overall space complexity of the trie-based bloom filter involves traversing through L elements.

**Performance Comparison:**

* The time complexity of a trie-data structure generally for insertion and searching is O(n).
* The space complexity is O(n\*k) where n is the number of nodes and k is the number of unique characters.
* A bloom filter is very efficient when it comes to the usage of time and space.
* The time complexity of a bloom filter with k hash functions for insertion and searching is O(k). In a bloom filter, the time complexity does not depend on the number of elements.
* To truly understand the space complexity of the bloom filter, you have to first choose your parameters. You could make a bloom filter with k=1 and it would just be a hash table that ignores collisions. However, you would have a very large O(m) if you wanted to keep your false positive rate low. The space of the actual data structure is simply O(m).
* Firstly, considering the Space efficiency, the trie-based bloom filter offers better space efficiency i.e., uses less space compared to the trie and bloom filter data structures individually. This means it uses less memory space.
* The trie-based bloom filter, however, uses multiple hash functions which can increase the possibility of collisions.
* The lookup efficiency of this hybrid data structure is slightly slower than that of the regular bloom filter. This is because an additional check is required in the trie to confirm the presence of the element which may increase the time a bit.
* The insertion efficiency of a trie-based bloom filter is almost similar to or slightly less than a bloom filter. This is because the trie-based bloom filter may require additional node updates in the trie structure which may reduce the efficiency a bit.
* In the case of memory usage, the trie-based bloom filter requires less memory space than its components mainly because it eliminates the need to store the complete elements in the trie resulting in saving space.
* So overall, a trie-based bloom filter has better space efficiency and memory usage than its individual components i.e., trie and bloom filter. However, it lacks a bit in the case of lookup efficiency and insertion efficiency.

**EXPERIMENTAL EVALUATION :**

1. **Performance Metrics:**

Performance metrics include factors such as execution time, memory usage, and scalability.

1. **Test data:**

This is about ensuring that the dataset is diverse, covering different patterns of inputs. Here in the code implemented, we do not explicitly specify a predefined dataset. Instead, it allows the user to interactively input data items to be inserted into the Trie Bloom Filter. Since the code allows the user to input their own dataset, the specific dataset will depend on the user’s input during runtime.

1. **Implementation Testing:**

This includes verifying the correctness of the Trie-based Bloom Filter by running tests on individual operations like insertion, deletion, and search and ensuring that the expected results match the actual results.

**The practicality and effectiveness of the Trie-based Bloom Filter in real-world scenarios:**

The hybrid data structure that was created, a Trie Bloom Filter, combines the benefits of a trie with a bloom filter to produce a data structure that is both space-efficient and quick to perform lookup operations. However, a number of factors affect how useful and successful this hybrid data structure is in practical situations.

* Dataset Characteristics:

A Trie Bloom Filter's efficiency is dependent on the features of the dataset it is employed with. It works well when the dataset contains a lot of distinct components and shows a lot of prefix similarity. Examples include word lists, URL databases, dictionaries, and any situation where the elements share a lot of prefixes. The trie structure can effectively store and search for items in certain circumstances, while the bloom filter lowers the number of unnecessary trie traversals.

* Memory Restrictions:

The Trie Bloom Filter offers memory efficiency compared to a pure trie data structure by using a Bloom Filter to reduce the number of unnecessary trie traversals. This makes it suitable for scenarios where memory is limited, and efficient memory utilization is crucial. However, it's important to consider the trade-off between memory usage and the desired false positive rate. Increasing the bloom filter size reduces false positives but increases memory consumption.

* Insertion and Deletion Overhead:

The insertion and deletion operations require a traversing trie structure, while the Trie Bloom Filter provides fast search operations. To insert or remove a word, the trie node must be configured to have an overhead value that is proportional to its length. The overhead of inserting and deleting data should thus be taken into account when it is assumed that there will be repeated updates to the database.

* False Positive Rate:

The possibility of false positives is introduced by the bloom filter part of the Trie Bloom Filter. When the filter reports a word as being present, even though it is not, false positives occur. The bloom filter size and number of elements that have been added increase the False Positive Rate. An analysis of the impact of false positiveness on a given application or use case is essential. Alternative data structures may be more appropriate when false positive values are not acceptable.

* Scalability:

The Trie Bloom Filter's scalability is determined by dataset size and the bloom filter's capacity. False positive rates can increase and affect the accuracy of searching operations when data size is larger than bloom filter's capacity. Therefore, the appropriate size of the bloom filter and the allocation of the memory should be carefully considered for large scale datasets.

**Real-world applications:**

* + **Spell checking:** In spell-checking algorithms, the Trie Bloom Filters can be used to efficiently identify the correct spelling of words that have been misspelt. Trie architecture allows quick prefix search and retrieval of similar words, while Bloom filter helps to remove unintelligible words.
  + **Network Routing and IP Lookups:** Trie Bloom Filters can be used for network router and IP address searches. They are capable of easily storing and searching for routing prefixes or IP addresses so that they can take swift decisions on route selection and forwarding.
  + **DNA Sequence Matching:** Trie Bloom Filters find applications in DNA sequence matching and bioinformatics. In order to store and search DNA sequences it is possible to use them, so that they are quickly identified as known sequences as well as filtering out unlikely matches.
  + **Data Deduplication:** if redundant data is to be identified and eliminated, Trie Bloom Filters may help in deduping tasks. The deduplication process can be more efficient if you store individual data items in the Trie, and use Bloom filters for rapid identification of possible duplicates.
  + **Caching and searching optimisation:** Trie Bloom filters can be used in cache systems to enhance your search efficiency. Cache systems can reduce the cost of back office searches and provide faster responses, by archiving frequently accessed data items in the Trie via Bloom filter to exclude unidentifiable ones.
  + **Keyword Filtering and Profanity Detection:** Trie Bloom Filters can be employed in content filtering systems to quickly identify and block offensive or inappropriate keywords. They will make them useful for the detection of swear words and content moderation, as they can match incoming text with a defined set of keywords.

**Limitations, challenges and potential future improvements:**

1. **False Positive Rate:** The bloom filter component of the Trie Bloom Filter introduces the possibility of false positives. While the false positive rate can be controlled by adjusting the size of the bloom filter, achieving a very low false positive rate may require a larger bloom filter size, leading to increased memory usage. Balancing false positive rates and memory usage is a challenge, and future improvements could focus on developing more sophisticated techniques to reduce false positives without significantly increasing memory requirements.
2. **Memory consumption:** The Trie Bloom Filter uses both the trie filter structure and the blooming filter so that it incurs additional memory overhead in comparison to a simple trie or bloom filter. Although it provides memory efficiency compared to a pure trie, memory usage can still be a limitation in scenarios with large datasets. The next area of improvement is the exploration of techniques to optimize memory utilisation in a way that does not compromise performance.
3. **Scalability:** The functionality of the Trie Bloom Filter depends on your bloom filter's capacity. False positive rates are increased in cases where a dataset is too large for bloom filters, affecting the accuracy of search results. To cope with growth in data, the next improvements could be focused on dynamic resizes of bloom filters to improve scaling and maintain a desired level of false positives.
4. **Update efficiency:** To perform insertion and deletion operations in a Trie Bloom Filter, it is necessary to cross the trie structure that can introduce overheads, especially for large words or frequent updates. The effectiveness of updates can be improved by optimising insertion and deletion operations so that they reduce the traversal and modification complexity of the trie structure.
5. **Flexibility and unification:** Trie Bloom filters have been applied in particular to strings. The flexibility and applicability in a broader range of scenarios can be enhanced by increasing the data structure's ability to work with different types of data or allowing for customizable key value Maps.
6. **Parallelized and distributed systems:** The performance may be improved by using complementary techniques or the distribution of Trie Bloom Filters on a number of computers, which can make it possible to handle bigger data sets. Future improvements could look at ways to parallelize searches, inserts and deletions operations as well as distribute the data structure between several nodes.

**CONCLUSION :**

* A trie-based bloom filter is thus a hybrid data structure that can be used in today’s world to improve efficiency in providing solutions to several real life problems.
* This hybrid data structure could be used more widely as it is being under utilized now. It helps to solve even complex problems by using properties of both the trie data structure and the bloom filter.
* Thus, a trie based bloom filter is a very useful data structure which could be used in day to day lives.

**REFERENCES :**

<https://www.geeksforgeeks.org/introduction-to-trie-data-structure-and-algorithm-tutorials/>

<https://llimllib.github.io/bloomfilter-tutorial/>