Cuckoo Filter and its improvement on number of relocations

Zdravko Grgić

Mladen Džida

May 2022

1 Introduction

With advances in technology it is easier to store and acquire large amounts of data. Analyzing these mass amounts of data can quickly become very difficult if one wants to do extensive search. This paper tries to solve this problem for a specific application of designing a space and computation efficient set representation and and for membership testing. Simply put, in this paper we will describe an algorithm that can store items and check whether an item exits in this data structure in an computationally and memory efficient way for big data. This algorithm is called Cuckoo filtering. Idea behind it is that it will give out an approximate solution, i.e. it works as a probabilistic data structure. For example if we would want to check if an item is inside a data structure we would not always get right answer since it is a probabilistic structure and it does not give out an exact answer. This might seem like a problem and a bad solution, but the amount of computation time and memory that these structures save compensate quite a lot for relatively small number of cases where this algorithm will fail. In section 2 we will define what Cuckoo Filter is and in section 3 we will talk about our own implementation and results.

2 Cuckoo Filter

Cuckoo filter is a type of probabilistic data structure that uses Cuckoo Hashing for storing items and later checking if they are stored in a fast and memory efficient way. The name of this type of hashing comes from habits of a bird called cuckoo where the cuckoo chick pushes the other eggs or young out of the nest when it hatches. Similar idea is implemented for inserting elements in a data structure. This data structure will be defined as a hash map and a general idea behind this algorithm is: each item that should be put in a hash map will be hashed by two hash functions and therefore we get to hashes which will serve as indexes in a hash map where this item might be stored. Algorithm does not store item in a raw format, but also saves only a hash of the item to save space and this form of an item is called fingerprint. So, each item can be stored only in two locations and if the fingerprint associated with this item is not in any of the two locations, we can safely say that this item is not stored in this data structure. But, if we find a fingerprint that relates to an item in first or second location or bucket, we can't be sure that this item really is stored in this structure because some items might have the same fingerprints so it is clear that this algorithm will not produce false negatives, but it can produce false positives. Also, when storing items, there might not be free space in any of the two buckets and therefore collision might occur and the way we handle them is similar to the way the cuckoo chicks behave, where in this example a new item will kick one already stored item and this new item will go on to another bucket where it might kick another out and so forth. In next two subsections we will more formally define insert and lookup function for this data structure.

2.1 Insert function

First, we will expand the said structure by allowing each bucket to store more than one fingerprint by setting each bucket to have a fixed number of compartments. Inserting a new element is done in three different ways, where each is associated with a different case regarding whether each of these buckets have free space to store a new fingerprint of an item. First case is when both of the buckets have space to store fingerprint and the way inserting is done is by randomly choosing the bucket to store the fingerprint for this item. There is a more efficient way to this and it involves more than just randomly choosing the bucket to store fingerprint, but rather a better way is to choose a bucket which has more free space and by this we will see that we will actually reduce the number of collisions and number of fingerprint relocations. This is essentially a technique we will use to handle the structure space more efficiently and the source for the title of this research. Second case occurs when one bucket has space to store fingerprint and the other does not. This case is solved by simply inserting a fingerprint in a bucket which has space. And the third case occurs when neither of the two buckets have space to store new fingerprint. If this happens, a bucket is randomly chosen out of the two and inside the chosen bucket we again randomly choose a fingerprint inside which will be relocated and the new fingerprint will be stored in its place. Now is a good time to talk more about hash functions used to get indexes of the table. We said that we need two indexes in a table and for that we use two hash functions, but this statement is not completely true because we will need to be able to get second hash from the first hash and the fingerprint, where fingerprint will also be a hash of some sort. A common way this is done is getting hash from the item using arbitrary hash function, and then splitting this hash into two parts where one part will be used as first hash and the other part will be used a fingerprint. The second hash will be calculated from the first hash and the fingerprint by combinging them via the XOR function wherer we XOR the first hash with another hash we calculate from the fingerprint(using the same or a new hash function). This is presented on a figure 1 where x is the input and the greek letter in second equation is label for fingerprint.

Figure 1:

$$h_1(x) = hash(x)$$
$$h_2(x) = h_1(x) \oplus hash(\xi_x)$$

This way of calculating hashes is useful for this algorithm because we can easily handle relocations of the fingerprints. If we go back to the part when we said we will place new fingerprint in a compartment of another one which we will relocate, we quickly realise the benefits of getting second hash from the first one. If we need to relocate the fingerprint,we need to get the second location where this fingerprint might be and that means getting the second hash. Because we deal with fingerprints and not the data itself, we can not get the hash from the fingerprint itself since it is not guaranteed our fingerprint is unique. So, we use the fingerprint and the first hash to obtain the second hash. Also, it is important to here that indexes will be calculated from hashes from modulo operation with number of buckets and fingerprint will be obtained from modulo operation of hash with number of bits used for fingerprint.

Figure 2:

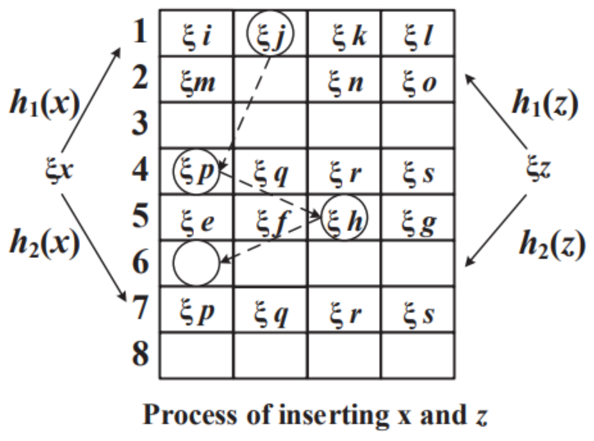
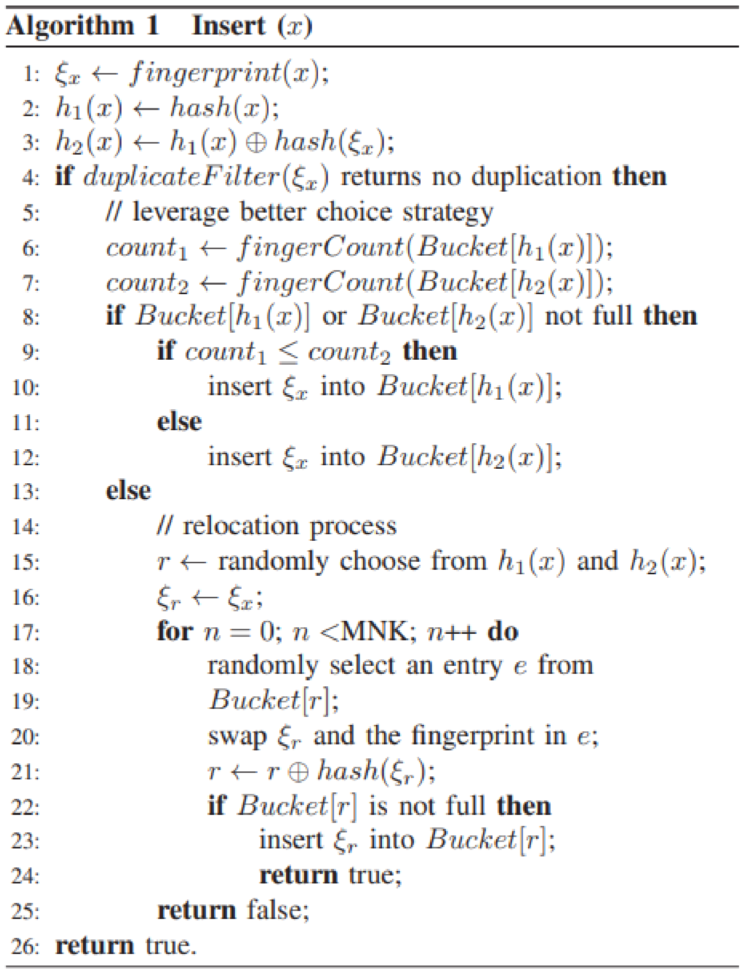


Figure 2 presents an example of inserting two elements, x and z where element x gets first hash to point to first location of the table and the second hash points to seventh location. Each of these locations does not have space to store a new element, so relocation of a random element(jth element) is performed and the process continues. Figure 3 shows complete pseudocode for inserting in cuckoo table.

Figure 3:



Algorithm 1 **Insert** (x)

```
$\xi_x \leftarrow fingerprint(x)$;
$h_1(x) \leftarrow hash(x)$;
$h_2(x) \leftarrow h_1(x) \oplus hash(\xi_x)$;
if $duplicateFilter(\xi_x)$ returns no duplication then
    // leverage better choice strategy
    $count_1 \leftarrow fingerCount(Bucket[h_1(x)])$;
    $count_2 \leftarrow fingerCount(Bucket[h_2(x)])$;
    if $Bucket[h_1(x)]$ or $Bucket[h_2(x)]$ not full then
        if $count_1 \leq count_2$ then
            insert $\xi_x$ into $Bucket[h_1(x)]$;
        else
            insert $\xi_x$ into $Bucket[h_2(x)]$;
    else
        // relocation process
        $r \leftarrow$ randomly choose from $h_1(x)$ and $h_2(x)$;
        $\xi_r \leftarrow \xi_x$;
        for $n = 0$; $n <$MNK; $n$++ do
            randomly select an entry $e$ from $Bucket[r]$;
            swap $\xi_r$ and the fingerprint in $e$;
            $r \leftarrow r \oplus hash(\xi_r)$;
            if $Bucket[r]$ is not full then
                insert $\xi_r$ into $Bucket[r]$;
                return true;
        return false;
return true.
```

Important thing to mention here is that if we get a lot of relocations when we insert a new item, we will stop the algorithm if it reaches maximum number of allowed relocations. This number is also called maximal number of kicks or MNK for short. If the algorithm reaches MNK, the last item that was kicked out and meant for relocation is simply left out of the table and the algorithm precedes to the next item for insertion.

2.2 Lookup function

Lookup function is much simpler than insert function. Everything it does is checking whether a fingerprint of an element exist in either of the two buckets in the table obtained from the two hashes. If fingerprint exists, the function returns boolean value true and if there isn't that exact fingerprint in either of the buckets, the function returns boolean value false. If an item is stored in the table, its fingerprint will be stored in one of the two possible locations, so we can be sure the item is not in the table if its fingerprint is not in the right buckets. But, if we conclude the item is inside the table from seeing the fingerprint in one of the two buckets, it does not have to mean that exact item was stored in the table because it is possible for multiple items to have the same fingerprint and for them to have one location the same. In short, false negatives aren't possible and false positives are possible.

3 Implementation and results

we implemented two versions of cuckoo filter, one using cuckoo filter with random choosing of bucket and one version where the bucket with more space is chosen to store the item. Our class of cuckoo filters takes an argument which is used to denote which version on cuckoo filter is used. Filters were implemented in C++. In the next two subsection we will describe in more detail how we implemented the algorithm and how did our results look like.

3.1 Implementation in C++

For the hash function, we used SHA1 hash function which we obtained from openssl library. From the hash function we took first 32 bits and used it as first hash and took second 32 bits and used those as fingerprint. First hash and fingerprint were also put in modulo functions with number of buckets and number of bits of fingerprint with each bit set to 1. Second hash was calculated from a XOR operation of these two sequences, but for the XOR we also calculated hash again from the fingerprint as explained above. For the data we used Escherichia coli genome for which we took sets of k-mers as items to be stored in the table. For testing, we randomly chose sequences of k length and checked whether they are stored in the table, where we might get positive result but the k-mer is not is the table since it might have been left out due to the algorithm reaching MNK. We implemented two classes: Table and CuckooFilter. Table class handles the processes of insertion and lookup, while CuckooFilter handles hashing, loading data and redirecting the data to the table. Each of these class has few more functionalities, but these are for printing the metrics we used for evaluation of the algorithm.

3.2 Hyperparameters and results

4 Conclusion

In this research paper we experimented with probabilistic data structure called Cuckoo Filter. This data structure has proved to be useful when dealing with big data since it reduces the time and space constraints of the problem. This does not come without a cost and for this we accept a certain probability that some of our results will be false positives. Nevertheless, the pros from this algorithm are greater than its cons.

References

[1] Feiyue Wang, Hanhua Chen, Liangyi Liao, Fan Zhang, Hai Jin: The Power of Better Choice: Reducing Relocations in Cuckoo Filter

[2] BIN FAN, DAVID G. ANDERSEN, AND MICHAEL KAMINSKY: Cuckoo Filter: Better Than Bloom