LSM Tree Optimization Through The Implementation of Cuckoo Filters

Humberto Cesar Villalta Valverde
KwangSik Kim
Kisu Kim
Jinman Kwon
Jaechoon Lim
Yongjoo Jun (report@dtonic.io)

Method Article

Keywords: probabilistic data structures, indexing algorithms, query optimization, cuckoo filters, bloom filters

Posted Date: August 8th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-3226421/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License. Read Full License

Additional Declarations: No competing interests reported.
LSM Tree Optimization Through The Implementation of Cuckoo Filters

Humberto Cesar Villalta Valverde\textsuperscript{1}, KwangSik Kim\textsuperscript{1}, Kisu Kim\textsuperscript{1}, Jinman Kwon\textsuperscript{1}, Jaechoon Lim\textsuperscript{1}, Yongjoo Jun\textsuperscript{1}\textsuperscript{*}

\textsuperscript{1*Dtonic, Seohyeonro, Seongnam, 13591, Gyeonggido, South Korea.}

\textsuperscript{*Corresponding author(s). E-mail(s): report@dtonic.io; humberto521336@gmail.com;}

Abstract

This study aims to develop an LSM tree that uses two different kinds of probabilistic data structures (PDS). These two data structures are the Bloom Filters and the state-of-the-art Cuckoo Filter released in 2014 by Fan. Cuckoo filters are a perfect choice for saving space and also for deleting keys that are not needed in the algorithm so the filters are synchronized with the keys of all the SSTables. The implementation in this study of Cuckoo filters showed three main advantages (1) Keys that are deleted in the SSTables are immediately reflected in the Cuckoo Filters (2) When SSTables are merged, Cuckoo Filters are also merged by measuring its load factor for availability (3) The performance of Cuckoo Filters compared to Bloom filters after deleting and query shows better performance despite that cuckoo filters are also being updated. This approach is expected to reduce storage space and the amount of time for queries and deletions on the overall data structure. This new optimized LSM tree is expected to be implemented in HBase as future work to check its performance with a real-world database for more empirical results.

Keywords: probabilistic data structures, indexing algorithms, query optimization, cuckoo filters, bloom filters

1 Introduction

At present, systems are scaling exponentially. Therefore developing a database or algorithm that can withstand the high insert and query operations is a matter of importance. One of the main purposes of Big data is to provide solutions for data
access in the shortest period possible but to also process that data as efficiently as possible with optimized tailored databases for that specific task. These databases can be scaled further to have a better data-distributed and operation-efficient cluster of regions as it is called for HBase.

Specialized indexing algorithms improve the overall performance of databases. However, indexing the data does not always guarantee the performance of a database maintaining the lowest time complexity. Algorithms such as red-black trees (RB), or Adelson-Velsky and Landis (AVL) trees have a time complexity of \(O(\log(n))\) for insert, delete, and query operations [1–4]. This study uses RB and AVL trees at the memory level. These two algorithms guarantee high insertion while sorting the data. However, the bigger the tree becomes, the more resources have to be used to complete needed tasks. Therefore, indexing algorithms are struggling into following the fast growth of data every year. Gartner estimated that information data presented on the internet will pass the brain capacity of the whole human beings living on Earth around 2025 [5]. Thus making emphasis on developing databases or algorithms capable of satisfying the processing and handling of fast-growing data.

Widely used databases such as HBase, Accumulo, and Google BigTable, provide a multi-level indexing method for handling huge amounts of data. These methods use traditional indexing algorithms such as B-trees and Log Structured Merged trees to efficiently insert, delete, and query the data [6]. It is called multi-level indexing because partitions of the database in the main table are taken place while generating a METATABLE in a manager node such as Zookeeper or Chubby for managing the partitioned tables. Using multi-level indexing in a master-slave structure is advantageous but when too many partitions are assigned to many slave nodes, search or query operations can be potentially affected.

Using probabilistic data structures [7] can potentially enhance the query performance of databases by being a quick, easy, and low-space-consuming solution. HBase is a well-known database for its outstanding performance on insert and query operations despite using a data structure whose main advantage is for fast data insertion. In HBase, probabilistic data structures are generated and loaded at the memory level after data has been flushed from memory to disk as a sorted key-value file called Hfile extended originally from the Sorted-String file (SSFile). These probabilistic data structures are extraordinarily space efficient and have a constant time complexity. Keys can be checked if they exist or not before wasting resources by opening the file with a time complexity of \(O(1)\) at the disk level. Meaning that from \(x\) amount of files in the system, it will open only the files where the probabilistic data structure returns true for set membership of the key that the client requests a query or deletion.

This study analyzes two probabilistic data structures (1) the bloom filters and (2) the cuckoo filters. These two probabilistic data structures are incorporated into an LSM tree to test their efficiency. Bloom filters were chosen because it is the traditional method to optimize query performance but also because they do not support deletions and cuckoo filters do provide that functionality. Therefore, during compaction and merge operations, cuckoo filters are expected to show better efficiency than bloom filters.
2 Literature Review

This section conducts a substantial study of the latest research on cuckoo filters and bloom filters for LSM trees to have a deep sight into the research that needs to be done.

2.1 Novel Cuckoo Filters Related Research

In [8] the authors make use of cuckoo filters by replacing the multiple bloom filters that are being used as a single main cuckoo filter in the LSM tree. This approach achieves the mentioned solution by using bits that should be used by the fingerprint to map data into auxiliary addresses in the LSM tree. They argue that the false positive rate is higher because fewer bits are used in the fingerprint. To address this issue, instead of using bits that are designated to the fingerprints, encoding is used to mitigate the high false positive rate. However, one of the main challenges that [8] have is that the LSM tree can grow in size over time. Therefore, the higher the level, the more bits are needed. Thus, the false positive rate is affected if the tree size grows considerably.

Research in [9] demonstrates that cuckoo filters can present false negatives the higher the load factor, the more evictions happen during the insertions. Therefore, cuckoo filters present a trade-off between space efficiency and false negatives. Thus, [9] shows that for a low false positive rate of 3 percent, bloom filters have better space efficiency than cuckoo filters.

Research in [10] shows mathematically and experimentally that Cuckoo filters can suffer from performance degradation during element insertion because of its random eviction strategy for the candidate buckets. This random selection also brings load-balancing problems to the data structure and thus leads to frequent relocations. This issue is addressed by choosing beforehand which bucket is a better candidate. This approach showed to better balance the load in the filter and thus the work needed for relocating fingerprints along the data structure is reduced.

Research in [11] shows that existing techniques for approximate set representation such as Cuckoo filters or Bloom filters have an insufficiency to satisfy the requirements of a dynamic set. Normally, a dynamic set is a data structure that can support modifications that a simple set cannot because of its immutability properties. To address the issue, this study proposed a dynamic cuckoo filter (DCF) to support reliable delete operation and elastic capacity for dynamic sets by using monopolistic fingerprints to represent an item. This study showed a 75 percent reduction in memory cost, a 50 percent improvement in construction speed, and an 80 percent improvement in the speed of membership query.

Research in [12] shows that cuckoo filters have an issue with the exclusive-OR (XOR) operation used by the Cuckoo filter because it requires the total number of buckets to be a power of two, leading to space inflation. To address the issue, the addition and subtraction operations are used instead of the XOR operation to compute the candidate buckets of an item. The proposed filter is called the additive and subtractive cuckoo filter (ASCF). By using this approach, the needed space is reduced while maintaining the same search and delete performance as the state-of-the-art cuckoo filter [20].
In [13] the cuckoo filters are extended to integrate a bloom filter that is used to improve the performance of insertions. The proposed CFBF does not require additional memory access for lookup operations and preserves the support for the deletion of the original cuckoo filter. This approach proves that it can be used to reduce worst-case insertion time by a factor of ten and achieve an average insertion time similar to that of a lookup.

### 3 Data Structures Overview

This section gives an overview of how data structures like the Log-Structure Merge Tree, AVL Tree, and Red-Black Tree manage basic operations. This section is mainly to provide the necessary concepts behind these algorithms.

#### 3.1 Log-Structure Merge Tree

The Log-Structured Merge (LSM) tree is a data structure optimized for fast writes using memory and disk dynamically [6]. In memory, the LSM tree usually makes use of a tree-based indexing algorithm such as Red-black [1, 4] or AVL trees [2, 3] to insert, delete, and read data. Therefore sorting and balancing are done beforehand prior to flushing the data structure into the disk. After the data structure is flushed into the disk, it automatically becomes a key-value sorted file called a sorted string table or (SSTable). This file is flushed in a sorted state because the indexing algorithm sorted the keys every time a client placed a request for insertion and deletion. However, some of the disadvantages of the LSM tree is the query performance. Every SSTable in the disk has a sparse index that enables the search of a certain key with O(log(n)) time complexity. However, LSM trees are a log-based structure, meaning that SSTables in the disk are sorted automatically from youngest to oldest. Therefore, every file must be checked to find the key requested from the client in the data structure causing a time complexity of O(n) and non-desired waiting time for the client. The LSM tree is usually seen in widely used databases such as HBase [14], Accumulo [15], Google Big Table [16], and Cassandra [17]. However, these databases all solved the long query period of keys by implementing a probabilistic data structure called bloom filters (refer to Subsection 4.1).

#### 3.2 AVL and Red-Black Trees

AVL tree is an extension of the binary search tree (BST) [2, 3]. This data structure supports a self-balancing function to prevent tree skewness. Therefore, the time complexity is maintained at O(log(n)). AVL trees prevent unbalancing by rotating the tree and balancing both sides of the tree according to a balance factor presented in every node of the tree. The balance factor executes balancing operations for the tree when its value is equal to or bigger than two. However, because this load factor is usually surpassed too often during insertion and deletion operations, the AVL tree tends to have a lower insertion performance than Red-Black trees. Some important properties of the AVL trees are shown next:

- Every node must have at least two children except the root
• AVL trees rotate to self-balance.
• Every node must have a balance factor
• All operations have an \( O\log(n) \) time Complexity

Red-black (RB) trees are also an extension of the BSTs. However, RB trees have higher insertion performance compared to AVL trees [1, 4]. This higher insertion is possible because the RB trees have two self-balancing approaches as shown in Table 1. The RB tree executes the balancing operations by applying properties as (1) The tree is composed of red and black nodes, the root always being black, and every newly inserted key is red on purpose to corrupt the tree and conduct balancing if needed because a red parent node cannot have a red child (2) If the red and black method did not prevent skewness, a rotation is executed to balance the tree. The RB trees have a better insertion performance than the AVL tree by trading off (1) space because an extra bit is added for assigning the color (2) skewness is more likely but not severe due to the color balancing method that provides. Some important properties of the RB trees are shown next:

• Every node has a color (red or black).
• A red node must always have only black children.
• RB Trees provided color and rotation self-balancing methods.
• All operations have an \( O\log(n) \) time Complexity

<table>
<thead>
<tr>
<th>Tree Type</th>
<th>Red-Black Tree</th>
<th>AVL Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Complexity</td>
<td>Self-balancing binary search tree ( O\log(n) )</td>
<td>Self-balancing binary search tree ( O\log(n) )</td>
</tr>
<tr>
<td>Child Number</td>
<td>No more than 2</td>
<td>No more than 2</td>
</tr>
<tr>
<td>Self-Balancing Methods</td>
<td>1. Rotation (left or right)</td>
<td>Rotates according to the balancing factor in every node.</td>
</tr>
<tr>
<td></td>
<td>2. Recoloring (red or black)</td>
<td></td>
</tr>
<tr>
<td>Tree Height</td>
<td>Represented as ( \log(N) ) where ( N ) is the depth of the tree by its amount of present black nodes.</td>
<td>Represented as ( \log(N) ) where ( N ) is the number of nodes in the tree.</td>
</tr>
</tbody>
</table>

4 Probabilistic Data Structures

Probabilistic data structures are bit arrays of low memory cost that are used to optimize the query performance of the databases. There are several types of data structures such as hyperloglog [18], cuckoo filters [19–21], bloom filters [21–23], and KD-tree filters [24]. These data structures are most useful when strategically placed in high-ingestion systems, in parts of the application where they can prevent expensive disk
seeks. For example, having an application perform a lookup of an element in a large table on a disk can easily bring down the throughput of an application from hundreds of thousands of operations per second to only a couple of thousands of operations per second. This study analyzes mainly cuckoo and bloom filters for set memberships. These two data structures share a very similar structure because (1) both of them insert keys in their slots using hash functions for obtaining indexes (2) optimize memory usage and reading speed by changing keys into bits (3) are probabilistic because their false positive rate is measure on how much hash collision can be accepted within the keys. Table 2 shows an overview of the characteristics of the probabilistic data structures to be applied in this study and their main differences.

Table 2 Bloom and Cuckoo Filter Comparison

<table>
<thead>
<tr>
<th></th>
<th>Hash Number</th>
<th>Operation List</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cuckoo Filter</td>
<td>2</td>
<td>- Deletions, - Insertions, - Confirmations</td>
<td>O(1)</td>
</tr>
<tr>
<td>Bloom Filter</td>
<td>k</td>
<td>- Insertions, - Confirmations</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

4.1 Bloom Filters

Bloom filters [21–23] are probabilistic data structures used to test if an element is part of a set member. This data structure is probabilistic because is possible to have false positives meaning that the actual element is indeed not part of the set but the result is otherwise. A bloom filter is composed of hash functions, bits, and a bit array with all bits initially being zero. Bloom filters only support two operations:

- Insertions: New elements are added and hashed multiple times while changing the corresponding bits from zero to one.
- Checkings: The element is hashed and if the corresponding bit values are all one, the element is likely to be part of the set membership. If not, then the element is definitely not part of the set membership because bloom filters do not have false negatives.

Bloom filters are much better for performing fast look-ups on a big-scale database and save a lot of space because keys are not saved on the array. However, bloom filters have a fixed false positive rate. Therefore, there is a non-zero probability of falsely indicating that an item is in the set. Also, if an item is added to a bloom filter, removing the item is not possible. Thus, leading to an increase in the false positive rate as more values are inserted. Bloom Filters are calculated according to (1) the level of false positive rate acceptance (2) a bit array of m length (3) an optimal hash function number k calculated according to the bit array size and insertion number [25, 26].
\[ FPR = (1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{-\frac{kn}{m}})^k \]  

(1)

\[ g = \ln(f) = k \ln(1 - p) = k \ln(1 - e^{-\frac{kn}{m}}) \]  

(2)

Optimal \[ k = \frac{m}{n} \ln(2) \]  

(3)

Optimal \[ m = -\frac{n \ln(FPR)}{\ln(2)^2} \]  

(4)

In Eq. 1 the false positive rate of bloom filters is calculated. The probability of a bit being zero is \( 1 - \frac{1}{m} \), “m” being the bit array size of the filter. However, bloom filters have one to “k” number of hash functions and an “n” number of items that can be inserted. Therefore, the probability must be elevated to “k” times “n”. Thus, the probability of a false positive is the probability of having a bit not being zero elevated by the “k” number of hash functions as \( (1 - (1 - \frac{1}{m})^{kn})^k \). In Eq. 2 and Eq. 3 the optimal “k” number of hash functions is calculated. Equation 2 is converted as a logarithmic expression for calculating its derivative. Only the derivative of Eq. 2 can lead to the optimal “k” number of hash functions expression as shown in Eq. 3. Equation 4 calculates the optimal bit array size of a bloom filter. Thus, the optimal bit array size is the conversion of the variable “k” of Eq. 1 by the expression in Eq. 3.

### 4.2 Cuckoo Filters

Cuckoo filters [19–21] are a probabilistic data structure that has been widely used in network applications since 2014. Like bloom filters, cuckoo filters are used to test whether an element is “probably” a member of a set or “definitely” not. Cuckoo filters are composed of slots, buckets, and fingerprints (bit arrays). The name cuckoo comes from the bird that lay its eggs in another nest and after the chick hatches, it drops the other eggs out of the nest or the other chicks that hatched.

Cuckoo filters provide similar space and time complexity with less hashing overhead than bloom filters (because cuckoo filters only have 2 hash functions and bloom filters have k number of functions). However, the main advantage is that cuckoo filters can delete items from the membership set because of its two-dimensional and fingerprint insertion. This could further improve databases such as HBase where SSTables are constantly being merged and compacted. Cuckoo filters have a load parameter that shows the percentage of slots already used in the filter.

A cuckoo filter with a load of 75 percent already has 75 percent of its buckets in use. However, research shows that the higher the load factor of a cuckoo filter, the more false negatives will start appearing [9, 10]. During our experimentation, a load factor lower than 65 percent prevented the system to generate false negatives when checking the existence of keys. Therefore, the load factor must be taken into consideration for checking if resizing or also other measures are needed to handle the filters in the memory level.

\[ I_1(x) = \text{hash}(x) \]  

(5)
\[ I_2 = I_1(x) \oplus \text{hash}(x's\ fingerprint) \] (6)

In Eq. 5 and Eq. 6 the index of the buckets where the fingerprint will be inserted is calculated. The first function will calculate the index of the first bucket by passing the key into a hash function. However, the second index is calculated by calculating the hash value of the fingerprint followed by an XOR operation in the index of the first bucket. Index one and two must be matched with the capacity that is set before generating a new cuckoo filter.

5 LSM Tree and Testing Tools Development

This section explains (1) the development behind the new LSM tree that implements cuckoo filters and the original LSM tree that implements bloom filters where several new functions are developed to make possible the merging and generation of the cuckoo filters, for bloom filters, only generation is supported (2) the environment that is used for the development and testing (3) the testing functions for checking the performance of the data structure.

5.1 Environment Settings

This study develops the LSM trees by using Python. The software environment required for execution includes Python version 3.6.8. Additionally, the project relies on several key libraries and packages, such as numpy version 1.19.5, pandas version 1.1.5, matplotlib version 3.3.4, pyarrow version 6.0.1, and mmh3 version 3.0.0.

For successful execution, the project should be run on an operating system like Centos 7 or a compatible alternative. The hardware environment should include a processor, such as an Intel Core i7-1165G7 processor (4 cores/8 threads, 12M Cache, up to 4.70 GHz), and a memory size of 64 GB [27].

5.2 LSM Tree Details

This study developed an LSM tree to implement probabilistic data structures to optimize its query performance. The algorithm has a memory and disk level. An RB tree is used at the memory level to process the client requests. At the disk level, the flushed trees are converted to an SSTable format and managed into 3 different levels. These three levels represent the age and size of the SSTables. The first level will have the recently flushed keys and values, making them accessible if the key is not in the memory. Users can decide the megabyte threshold of the files in the first, second, and third levels. According to these settings, the files will move to deeper levels. Thus, one can say that an SSTable is old according to how many times it has been merged.

The algorithm loads the filters in the memory immediately after an SSTable is generated. The algorithm then manages these filters with META information that is also placed in the memory for faster access.

For the LSM tree that implements bloom filters, every time an SSTable is merged, the bloom filters of the merged SSTables are assigned to the new SSTable as a linked
list. However, if the merged SSTables continue to merge with other SSTables containing one or more filters, the linked list is then going to have a higher time and space complexity. Therefore, the algorithm uses a function that reads the SSTables containing all the bloom filters and makes a new bigger filter with all the keys in the SSTable. With this approach, if a key is deleted, an update is possible to be placed only when the merging procedure starts. Therefore, the false positive rate of the overall algorithm is increased by this phenomenon. Because bloom filters only support insertions and searches, the algorithm becomes less flexible when it processes query requests from clients. Therefore, instead of implementing bloom filters, a new algorithm implementing cuckoo filters is developed.

This new algorithm has two different functions for managing the cuckoo filters in the data structure. The first function has the same anatomy as shown in the algorithm implementing bloom filters. However, the second function has a special feature that adds value to the overall structure. This function is applied whenever a merging procedure starts between the SSTables. In the case of bloom filters, when more data is inserted and the number of insertions exceeds, the bloom filters will have an increment of false positives. However, cuckoo filters can guarantee a small false positive rate while having a high load factor. Therefore, this function merged the two filters by taking the fingerprints of the second filter and appending them to the first filter but always taking into account that the load factor of the filter does not surpass the allowed limit. This is important to keep because if the load factor of the filter is more than 65 percent, the filter will return false negatives. Thus, affecting the overall reliability of the algorithm.

One of the biggest advantages is that cuckoo filters support deletions. This allows the algorithm to reflect the deletion directly in the filter for future queries. Meaning that if the deleted key is queried again, the cuckoo filters will be up to date and will confirm if the key is a member of the algorithm. For bloom filters, it will cause a false positive and will have to open the SSTable causing more use of resources.

### 5.3 Testing Tools Details

The functions developed for testing the algorithm measure (1) insertion time per request (2) query time before deletion per request (3) deletion time per request and (4) query time after deletion per request. This function takes as parameters the algorithm, testing data, the insertion and query rate, and the unit for measuring the time. In this study, microseconds are used to measure the time for single transactions. The function returns a list of the number of inserts and queries specified beforehand. While testing, the algorithms showed too much noise. Therefore, a secondary function that removes the outliers is developed to have the most accurate average and standard deviation. As shown in Eq. 7, a Z-score technique is used to calculate the distance of all the values according to their average. A threshold of 3 is chosen because it is considered a cut-off value to set the limit by preventing not to get too much data from being erased. In this study, more than 99 percent of data is saved by this threshold. Only less than 1 percent is not considered to have accurate results.
This study uses public geo-data that measures the internet speed of global fixed broadband and mobile networks to test the performance of the algorithm. The total size of this dataset is 15 gigabytes. However, we only use a portion of the data set that contains around 4.2 million rows. Future adaptations to other programming languages such as Java or C++ will improve the overall speed and performance of the algorithm rather than implementing Python.

6 Cuckoo Vs Bloom LSM Tree Performance Results

This section presents the final results of the study. Explanations of how the tests are configured and the results of the set configurations during the testing are shown.

6.1 Tests Configuration

This study performed a total of four tests to check the performance during insertions, queries, and deletions. The basic time unit used in this study is on milliseconds and a maximum time threshold is set to 180 seconds for merging operations. In Table 3, the tests are performed 10 times for different settings for checking query and delete operations. The LSM tree is divided into 5 sizes, the batch size for the segments is one-tenth of the size of the LSM tree, and the query and delete testing starts from 15 thousand and increments every 15 thousand according to the increase of the size of the LSM tree. In Table 4, tests are performed 7 times for testing insertion performance. The test starts from 25 thousand keys and doubles up after finishing each test and the batch size is one-tenth of the size of the total insertion test size.

<table>
<thead>
<tr>
<th>Test Number</th>
<th>LSM tree Size (Keys)</th>
<th>Query, Deletion Test Size (Keys)</th>
<th>Batch Test Size (Keys)</th>
<th>Level Test Threshold (Megabytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>200k</td>
<td>15k</td>
<td>20k</td>
<td>lvl1: 40, lvl2: 80</td>
</tr>
<tr>
<td>2</td>
<td>200k</td>
<td>30k</td>
<td>20k</td>
<td>lvl1: 50, lvl2: 100</td>
</tr>
<tr>
<td>3</td>
<td>400k</td>
<td>45k</td>
<td>40k</td>
<td>lvl1: 60, lvl2: 120</td>
</tr>
<tr>
<td>4</td>
<td>400k</td>
<td>60k</td>
<td>40k</td>
<td>lvl1: 70, lvl2: 140</td>
</tr>
<tr>
<td>5</td>
<td>600k</td>
<td>75k</td>
<td>60k</td>
<td>lvl1: 80, lvl2: 160</td>
</tr>
<tr>
<td>6</td>
<td>600k</td>
<td>90k</td>
<td>60k</td>
<td>lvl1: 90, lvl2: 180</td>
</tr>
<tr>
<td>7</td>
<td>800k</td>
<td>105k</td>
<td>80k</td>
<td>lvl1: 100, lvl2: 200</td>
</tr>
<tr>
<td>8</td>
<td>800k</td>
<td>120k</td>
<td>80k</td>
<td>lvl1: 110, lvl2: 220</td>
</tr>
<tr>
<td>9</td>
<td>1000k</td>
<td>135k</td>
<td>160k</td>
<td>lvl1: 120, lvl2: 240</td>
</tr>
<tr>
<td>10</td>
<td>1000k</td>
<td>150k</td>
<td>160k</td>
<td>lvl1: 130, lvl2: 260</td>
</tr>
</tbody>
</table>
Table 4 Insertion Test Main Configurations

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Insertion Size (Keys)</th>
<th>Batch Size (Keys)</th>
<th>Level Test Threshold (Megabytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25k</td>
<td>2.5k</td>
<td>lvl1: 10, lvl2: 20</td>
</tr>
<tr>
<td>2</td>
<td>50k</td>
<td>5k</td>
<td>lvl1: 20, lvl2: 40</td>
</tr>
<tr>
<td>3</td>
<td>100k</td>
<td>10k</td>
<td>lvl1: 30, lvl2: 60</td>
</tr>
<tr>
<td>4</td>
<td>200k</td>
<td>20k</td>
<td>lvl1: 40, lvl2: 80</td>
</tr>
<tr>
<td>5</td>
<td>400k</td>
<td>40k</td>
<td>lvl1: 50, lvl2: 100</td>
</tr>
<tr>
<td>6</td>
<td>800k</td>
<td>80k</td>
<td>lvl1: 60, lvl2: 120</td>
</tr>
<tr>
<td>7</td>
<td>1600k</td>
<td>160k</td>
<td>lvl1: 70, lvl2: 140</td>
</tr>
</tbody>
</table>

In the tests, level test thresholds are set for merging and moving the segments of the tree along the disk to give lower priority to those that are older. However, during our testing, we randomly pick up keys from the dataset that we are loading to have better empirical results of how efficient can be the algorithm in such scenarios. This counts up that very old and young keys are selected during the testing. The tests in Tables 3 and 4 are conducted twice for 0.1 and 5 percent false positive rates for comparing the impact on the efficiency that a high or low false positive rate could generate in the overall performance of the LSM tree. The two false positive rate tests are crucial because [20] states that cuckoo filters are more efficient when used at lower false positive rates. On the other hand, bloom filters achieve better efficiency when rather higher false positive rates are set but other studies [9] showed that bloom filters can achieve better results than cuckoo filters also when having 3 percent or lower false positive rates.

6.2 LSM Tree Testing Results

The tests were conducted for a false positive rate of 0.1 percent and 5 percent each. The reason for conducting the test with this method is that bloom filters have a better performance when there is a relatively high false positive rate and cuckoo filters have a better performance when low false positive rates are needed.

In Figure 1 Part A we can see that the difference between the query performance of the cuckoo filters and bloom filters is minimum for a false positive rate of 0.1 percent. However, in Figure 1 Part B cuckoo filters tend to show a better query response compared to bloom filters when setting a false positive rate of 5 percent. Therefore, it opens the possibility to use cuckoo filters not only for data-sensitive databases but to also apply cuckoo filters for faster queries and better space complexity by trading off a higher false positive rate.

Depending on the size of the test, the average of both filters differ slightly and the bigger the test is, the actual query time tends to increment or decrement. In this stage, the bloom and cuckoo filters have only received insertions. Bloom filters are more...
insert-friendly because of the structure that it was designed on. It only needs to change the bits from zero to one. If the bit was already changed to one, there is no further action. However, for cuckoo filters, if one of the candidate buckets cannot receive one more fingerprint, eviction actions are executed and a loop with a threshold is set for then returning that the filter is too loaded and regeneration actions are needed.

In Figure 2 Part A and B, we can appreciate an interesting performance when deleting keys from the tree. Figure 2 Part B shows a slightly better performance than Figure 2 Part A for higher false positive rates. The reason for such performance is that cuckoo filters can support deletions by removing the fingerprint of a key from the candidate buckets while checking the existence of the key. This process is performed as bloom filters also execute checkings. The only difference is that after locating the fingerprint, cuckoo filters delete immediately the fingerprint. This approach uses the same resources as the checking method for updating the filter. In other words, cuckoo
filters perform almost the same when deleting keys in the LSM tree while also updating themselves.

Another important thing to point also is that in the case of bloom filters, such an update can be placed only if the filter is regenerated. This issue with the bloom filter tends to present not only bottlenecks but also because it is not updated, tends to return more false positives to the clients. Thus, leading to higher latency.

In Figure 3 Part A and B, the query performance is compared according to the deleted keys from the LSM tree. Figure 3 Part B shows a more consistent or less deviated average than Figure 3 Part A when looking at their standard deviation. As mentioned, bloom filters cannot be updated by simply deleting the bits as cuckoo filters.
where updates can be placed by removing the fingerprint contained in the candidate bucket(s).

![Diagram](A) 0.1 Percent False Positive Rate

![Diagram](B) 5 Percent False Positive Rate

Fig. 3 Query Performance After Deletions

The overall performance of cuckoo filters is 70x better than the performance of bloom filters. This performance gap can be demonstrated in Figure 5. As we can see, a client requests to query the data. Then, the key is checked in chronological order in the memory level filters. If the key is in the filter as a fingerprint, the SSFile corresponding to that filter is checked. Thus, if the key is in the SSFile, the key is returned to the client. However, if the key has been deleted but another client queries the same key, disk resources are then used to check if the key is in the file or not. Therefore, this can cause a considerable bottleneck because LSM trees are known to have a query time complexity of $O(n)$. Thus, LSM tree developers applying bloom filters must expect
that this configuration can cause a higher latency when querying data. Whereas, if cuckoo filters are placed instead, this whole disk process can be avoided because the fingerprints are deleted as soon as a key is deleted in the SSFile. Thus, achieving a performance 70x times higher than bloom filters when deletions and updates are placed in the LSM tree.

In Figure 4 Part A and B, we compared the insertion performance of the LSM tree that implements cuckoo and bloom filters. The performance gap between low and high false positive rates shows that there is almost no difference. Findings still show that bloom filters offer on average 1.5x times faster insertion performance than the developed LSM tree implementing cuckoo filters for high and low false positive rates. Other studies [10–13] showed other extensions of newly developed cuckoo filters that can guarantee better space efficiency and low latency by doing a different set of operations when choosing candidate buckets than the original cuckoo filters [20].
Table 5 Average (Avg) and Standard Deviation (Std) Performance of Bloom and Cuckoo Filters

<table>
<thead>
<tr>
<th>Operation</th>
<th>FPP (%)</th>
<th>Bloom Filter Avg±Std (µs)</th>
<th>Cuckoo Filter Avg±Std (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertions</td>
<td>0.1</td>
<td>0.0242±0.0030</td>
<td>0.0361±0.0073</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.0218±0.0031</td>
<td>0.0360±0.0074</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.0234±0.0001</td>
<td>0.0357±0.0001</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>0.0235±0.0002</td>
<td>0.0356±0.0001</td>
</tr>
<tr>
<td>Query Before Deletions</td>
<td>0.1</td>
<td>13.4821±4.5161</td>
<td>13.9298±4.6660</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>16.4116±6.9105</td>
<td>15.8695±5.9005</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>17.7700±6.4704</td>
<td>16.2345±3.8368</td>
</tr>
<tr>
<td>Deletions</td>
<td>0.1</td>
<td>30.1270±10.1742</td>
<td>30.131±10.1767</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>33.0068±13.9864</td>
<td>31.1241±11.7699</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>36.5020±13.3576</td>
<td>32.0134±7.5333</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>45.3763±19.5963</td>
<td>33.3968±8.8692</td>
</tr>
<tr>
<td>Query After Deletions</td>
<td>0.1</td>
<td>13.0037±4.3471</td>
<td>0.0823±0.0055</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>16.9102±7.5339</td>
<td>0.0872±0.0098</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>20.4678±7.8352</td>
<td>0.5460±3.1126</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>28.4971±10.2785</td>
<td>1.6981±3.8583</td>
</tr>
</tbody>
</table>

In the original study [20], cuckoo filters choose the bucket candidates randomly, and as stated in [9], this affects the load balance of the filter. Also, relocations are one of the biggest issues for cuckoo filters. In our study, we put an arbitrary 500 kick count number representing the number of times that relocating is allowed. Further optimization in this area will be needed for our system. Also, the cuckoo filters in [10–13] can provide a better performance result than using the cuckoo filters in [20]. This solution can close the breach of performance between cuckoo filters and bloom filters and enhance performance when querying and deleting keys in the LSM tree.

In Table 5 we can appreciate a more comprehensive comparison of the performance of the bloom filters and cuckoo filters by the average of each conducted test for 0.1, 5, 10, and 20 percent false positive rates. We see that the higher the false positive rate, the slower the bloom filters but the better the cuckoo filters can perform. However, bloom filters on average are 1.5x faster than cuckoo filters. The results of this study show that bloom filters are a better choice for high-insertion systems that do not require continuous queries. On the other hand, the results show that cuckoo filters have on average faster performance than bloom filters for querying data before deletions, during deletions, and also after deletions. Also, results show that the stability of the queries and deletions is better than the one of the bloom filters. Lastly, the results
Fig. 5 LSM tree Disk/Memory-level Structure

also show that cuckoo filters have a greater advantage than bloom filters because are able to update information in the filter. This prevents the system to open SSFiles and wasting unnecessary resources.

7 Conclusion

Some databases apply cuckoo filters to improve query performance [28]. This study also proposed an improved algorithm that uses cuckoo filters as a linked list, but in the mean, it also merges the filters to improve the space complexity. As future research follows this study, applying cuckoo filters into widely used databases such as HBase, Accumulo, or Cassandra can improve their query performance for areas focused on analytics. The findings of this study show that although the insertion performance of the LSM tree bloom filters is on average 1.5x times faster, cuckoo filters can achieve up to 70x times faster query performance in a database that experienced a lot of deletes and updates. Thus, this study shows that state-of-the-art cuckoo filters are becoming the future candidate to replace bloom filters in existing databases to improve overall efficiency. On the mean trading-off insertion for systems focused on analytical areas will not affect efficiency and it will improve (1) space complexity (2) a lower probability of false positives after deletions (3) constant merges in between cuckoo filters that have low load factor.

8 Declarations

8.1 Ethics Approval and Consent to Participate

Not applicable
8.2 Consent for Publication
Not applicable

8.3 Availability of Data and Materials
The developed algorithm was tested with publicly available data from a widely used data science website called “Kaggle”. The data used in this study can be downloaded from the website below:

• https://www.kaggle.com/datasets/dhruvildave/ookla-internet-speed-dataset

This study used the next device with the next specifications to conduct the development and testing of the algorithm:

<table>
<thead>
<tr>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS: Centos7</td>
</tr>
<tr>
<td>CPU: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz</td>
</tr>
<tr>
<td>GPU: UHD Graphics [Intel Corporation]</td>
</tr>
<tr>
<td>MEM: 64GB</td>
</tr>
<tr>
<td>Storage: 900GB</td>
</tr>
</tbody>
</table>

8.4 Competing Interests
Not applicable

8.5 Funding
Not applicable

8.6 Authors’ Contributions
We, the authors of this manuscript, would like to provide a comprehensive statement detailing our individual contributions to the research, writing, and development of this work. Each of us has played a crucial role in different aspects of the project, combining our expertise to produce a cohesive and impactful study. Below is a summary of our respective contributions:

• Humberto Cesar Villalta Valverde: As the lead author, he was responsible for conceptualizing the research idea, formulating the research questions, and designing the study. He coordinated the overall research process, conducted data collection and analysis, and interpreted the results. Additionally, he actively participated in writing the manuscript, preparing figures and tables, and revising the content based on the feedback received from other authors.
• **Kwangsik Kim**: He contributed significantly to the literature review, providing critical insights into the existing body of knowledge related to our research topic. He also played a key role in the data analysis and interpretation. Furthermore, he collaborated with Humberto Cesar Villalta Valverde in preparing the manuscript.

• **Kisu Kim**: He played an integral role in organizing and ensuring all contents, papers, and articles obtained from other sources had the highest quality possible. He actively participated in analyzing the results and getting rational interpretations. Additionally, he contributed to the section “Cuckoo Vs Bloom LSM Tree Performance Results”, “Data Structures Overview”, and “Probabilistic Data Structures” of the manuscript, drawing connections between the findings and the research objectives.

• **Jinman Kwon**: He contributed by reviewing the progress of the research and giving insightful feedback on the conduction of the research. He also continually contributed by providing crucial content and references that helped this research to become solid.

• **Jaechoon Lim**: He contributed to the data visualization and partially to the whole writing process of the manuscript.

• **Yongjoo Jun**: He contributed to the generation of the table of contents and itemization of the paper. He participated actively in the gathering of references for a rational literature review. Furthermore, he contributed to the conclusion section to state the final results and give feedback on the future work that this paper needs.

All authors have read and approved the final version of the manuscript and have agreed to be accountable for the work’s accuracy and integrity. We believe that our combined efforts have resulted in a robust and well-grounded study that adds significant value to the existing literature.

8.7 Acknowledgements

Not applicable

8.8 Authors’ Information

Not applicable

References


[19] Rasmus Pagh, “Cuckoo hashing”, Elsevier Science, IT University of Copenhagen, Rued Langgaardsvej 7, 2300 København S, Denmark, pagh@itu.dk, December, 2003


