**Research Project-Multi Copy Cuckoo Hashing**

**Group Number:** 17

**Name of the Group Members**

Anagha HC (211IT08)

Sachin Prasanna (211IT058)

Soumya Sangam Jha (211IT068)

Vartika T Rao (211IT077)

Abhayjit Singh Gulati (211IT085)

|  |
| --- |
| **Base Research Paper** |
| [D. Li, R. Du, Z. Liu, T. Yang and B. Cui, "Multi-copy Cuckoo Hashing," 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019, pp. 1226-1237, doi: 10.1109/ICDE.2019.00112.](https://ieeexplore.ieee.org/document/8731423) |
|  |
|  |
| **What is the Problem?** |
| Cuckoo Hashing is a technique for implementing a hash table. Unlike most other hash tables, it achieves constant time worst-case complexity for lookups. However, insertion might lead to cycles that are tough to break out of and causes computation of the hash function multiple times. These problems can be handled using Multi-copy cuckoo hashing.  Multi-Copy Cuckoo Hashing inserts copies of keys into multiple hash tables simultaneously, so when multiple candidate buckets are available, it is not necessary to randomly select a candidate bucket when inserting. This improves lookup time. However, the probability of rehashing stays the same; hence, an additional stash is used to store keys to temporarily delay the need for rehashing. But lookup in a big stash would increase the worst-case time complexity to linear, and once an inserted item is deleted from the stash, it must again be traversed during a lookup. For this, a bloom filter-like structure is used to reduce the frequency of lookup in stash; however, this causes the frequency of false positives. |
|  |
|  |
| **What is the Possible Solution?** |
| **Linear Probing in Stash:**  A stash-based approach is implemented to store the items that failed to be inserted during the insertion process. While this method helps to avoid the costly rehashing operation, it comes at the cost of increased time for lookup.  To ensure that the lookup in the stash is constant, Linear Probing can be used. The hash function helps reduce the time complexity of lookup to constant.  **Flags- Counting Bloom Filter:**  An array of flags will be made, which will work in a fashion similar to a counting bloom filter. A counting bloom filter is a modification of a bloom filter as it does not give any false negative results but also reduces the probability of false positives. In case of deletion, the counter value is decremented. This ensures that the false positive rate does not increase simultaneously, ensuring no false negatives are there. |
|  |
|  |
| **Which Data Structure to Use for the Solution (Details of the Data Structure)** |
| The solution has four parts:   * Off-chip hash table * On-chip counters * Off-chip stash * Flags (bloom filters) * **Off-chip hash table**: The off-chip table is the hash table that stores the real items in the sub-tables. It stores the items according to the corresponding hash function for the table. **The hash table is implemented using 2 fixed length one-dimensional integer arrays of size M.** We use two independent hash functions for the two tables. * **On-chip counters**: The on-chip counters contain the counters that are one-to-one mapped to the off-chip candidate buckets. The counters are initially set to 0 and are updated with each insertion and deletion. **The on-chip counters are implemented using 2 fixed length one-dimensional short int type arrays of size M.** * **Off-chip stash:** A stash is a hash table that uses a simple hash function to store the items from the failed insertions into the off-chip table, to avoid costly rehashing. **We have used a fixed-length one-dimensional integer array for a stash with linear probing of size S. A separate hash function is used to map keys in the stash.** * **Flags:** The flags is an array that is initially set to 0 and work in the fashion of a counting-bloom filter, when an item fails in insertion and is put to the stash, the flags of its candidate buckets are incremented, so at a later time when we want to decide whether to check the stash, we will see first if any of the flags of the associated buckets are 0, if yes we know for sure it is not there without accessing the stash. The stash flags are assigned one-to-one to each candidate bucket. **The flag table was implemented using 2 fixed length one-dimensional short int type arrays of size M.** |
|  |
|  |
| **Justify Why This is the Best Data Structure for this Application?** |
| **Offchip Hashtable-** The off-chip table is the main Cuckoo hash table that stores the real items in its sub-tables. We have used 2 subtables implemented using 2 one dimensional integer arrays because of its O(1) time complexity for accessing the element. Since the index of the candidate buckets (positions where the key can be inserted) is obtained using hash functions, the candidate buckets can be accessed in O(1) time using arrays. When we insert a key into a hash table, the hash function is used to determine the index of the candidate bucket. Once the index is determined, the element can be inserted into the array at that location in constant time, O(1). Similarly, when we want to retrieve the key from a hash table, the hash function is used to determine the index at which the element should be located, and the element can be retrieved from that index in constant time, O(1).  **Onchip Counters**- The counters must be maintained in the fast on-chip memory to minimize unnecessary operations to the main table in the slow off-chip memory. Since we are using only 2 tables, we only need 2 bits of space to store the counter corresponding to each candidate bucket, hence we used a one-dimensional short integer type array of the same size as the offchip table. Arrays help in maintaining a deterministic one-to-one mapping between on-chip counters and off-chip tables which makes it much easier to determine the status of the buckets on-chip and execute the results from the counter logic off-chip since the corresponding counter indices can be accessed using the same index as that of the off-chip bucket.  **OffChip Stash-** The offchip stash is a small hash table implemented using an array using the linear probing collision resolution technique. Arrays are used to implement linear probing because they allow for the elements in the hash table to be stored in a contiguous block of memory, which makes it easy to search for the next available empty slot. When a collision occurs and an element needs to be inserted into the hash table, the algorithm starts at the index that was determined by the hash function and then searches sequentially through the array until it finds an empty slot.  **Flags**- A 1-bit flag corresponding to each bucket can be used that will work in the fashion of a bloom filter. However, on deletion, these flags cannot be updated as that might lead to false negatives in addition to the already present false positives. Hence, we are using a counting bloom filter that supports deletion, unlike a standard bloom filter. To replicate the working of a counting bloom filter we are using an array of short type of the same size as the main tables. We are using short type as the maximum value that is stored in their flags is the length of the stash. Arrays make it easier to have a one-to-one mapping to the main table. |
|  |
|  |
| **Implementation Details** |
| 1. **Insertion**   The element is inserted in all its empty candidate buckets, and their corresponding counters are updated. The insertion operation works on the following principles:   * All the empty candidate buckets must be occupied. * Buckets with elements of count 1 are never overwritten. * The rest are overwritten in the decreasing order of their count until the overwriting results in more copies of the inserted item than the overwritten one. * In case of collision, the key is stored in the stash. The insertion in the stash is implemented using **linear probing**.  1. **Look-Up**   The look-up operation works on the following principles:   * All the buckets are skipped, and a negative is returned for any candidate bucket with count 0. * The non-zero candidate buckets are partitioned according to their count. The partitions whose size is smaller than the associated count are skipped. * For each of the remaining partitions, if the size is S and the associated count is C, S-C+1 buckets are checked in the partition. The queried item is returned if it is found. Otherwise, a negative is returned. * If the queried item is not present in the main table, the stash is checked.  1. **Deletion**   The deletion operation works on the following principles:   * The counters of the key to be deleted are set to 0 without visiting the main off-chip table. * If the queried item is not present in the main table, the stash is checked. Upon finding the item, it is deleted. Otherwise, a negative is returned. |
|  |
|  |
| **Complexity/Performance Analysis** |
| **Insertion**: The amortised time complexity for insertion operation is *O(1)*. There is a possibility of rehashing when the table is filled, or the load ratio is high. This requires re-inserting all elements of the table into a new table. However, a small stash of size *s* improves the probability of rehashing from *O(1/n)* down to *O(1/ns+1)* [*[1]*](https://ieeexplore.ieee.org/document/8731423)  **Lookup:** The worst-case time complexity of lookup in the main table in multi-cuckoo hashing is O(1) as only 2 buckets need to be accessed.  Since we are using a stash-based approach to minimise the need for rehashing, and improve the insertion time complexity, the worst-case lookup in the stash with linear probing is O(n). However, the amortised time complexity is still O(1).  But by using flags that function as counting bloom filters, we reduce the scope for lookup in the stash. Thus the time complexity of the lookup is O(1).  **Deletion:** The worst-case time complexity of deletion in the main table in multi-cuckoo hashing is O(1) as only the counters corresponding to the candidate bucket need to be set to 0.  However, in the stash, the worst-case lookup in the stash with linear probing is O(n) but amortised time complexity is O(1).  Thus the amortised time complexity of deletion is O(1). |
|  |
|  |
| **Observations and Conclusions** |
| Multicopy Cuckoo Hashing had problems in cases where Deletion in the Multicopy Cuckoo Hash Table was more frequently done. Our solution, the  "Counting Bloom Filter" improves the performance of the Hash Tables in cases where Deletions are more frequent. Two examples where deletion is as frequent as insertion are:   1. Active Covid Cases: Supposing the people with Covid are stored in the Hash tables. Since people recover within 14 days and assume a constant caseload, our solution can prove to be beneficial. 2. Air Traffic Control Management: In busier airports where airplanes are to land in the airport, they can be stored in Hash Tables for faster lookup in case any communication is required. Once the airplanes land, they need to be removed from the table, so deletion will be as frequent as insertion and our solution can come in handy again.   Finally, keys in the stash can be looked up or deleted more easily than conventional Multicopy Cuckoo Hashing because of linear probing implemented in the stash. |

|  |
| --- |
|  |
|  |
| **References** |
| [1] [D. Li, R. Du, Z. Liu, T. Yang and B. Cui, "Multi-copy Cuckoo Hashing," 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019, pp. 1226-1237, doi: 10.1109/ICDE.2019.00112.](https://ieeexplore.ieee.org/document/8731423)  [2] [Pagh, Rasmus, and Flemming Friche Rodler. "Cuckoo hashing." *Journal of Algorithms* 51.2 (2004): 122-144.](https://www.itu.dk/people/pagh/papers/cuckoo-jour.pdf)  [3] [Luo, Lailong, et al. "Optimizing bloom filter: Challenges, solutions, and comparisons." *IEEE Communications Surveys & Tutorials* 21.2 (2018): 1912-1949.](https://ieeexplore.ieee.org/abstract/document/8586915)  [4] [Rottenstreich, Ori, Yossi Kanizo, and Isaac Keslassy. "The variable-increment counting Bloom filter." *IEEE/ACM Transactions on Networking* 22.4 (2013): 1092-1105.](https://ieeexplore.ieee.org/abstract/document/6576292) |