Mitigating Asymmetric Read and Write Costs in Cuckoo Hashing for Storage Systems

Yuanyuan Sun, Yu Hua*, Zhangyu Chen, Yuncheng Guo
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
**Corresponding Author: Yu Hua (csyhua@hust.edu.cn)*

Abstract

In storage systems, cuckoo hash tables have been widely used to support fast query services. For a read, the cuckoo hashing delivers real-time access with $O(1)$ lookup complexity via open-addressing approach. For a write, most concurrent cuckoo hash tables fail to efficiently address the problem of endless loops during item insertion due to the essential property of hash collisions. The asymmetric feature of cuckoo hashing exhibits fast-read-slow-write performance, which often becomes the bottleneck from single-thread writes. In order to address the problem of asymmetric performance and significantly improve the write/insert efficiency, we propose an optimized Concurrent Cuckoo hashing scheme, called CoCuckoo. To predetermine the occurrence of endless loops, CoCuckoo leverages a directed pseudoforest containing several subgraphs to leverage the cuckoo paths that represent the relationship among items. CoCuckoo exploits the observation that the insertion operations sharing a cuckoo path access the same subgraph, and hence a lock is needed for ensuring the correctness of concurrency control via allowing only one thread to access the shared path at a time; Insertion operations accessing different subgraphs are simultaneously executed without collisions. CoCuckoo improves the throughput performance by a graph-grained locking to support concurrent writes and reads. We have implemented all components of CoCuckoo and extensive experiments using the YCSB benchmark have demonstrated the efficiency and efficacy of our proposed scheme.

1 Introduction

Efficient query services are demanding and important to storage systems, which hold and process much more data than ever and the trend continues at an accelerated pace. The widespread use of mobile devices, such as phones and tablets, accelerates the generation of large amounts of data. There exist 1.49 billion mobile daily active users on Facebook in September 2018, with an increase of 9% year-over-year. In each minute, 300 new profiles are created and more than 208 thousand photos are uploaded to Facebook [5]. The explosion of data volume leads to nontrivial challenge on storage systems, especially on the support for query services [8, 12, 49]. Moreover, write-heavy workloads further exacerbate the storage performance. Much attention has been paid to alleviate the pressure on storage systems, which demands the support of low-latency and high-throughput queries, such as top-k query processing [30, 33], optimizing big data queries via automated program reasoning [42], offering practical private queries on public data [47], and optimizing search performance within memory hierarchy [9].

In order to improve the performance of query services for storage systems, efficient hash structures have been widely used. In general, each hash function maps each item to a unique bucket in a hash table, which needs to support constant-scale and real-time access. However, items may be hashed into the same bucket by hash functions, called **hash collisions**. Due to the use of efficient open-addressing design, cuckoo hashing [38] is able to mitigate hash collisions with amortized constant-time insertion overhead and lookup consumption to meet the throughput needs of real-world applications. Unlike conventional hashing schemes that offer only one bucket for each item, the cuckoo hashing provides multiple (usually two in practice [11, 19, 37, 40]) candidate positions for each item to reduce the probability of hash collisions. To perform a lookup operation, at most two positions are probed, and the worst-case time is constant-scale, thus delivering high performance especially in terms of low-latency read and lookup operations [10, 16, 18, 20, 25, 31, 44]. However, to insert an item, the cuckoo hashing has to probe the two candidate buckets for finding an empty position. If an empty slot does not exist, recursive replacement operations are needed to kick items out of their current positions until a vacant bucket is found, which forms a **cuckoo path**. There exists a certain probability of producing an **endless loop**, which occurs after a large number of step-by-step kick-out operations and turn out to be an insertion failure, thus resulting in slow-write performance. The property of asymmetric reads and writes in the cuckoo hashing becomes a potential performance bottlenecks for storage systems.

The endless loops not only lead to the slow-write operation, but also make storage systems unable to efficiently support **concurrent** operations on both reads and writes, which results in poor performance.

As the number of cores is increasing in modern processors, concurrent data structures are promising approaches to provide high performance for storage systems [13, 15, 21, 22, 43, 48, 50]. In general, locks are often utilized to ensure the consistency among multiple threads [4]. To mitigate the hash collisions for writes, most existing hash tables store items in a linked list with coarse-grained locks for entire tables [19], fine-grained locks for per bucket [2, 32] or Read-Copy-Update (RCU) mechanisms [35, 36, 46] for concurrency control. However, the coarse-grained locks for entire tables lead to poor scalability on multi-core CPUs due to the long lock time, and fine-grained per-bucket locks result in substantial resource consumption due to frequent locking and unlocking operations in a cuckoo path. RCU [34] works well for read-heavy workloads, but inefficiently for the workloads with more writes than reads. Moreover, the cuckoo hashing with per-bucket locks suffers from substantial performance degradations due to the occurrence of endless loops. The read operation has to wait for the write operation on the same bucket to complete and then release the lock before being executed. Hence, it is inefficient to frequently lock and unlock buckets during the long cuckoo paths of endless loops.

In order to offer a high-throughput and concurrency-friendly cuckoo hash table, we need to address two main challenges.

Poor Insertion Performance. Cuckoo hashing executes recursive replacing operations to kick items from their storage positions to the candidate positions for finding an empty bucket during an insertion procedure. Moreover, all efforts become useless when encountering an endless loop. To ensure the correctness of concurrency control, two continuous buckets in a path have to be locked for each kick-out operation. The frequent locking and unlocking on a cuckoo path will lead to high time overhead, which decreases the insertion performance.

Poor Scalability. The cuckoo path is possible to be very long in real-world applications. For example, the kick-out threshold is 500 in MemC3 [19], and thus the longest cuckoo path contains 500 buckets. All kick-out operations have to be completed in the buckets protected by the locks. Due to frequent use of locks to ensure consistency among multiple threads, the concurrent hash table results in poor scalability.

In fact, only insertion operations sharing the cuckoo path require locks for ensuring the consistency. Insertion operations through different cuckoo paths can be simultaneously executed. Based on this observation, we propose a lock-efficient concurrent cuckoo hashing scheme, named CoCuckoo. It leverages a directed pseudoforest containing several subgraphs to represent items' hashing relationship, which is further used to indicate the cuckoo paths of insertion operations. In the pseudoforest, each vertex corresponds to a bucket in the hash table, and each edge corresponds to an inserted item from its storage vertex to its backup vertex. In our CoCuckoo, the vertices corresponding to candidate positions of the path-overlapped items must be in the same subgraph. The path-overlapping can be interpreted that two or more cuckoo paths of items share the same nodes during insertion. In our design, each node only exists in a subgraph with a unique subgraph number. Hence, the operations on the path-overlapped nodes access the same subgraph. We leverage a graph-grained, rather than per-bucket, locking scheme to avoid potential contention. Once locking a subgraph number, all buckets with the same number cannot be accessed by different threads at the same time. If one thread intends to access a bucket, it first obtains the number of the subgraph to which the bucket belongs, to check if the subgraph number has been locked.

Specifically, we have the following contributions.

High Throughput. CoCuckoo not only retains cuckoo hashing's strength of supporting constant-scale lookups via open addressing, but also delivers high throughput by a graph-grained locking to support concurrent insertions.

Contention Mitigation. We optimize the graph-grained locking mechanism to pursue low lock overheads for different cases of insertions and release the locks as soon as possible to ease the contention. CoCuckoo is able to predetermine the insertion failures without the need of carrying out continuous kick-out operations, and thus avoids many unnecessary locking operations.

System Implementation. We have implemented all the components and algorithms of CoCuckoo. Moreover, we compared CoCuckoo with state-of-the-art and open-source scheme, *libcuckoo* [32], which offers multi-reader/multi-writer service.

2 Backgrounds

2.1 The Cuckoo Hashing

Cuckoo hashing [38] is an open-addressing technique with $O(1)$ amortized insertion and lookup time. In order to mitigate hash collisions, items can be stored in one of two buckets in a hash table. If one position is occupied, the item can be kicked out to the other [17, 26]. The frequent kick-out operations help items find empty positions for insertion with the costs of possible extra latency. On the other hand, we can definitely read the queried data in one of two hashed positions, thus obtaining constant-scale query time complexity.

Definition 1 ***Conventional Cuckoo Hashing.*** *Suppose that k is the number of hash functions, and S is a set of items. For the case of $k = 2$, conventional cuckoo hashing table H uses two independent and uniformly distributed hash functions h_1, h_2: $S \to \{0, \ldots, n-1\}$, where n is the size of the hash table. An item x can be stored in any of Buckets $h_1(x)$ and $h_2(x)$ in H, if being inserted successfully.*

The operation for inserting Item x proceeds by computing two hash values of Item x to find Buckets $h_1(x)$ and $h_2(x)$ that could be used to store the item. If either of the two buckets is empty, the item is then inserted into that bucket. Otherwise, an item is randomly chosen from the two candidate buckets and kicked out by Item x. The replaced item is then relocated to its own backup position, possibly replacing another item, until an empty bucket is found or a kick-out threshold is reached. The sequence of replaced items in an insertion operation is called a *cuckoo path*. For example, "$b \rightarrow k \rightarrow \phi$" is one cuckoo path to identify one bucket available to insert Item x as illustrated in Figure 1, which shows a standard cuckoo hash table with two hash functions. The start point of an edge (arrow) represents the actual storage position of an item, and the end point is the backup position. For instance, the bucket storing Item c is the backup position of Items a and n.

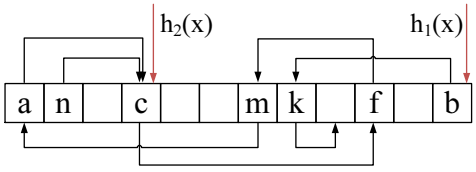


Figure 1: The conventional cuckoo hashing table.

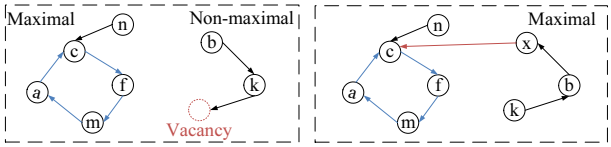
A cuckoo graph is formed by considering each bucket in the hash table as a vertex and each item as an edge. The cuckoo graph can be transformed into a pseudoforest based on graph theory [29].

2.2 Pseudoforest Theory

A pseudoforest is an undirected graph where each vertex only corresponds to at most an edge, and each of maximally connected components, named **subgraphs**, has at most one cycle [7, 23]. The cycle formation starts from a vertex and returns to the vertex through connected edges. Namely, each subgraph in a pseudoforest has no more edges than vertices.

In order to clearly illustrate the direction of a cuckoo path in insertion operations, we extend the pseudoforest into a directed graph by adding the directions from storage positions of items to their backup positions. In the directed pseudoforest, each vertex corresponds to a bucket, and each edge corresponds to an inserted item from the storage vertex to its backup vertex. In the conventional cuckoo hash tables, each bucket stores at most one item, and thus each vertex in a directed pseudoforest has an outdegree of at most one. If the outdegree of a vertex is zero, the vertex corresponds to a vacant bucket, and the subgraph having the vertex contains no cycles, which is called **non-maximal subgraph**. Otherwise, if the outdegrees of all vertices are equal to one, the number of edges of the subgraph is equal to that of vertices, and thus the subgraph contains a cycle, which is a **maximal subgraph**. Any insertion into the subgraph containing a cycle leads to an endless loop [28]. Therefore, if the states of corresponding subgraphs are known before the item is inserted, the insertion result can be predetermined. Based on this property, we can accurately predetermine the occurrence of endless loops without the need of brute-force checking in a step-by-step way.

Figure 2a shows the corresponding pseudoforest of the cuckoo hash table before inserting Item x in Figure 1, which contains one maximal subgraph and one non-maximal subgraph. After inserting Item x, the two subgraphs are merged into a maximal subgraph, as shown in Figure 2b. In particular, after executing the kick-out operations during the insertion, the original vacant vertex becomes the storage position of Item k, and the vertex currently storing Item b becomes the backup position of Item k. The arrow between two vertices needs to be reversed.



(a) Before inserting x. (b) After inserting x.

Figure 2: The directed pseudoforest.

In order to maintain the relationship of subgraphs in the pseudoforest, we assign a subgraph number to each bucket and its corresponding vertex, and hence the buckets with the same subgraph number belong to the same subgraph. However, as the subgraphs are merged during item insertion, the vertices with different subgraph numbers are merged into the same subgraph and need to be represented by the same subgraph number. Frequent updates to subgraph numbers of vertices cause severe insertion latency. In order to address this problem, we leverage **the disjoint-set data structure** [24] to maintain the relationship of subgraphs without updating the subgraph numbers.

2.3 The Disjoint-set Data Structure

In general, a disjoint-set data structure [24] is a tree-based structure that handles merging and lookup operations upon disjoint (non-overlapping) subsets, like our design goal. This structure offers near-constant-time complexity in adding new subsets, merging existing subsets, and determining if two or more elements exist in the same subset. Each element in the structure stores an *id*, and a parent pointer. If an element's parent pointer does not point to any other elements, this element is called the root of the tree and becomes the representative member of its subset. A subset may contain only one element. However, if the element has a parent, the element is part of the subset that is identified by uptracking the parents' chain until a representative element (without a parent) is found at the root of the tree.

Three operations can be performed efficiently on the disjoint-set data structure:

MakeSet(x) creates a subset of a new element x, which has a unique *id*, and a parent pointer to itself. The parent

pointer to itself indicates that the element is the representative member of its own subset. The *MakeSet* operation has $O(1)$ time complexity.

Find(x) follows the chain of parent pointers from a leaf element x up to the tree until it reaches the representative member of the tree to which x belongs. In order to make *Find* operations time-efficient, **path compression** operation is used to flatten the tree-based structure by allowing each element to point to the root whenever *Find* is performed. This is valid because each element accessed on the way to the root is part of the same subset. The resulting flatter tree not only speeds up the future operations on these elements, but also accelerates the operations that reference them.

Union(x,y) uses **Find(x)** and **Find(y)** to determine the roots of x and y. If the roots are distinct, the two corresponding trees are combined by attaching the root of one to that of the other, e.g., the root of the tree with fewer elements to the root of the tree having more elements.

The unique number of each subgraph in the directed pseudoforest is viewed as an element in the disjoint-set data structure. When a new subgraph with a unique number is generated, the *MakeSet* operation is called to generate a new corresponding subset. The *Find* operation is performed when we want to know if the subgraphs of two vertices belong to the same subgraph. Moreover, the *Union* operation is triggered when subgraphs are merged.

3 The CoCuckoo Design

The cost-efficient CoCuckoo improves throughput performance via a graph-grained locking to support concurrent insertions and lookups. CoCuckoo leverages a directed pseudoforest containing several subgraphs to represent items' hashing relationship, which is used to indicate the cuckoo paths in insertion operations. A subgraph consists of the vertices corresponding to buckets, as well as the edges corresponding to the inserted items from their storage positions to their backup positions. The disjoint-set data structure is used to maintain the relationship among subgraphs and stored in memory. Meanwhile, the pseudoforest is just a variant of the cuckoo hash table and is not stored. Figure 3 shows the framework of CoCuckoo. A key and its corresponding metadata are stored in each bucket of the hash table (**H_Table**). The metadata per bucket include the position of the value corresponding to the key (*v_pos*), and the subgraph number of the subgraph to which the bucket belongs (*sub_id*). The *sub_id*s are initialized to -1. Moreover, the disjoint-set data structure called **UF_Array** is described in Section 3.3.2.

Each candidate bucket of an item to be inserted into the hash table possibly corresponds to a vertex in the pseudoforest. If a bucket in the hash table does not correspond to any vertices in the pseudoforest, it means that this bucket has not been visited before and is not a candidate bucket for any inserted items. In general, this bucket corresponds to an *EMPTY* subgraph, and its *sub_id* is -1. Hence, an item

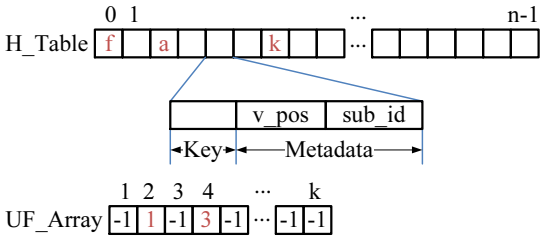


Figure 3: The CoCuckoo framework.

can be directly inserted into a bucket corresponding to an *EMPTY* subgraph. There are at most two *EMPTY* subgraphs for each item insertion due to the existence of two candidate positions. According to the number of corresponding *EMPTY* subgraphs, we classify item insertions into three cases, namely, *TwoEmpty*, *OneEmpty*, and *ZeroEmpty*, respectively showing 2, 1, and 0 *EMPTY* subgraphs.

Moreover, only insertion operations sharing the same cuckoo path require the locks for guaranteeing the correctness of concurrent insertions. Insertion operations through different cuckoo paths access different subgraphs, which can be simultaneously executed and have no lock contention. Based on this observation, CoCuckoo allows the vertices, which correspond to candidate positions of the path-overlapped items, to exist in the same subgraph. We leverage graph-grained locking to avoid potential collisions. These threads won't conflict as long as they manipulate different subgraphs. Most subgraphs are small enough as demonstrated in Figure 6, and hence only a few vertices are constrained at a time.

3.1 Intra-thread Operation

Items inserted into cuckoo hashing form a *cuckoo graph*, which is represented as a directed pseudoforest in our CoCuckoo. Each vertex in the pseudoforest corresponds to a bucket of the hash table and each edge corresponds to an item. An inserted item generates an edge from its actual storage position to its backup position. The pseudoforest reveals cuckoo paths of kick-out operations for item insertion. Hence, the directed pseudoforest can be used to track and exhibit path overlapping of items in advance to avoid potential collisions by a graph-grained locking.

3.1.1 The Case of *TwoEmpty*

When two candidate positions of an item have not yet been used and represented by any vertices in subgraphs of the pseudoforest, i.e., two *EMPTY* subgraphs, the item can be directly inserted into one bucket (the position hashed by the first function by default). Hence, the insertion needs to create a new subgraph, which is non-maximal and ends up with an empty vertex, as shown in Figure 4a to insert Items a, f, and k, respectively. To clearly show these cases, we leverage two hash tables in the following examples. Items hashed by the second function is inserted into the second hash table.

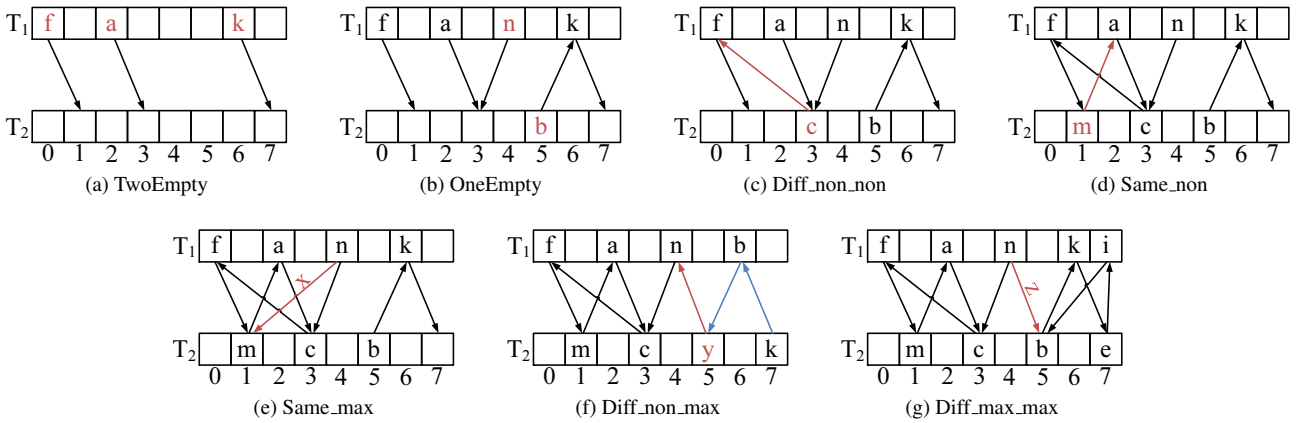


Figure 4: The cases of item insertions.

3.1.2 The Case of *OneEmpty*

One of two candidate buckets of an item corresponds to an existing vertex in the pseudoforest, and the other corresponds to an *EMPTY* subgraph, which will become a new vertex after inserting the item. As shown in Figure 4b, Items b and n can be directly inserted into Buckets $T_2[5]$ and $T_1[4]$ respectively due to available vacancy.

3.1.3 The Case of *ZeroEmpty*

Two candidate positions of an item correspond to two existing vertices in the pseudoforest, which exist either in the same subgraph or both subgraphs. Moreover, each subgraph is either maximal or non-maximal. According to the states and number of subgraphs, we classify the case of *ZeroEmpty* into four subcases.

Diff_non_non: If the vertices corresponding to two candidate positions of an item exist in two non-maximal subgraphs, the item can be successfully inserted into the hash table due to the existence of vacancies. The two subgraphs will be further merged, i.e., inserting Item c into Bucket $T_2[3]$, as shown in Figure 4c.

Same_non: When the two vertices are in the same non-maximal subgraph, there exists a vacant bucket for the item. It will be inserted into the hash table and the corresponding subgraph becomes maximal. For example, Item m is inserted into the hash table and we add an edge from Bucket $T_2[1]$ to Bucket $T_1[2]$. Hence the corresponding subgraph forms a loop, as shown in Figure 4d.

Max: If the two vertices exist in one maximal directed subgraph (named **Same_max**, e.g., Item x in Figure 4e), or two maximal subgraphs (named **Diff_max_max**, e.g., Item z in Figure 4g), no vacancies are available for the new item. The insertion fails even if executing many kick-out operations within a loop. Unlike it, CoCuckoo predetermines the failure in advance and store the item in temporary space (e.g., a stash) [16, 26].

Diff_non_max: One vertex exists in a maximal subgraph and the other is in a non-maximal subgraph in this case. There exists a vacant bucket for the item. The insertion will be successful after several kick-out operations. For example, as shown in Figure 4f, the cuckoo path is "$b \rightarrow k \rightarrow \phi$" when inserting Item y. The two subgraphs are further merged into a new maximal directed subgraph after the insertion.

3.2 Inter-thread Synchronization Optimization

Each thread gets a request from its own task queue each time. Only insertion operations sharing the same cuckoo path require locks for guaranteeing the correctness of concurrent insertions. If two or more insertion requests are path-overlapped, the corresponding threads access to the same subgraph. Hence, the threads that visit the same subgraph have to wait for locks due to guaranteeing the correctness, and the threads that visit different subgraphs can be executed concurrently. To further alleviate the lock contention and overheads, we optimize the operations on different cases of insertions to release locks as soon as possible.

Algorithm 1 shows the steps involved in the insertion of Item x. First, we predetermine the insertion failure if the case is *MAX*, and then lock the subgraphs as shown in Algorithm 2. Second, we determine the case of item insertion and execute corresponding insertion operations.

3.2.1 The Case of *TwoEmpty*

The corresponding thread allocates a new subgraph number, which is locked and assigned to the two *EMPTY* subgraphs. The subgraph number is used to identify a unique subgraph. Since the buckets of *EMPTY* subgraphs can be accessed without acquiring locks, other threads may occupy these buckets before the thread assigns a subgraph number to them. We utilize two atomic subgraph number assignment operations (based on *Compare-And-Swap* instructions) for consistency. If both atomic operations are successful, the two *EMPTY* subgraphs are assigned the same subgraph number. The item can be inserted directly into one of its candidate buckets without kick-out operations. Finally, a non-maximal subgraph is produced. Once the two *EMPTY* subgraphs are

Algorithm 1 Insert(Item x, Hash a, Hash b)

```
1: /*a and b are two indexes of Item x's candidate positions*/
2: if SG_a is maximal && SG_b is maximal then
3:     Return; /*Failure predetermination*/
4: end if
5: LockGraphs(a,b);
6: Result ← False;
7: if SG_a and SG_b are EMPTY then
8:     Result ← InsertTwoEmpty(x,a,b);
9: else if SG_a is EMPTY || SG_b is EMPTY then
10:    if SG_a is EMPTY then
11:        Result ← InsertOneEmpty(x,a,b);
12:    else
13:        Result ← InsertOneEmpty(x,b,a);
14:    end if
15: else
16:    if SG_a is maximal && SG_b is maximal then
17:        Result ← InsertMax(x,a,b);
18:    else if SG_a is non-maximal && SG_b is non-maximal then
19:        if SG_a == SG_b then
20:            Result ← InsertSameNon(x,a,b);
21:        else
22:            Result ← InsertDiffNonNon(x,a,b);
23:        end if
24:    else if SG_a is non-maximal then
25:        Result ← InsertDiffNonMax(x,a,b);
26:    else
27:        Result ← InsertDiffNonMax(x,b,a);
28:    end if
29: end if
30: if Result == False then
31:    Goto 2;
32: end if
```

Algorithm 2 LockGraphs(Int a, Int b)

```
1: while True do
2:    if SG_b < SG_a then
3:        SWAP(SG_a,SG_b);
4:    end if
5:    if SG_a is EMPTY then
6:        Return;
7:    else
8:        /*Lock subgraphs in order to avoid deadlocks*/
9:        LOCK(SG_a);
10:       if SG_a ≠ SG_b then
11:           LOCK(SG_b);
12:       end if
13:   end if
14:   if SG'_a == SG_a&&SG'_b == SG_b then
15:       /*SG'_a and SG'_b are subgraph numbers after locking*/
16:       break;/*Double check*/
17:   else
18:       UNLOCK(SG_a);
19:       if SG_a ≠ SG_b then
20:           UNLOCK(SG_b);
21:       end if
22:   end if
23: end while
```

found with different subgraph numbers, the atomic operations fail, and the *Insert* operation has to be re-executed as shown in Algorithm 3.

Algorithm 3 InsertTwoEmpty(Item x, Hash a, Hash b)

```
1: LOCK(SG); /*The corresponding subgraph number*/
2: if AtomicAssign(&SG_a,SG)&&AtomicAssign(&SG_b,SG)
   then
3:     DirectInsert(x,Bucket[a]);/*Insert directly into B[a]*/
4:     Tag[SG] ← NON_MAX_MARK;
5:     UNLOCK(SG);
6:     Return True;
7: else
8:     UNLOCK(SG);
9:     Return False;
10: end if
```

3.2.2 The Case of *OneEmpty*

One candidate bucket of the item to be inserted corresponds to an *EMPTY* subgraph, which is vacant. The other candidate bucket of the item corresponds to an existing vertex of a subgraph. The item can be directly inserted into the *EMPTY* subgraph, no matter what the state of another subgraph is. Hence, we utilize two atomic operations without locks to execute the insertion operation. The *Insert* operation atomically assigns the number of the existing subgraph to the new vertex, and inserts the item into the new vertex by an atomic write operation as shown in Algorithm 4. Moreover, the state of the final merged subgraph depends on the pre-merged subgraph without changes. If the vertex has been already occupied by another item, the subsequent atomic write operation of insertion fails, which means that the original *EMPTY* subgraph has been merged with another subgraph and becomes not empty. The *Insert* operation has to be restarted, and the insertion case becomes *ZeroEmpty*. Hence, the insertion protocol ensures forward progress and doesn't produce repeated atomic operation failures.

Algorithm 4 InsertOneEmpty(Item x, Hash a, Hash b)

```
1: if AtomicAssign(&SG_a,SG_b) then
2:     if AtomicInsert(x,Bucket[a]) then
3:         Return True;
4:     else
5:         Return False;
6:     end if
7: else
8:     Return False;
9: end if
```

3.2.3 The Case of *ZeroEmpty*

Diff_non_non: The *Insert* operation locks the two corresponding non-maximal subgraphs, inserts the item after several kick-out operations, and then releases the lock after the

two subgraphs have been merged as shown in Figure 5. The merged subgraph is non-maximal, which has a vacant bucket for another item to be inserted.

Algorithm 5 InsertDiffNonNon(Item x, Hash a, Hash b)

1: $Kick\text{-}out(x, Bucket[a])$; /*Enter from $B[a]$*/
2: $Union(SG_a, SG_b)$;
3: $UNLOCK(SG_a)$;
4: $UNLOCK(SG_b)$;
5: Return $True$;

Same_non: The merged subgraph is maximal after the item is inserted, and no vacancies are available for other items to be inserted. Hence, to ease the lock contention, our optimization is to lock the corresponding subgraph and mark it to be maximal, and unlock the subgraph before executing kick-out operations of the insertion, as shown in Algorithm 6.

Algorithm 6 InsertSameNon(Item x, Hash a, Hash b)

1: $Tag[SG] \leftarrow MAX\_MARK$;
2: $UNLOCK(SG)$;
3: $Kick\text{-}out(x, Bucket[a])$;/*Enter from $B[a]$*/
4: Return $True$;

Max: No vacancies are available in the corresponding subgraphs for items in this case, and the *Insert* operation will always walk into a loop and be predetermined to a failure. We just unlock the corresponding subgraph(s) without any other operations, as shown in Algorithm 7.

Algorithm 7 InsertMax(Item x, Hash a, Hash b)

1: $UNLOCK(SG_a)$;
2: **if** $SG_a \neq SG_b$ **then**
3: $\quad UNLOCK(SG_b)$;
4: **end if**
5: Return $True$;

Diff_non_max: There exists only one vacancy for an item, and the state of the merged subgraph is predetermined to be maximal after the item is inserted in the case. Other threads accessing the subgraph first obtain the subgraph state and will not insert items when the subgraph is maximal. For the *Insert* operation, the corresponding thread obtains the lock of the non-maximal subgraph and marks it to be maximal, and then releases the lock immediately. The insertion with several kick-out operations and the merging operation of two subgraphs complete outside the lock, as shown in Algorithm 8.

Algorithm 8 InsertDiffNonMax(Item x, Hash a, Hash b)

1: $Tag[SG_a] \leftarrow MAX\_MARK$;
2: $UNLOCK(SG_a)$;
3: $UNLOCK(SG_b)$;
4: $Kick\text{-}out(x, Bucket[a])$;/*Enter from $B[a]$*/
5: $Union(SG_a, SG_b)$;
6: Return $True$;

3.3 Subgraph Management

3.3.1 Subgraph Number Allocation

We allocate each newly created subgraph a new number for identification. Each subgraph number represents a unique subgraph, and each vertex of the subgraph records the number of its corresponding bucket. All threads allocate subgraph numbers concurrently. When a thread requests a subgraph number, we lock the number generator. Other threads have to wait for unlocking, thus increasing the response time of requests and possibly becoming performance bottleneck. Moreover, we only need to confirm that the subgraph numbers are unique without the need of continuity. In order to decrease the response time without locks and ensure consistency of subgraph number allocation, we leverage a simple modular function to compute the subgraph numbers for all threads. In the modular function, the modulus is the total number of threads p, and the remainder is the number of each thread r. Hence, the subgraph number allocated by each thread is $n = kp + r$, while the parameter k is an accumulator. A subgraph number generator serves for a thread. For example, in the 8-thread CoCuckoo, the subgraph numbers allocated by $Thread$ 2 is 2, 10, 18, and so on. Hence, we allocate subgraph numbers for each thread in order, and then add one to the accumulator k after creating a new subgraph.

3.3.2 Subgraph Merging

When vertices corresponding to two candidate positions of an inserted item exist in two subgraphs, they are merged after item insertion. To avoid exhaustively searching for all vertex members of corresponding subgraphs and updating their subgraph numbers (*sub_id*s in metadata) when subgraphs are merged, a tree-based data structure called **disjoint-set data structure** is utilized to maintain the relationship between subgraphs. Each node in the tree stores a subgraph number and a parent pointer. In order to avoid deadlocks, we always merge the subgraphs with bigger numbers into that with smaller numbers. After subgraphs are merged, the parent pointer of the node with bigger number points to the node with smaller number, which becomes the representative of the tree. The newly merged subgraph is possible to be merged again with others. Finally, the subgraph number of the tree's root represents the number of the subgraph merged from all prior subgraphs.

The $Union$ operation is called when merging subgraphs is needed. $Union(sub\_id1, sub\_id2)$ uses $Find(sub\_id1)$ and $Find(sub\_id2)$ to determine the roots of the nodes with sub_id1 and sub_id2. If the roots are distinct, the trees are combined by attaching the tree whose root has the bigger subgraph number to the root of the smaller one. Furthermore,

$Find(sub\_id)$ follows the chain of parent pointers from a leaf node with sub_id1 up to the tree until it reaches a root node, whose parent is itself. The subgraph number stored in the root node is the actual subgraph number of all nodes in the chain.

An array **UF_Array** is used to implement the disjoint-set data structure. The length of the array is the number of subgraphs, and the indexes of the array indicate the subgraph numbers. The values of elements are initialized to -1, and updated to the indexes of their parents after merging. Specifically, there are two cases for the value of an element in the array: (1) If the value of an element is equal to -1, the subgraph number represented by the index is a root node; (2) If the value of an element is larger than 0, the subgraph number corresponding to the value is the parent node of the subgraph number represented by the index. As shown in Figure 3, $UF\_Array[1] = -1$ means that the subgraph number 1 is the root node; $UF\_Array[2] = 1$ means that the subgraph number 1 is the parent node of the subgraph number 2. Therefore, all buckets with subgraph number 2 have the actual subgraph number of 1.

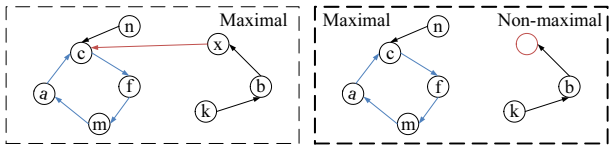
3.3.3 Item Deletion and Subgraph Splitting

For an item, when the vertex of the storage position is not on a cycle, the corresponding subgraph is split into two subgraphs when deleting the item [6]. To avoid reconstructing the hash table and updating *sub_ids* of all buckets, the *re_edge* information in metadata per bucket is added to record the related edges of the bucket in the corresponding subgraph. When deleting Item x, we first compute the two corresponding buckets of Item x, i.e., determine the storage and backup positions. Item x is deleted from the storage bucket in the hash table, and the corresponding edge is deleted from the *re_edge* in two buckets. All related edges and buckets are then recursively searched from the *re_edge* in the candidate buckets. Finally, all searched buckets are updated with a new *sub_id*. In order to optimize the delete operation after splitting subgraphs, all buckets in the subgraph containing the storage position of Item x do not need to update their *sub_ids*. Their actual *sub_ids* can be searched by UF_Array. The thread that performed the delete operation acquires the lock of the corresponding subgraph until the operation completes. Figure 5 shows an example of splitting subgraph when deleting Item x. The backup position of Item x (namely, the storage position of Item c) is found by hash computation, and all related nodes in the left subgraph are iteratively searched by the *re_edge* information in the metadata of related buckets. We then update all *sub_id*s of searched related buckets in hash tables.

4 Performance Evaluation

4.1 Experimental Setup

The server used in our experiments is equipped with an Intel 2.4GHz 16-core CPU, 24GB DDR3 RAM, and 2TB hard disk. The L1 and L2 caches of the CPU are 32KB and 256KB, respectively. The programming language of all functional components of CoCuckoo scheme is C/C++. Moreover, multi-threading is implemented by the *pthread* via a *pthread.h* header and a thread library.



(a) Before deleting x. (b) After deleting x.

Figure 5: An example of splitting subgraph.

Workloads: The widely-used industry standard in evaluating the performance of key-value stores is the *Yahoo! Cloud Serving Benchmark* (YCSB) [14]. We use this benchmark to generate five workloads, each with different proportions of *Insert* and *Lookup* queries to represent real-world scenarios, as shown in Table 1. Especially, the INS workload is the worst case for cuckoo hashing (resulting in a nearly full table), and the remaining four workloads are common cases with *Lookup* operations. Moreover, each workload has two million key-value pairs. Each key in workloads is 16 bytes and each value is 32 bytes for most experiments (except in Section 4.2.4 that uses various sizes for evaluating the effect of size on throughput). The default cuckoo hash table has $2^{21} = 2,097,152$ slots, which consumes about 96MB memory in total.

Table 1: Distributions of different queries in each workload.

Workload	Insert	Lookup
Insert-only (INS)	100%	0%
Insert-heavy (IH)	75%	25%
Insert-lookup balance (ILB)	50%	50%
Lookup-heavy (LH)	25%	75%
Lookup-only (LO)	0%	100%

Threads: In our experiments, there are five settings for the number of threads, including a single thread, 4, 8, 12, and 16 threads, to evaluate the concurrency performance.

Comparisons: We compare our proposed CoCuckoo with open-source *libcuckoo* [3, 32], which is optimized to offer multi-reader/multi-writer service through spin_locks based on concurrent multi-reader/single-writer cuckoo hashing used in *MemC3* [19]. Since *libcuckoo* has multiple slots per bucket to mitigate collisions [19,39,41,51], we also evaluate the performance of 2-, 4-, 8- and 16-way *libcuckoo*. We follow the default configuration of *libcuckoo* in the original paper [32] and only adjust the numbers of threads and slots to facilitate fair comparisons in concurrency. The results of CoCuckoo and *libcuckoo* come from in the same experimental environment.

We focus on the performance improvements from our design in the context of workloads with concurrent insertions and lookups by measuring the throughput and latency of

multiple threads accessing the same hash table. In general, the request response on the cuckoo hashing table becomes slower as the load factor increases, since more items have to be moved [32]. Hence, we measure the throughput and latency for certain load factor intervals (from 0 to 80%), and average throughput and latency.

4.2 Results and Analysis

4.2.1 Lock Granularity

A graph-grained lock is used for concurrency control in our CoCuckoo. Since most subgraphs are small, the granularity of graph-grained locks is acceptable, which only constrain a very small number of buckets. We have measured the number of subgraphs in each size interval with Insert-only workload. Figure 6 demonstrates that most subgraphs are small. For example, 44.25% subgraphs contain only 3 vertices, and about 99% subgraphs contain no more than 10 vertices. Very few buckets are constrained once for ensuring the correctness of concurrency control of the multi-thread hash tables. We obtained identical experimental results under single-threaded and multi-thread conditions. The reason is that for given hash functions, the hash location of each item is determined. Based on hashed locations, the cuckoo subgraphs are also determined, except the difference of the order in which the vertices are added into subgraphs due to the concurrency of threads.

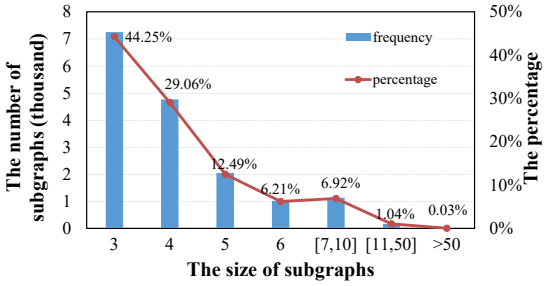


Figure 6: The number of subgraphs in each size interval.

4.2.2 Throughput

This subsection evaluates the average throughput under an increasing number of threads with five workloads and throughput at different table occupancies with Insert-only workload. The throughput is the average number of requests completed per second.

Figure 7 shows the average throughput under an increasing number of threads with five different workloads. The average throughput increases with the increasing number of threads in all cuckoo hashing tables, due to the multi-thread acceleration. In *libcuckoo*, the lower associativity improves the throughput, because each lookup checks fewer slots in order to find the key, and each insertion needs to probe fewer slots in a bucket for an empty slot. In 4-way *libcuckoo*, each *Lookup* requires at most two cache-line reads to find the key and one more cache-line read to obtain the value. Furthermore, in 16-way *libcuckoo*, each *Lookup* requires at most eight cache-line reads to find the key and one more cache-line read to obtain the value. With the increasing number of threads, we observe that CoCuckoo significantly increases the average throughput over *libcuckoo* by 50% to 130% as shown in Figure 7f. In particular, the growth rate is defined as the throughput growth ratio of CoCuckoo relative to 2-way *libcuckoo*.

Figure 8a illustrates that the impact of load factors on 16-thread cuckoo hashing throughput for Insert-only workload. With the increase of load factors, the throughput decreases, since each *Insert* operation has to probe more buckets for finding an empty slot, and requires more item replacements to insert the new item. The *libcuckoo* utilizes *Breadth-First Search* (BFS) to find an empty slot for item insertion, and the path threshold is 5 in open-source implementation [3]. In 1-way *libcuckoo*, the threshold is reached at lower load factor, and expensive rehashing operations are executed, which results in poor performance. However, a higher load factor (more than 0.65) results in performance decrease in terms of throughputs of CoCuckoo. We argue that as the load factor increases, the subgraph merge operations become more frequent, thus leading to the decreasing number of subgraphs and the increasing number of vertices included in a single subgraph. Hence, more vertices are constrained by one thread that handles the *Insert* operation due to the graph-grained locking. Compared with *libcuckoo*, the proposed CoCuckoo obtains significant performance improvements in terms of throughputs.

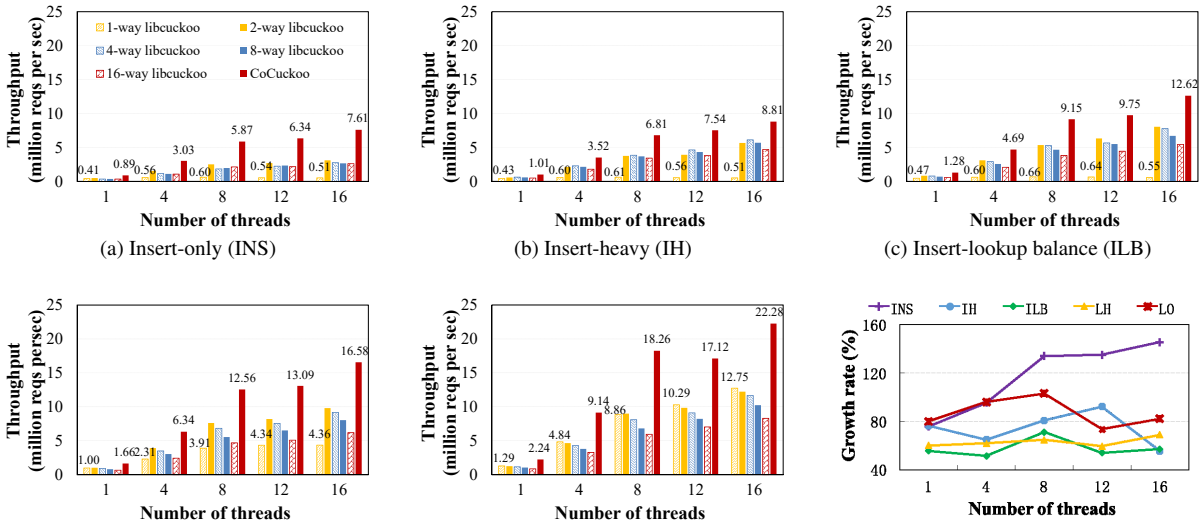
4.2.3 Predetermination for Insertion

Table 2 shows the fractions of all cases with different workloads of containing *Insert* operations in 16 threads. In each workload, the two cases of *TwoEmpty* and *OneEmpty* account for a large proportion, which means that most insertion operations probe new and empty buckets, and add new vertices into corresponding subgraphs in the pseudoforest. Hence, with executing these insertion operations, the threads leverage short-term (*TwoEmpty*) or no (*OneEmpty*) locks the shared buckets, which alleviates the lock contention and further improves the throughput of CoCuckoo. The *Max* case occurs in INS and IH workloads, which means the proposed CoCuckoo predetermines insertion failures and releases locks without any kick-out operations to ease the contention.

4.2.4 Different Key Sizes

All prior experiments used the workloads with 16-byte keys and 32-byte values. We further evaluate the average throughput of cuckoo hash tables with different key sizes at the load factor of 0.8, as shown in Figure 8b- 8f.

In each figure, we show the average throughput, as the key size increases from 8 bytes to 64 bytes with Insert-only workload. The throughput decreases, as the key size increases due to the increased *String_copy* and *String_comp* overheads, as well as memory bandwidth overhead. More-



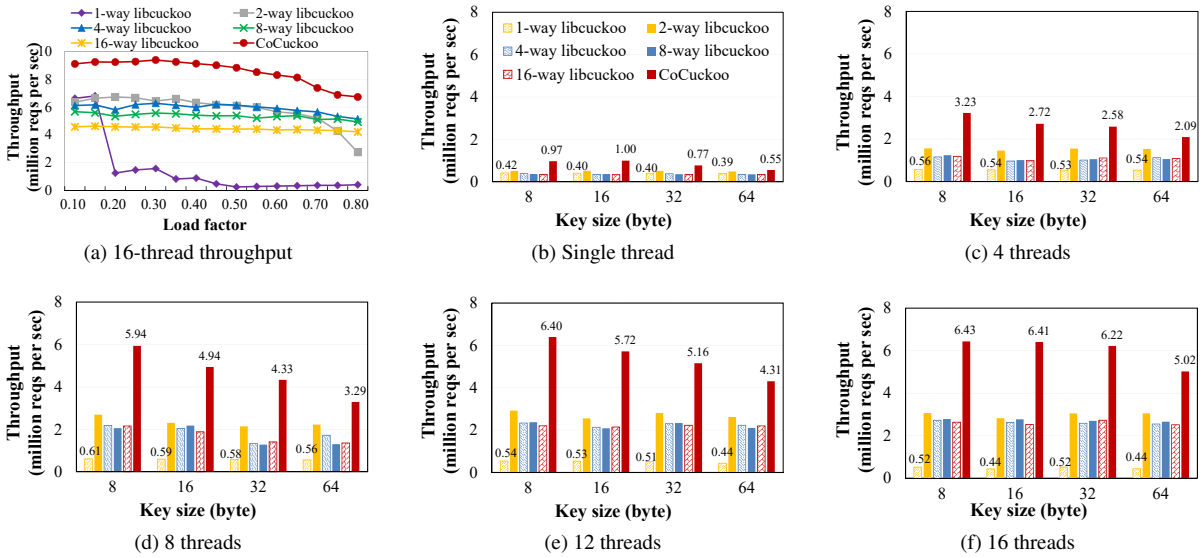
(a) Insert-only (INS) (b) Insert-heavy (IH) (c) Insert-lookup balance (ILB)

(d) Lookup-heavy (LH) (e) Lookup-only (LO) (f) Growth rate compared with libcuckoo.

Figure 7: The average throughput and growth rate compared with 2-way *libcuckoo*.

Table 2: The fractions of all cases in 16 threads.

Workloads	TwoEmpty	OneEmpty	Same_non	Max	Diff_non_non	Diff_non_max
Insert-only	25.673%	37.9628%	0.0003%	13.9802%	13.1447%	9.239%
Insert-heavy	32.9343%	40.4907%	0.0004%	3.5921%	16.7513%	6.2312%
Insert-lookup balance	44.675%	39.6011%	0.0002%	0%	15.7235%	0.0002%
Lookup-heavy	64.4448%	30.1658%	0%	0%	5.3894%	0%



(a) 16-thread throughput (b) Single thread (c) 4 threads

(d) 8 threads (e) 12 threads (f) 16 threads

Figure 8: The throughput in different table occupancies and sizes of keys with Insert-only workload.

over, our concurrent cuckoo hashing becomes much less effective with large keys. For example, in the case of 16 threads, the throughput of CoCuckoo is 6.43 million requests per second with 8-byte keys, which is 135% higher than 4-way *libcuckoo*. The throughput of CoCuckoo is only 95% higher than 4-way *libcuckoo* with 64-byte keys. Similarly, hyperthreading also becomes much less effective with larger keys. For example, with 64-byte keys, the 4-thread throughput of CoCuckoo is 2.09 million requests per second, 8-thread throughput is more than 57% higher than 4-thread throughput, but 16-thread throughput is only 50% higher than 8-thread throughput. In order to support large-size keys, the full keys and values can be stored outside the table and referenced by a pointer, which possibly damages lookup performance. A null pointer indicates that the bucket is unoccupied.

4.2.5 Extra Space Overhead and Impact

Extra space overhead comes from the auxiliary structure of CoCuckoo, which includes two parts. One is the UF_Array to maintain the relationship among subgraphs. The other is the *sub_id* (subgraph number) stored in each bucket of the cuckoo hash table. Specifically, the UF_Array length is the number of subgraphs, and the indexes of the array indicate the subgraph numbers. The maximum number of subgraphs is equal to that of buckets in hash tables. The subgraph number is an *Int* type data in our implementation, which is usually 4 bytes in currently compilers (e.g., GCC). For subgraph numbers, the *sub_id* is stored in each bucket as metadata.

The default cuckoo hash table has 2^{21} slots. Therefore, the extra space overhead is totally $2^{21} * (4+4)B = 16MB$, which is deterministic and very small, compared with current memory capacity. Moreover, the pseudoforest is the theoretical transformation form of the cuckoo hash table, which is not stored in memory. In essence, we leverage acceptable space overhead to obtain significant performance improvements, which is a suitable trade-off. Moreover, in order to examine its impact upon system performance, we evaluate the throughput with the extra space through the Insert-only workload. To facilitate fair comparisons, we define the identical space available for both *libcuckoo* and CoCuckoo. As shown in Figure 9, we observe that CoCuckoo increases the throughput over 2-way *libcuckoo* by 73% to 159%, which is comparable to 75%-150% in Figure 7a in Section 4.2.2, thus exhibiting little impact.

We also evaluate the average execution time per request. As shown in Figure 10, the average time per request in 16-thread CoCuckoo is 1.66μs, which is much shorter than 16.37μs in *libcuckoo*. Frequent rehash operations occur during the item insertion of 1-way *libcuckoo*, resulting in longer insertion time.

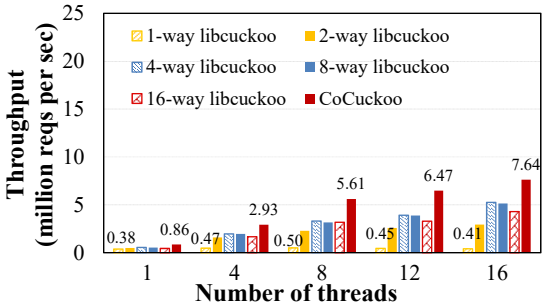


Figure 9: The average throughput with the same space overhead at the load factor of 0.8.

4.2.6 Deletion Latency

The *re_edge* information in metadata per bucket is included for supporting deletion. We evaluate the impact of deletion on performance by measuring request latency. The latency is defined as the time required to be executed in the concurrent program operation per request except the time in a serial program operation. Specifically, for a request, the latency is

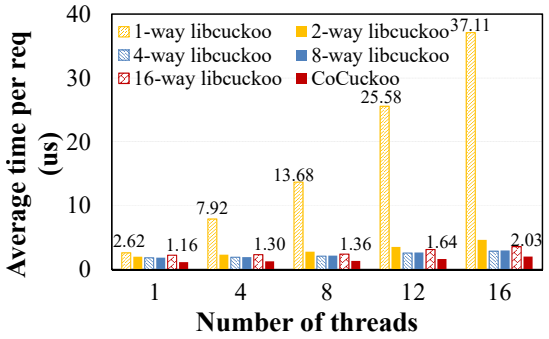


Figure 10: The average time per request with the same space overhead.

equal to the execution time of a thread in a concurrent program minus the time it consumes in a serial program. We use the YCSB benchmark to generate two workloads: (1) 5% *Delete*: 95% *Insert* and 5% *Delete* requests; (2) 10% *Delete*: 90% *Insert* and 10% *Delete* requests, each with one million key-value pairs with deletion.

Figure 11 shows the average request latency of CoCuckoo with six workloads containing *Insert* and *Delete* operations. With the increasing number of threads, the average latency increases due to more intense lock contentions. The time waiting for locks increases with the increasing number of threads. For example, with Insert-only workload, the single-thread latency is 1.16μs per request, 8-thread latency is 2.6% longer than single-thread latency, and 16-thread latency is 50.0% longer than 8-thread latency. However, with the same number of threads, the average latency of different workloads incurs slight changes. Deletions are generally executed within locks, which exacerbates lock contentions. The latency of *Delete* is similar to that of *Insert* and larger than that of *Lookup*.

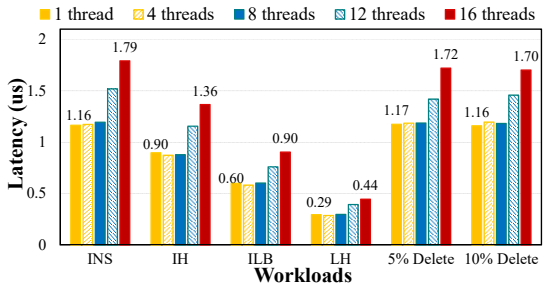


Figure 11: Average latency in different workloads.

Figure 12 illustrates the impact of load factors on latency performance of CoCuckoo with 5% *Delete*. As the load factor increases, there is a slight decrease in latency. Due to our optimization in Section 3.2.2, the average latency of Case *OneEmpty* is smaller than that of Case *TwoEmpty*. The proportion of Case *OneEmpty* increases while the proportion of Case *TwoEmpty* significantly decreases as the load factor increases. Hence, the overall latency decreases. In the meantime, due to appropriate load factors, the decrease of request latency is slow. For example, 16-thread latency is 1.73μs at

a load factor of 0.10 for the hash table, and 1.68μs at a load factor of nearly 0.50, which is only 2.9% smaller than that at a load factor of 0.10.

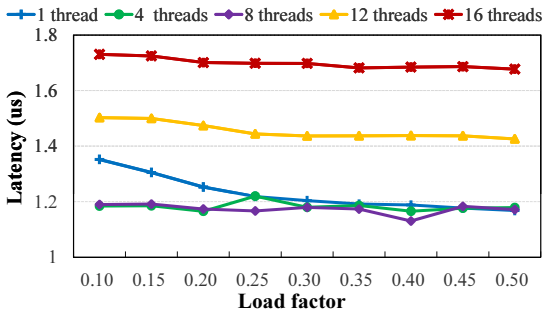


Figure 12: The latency at different table occupancies with 5% *Delete*.

5 Related Work

Performance-constrained single-thread hash tables. The Google's *dense_hash_map*, which is available in the Google SparseHash [1] library, supports fast lookup services. The dense hash uses open addressing with internal quadratic probing and achieves space efficiency for extremely high speeds. Moreover, this table stores items in a single large array and maintains a maximum 0.5 load factor by default.

Horton table [11] is an enhanced bucketized cuckoo hash table for reducing the number of CPU cache lines that are accessed in each lookup, thus achieving higher throughput. Most items are hashed by only a single function and therefore are retrieved by accessing a single bucket, namely, a single cache line. For negative lookups, item remapping enables low access costs to access one cache line.

SmartCuckoo [45] is a cost-efficient cuckoo hash table for accurately predetermining the status of cuckoo operations and the occurrence of endless loops. A directed pseudoforest representing the hashing relationship is utilized to track item placements in the hash table, and further avoid walking into an endless loop.

However, the performance of these hash tables does not scale with the number of cores in the processor, due to only a single thread permitted in execution, which suffer from the performance bottleneck of slow writes in storage systems. Unlike them, the design goal of our CoCuckoo is to efficiently support concurrent operations and deliver high performance in storage systems via concurrency-friendly hash tables.

High-performance concurrent hash tables. *Relativistic hash table* [46] is the data structure that supports shrinking and expanding while allowing concurrent, wait-free and linearly scalable lookups. The proposed resize algorithms reclaim memory as the number of items decreases, and enable Read-Copy Update (RCU) [34–36, 46] hash tables to maintain constant-scale performance as the number of item increases, without delaying or disrupting readers.

*MemC*3 [19] is designed to provide caching for read-mostly workloads and leverages the optimistic cuckoo hashing, which supports multiple readers without locks and a single writer. Instead of moving "items" forward along the cuckoo path, the cuckoo hashing used in *MemC*3 moves "vacancies" backwards along the cuckoo path. The backward method ensures that an item can always be found by a reader.

The *libcuckoo* [32] redesigns *MemC*3 to minimize the size of the hash table's critical sections and allow for significantly increased parallelism, which supports two concurrency control mechanisms, i.e., fine-grained locking and hardware transactional memory.

These schemes are dedicated to improve lookup performance and however fail to work well for tables with more insertion than lookup operations due to the occurrence of hash collisions and endless loops. Our proposed CoCuckoo focuses on write-heavy workloads for supporting concurrent insertions and lookups.

Moreover, CoCuckoo currently addresses the performance bottleneck for cuckoo hashing with two hash functions. The setting of more than two hash functions would significantly increase operation complexity [16,52], which can be reduced to two using techniques such as double hashing [27].

6 Conclusion

Most existing concurrent cuckoo hash tables are used for read-intensive workloads and fail to address the potential problem of endless loops during item insertion. We proposed CoCuckoo, an optimized concurrent cuckoo hashing scheme, which represents cuckoo paths as a directed pseudoforest containing multiple subgraphs to indicate items' hashing relationship. Insertion operations sharing the same cuckoo path need to access the same subgraph, and hence a lock is needed for ensuring the correctness of concurrent operations. Insertion operations accessing different subgraphs enable simultaneous execution. CoCuckoo classifies item insertions into three cases and leverages a graph-grained locking mechanism to support concurrent insertions and lookups. We further optimize the mechanism and release locks as soon as possible to mitigate the contention for pursuing low lock overheads. Extensive experiments using the YCSB benchmark demonstrate that the proposed CoCuckoo achieves higher throughput performance than state-of-the-work scheme, i.e., *libcuckoo*.

7 Acknowledgments

This work was supported by National Key Research and Development Program of China under Grant 2016YFB1000202, and National Natural Science Foundation of China (NSFC) under Grant No. 61772212. The authors are grateful to anonymous reviewers and our shepherd, Haris Volos, for their constructive feedbacks and suggestions.

References

[1] Google SparseHash. https://github.com/sparsehash/sparsehash.

[2] Intel Threading Building Block. https://www.threadingbuildingblocks.org/.

[3] Libcuckoo library. https://github.com/efficient/libcuckoo.

[4] Memcached. A distributed memory object caching system. http://memcached.org/, 2011.

[5] The Top 20 Valuable Facebook Statistics. https://zephoria.com/top-15-valuable-facebook-statistics/, Updated March 2019.

[6] ALSTRUP, S., GØRTZ, I. L., RAUHE, T., THORUP, M., AND ZWICK, U. Union-Find with Constant Time Deletions. In *International Colloquium on Automata, Languages, and Programming* (2005), Springer, pp. 78–89.

[7] ÀLVAREZ, C., BLESA, M., AND SERNA, M. Universal Stability of Undirected Graphs in the Adversarial Queueing Model. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures (SPAA)* (2002), ACM, pp. 183–197.

[8] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. A View of Cloud Computing. *Communications of the ACM 53*, 4 (April 2010), 50–58.

[9] AYERS, G., AHN, J. H., KOZYRAKIS, C., AND RANGANATHAN, P. Memory Hierarchy for Web Search. In *Proceedings of 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2018), IEEE, pp. 643–656.

[10] BRESLOW, A. D., AND JAYASENA, N. S. Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity. *Proceedings of the VLDB Endowment 11*, 9 (2018), 1041–1055.

[11] BRESLOW, A. D., ZHANG, D. P., GREATHOUSE, J. L., JAYASENA, N., AND TULLSEN, D. M. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In *Proceedings of 2016 USENIX Annual Technical Conference (USENIX ATC)* (2016), USENIX Association, pp. 281–294.

[12] BYKOV, S., GELLER, A., KLIOT, G., LARUS, J. R., PANDYA, R., AND THELIN, J. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)* (2011), ACM.

[13] CALCIU, I., SEN, S., BALAKRISHNAN, M., AND AGUILERA, M. K. Black-box Concurrent Data Structures for NUMA Architectures. In *Proc. ASPLOS* (2017), ACM, pp. 207–221.

[14] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)* (2010), ACM, pp. 143–154.

[15] DAVID, T., DRAGOJEVIC, A., GUERRAOUI, R., AND ZABLOTCHI, I. Log-free concurrent data structures. In *Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC)* (2018), USENIX Association, pp. 373–386.

[16] DEBNATH, B. K., SENGUPTA, S., AND LI, J. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *Proceedings of 2010 USENIX Annual Technical Conference (USENIX ATC)* (2010), pp. 1–16.

[17] DEVROYE, L., AND MORIN, P. Cuckoo hashing: Further analysis. *Information Processing Letters 86*, 4 (2003), 215–219.

[18] EPPSTEIN, D., GOODRICH, M. T., MITZENMACHER, M., AND TORRES, M. R. 2-3 Cuckoo Filters for Faster Triangle Listing and Set Intersection. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)* (2017), ACM, pp. 247–260.

[19] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the Symposium on Network System Design and Implementation (NSDI)* (2013), vol. 13, pp. 385–398.

[20] FAN, B., ANDERSEN, D. G., KAMINSKY, M., AND MITZENMACHER, M. D. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014), ACM, pp. 75–88.

[21] FATOUROU, P., KALLIMANIS, N. D., AND ROPARS, T. An Efficient Wait-free Resizable Hash Table. In *Proc. SPAA* (2018), ACM.

[22] FRIEDMAN, M., HERLIHY, M., MARATHE, V., AND PETRANK, E. A Persistent Lock-Free Queue for Non-Volatile Memory. In *Proc. PPoPP* (2018), ACM, pp. 28–40.

[23] GABOW, H. N., AND WESTERMANN, H. H. Forests, Frames, and Games: Algorithms for Matroid Sums and Applications. *Algorithmica 7*, 1 (1992), 465–497.

[24] GALLER, B. A., AND FISHER, M. J. An Improved Equivalence Algorithm. *Communications of the ACM 7*, 5 (1964), 301–303.

[25] HUA, Y., XIAO, B., AND LIU, X. Nest: Locality-aware Approximate Query Service for Cloud Computing. In *Proc. INFOCOM* (2013), IEEE, pp. 1303–1311.

[26] KIRSCH, A., MITZENMACHER, M., AND WIEDER, U. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM Journal on Computing 39*, 4 (2009), 1543–1561.

[27] KNUTH, D. E. *The Art of Computer Programming: Sorting and Searching*, vol. 3. Pearson Education, 1997.

[28] KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. Efficient Parallel Algorithms for Graph Problems. *Algorithmica 5*, 1 (1990), 43–64.

[29] KUTZELNIGG, R. Bipartite Random Graphs and Cuckoo Hashing. In *Discrete Mathematics and Theoretical Computer Science* (2006), Discrete Mathematics and Theoretical Computer Science, pp. 403–406.

[30] LEUNG, A. W., SHAO, M., BISSON, T., PASUPATHY, S., AND MILLER, E. L. Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)* (2009), vol. 9, pp. 153–166.

[31] LI, Q., HUA, Y., HE, W., FENG, D., NIE, Z., AND SUN, Y. Necklace: An Efficient Cuckoo Hashing Scheme for Cloud Storage Services. In *Proceedings of the 22nd International Symposium of Quality of Service (IWQoS)* (2014), IEEE, pp. 153–158.

[32] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)* (2014), ACM, p. 27.

[33] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZIS, G., AND CAMBLE, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)* (2009), vol. 9, pp. 111–123.

[34] MCKENNEY, P. E. RCU vs. Locking Performance on Different CPUs. In *linux.conf.au* (2004).

[35] MCKENNEY, P. E., APPAVOO, J., KLEEN, A., KRIEGER, O., RUSSELL, R., SARMA, D., AND SONI, M. Read-Copy Update. In *Ottawa Linux Symposium* (2002), pp. 338–367.

[36] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems* (1998), pp. 509–518.

[37] MITZENMACHER, M. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems 12*, 10 (2001), 1094–1104.

[38] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[39] POLYCHRONIOU, O., RAGHAVAN, A., AND ROSS, K. A. Rethinking SIMD Vectorization for In-Memory Databases. In *Proc. SIGMOD* (2015), ACM, pp. 1493–1508.

[40] RICHA, A. W., MITZENMACHER, M., AND SITARAMAN, R. The Power of Two Random Choices: A Survey of Techniques and Results. *Combinatorial Optimization 9* (2001), 255–304.

[41] ROSS, K. A. Efficient Hash Probes on Modern Processors. In *IEEE 23rd International Conference on Data Engineering (ICDE)* (2007), IEEE, pp. 1297–1301.

[42] SCHLAIPFER, M., RAJAN, K., LAL, A., AND SAMAK, M. Optimizing Big-Data Queries Using Program Synthesis. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (2017), ACM, pp. 631–646.

[43] SHAVIT, N. Data Structures in the Multicore Age. *Communications of the ACM 54*, 3 (2011), 76–84.

[44] SUN, Y., HUA, Y., FENG, D., YANG, L., ZUO, P., AND CAO, S. MinCounter: An Efficient Cuckoo Hashing Scheme for Cloud Storage Systems. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST)* (2015), IEEE, pp. 1–7.

[45] SUN, Y., HUA, Y., JIANG, S., LI, Q., CAO, S., AND ZUO, P. SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC)* (2017), USENIX Association, pp. 553–565.

[46] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of 2011 USENIX Annual Technical Conference (USENIX ATC)* (2011), USENIX, pp. 145–158.

[47] WANG, F., YUN, C., GOLDWASSER, S., VAIKUNTANATHAN, V., AND ZAHARIA, M. Splinter: Practical Private Queries on Public Data. In *Proc. NSDI* (2017), pp. 299–313.

[48] WINBLAD, K., SAGONAS, K., AND JONSSON, B. Lock-free Contention Adapting Search Trees. In *Proc. SPAA* (2018), ACM, pp. 121–132.

[49] WU, S., LI, F., MEHROTRA, S., AND OOI, B. C. Query Optimization for Massively Parallel Data Processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)* (2011), ACM.

[50] WU, Y., AND TAN, K.-L. Scalable In-Memory Transaction Processing with HTM. In *Proceedings of 2016 USENIX Annual Technical Conference (USENIX ATC)* (2016), pp. 365–377.

[51] ZHANG, K., WANG, K., YUAN, Y., GUO, L., LEE, R., AND ZHANG, X. Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1226–1237.

[52] ZUO, P., AND HUA, Y. A Write-friendly Hashing Scheme for Non-volatile Memory Systems. In *Proc. MSST* (2017).