A Collision-Mitigation Cuckoo Hashing Scheme for Large-Scale Storage Systems

Yuanyuan Sun, Yu Hua, *Senior Member, IEEE*, Dan Feng, *Member, IEEE*, Ling Yang, Pengfei Zuo, Shunde Cao, and Yuncheng Guo

Abstract—With the rapid growth of the amount of information, cloud computing servers need to process and analyze large amounts of high-dimensional and unstructured data timely and accurately. This usually requires many query operations. Due to simplicity and ease of use, cuckoo hashing schemes have been widely used in real-world cloud-related applications. However, due to the potential hash collisions, the cuckoo hashing suffers from endless loops and high insertion latency, even high risks of re-construction of entire hash table. In order to address these problems, we propose a cost-efficient cuckoo hashing scheme, called MinCounter. The idea behind MinCounter is to alleviate the occurrence of endless loops in the data insertion by selecting unbusy kicking-out routes. MinCounter selects the "cold" (infrequently accessed), rather than random, buckets to handle hash collisions. We further improve the concurrency of the MinCounter scheme to pursue higher performance and adapt to concurrent applications. MinCounter has the salient features of offering efficient insertion and query services and delivering high performance of cloud servers, as well as enhancing the experiences for cloud users. We have implemented MinCounter in a large-scale cloud testbed and examined the performance by using three real-world traces. Extensive experimental results demonstrate the efficacy and efficiency of MinCounter.

Index Terms—Cuckoo hashing, cloud storage, data insertion and query

1 Introduction

In the era of Big Data, cloud computing servers need to process and analyze large amounts of data timely and accurately. According to the report of International Data Corporation (IDC) in 2014, the digital universe is doubling in size every two years from now until 2020, and the data we create and copy annually will reach 44 ZettaBytes in 2020. The digital bits in data universe will be as many as stars in the physical universe [1]. Industrial companies have already begun to deal with terabyte-scale and even petabytes-scale data everyday [2], [3], [4]. Large fractions of massive data come from the popular use of mobile devices [1]. Due to the constrained energy and limited storage capacity, real-time processing and analysis are nontrivial in the context of cloud-based applications.

Although cloud computing systems consume a large amount of system resources, it is still challenging to obtain accurate results for query requests in a real-time manner [5], [6], [7]. In order to improve entire system performance and storage efficiency, existing schemes have been proposed, such as hierarchical Bloom filter index to speed up the searching process [8], continuously monitoring query execution to optimize the cloud-scale query [9], query optimization for parallel data processing [4], approximate membership query over cloud data [10], multi-keyword search over encrypted cloud data [11], [12], similarity search in file systems [13], minimizing retrieval latency for content cloud information [14] and retrieval for ranked queries over cloud data [7]. Due to space inefficiency and high-complexity hierarchical addressing, these schemes fail to meet the needs of real-time queries.

In order to support real-time queries, hashing-based data structures have been widely used in constructing the index due to constant-scale address\ing complexity and fast query response. Unfortunately, hashing-based data structures cause low space utilization, as well as high-latency risk of handling hashing collisions. Traditional techniques used in hash tables to deal with hashing collisions include open addressing [15], [16], [17], [18], [19], chaining [20], [21], [22] and coalesced hashing [23], [24], [25]. Unlike conventional hash tables, cuckoo hashing [26] addresses hashing collisions via simple "kicking-out" operations (i.e., flat addressing), which moves items among hash tables during insertions, rather than searching the linked lists (i.e., hierarchical addressing). Architecture-conscious hashing [27] has demonstrated that cuckoo hashing is much faster than the chaining hashing with the increase of load factors. The cuckoo hashing makes use of $d \geq 2$ hash tables, and each item has d buckets for storage. Cuckoo hashing selects a suitable bucket for inserting a new item and alleviates hash collisions by dynamically moving items among their d candidate positions respectively in hash tables. Such scheme ensures a more even distribution of data items among hash tables than uses only one hash function in conventional hash tables. Due to the salient feature of flat addressing with constant-scale complexity, cuckoo hashing needs to probe all hashed buckets only once and obtains the query results. Even probing at most d buckets in the worst case, the cuckoo hashing guarantees constant-scale query time

• *Y. Sun, Y. Hua, D. Feng, L. Yang, P. Zuo, S. Cao, and Y. Guo are with Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China.*
E-mail: {sunyuanyuan, csyhua, dfeng, yling, pfzuo, csd, ycguo}@hust.edu.cn.

Manuscript received 28 Feb. 2016; revised 20 July 2016; accepted 21 July 2016. Date of publication 27 July 2016; date of current version 15 Feb. 2017.
Recommended for acceptance by Z. Chen.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2594763

1045-9219 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

complexity and constant amortized time for insertion and deletion process, which is also considered by Rivest in [18]. Cuckoo hashing thus improves space utilization without the increase of query latency.

In order to implement data structures of hash tables to adapt to concurrent hardware, e.g., multiprocessor machines, efficient synchronization of concurrent access to data strutures is essential and significant. More and more studies focus on proposing concurrent hash tables [28], [29], [30], [31], [32], [33].

In practice, due to the essential property of hash functions, the cuckoo hashing fails to fully avoid hash collisions. Existing work to handle hash collisions mainly leverages random-walk approach [34], [35], which suffers from redundant migration operations among servers on account of unpredictable random selection. The random-walk schemes cause endless loops and high latency for re-construction of hash tables eventually, like *ChunkStash* [36]. *ChunkStash* uses an in-memory hash table to index metedata, and the hash collisions are mitigated by cuckoo hashing. The *ChunkStash* system addresses hash collisions via a small number of key relocations to extra linked list (or hash table). The approach alleviates hash collisions with the extra spatial costs. In order to deliver high performance and support real-time queries, we need to deal with three main challenges.

- **Intensive Data Migration**. When new data items are inserted into storage servers via cuckoo hashing, a kicking-out operation may incur intensive data migration among servers if hash collisions occur [32]. The kicking-out operation needs to migrate the selected item to its other candidate buckets and kick out another existing item until an empty slot is found. Frequent kicking-out operations cause intensive data migration among multiple buckets of hash tables. Conventional cuckoo hashing based schemes heavily depend on the timeout status to identify an insertion failure. They complete the insertion process only after experiencing numerous random-walk based kicking-out operations, thus resulting in endless loops and consuming substantial system resources. Hence, we need to avoid or alleviate the occurrence of intensive data migration.
- **Space Inefficiency**. In general, when data collisions occur during the insertion process, we randomly choose a position from d candidates to kick out the existing item, but we cannot predict in advance whether there exists an empty slot for the kicked-out item. Due to the unpredictable random selection in traditional cuckoo hashing, there is always some small but practically significant probability that a failure occurs during the insertion of an item. None of the d buckets are or can easily be made empty to hold the data [37]. In this case, conventional cuckoo hashing has to rehash all items in the hash tables or leverage extra space to store insertion-failure items, leading to space inefficiency of hash tables.
- **High Insertion Latency**. The cuckoo hashing schemes based on random-walk approach move items randomly among their d candidate positions in hash tables [35]. This exacerbates the uncertainty of hash addressing when all candidate positions are occupied. The random selection in kicking-out operations may cause repetitions and infinite loops of kicking-out paths [36], which results in high latency in insertion operation.

In order to address these challenges, we propose a MinCounter scheme for cloud storage systems to mitigate the actual hash collisions and high latency in the insertion process. MinCounter allows each item to have d candidate buckets. An empty bucket can be chosen to store the item. In order to record kicking-out times occurring at the bucket in real time, we allocate a counter for each bucket. When hash collisions occur during the insertion operation, and all candidate buckets are not empty, the item selects the bucket with the minimum counter to kick out the occupied item to reduce or avoid endless loops. The rationale of MinCounter is to select unbusy kicking-out routes independently rather than randomly choose and seek the empty buckets as quickly as possible. Moreover, in order to reduce the frequency of rehashing, we temporarily store insertion-failure items into in-memory cache, rather than directly rehash the entire structure. Specifically, this paper has made the following contributions.

- **Alleviating Hash Collisions and Data Migration**. We propose a novel cuckoo hashing based scheme, called MinCounter, in the cloud storage servers. MinCounter allocates a counter per bucket of hash tables, to record the kicking-out times occurring in the buckets. When d candidate positions of a new item to be inserted are all occupied by other items, MinCounter selects the minimum counter, rather than randomly choose, to execute the replacement operation. This scheme can significantly alleviate hash collisions and decrease data migration by balancing items in hash tables.
- **Improving Space Efficiency and Decreasing Insertion Latency**. MinCounter demonstrates salient performance superiority in terms of insertion latency. It achieves load balance by kicking items out to "cold" (infrequently accessed) positions when hash collisions occur. We can mitigate data collisions and reduce total times of kicking-out operations. MinCounter hence improves space efficiency and decreases insertion latency.
- **Practical Implementation**. We have implemented the MinCounter prototype and compared it with the state-of-the-art cuckoo hashing scheme, ChunkStash [36], in a large-scale cloud computing testbed. We use two real-world traces and a dataset of randomly generated numbers to examine the practical performance of the proposed MinCounter. The results demonstrate significant performance improvements in terms of utilization of hash tables and insertion latency.

The rest of this paper is organized as follows. Section 2 shows the research background. Section 3 presents the MinCounter design and practical operations. Section 4 illustrates the performance evaluation and Section 5 shows the related work. Finally, we conclude our paper in Section 6.

2 Background

The cuckoo hashing was described in [26] as a dynamization of a static dictionary. Cuckoo hashing leverages two or more hash functions for handling hash collisions to mitigate

the computing complexity of using the linked lists of conventional hash tables. An item x has two candidate positions to be placed, i.e., $h_1(x)$ and $h_2(x)$, instead of only one single position in cuckoo hashing scheme. A bucket stores only one item and hash collisions can be decreased. For a general lookup, we only probe whether the queried item is in one of its candidate buckets.

A hash collision occurs when all candidate buckets of a newly inserted item have been occupied. Cuckoo hashing needs to execute "kicking-out" operations to dynamically move existing items in the hashed buckets and select a suitable bucket for the new item. The kicking-out operation is similar to the behavior of cuckoo birds in nature, which kicks other eggs or young birds out of the nest. In the similar manner, the cuckoo hashing recursively kicks items out of their buckets and leverages multiple hash functions to offer multiple choices and alleviates hash collisions.

However, cuckoo hashing fails to fully avoid hash collisions. An insertion of a new item causes a failure and an endless loop when there are collisions in all probed positions until reaching the timeout status. To break the endless loop, an intuitive way is to perform a full rehash if this rare incident occurs. In practice, the expensive overhead of performing a rehashing operation can be dramatically reduced by taking advantage of a very small additional constant-size space [37].

2.1 Standard Cuckoo Hashing

The cuckoo hashing is a dynamization of a static dictionary and supports fast queries with the worst-case constant-scale lookup time due to flat addressing for an item among multiple choices.

Definition 1. *Standard Cuckoo Hashing. Conventional cuckoo hashing uses two hash tables, T_1 and T_2, with the length m, and two hash functions h_1, h_2: $U \rightarrow \{0, \ldots, m-1\}$. Each item $x \in S$ is kept in one of the two buckets: $h_1(x)$ of T_1, $h_2(x)$ of T_2, but never in both. The hash functions h_i, $i = 1, 2$, meet the conditions of independent and random distribution.*

Fig. 1 shows an example of two hash tables to illustrate the practical operations of standard cuckoo hashing. We use arrows to show possible destinations for moving items as shown in Fig. 1a. If item x is inserted into hash tables, we first check whether there exists any empty bucket of two candidates of item x. If not, we randomly choose one from candidates and kick out the original item. The kicked-out item is inserted into $Table_2$ in the same way. The process is executed in an iterative manner, until all items find their buckets. Fig. 1b demonstrates the running process that the item x is successfully inserted into the $Table_1$ by moving items a and b from one table to the other. While, as shown in Fig. 1c, endless loops may occur and some items fail to find a suitable bucket to be stored. Therefore, a threshold "$MaxLoop$" is necessary to specify the number of iterations. If the iteration times are equal to the pre-defined threshold, we can argue the occurrence of endless loops, which causes the re-construction of entire structure. The theoretical analysis of $MaxLoop$ has been shown in Section 4.1 in [38]. Moreover, due to essential property of random choice in hash functions, hash collisions can not be fully avoided, but significantly alleviated [35].

It is shown in [39] that if two hash tables are almost half full, i.e., $m \geq (1+\varepsilon)n$ for a constant parameter $\varepsilon > 0$, and

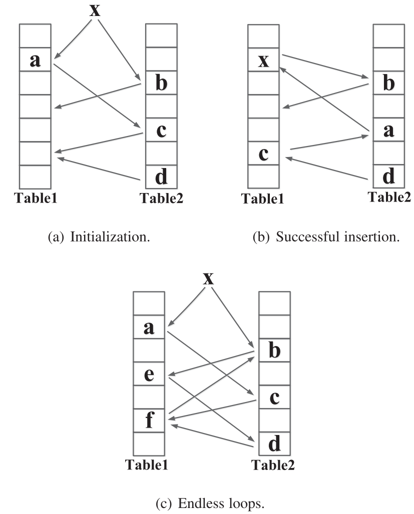


Fig. 1. The example of item insertion in the cuckoo hashing.

h_1 and h_2 are chosen uniformly and randomly from an $(O(1), O(\log n))$-universal family, the probability of being unable to arrange all items of dataset S is $O(1/n)$, according to h_1 and h_2.

2.2 Improved Cuckoo Hashing

There is an improvement in cuckoo hashing and allows each item to own $d > 2$ candidate locations, which has been widely used in real-world applications [32], [36], [37], [40], [41], [42], [43], [44].

Definition 2. *Improved Cuckoo Hashing. Improved cuckoo hashing leverages d hash tables, $T_1, T_2, \ldots, T_d$, and d hash functions $h_1, h_2, \ldots, h_d : U \rightarrow \{0, \ldots, m-1\}$, where m is the length of each hash table. Each item $x \in S$ is kept in one of the d buckets: $h_1(x)$ of T_1, $h_2(x)$ of T_2, ..., $h_d(x)$ of T_d, but never in d buckets. $d(>2)$ is a small constant.*

2.3 Concurrency Control Using Locking

It's important to efficiently support concurrent access to a cuckoo hash table. Some previously proposed schemes are used to improve concurrency [32], [33], [44]. In order to support arbitration for concurrent access in data structures, locking is an efficient mechanism [31], [45], [46], [47].

Multi-thread applications achieve high performance through taking advantage of more and more cores. In order to ensure thread correctness and safety, a critical section controlled by a lock is used to allow the operations of multiple threads to be serialized when accessing the shared part of data.

- **Coarse-grained Lock [32], [33], [45]**. A simple way of locking is to attach a coarse-grained lock to the entire shared data structure, and only one thread can possess the lock in the meanwhile. The thread with the lock prevents other threads from accessing the shared data, even through others only want to read the shared data and have no updates, which has negative influence on concurrent performance.
- **Fine-grained Lock [31], [46], [47]**. Another locking is to divide the coarse-grained lock into multiple fine-grained locks. Each fine-grained lock protects only one part of the shared data structures. Hence,

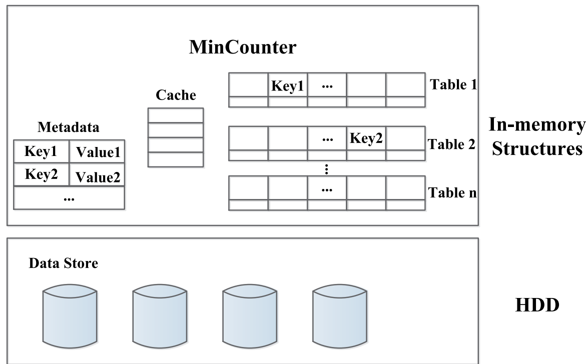


Fig. 2. The architecture of MinCounter storage system.

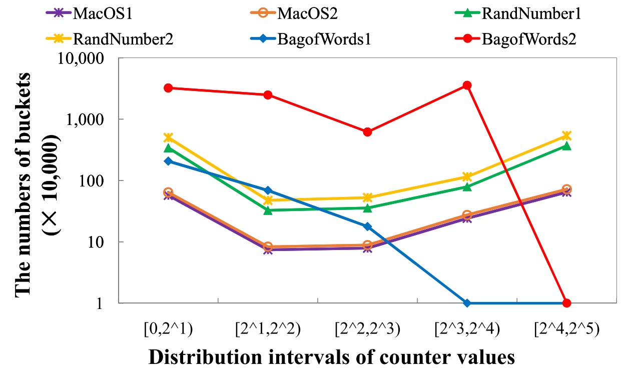


Fig. 3. The unbalanced distribution of buckets.

multiple threads can work on different parts of the structures without conflicts. Fine-grained locking has positive influence on concurrent performance compared with the coarse-grained locking. However, careful design and implementation are needed to avoid deadlock, livelock, starvation, etc.

It is challenging to effectively support concurrent access to cuckoo hash tables. In order to support the concurrent execution of the single-thread cuckoo hashing algorithm, while still maintaining the high space efficiency of cuckoo hashing, we need to deal with two problems.

- **Deadlock Risk(writer/writer)**. Multiple insertion operations process in the concurrent cuckoo hashing. An insertion operation may modify several buckets when moving the items among hash tables until each item has its available bucket. There is a situation in which two or more insertion operations require locks for buckets which are owned by each other. They have to wait for the other to finish, but neither ever does, which results in a deadlock. Basic techniques support atomic insertion operation to avoid deadlock, e.g., acquiring all necessary locks previously, which is not appropriate due to reducing the overall performance of the concurrent system.
- **False Misses(reader/writer)**. Query operations lookup items while kicking-out operations in insertion occur in the concurrent cuckoo hashing. There is a case that an query operation searches an item which is kicked out from its original bucket and before being inserted to its candidate position. The item is unreachable from both buckets and temporarily unavailable. Thus if the insertion-operation is non-atomic, query operation may complete and return a false miss when an item is unreachable.

3 Design and Implementation Details

3.1 The MinCounter Architecture

MinCounter supports a fast and cost-effective cuckoo hashing scheme for data insertion. Due to the simplicity and ease of use, MinCounter has the salient features of high utilization, less data migration and less time overheads. The summarized structure fits into the main memory to improve overall performance. Fig. 2 shows the storage architecture of MinCounter.

We implement the MinCounter component as well as metadata structures in the DRAM. The metadata are in the form of key-value pairs. A key is the hashed value of a file ID and the value is the correlated metadata. The hard disk at the bottom stores and maintains the correlated files. The proposed MinCounter scheme is compatible to existing systems, such as the Hadoop Distributed File System (HDFS) and General Parallel File System (GPFS).

3.1.1 In-Memory Data Structure

We observe that the frequency of kicking-out operations in each bucket of hash tables is not uniform. Some buckets receive frequent kicking-out operations and some perform infrequently. We call the buckets where hash collisions occur frequently as "hot" buckets, and the buckets where hash collisions occur infrequently as "cold" buckets. The frequency is interpreted as the times of hash collisions occurring in the bucket during insertion operations of whole dataset. Fig. 3 illustrates the existence of unbalanced "hot" and "cold" buckets. During the insertion operation, we examine the counter values of buckets in six datasets, which are further classified into five intervals (there is no bucket whose counter value is larger than 2^5). We record the numbers of buckets whose counter values are in each interval. The counter value of each bucket indicates the frequency of hash collisions occurring in the bucket. The buckets where hash collisions frequently occur are "hot", and otherwise "cold". We take advantage of the characteristic to propose an effective scheme, called MinCounter, to alleviate hash collisions and endless loops via a frequency-aware method.

MinCounter is to place items in the cuckoo hashing based data structure. MinCounter selects the "cold", rather than random, buckets and avoids "hot" paths to alleviate the occurrence of endless loops in the data insertion process when hash collisions occur.

It is easy to understand the case of $d = 2$ in the cuckoo hashing. Each bucket contains only one item. When an item is kicked out from its occupied bucket, it has to choose to replace one of items in other candidate buckets due to avoiding self-kicking-out [38], [48]. In real-world applications, the cases of $d > 2$ are more important and widely exist, which is the focus in MinCounter.

The idea behind MinCounter is to judiciously alleviate the hash collisions in the data insertion procedure. Conventional cuckoo hashing can be carried out in only one large hash table or $d \geq 2$ hash tables. Each item of the set S is hashed to d candidate buckets of hash tables. When item x is inserted into the hash tables, we look up all of its d

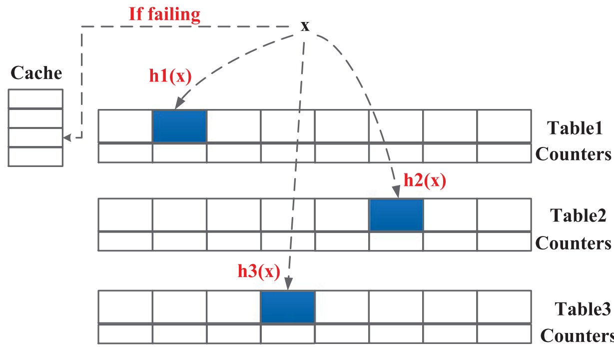


Fig. 4. The data structure of MinCounter.

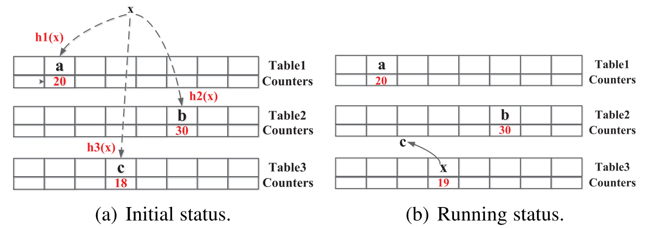
candidate buckets in order to observe whether there is at least one empty slot to insert. Otherwise, we have to replace one in occupied buckets with the item x. Traditionally, we choose to use random-walk cuckoo hashing [34], [35] to address hash collisions. When there is no empty bucket for item x, it randomly chooses the bucket from its candidates to perform replacement operation. In general, we avoid choosing the one that was occupied just now due to avoiding self-kicking-out. Due to the randomness of the random-walk cuckoo hashing, endless loops and repetitions cannot be avoided actually, which affects the overall performance of structures, such as load factor and insertion latency.

MinCounter is a multi-choice hashing scheme to place items in hash tables as shown in Fig. 4. It leverages cuckoo hashing to allow each item to have d candidate buckets. Each item chooses only one bucket to locate. The used hash functions are chosen independently from an appropriate universal hash family. We allocate a counter for each bucket to record the kicking-out times occurring at the bucket during insertion processes of whole dataset. If no empty buckets are available, the item needs to select one with the minimum counter and kick out the occupied item to reduce or avoid the endless loop. But there are always some opportunities that in the insertion of a new item, none of the d candidate positions are empty, thus causing an insertion failure. We temporarily store insertion-failure items into a constant-sized cache [37] rather than rehash the structure immediately to reduce the expensive overhead of structure reconstruction with a slight cost of extra queries.

Fig. 4 illustrates the data structure of MinCounter (e.g., $d = 3$). The blue buckets are the hit positions by hash computation of item x. If all positions $h_i(x) (i = 1, 2, 3)$ are occupied by other items, the item has to replace one through the MinCounter scheme. Furthermore, one item has to be inserted into the extra cache when insertion failure occurs.

3.1.2 The MinCounter Working Scheme

In order to alleviate the occurrence of endless loops in cuckoo hashing, we improve the conventional cuckoo hashing by allocating a counter for each bucket of hash tables. We utilize the counters to record kicking-out times occurring at buckets in history. When a hash collision occurs in a bucket, the corresponding counter increases by 1. If an item x is inserted into the hash tables without the availability of empty candidate buckets, we choose the bucket with the minimum counter to execute the replacement. Particularly, if more



(a) Initial status. (b) Running status.

Fig. 5. The improved cuckoo hashing table structure.

than one counter has the minimum, we choose the bucket with the minimum number of hash tables by default.

As shown in Fig. 5, we take $d = 3$ to give an example. When the item x is inserted into hash tables, we first check the buckets of $h_1(x)$, $h_2(x)$ and $h_3(x)$ in each hash table respectively to find an empty bucket. Each candidate bucket of x is occupied by a, b, c respectively (as shown in Fig. 5a). Moreover, we compare the counters of candidate buckets and choose the minimum one (i.e., 18 in this example), and further replace item c with x. In the meantime, the counter of the bucket of $h_3(x)$ increases by 1 up to 19 (Fig. 5b). The kicked-out item c becomes the one needed to be inserted, and the insertion procedure goes on, until an empty slot is found in hash tables.

MinCounter allows items to be inserted into hash tables to improve the storage space efficiency, but fails to fully address endless loops. Like ChunkStash [36], we leverage an extra space to temporarily store the insertion-failure items rather than rehash the structure immediately. The viability of this approach has also been established in [37], where the authors show, through massive analysis and simulations, that a very small constant-size extra space yields significant improvements, dramatically reducing the insertion failure probabilities associated with cuckoo hashing and enhancing cuckoo hashing's practical viability in both hardware and software. Some schemes also use the similar mechanism with a *stash* [49], [50], [51].

Due to the negligible extra space, we construct a small hash table, in memory. The insertion-failure items store in the table through a random hash function for supporting operations in expected constant time.

3.1.3 Concurrent Cuckoo Hashing

In order to address those challenges presented in Section 2.3 and support multiple writes and reads in a concurrent manner, we improve conventional single-thread cuckoo hashing scheme by fine-grained lock mechanism and an auxiliary space.

We allocate a lock for each bucket of hash tables, and each lock protects only one bucket of hash tables. Insertion operations request locks for buckets when kicking-out operations occur. Different buckets can be protected by different locks, and the operations on different buckets can proceed concurrently. We add extra space in memory as an overflow buffer to temporarily store items kicked out from their original buckets and before being inserted to candidate positions. The size of extra space is equal to the number of threads, so the space overhead is negligible.

During insertion, each operation requests only one lock for bucket where kicking-out operation occurs, and stores the kicked-out item in buffer, which avoids writer/writer

```
Insert(Item x, kickcount, exclude)
  if DirectInsert(Item x) then
    Return
  end if
  if kickcount ≤ MaxLoop then
    FindMinCounter(x, exclude) → k
    lock()
    D[h_k(x)] → y
    x → D[h_k(x)]
    unlock()
    C[h_k(x)] ++
    Insert(y, kickcount+1, k)
  else
    x → S[x]
  end if
```

Fig. 6. The algorithm of item insertion.

deadlock effectively. To avoid misses in reader/writer, we need to check both hash tables and the buffer during query operations. The lock mechanism may have a negative effect on time overhead for one insertion operation, but it improves the overall time performance, compared with no concurrency. For a query, we need to check both the hash tables and the extra space ($stash$ and buffer) to guarantee the query accuracy.

3.2 Practical Operations

We describe practical operations of MinCounter to support item insertion, item query and item deletion.

3.2.1 Insertion

The insertion operation places items into their empty hash buckets to achieve load balance. Fig. 6 illustrates the iterative insertion algorithm for item x. We use the parameter $kickcount$ as the counter to record the cumulative kicking-out times, and the parameter $exclude$ to record the current kicking-out position due to avoiding self-kicking-out. The two parameters are initialized to 0. We first find an empty bucket for the item x to be inserted from its candidates. If there exists an empty bucket and no hash collisions occur, the item x can be directly inserted as shown in Fig. 7. Moreover, if there is no empty bucket among the candidate positions, MinCounter needs to employ the kicking-out operations to find an empty bucket for item x as shown in Fig. 8. If the recursive times reach the threshold $MaxLoop$, we store the insertion-failure item into the extra cache.

We denote $D[*]$ to be the data in the bucket, $C[*]$ to represent the counter of the bucket, and $S[*]$ to represent the

```
DirectInsert(Item x)
  i := 1
  while D[h_i(x)] != NULL and i ≤ d do
    i ++
  end while/* find an empty bucket*/
  if i ≤ d then
    lock()
    x → D[h_i(x)]
    unlock()
    Return True
  else
    Return False
  end if
```

Fig. 7. The algorithm of direct item insertion.

```
FindMinCounter(Item x, exclude)
  k = 1, m = 1
  min = INTMAX
  while m ≤ d do
    if m! = exclude and C[h_m(x)] ≤ min then
      C[h_m(x)] → min
      m → k
    end if
    m++
  end while
  Return k /* find the minimum counter*/
```

Fig. 8. The algorithm of item insertion via MinCounter.

data in the extra space. We need to decide how to choose the replaced item if d candidate buckets for the item x to be inserted are not empty. Our approach is to compare the d counters and select the minimum one, and kick out the occupied item y by x. We further need to locate y in one of its other $d-1$ candidate buckets (except the one that now occupied by x, to avoid the obvious loop).

3.2.2 Query

The query operation needs to traverse all candidate buckets of the item to be queried and the extra space. MinCounter consumes the constant-scale query time of cuckoo hashing in a simple way. Fig. 9 illustrates the details of query operation. We first search all candidate buckets of the item. If the item is found in hash tables, we return the result and finish the query operation. If the query failure occurs, we have to traverse the extra space to search the queried item. We return the result when the item is found in the space. Otherwise, the query operation fails eventually.

3.2.3 Deletion

In the deletion operation, we need to lookup the item to be deleted and remove it from the bucket of hash tables or extra space directly as shown in Fig. 10. We first search the item to be deleted in its all candidate positions of hash tables, and delete it at once when the item is found. If we fail to find the item in the hash tables, we traverse the extra space to delete it. If the item can not be found, it means the item doesn't exist in the data structure.

```
Query(Item x)
  Result = ϕ
  i := 1
  while i ≤ d do
    if D[h_i(x)] == x then
      D[h_i(x)] → Result
      Return Result
    end if
    i++
  end while
  j :=1 /*traverse the extra space, M is the capacity of the space*/
  while j ≤ M do
    if S[j] == x then
      S[j] → Result
    end if
    j++
  end while
  Return Result
```

Fig. 9. The algorithm of item query.

Delete(Item x)

```
i = 1
while i ≤ d do
    if (D[h_i(x)] == x) then
        Delete x
        Return
    end if
    i ++
end while
j = 1
while j ≤ M do
    if (S[j] == x) then
        Delete x
        Return
    end if
    j ++
end while
Return Result
```

Fig. 10. The algorithm of item deletion.

4 Performance Evaluation

4.1 Experimental Setup

We implemented the MinCounter scheme in a 128-node parallel cluster system. Each node is equipped with an Intel 2.8 GHz 16-core CPU, a 16 GB DRAM and 500 GB hard disk. The prototype is developed under the Linux kernel 2.6.18 environment and we implemented all functional components of MinCounter in the user space. In order to demonstrate the efficiency and effectiveness of the proposed MinCounter scheme, we use three datasets and two initial rates, i.e., 1.1 and 2.04, in hash tables. The initial rate means the multiple we set based on the size of datasets to create hash tables. When the rate is 1.1, hash collisions occur frequently, and hardly when the rate is 2.04.

4.2 The Kicking-Out Threshold Settings

Existing cuckoo hashing schemes fail to fully avoid endless loops due to the essential property of hash collisions. In order to alleviate the endless loops and reduce temporal and spatial overheads in the item insertion operations, a conventional method is to pre-define an appropriate threshold to represent the tolerable maximum times of kicking-out per insertion operation. However, it is nontrivial to obtain the suitable threshold value that depends on the application requirements and system status. In order to carry out meaningful experiments, we choose to use experimental approaches to determine the appropriate threshold value.

We use the utilization ratio as the performance metric to identify a suitable threshold. Fig. 11 shows the utilization ratio of cuckoo hash tables increases with the threshold of kicking-out times when insertion failures occur. We observe that the utilization ratio exhibits smoothness and achieves a satisfactory level starting from 80. This means no matter how to increase the threshold, the utilization ratio of hash tables has no significant changes.

In the meantime, as shown in Fig. 12, total kicking-out numbers increase significantly with the increasing threshold of kicking-out times. Intuitively, the extra overhead of kicking-out operations increases with the threshold, we thus need to choose the smallest threshold reaching the satisfactory utilization ratio of hash tables. Therefore, the following experiments are performed with the optimal

TABLE 1
The Dataset of Randomly Generated Numbers

Groups	Range	Size
group1	0-100,000,000	1,314,404
group2	0-80,000,000	2,017,180
group3	0-100,000,000	2,517,415
group4	0-200,000,000	4,960,610
group5	0-500,000,000	7,666,282
group6	0-500,000,000	11,184,784

TABLE 2
The MacOS Trace

Groups	Date	Size
group1	20121101	1,005,000
group2	20130101	1,020,673
group3	20130201	1,027,657
group4	20130401	1,037,681

TABLE 3
The Bag of Words Trace

Groups	Text collections	D	W	N
group1	KOS blog entries	3,430	6,906	353,160
group2	NIFS full papers	1,500	12,419	746,316
group3	Enron Emails	39,861	28,102	3,710,420
group4	NYTimes news articles	300,000	102,660	69,679,427

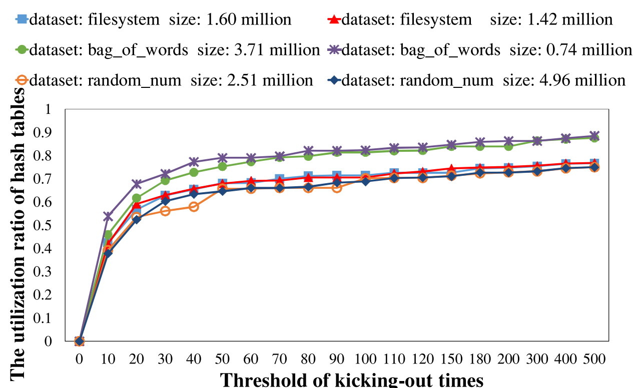


Fig. 11. The utilization ratio of cuckoo hash tables with different thresholds of kicking-out times.

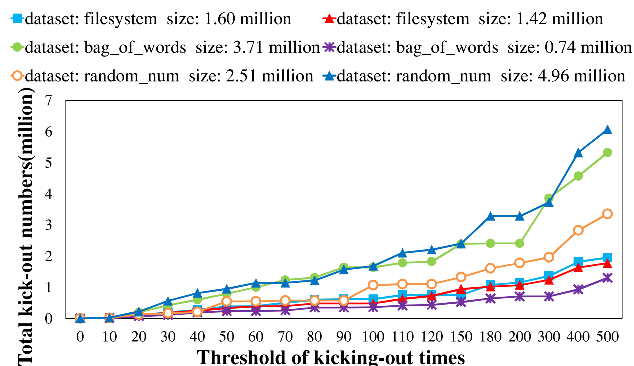


Fig. 12. The total kicking-out numbers of the whole dataset insertion operations with different thresholds of kicking-out times.

TABLE 4
The Distribution of Counter Values

Dataset	Scheme	$[0, 2)$	$[2, 4)$	$[4, 8)$	$[8, 16)$	$[16, 32)$	$[32, 64)$	$[64, 128)$	$[128, +\infty)$	Total
MacOS1	MinCounter	572,095	73,852	78,731	242,152	646,779	0	0	0	15,478,911
	ChunkStash	607,602	35,603	158,600	286,433	313,276	129,008	6,583	6	17,091,350
MacOS2	MinCounter	638,199	83,278	88,425	274,065	719,329	0	0	0	17,214,994
	ChunkStash	677,515	40,334	178,197	321,110	350,754	142,416	7,076	6	19,020,956
RandNum1	MinCounter	3,412,984	326,677	356,746	793,323	3,701,119	0	0	0	102,046,710
	ChunkStash	3,569,932	168,758	643,825	1,171,156	1,508,509	1,018,735	19,1162	2,932	111,370,874
RandNum2	MinCounter	4,996,954	475,067	525,734	1,150,821	5,381,882	0	0	0	149,183,728
	ChunkStash	5,224,727	247,873	933,485	1,702,033	2,196,590	1,482,143	283,787	5,432	162,740,748
BagofWords1	MinCounter	2,064,669	689,473	177,673	0	0	0	0	0	5,846,576
	ChunkStash	2,158,737	1,839,122	790,557	399,215	43,269	213	0	0	11,174,442
BagofWords2	MinCounter	32,298,060	24,833,952	6,188,804	35,676,562	0	0	0	0	452,870,992
	ChunkStash	34,678,813	5,324,479	10,258,177	13,922,742	10,141,169	2,274,767	47,232	5	538,695,011

threshold of 80, compared with other values of 50, 100 and 120 when the initial rate is 1.1. When the rate is 2.04, almost all items are inserted successfully and there is no significant difference of the utilization ratio of hash tables, so we only examine the optimal threshold of 80 in this situation.

4.3 The Counter Size Settings

First, we need to consider the bits per counter of per bucket in hash tables for space savings. Table 4 shows the distribution of counters' values. We randomly choose two groups of data from three datasets respectively for statistic analysis. Most values are distributed in the interval of 0 to 32 (namely 2^5). The values of counters larger than 32 are 0, and it is sufficient to allocate 5 bits per counter. The memory overflow may hardly occur, which means MinCounter leads to the equilibrium distribution. To demonstrate the efficiency of our MinCounter scheme, Fig. 13 shows the bits per counter and the average numbers of kicking-out times per bucket in hash tables when using the MinCounter scheme. We observe that at most 5 bits per bucket is sufficient for a large proportion of dataset.

4.4 The Cache Size Settings

Second, we need to carefully consider the cache size for temporarily storing insertion-failure items in MinCounter. Fig. 14 illustrates the success ratio of data insertion operations with different thresholds of kicking-out times in rate = 1.1. Based on examining 6 groups of data randomly chosen from three datasets, we observe that over 95 percent items in each dataset are inserted successfully into hash tables, and the last two real-world traces are even more than 99 percent. So we choose the constant-sized cache, which is equal to 5 percent of dataset size, for each set of data in MinCounter.

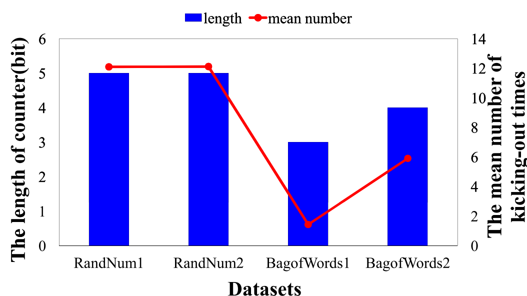


Fig. 13. The distribution of values of counters.

4.5 Experimental Results

4.5.1 *Single-Thread Results*

We show advantages of MinCounter over RandomWalk [35] and ChunkStash [36] by comparing their experimental results in terms of utilization ratio of hash tables when insertion failures occur, total kicking-out times and mean time overheads of whole insertion operations. The thresholds of kicking-out times are 50, 80, 100 and 120. In the following, MT is the threshold of kicking-out times in the MinCounter scheme, RT is the threshold in the RandomWalk scheme and CT is the threshold in the ChunkStash scheme. Meanwhile, numbers behind MT, RT and CT in following figures are thresholds set in experiments, such as MT80 is the MinCounter scheme with the threshold of 80.

- **Utilization Ratio**. Fig. 15 shows the utilization ratio of cuckoo hash tables when insertion failure first occurs by using the dataset of Randomly Generated Numbers. We observe that the average utilization ratio of MinCounter is 75 percent, which is higher than the percentage of 70 percent in RandomWalk and ChunkStash. Compared with RandomWalk and ChunkStash schemes, MinCounter obtains on average 5 percent utilization ratio promotion. RandomWalk and ChunkStash schemes need to choose kicking-out positions randomly when hash collisions occur. There is no guide for avoiding endless loops, and iterations may easily reach the threshold of

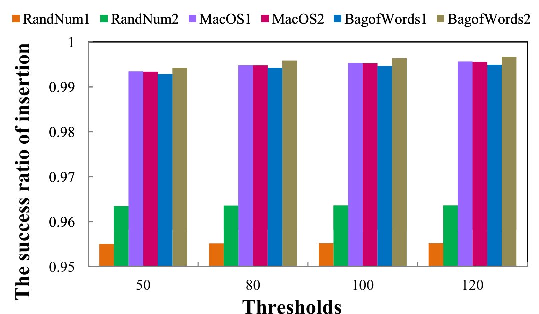


Fig. 14. The success ratio of data insertion operations with different thresholds of kicking-out times.

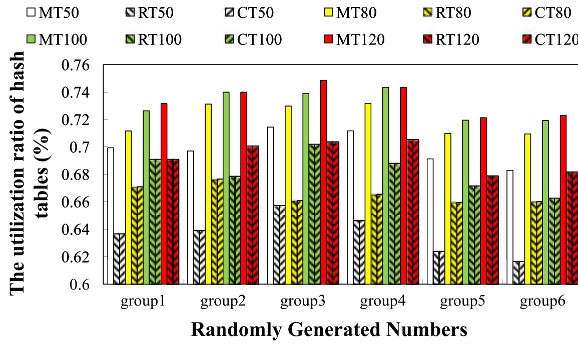


Fig. 15. The utilization ratio of cuckoo hash tables using the dataset of randomly generated numbers.

kicking-out times. Less items can be inserted into hash tables, which results in lower utilization ratio of hash tables. Furthermore, an insertion failure shows the occurrence of an endless loop. A rehash process is needed. MinCounter improves the utilization ratio of hash tables, which means the proposed scheme alleviates hash collisions and decreases the rehash probability. MinCounter optimizes the cloud computing system performance by improving the utilization of hash table and decreasing the rehash probability.

Fig. 16 shows the utilization ratio of cuckoo hash tables when using the trace of fingerprints of MacOS. We observe that compared with RandomWalk and ChunkStash schemes, MinCounter obtains on average 6 percent utilization improvement, while the average utilization ratio of MinCounter is 88 percent in the MacOS trace, and 82 percent in RandomWalk and ChunkStash schemes.

Fig. 17 illustrates the utilization ratio of cuckoo hash tables when using the Bag of Words trace. We observe that MinCounter obtains on average 5 percent utilization improvement, compared with RandomWalk and ChunkStash schemes, while the average utilization ratio of MinCounter is 88 percent in the Bag of Words trace, and 83 percent in RandomWalk and ChunkStash schemes.

- **Total Kicking-out Times**. We examine the total kicking-out numbers of MinCounter, RandomWalk and ChunkStash by using the dataset of Randomly Generated Numbers as shown in Fig. 18. When insertion failure occurs, we store the item into a temporary small additional constant-size cache in MinCounter

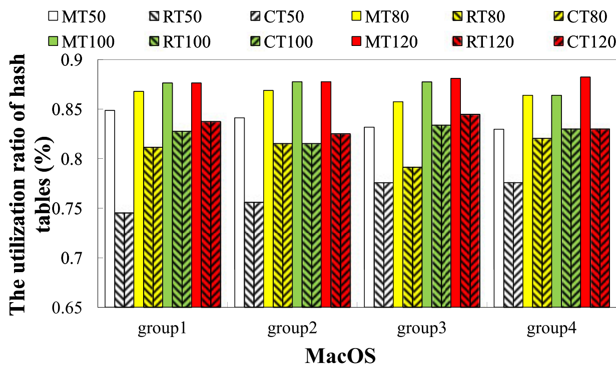


Fig. 16. The utilization ratio of cuckoo hash tables using the trace of MacOS.

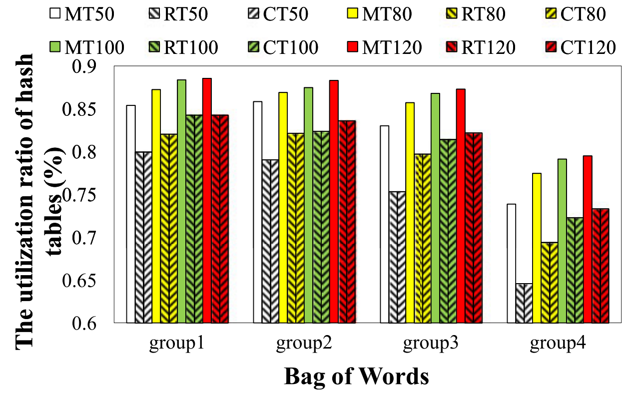
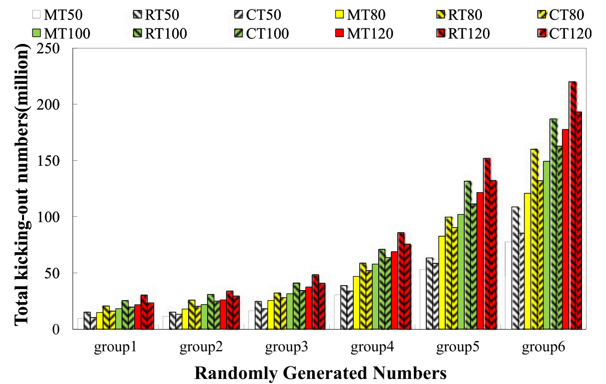


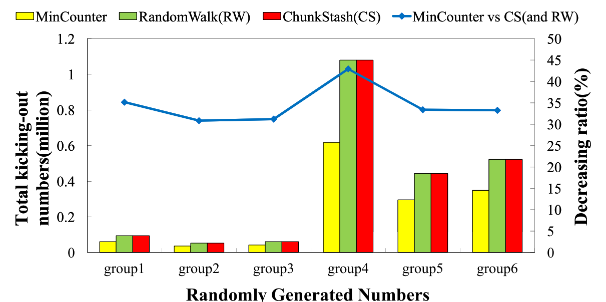
Fig. 17. The utilization ratio of cuckoo hash tables using the trace of Bag of Words.

like ChunkStash rather than rehash tables like RandomWalk immediately. This may cause slight extra space overhead, but obtains the benefit of reducing the failure probability. Compared with RandomWalk and ChunkStash, MinCouter significantly cuts down over 20 percent and 10 percent total kicking-out numbers in rate = 1.1 (in Fig. 19), and on average 37 percent in rate = 2.04 (in Fig. 18b). MinCounter enhances the experiences of cloud users through decreasing total kicking-out times.

We examine the total kicking-out numbers of MinCounter, RandomWalk and ChunkStash by using the trace of MacOS as shown in Fig. 20. Compared with RandomWalk and ChunkStash, MinCouter significantly cuts down almost 55 and 50 percent total kicking-out numbers in rate = 1.1 (in Fig. 21), and on



(a) Rate = 1.1.



(b) Rate = 2.04.

Fig. 18. The total kicking-out numbers of whole insertion operations using the dataset of randomly generated numbers.

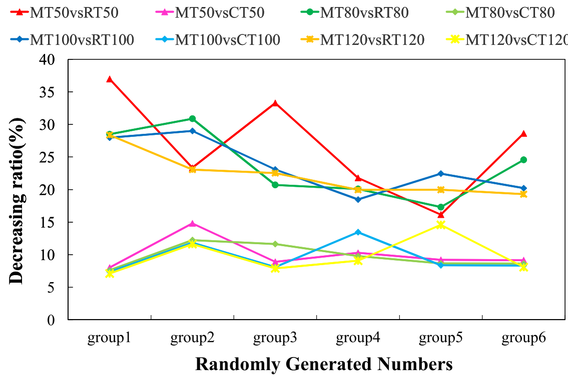


Fig. 19. The decreasing ratio of total kicking-out times of MinCounter using the dataset of randomly generated numbers in rate = 1.1.

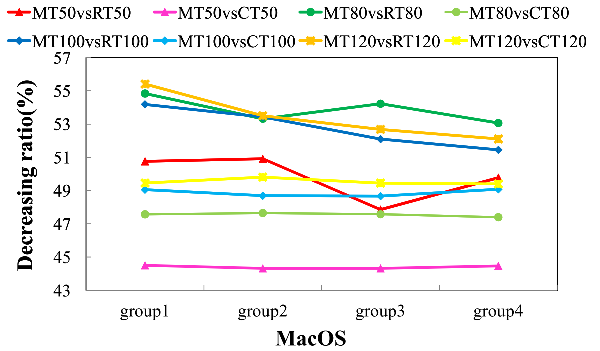


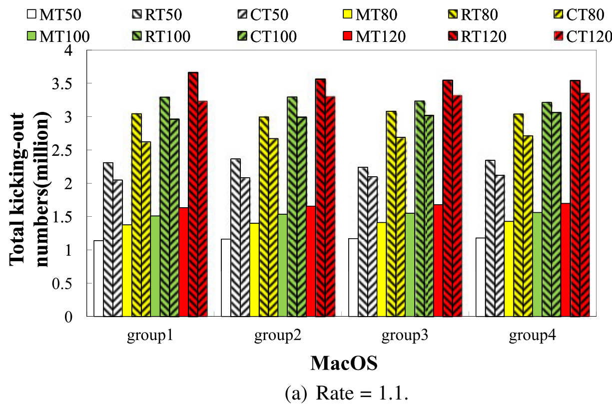
Fig. 21. The decreasing ratio of total kicking-out times of MinCounter using the trace of MacOS in rate = 1.1.

average 30 percent in rate = 2.04 (in Fig. 20b). Such significant improvement comes from the kicking-out balance between hash tables in MinCounter to avoid "hot" buckets and reduce kicking-out times. Moreover, we observe that the case of rate = 1.1 receives more decrease than the case of rate = 2.04. Items have lower probabilities of hash collisions due to having more space capacity in rate = 2.04.

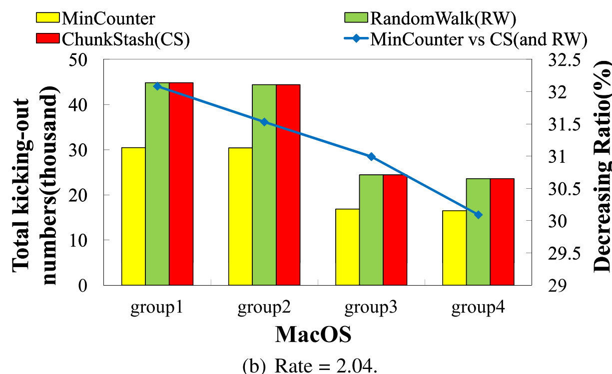
Fig. 22 shows the total kicking-out numbers in the Bag of Words trace. Compared with RandomWalk and ChunkStash, MinCouter reduces about 50 percent total kicking-out numbers in rate = 1.1 (in Fig. 23), and on average 30 percent in rate = 2.04 (in Fig. 22b).

4.5.2 Multi-Thread Reasult

- **Insertion Time Overhead**. *MMT* represents the threshold of kicking-out times in the multi-threads MinCounter scheme, and *SMT* is the threshold in the single-thread scheme. Fig. 24 shows mean time overheads of entire insertion operations by using the dataset of randomly generated numbers. We observe that compared with ChunkStash, single-thread MinCouter significantly reduces about 20 percent time overheads in rate = 1.1 rate = 2.04. Multi-threads MinCounter makes further improvment through cutting down more than 27 percent time overheads in rate = 1.1 (in Fig. 25) and more than 20

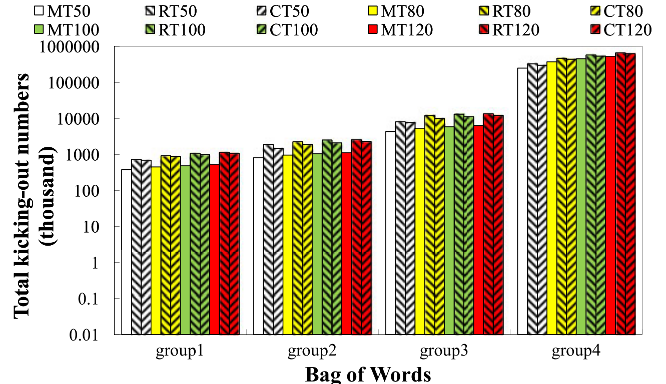


(a) Rate = 1.1.

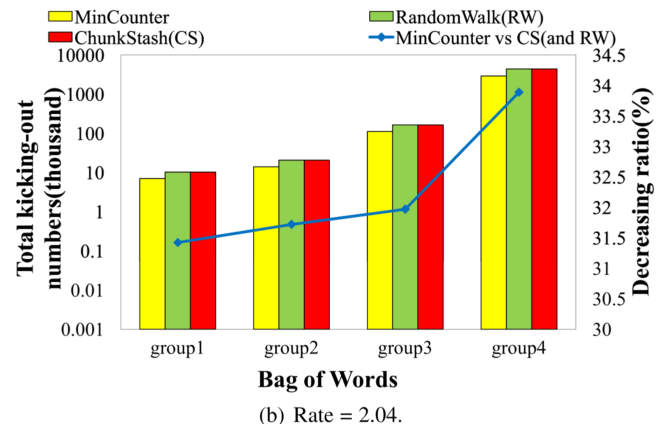


(b) Rate = 2.04.

Fig. 20. The total kicking-out numbers of whole insertion operations using the trace of MacOS.



(a) Rate = 1.1.



(b) Rate = 2.04.

Fig. 22. The total kicking-out numbers of whole insertion operations using the trace of Bag of Words.

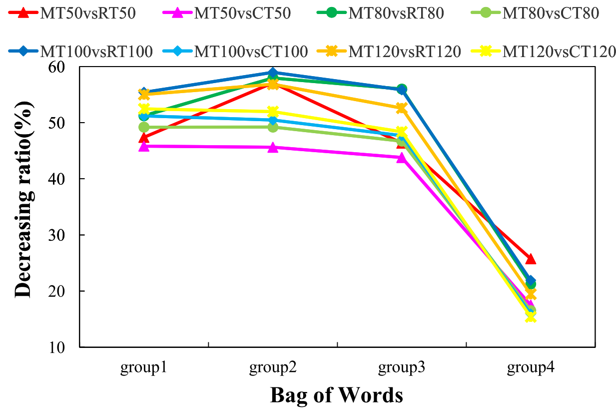
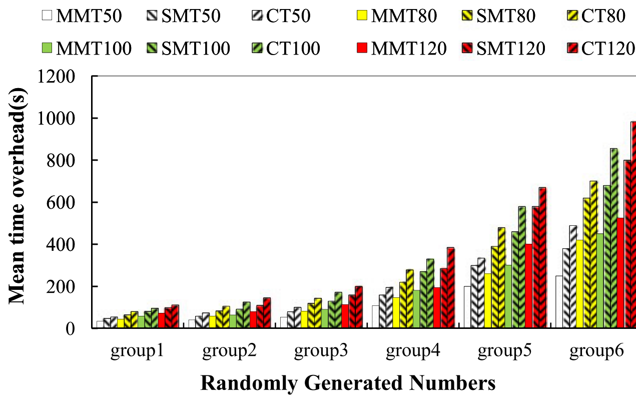
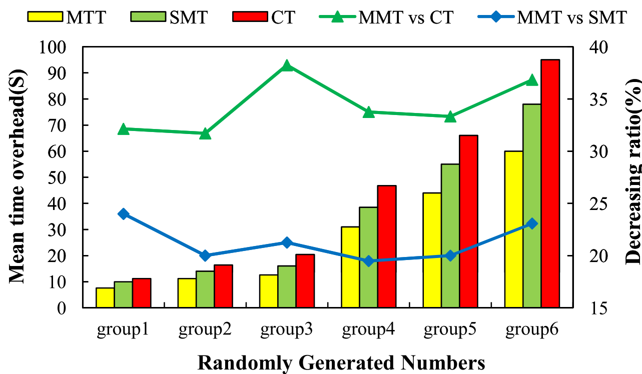


Fig. 23. The decreasing ratio of total kicking-out times of MinCounter using the trace of Bag of Words in rate = 1.1.



(a) Rate = 1.1.



(b) Rate = 2.04.

Fig. 24. The mean time overheads of whole insertion operations using the dataset of randomly generated numbers.

percent in rate = 2.04 (in Fig. 24b). MinCounter optimizes the cloud computing systems performance by decreasing the time overheads.

We examine the time overheads of MinCounter and ChunkStash by using the metric of mean time overheads of whole insertion operations in the trace of MacOS as shown in Fig. 26. In contrast with ChunkStash, single-thread MinCouter significantly decreases about 36 percent time overheads in rate = 1.1 and more than 25 percent in rate = 2.04. In addition, multi-threads MinCounter cuts down more than 32 percent time overheads in rate = 1.1 (in Fig. 27) and about 10 percent in tate = 2.04 (in Fig. 26b).

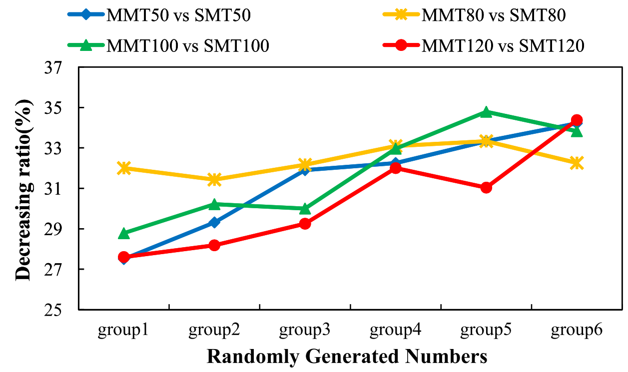
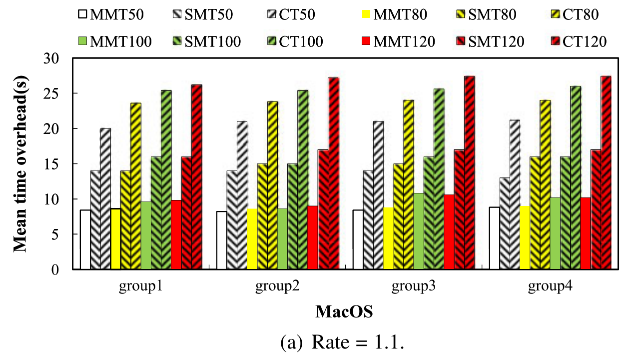
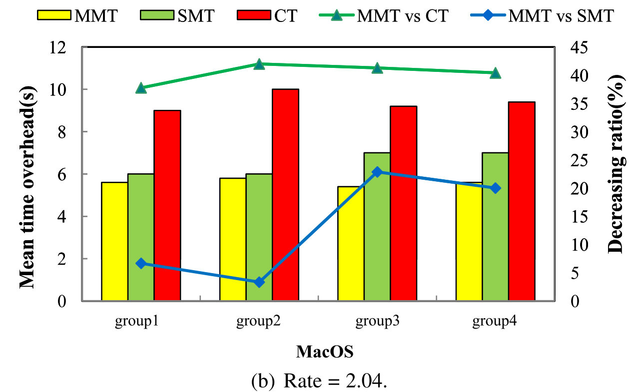


Fig. 25. The decreasing ratio of mean time overheads of whole insertion operations using the trace of randomly generated numbers in rate = 1.1.



(a) Rate = 1.1.



(b) Rate = 2.04.

Fig. 26. The mean time overheads of whole insertion operations using the trace of MacOS.

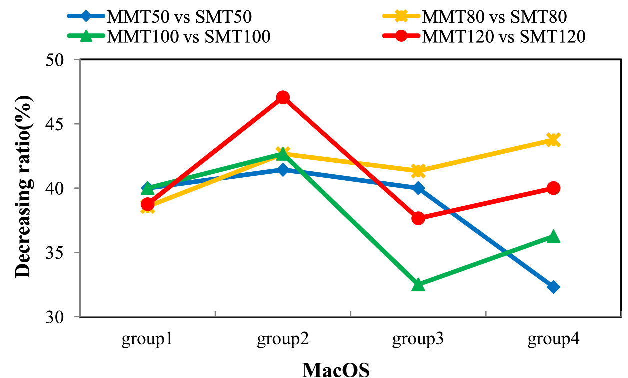
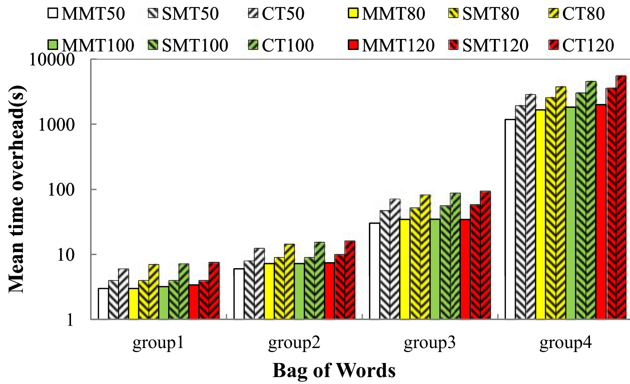
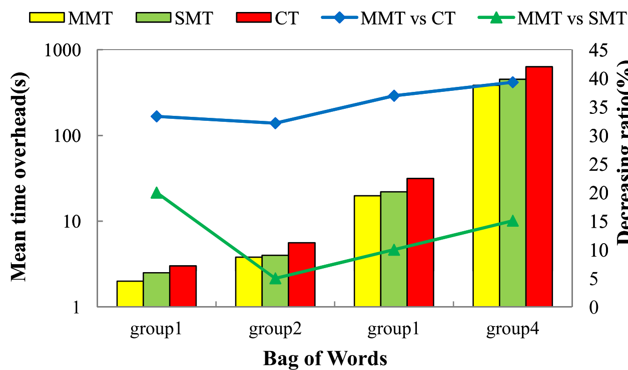


Fig. 27. The decreasing ratio of mean time overheads of whole insertion operations using the trace of MacOS in rate = 1.1.



(a) Rate = 1.1.



(b) Rate = 2.04.

Fig. 28. The mean time overheads of whole insertion operations using the trace of Bag of Words.

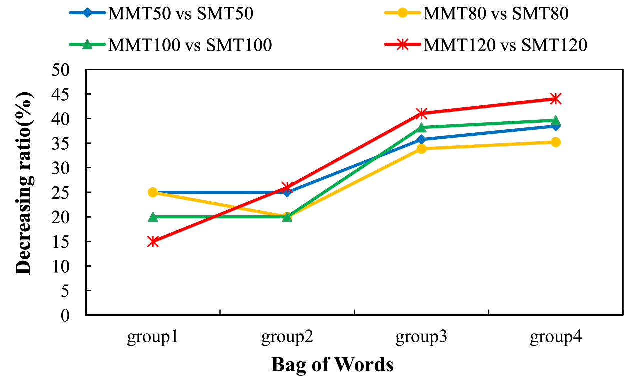


Fig. 29. The decreasing ratio of mean time overheads of whole insertion operations using the trace of Bag of Words in rate = 1.1.

Fig. 28 shows the mean time overheads of whole insertion operations of MinCounter and ChunkStash by using the trace of Bag of Words. Compared with ChunkStash, single-thread MinCounter significantly reduces more than 35 percent time overheads in rate $= 1.1$ and about 25 percent in rate $= 2.04$. Multi-threads MinCounter further decreases more than 15 percent time overheads in tate $= 1.1$ (in Fig. 29) and about 10 percent in rate $= 2.04$ (in Fig. 28b).

5 Related Work

Cuckoo hashing [38] is an efficient variation of the multi-choice hashing scheme. In the cuckoo hashing scheme, an item can be placed in one of multi-candidate buckets of hash tables. When there is no empty bucket for an item at any of its candidates, the item can kick out the item existing in one of the buckets, instead of causing insertion failure and overflow (e.g., using the linked lists). The kicked-out item operates in the same way, and so forth iteratively, until all items occupy one of buckets during insertion operations. Some researches discuss the case of multiple selectable choices of $d > 2$ as hypergraphs [54], [55].

Existing work about cuckoo hashing [56], [57] presents the theoretical analysis results. Simple properties of branching processes are analyzed in bipartite graph [56]. A study by M. Mitzenmacher judiciously answers the open questions to cuckoo hashing [57].

Further variations of cuckoo hashing are considered in [35], [37]. For handling hash collisions without breadth-first search analysis, the study [35] by A. Frieze et al. presents a more efficient method called random-walk. This method randomly selects one of candidate buckets for the inserted item, if there is no vacancy among its possible locations. In order to dramatically reduce the probability that a failure occurs during the insertion of an item, they propose a more robust hashing, that is cuckoo hashing with a small constant-sized *stash*, and demonstrate that the size of *stash* is equivalent to only three or four items and it has tremendous improvements through analysis and simulations [37]. Necklace [58] is an efficient variation of cuckoo hashing scheme to mitigate hash collisions in insertion operations.

In practice, a variant of cuckoo hashing [59] takes advantage of the similar technology with MinCounter. However, it is a wear-leveling technique for cuckoo hashing, and only focuses on the memory wear performance. MinCounter records the current kicking-out position and avoids self-kicking-out.

Cuckoo hashing has been widely used in real-world applications [36], [40], [60]. Cuckoo hashing is amenable to a hardware implementation, such as in a router. To avoid a large number of items to be moved during insertion operations causing expensive overhead in a hardware implementation, at most one item to be moved is acceptable [60]. ChunkStash improves advantages of a variant of cuckoo hashing to resolve hash collisions, which indexes chunk metadata using an in-memory hash table [36]. NEST [40] leverages cuckoo-driven hashing to achieve load balance.

Compared with the conference version [61], this paper presents the architecture of the storage system to elaborate the platform of MinCounter scheme. We show practical operations of MinCounter to support item insertion, item query, item deletion, and corresponding pseudocodes. We significantly extend the evaluation and discussion of the system implementation by adding a new trace, i.e., MacOS, and a new evaluation metric, i.e., the mean time overheads. Moreover, in order to support multiple writes and reads in a concurrent manner, we improve conventional single-thread cuckoo hashing scheme by fine-grained lock mechanism, as well as discribe two challenges for concurrent access.

6 Conclusion

In order to alleviate the occurrence of endless loops, this paper proposed a novel concurrent cuckoo hashing scheme, named MinCounter, for large-scale cloud computing systems. The

MinCounter has the contributions to three main challenges in hash-based data structures, i.e., intensive data migration, low space utilization and high insertion latency. MinCounter takes advantage of "cold" buckets to alleviate hash collisions and decrease insertion latency. MinCounter optimizes the performance for cloud servers, and enhances the quality of experience for cloud users. Compared with state-of-the-art work, we leverage extensive experiments and real-world traces to demonstrate the benefits of MinCounter. We have released the source code of MinCounter for public use in Github at https://github.com/syy804123097/MinCounter.

ACKNOWLEDGMENTS

This work is supported by National Key Research and Development Project under Grant 2016YFB1000202. This is an extended version of our manuscript published in the Proceedings of the International Conference on Massive Storage Systems and Technology (MSST), 2015. Yu Hua is the corresponding author.

REFERENCES

[1] V. Turner, J. Gantz, D. Reinsel, and S. Minton, "The digital universe of opportunities: Rich data and the increasing value of the Internet of Things," *International Data Corporation, White Paper, IDC_1672*, 2014.

[2] M. Armbrust, et al., "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[3] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud computing for everyone," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, pp. 16:1–16:14.

[4] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi, "Query optimization for massively parallel data processing," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, pp. 12:1–12:13.

[5] Y. Hua, B. Xiao, and J. Wang, "Br-tree: A scalable prototype for supporting multiple queries of multidimensional data," *IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1585–1598, Dec. 2009.

[6] C. Wang, K. Ren, S. Yu, and K. M. R. Urs, "Achieving usable and privacy-assured similarity search over outsourced cloud data," in *Proc. IEEE Conf. Comput. Commun.*, 2012, pp. 451–459.

[7] Q. Liu, C. C. Tan, J. Wu, and G. Wang, "Efficient information retrieval for ranked queries in cost-effective cloud environments," in *Proc. IEEE Conf. Comput. Commun.*, 2012, pp. 2581–2585.

[8] A. Crainiceanu, "Bloofi: A hierarchical bloom filter index with applications to distributed data provenance," in *Proc. Int. Workshop Cloud Intell.*, 2013, pp. 4:1–4:8.

[9] N. Bruno, S. Jain, and J. Zhou, "Continuous cloud-scale query optimization and processing," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 961–972, 2013

[10] Y. Hua, B. Xiao, B. Veeravalli, and D. Feng, "Locality-sensitive bloom filter for approximate membership query," *IEEE Trans. Comput.*, vol. 61, no. 6, pp. 817–830, Jun. 2012.

[11] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 222–233, Jan. 2014.

[12] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou, "Fuzzy keyword search over encrypted data in cloud computing," in *Proc. IEEE Conf. Comput. Commun.*, 2010, pp. 1–5.

[13] Y. Hua, B. Xiao, D. Feng, and B. Yu, "Bounded LSH for similarity search in peer-to-peer file systems," in *Proc. 37th Int. Conf. Parallel Process.*, 2008, pp. 644–651.

[14] M. Björkqvist, L. Y. Chen, M. Vukolić, and X. Zhang, "Minimizing retrieval latency for content cloud," in *Proc. IEEE Conf. Comput. Commun.*, 2011, pp. 1080–1088.

[15] W. W. Peterson, "Addressing for random-access storage," *IBM J. Res. Develop.*, vol. RD-1, no. 2, pp. 130–146, 1957.

[16] J. I. Munro and P. Celis, "Techniques for collision resolution in hash tables with open addressing," in *Proc. ACM Fall Joint Comput. Conf.*, 1986, pp. 601–610.

[17] J. Maddison, "Fast lookup in hash tables with direct rehashing," *Comput. J.*, vol. C-23, no. 2, pp. 188–190, 1980.

[18] R. L. Rivest, "Optimal arrangement of keys in a hash table," *J. ACM*, vol. ACM-25, no. 2, pp. 200–209, 1978.

[19] R. P. Brent, "Reducing the retrieval time of scatter storage techniques," *Commun. ACM*, vol. ACM-16, no. 2, pp. 105–109, 1973.

[20] J. Kelsey and B. Schneier, "Second preimages on n-bit hash functions for much less than 2^n work," in *Proc. 24th Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2005, pp. 474–490.

[21] X. Wang, H. Yu, and Y. L. Yin, "Efficient collision search attacks on SHA-0," in *Proc. 25th Annu. Int. Conf. Advances Cryptology*, 2005, pp. 1–16.

[22] I. Koltsidas and S. D. Viglas, "Flashing up the storage layer," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 514–525, 2008.

[23] J. S. Vitter, "Analysis of the search performance of coalesced hashing," *J. ACM*, vol. 30, no. 2, pp. 231–258, 1983.

[24] W.-C. Chen and J. S. Vitter, "Analysis of new variants of coalesced hashing," *Trans. Database Syst.*, vol. 9, no. 4, pp. 616–645, 1984.

[25] J. S. Vitter and W.-C. Chen, *The Design and Analysis of Coalesced Hashing*. Oxford, U.K.: Oxford Univ. Press, Inc., 1987.

[26] R. Pagh and F. F. Rodler, *Cuckoo Hashing*. Berlin, Germany: Springer, 2001.

[27] M. Zukowski, S. Héman, and P. Boncz, "Architecture-conscious hashing," in *Proc. Workshop Data Manage. New Hardware*, 2006, Art. no. 6.

[28] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proc. 14th Annu. ACM Symp. Parallel Algorithms Architectures*, 2002, pp. 73–82.

[29] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *J. ACM*, vol. 53, no. 3, pp. 379–405, 2006.

[30] J. Triplett, P. E. McKenney, and J. Walpole, "Scalable concurrent hash tables via relativistic programming," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 3, pp. 102–109, 2010.

[31] J. Triplett, P. E. McKenney, and J. Walpole, "Resizable, scalable, concurrent hash tables via relativistic programming," in *Proc. USENIX Annu. Tech. Conf.*, 2011 p. 11.

[32] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *Proc. 10th USENIX Networked Syst. Des. Implementation*, 2013, pp. 385–398.

[33] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent Cuckoo hashing," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 27:1–27:14.

[34] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space efficient hash tables with worst case constant access time," in *Proc. 20th Annu. Symp. Theoretical Aspects Comput. Sci.*, 2003, pp. 271–282.

[35] A. Frieze, P. Melsted, and M. Mitzenmacher, "An analysis of random-walk cuckoo hashing," *Approximation, Randomization, and Combinatorial Optimization: Algorithms and Techniques*. Berlin, Germany: Springer, 2009, pp. 490–503.

[36] B. K. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up inline storage deduplication using flash memory," in *Proc. USENIX Annu. Tech. Conf.*, 2010, p. 16.

[37] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM J. Comput.*, vol. 39, no. 4, pp. 1543–1561, 2009.

[38] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[39] R. Pagh, "On the cell probe complexity of membership and perfect hashing," in *Proc. 33rd Annu. ACM Symp. Theory Comput.*, 2001, pp. 425–432.

[40] Y. Hua, B. Xiao, and X. Liu, "NEST: Locality-aware approximate query service for cloud computing," in *Proc. IEEE INFOCOM*, 2013, pp. 1303–1311.

[41] Y. Hua, B. Xiao, X. Liu, and D. Feng, "The design and implementations of locality-aware approximate queries in hybrid storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 11, pp. 3194–3207, Nov. 2015.

[42] Y. Hua, H. Jiang, and D. Feng, "Fast: Near real-time searchable data analytics for the cloud," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2014, pp. 754–765.

[43] B. Debnath, S. Sengupta, and J. Li, "Flashstore: High throughput persistent key-value store," *Proc. VLDB Endowment*, vol. 3, no. 1/2, pp. 1414–1425, 2010.

[44] M. Herlihy and N. Shavit, "The art of multiprocessor programming," in *Proc. 25th Annu. ACM Symp. Principles Distrib. Comput.*, vol. 6, pp. 1–2, 2006.

[45] B. Fitzpatrick and A. Vorobey, "Memcached: A distributed memory object caching system," 2011, Available at: http://memcached.org

[46] C. Pheatt, "Intel threading building blocks," *J. Comput. Sci. Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[47] P. E. McKenney, et al., "Read-copy update," in *Proc. AUUG Conf.*, 2001, p. 175.

[48] R. Kutzelnigg, "Bipartite random graphs and cuckoo hashing," in *Proc. 4th Colloquium Mathematics Comput. Sci.*, 2006, pp. 403–406.

[49] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in) security of hash-based oblivious RAM and a new balancing scheme," in *Proc. 23rd Annu. ACM-SIAM Symp. Discrete Algorithms*, 2012, pp. 143–156.

[50] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious RAM simulation," in *Proc. 23rd Annu. ACM-SIAM Symp. Discrete Algorithms*, 2012, pp. 157–167.

[51] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *Proc. IEEE 17th Int. Symp. High Performance Comput. Architecture*, 2011, pp. 169–180.

[52] R. Baeza-Yates and G. H. Gonnet, *Handbook of Algorithms and Data Structures*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 1991.

[53] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok, "Generating realistic datasets for deduplication analysis," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 261–272.

[54] N. Fountoulakis, K. Panagiotou, and A. Steger, "On the insertion time of cuckoo hashing," *SIAM J. Comput.*, vol. 42, no. 6, pp. 2156–2181, 2013.

[55] N. Fountoulakis, M. Khosla, and K. Panagiotou, "The multiple-orientability thresholds for random hypergraphs," in *Proc. 23rd Annu. ACM-SIAM Symp. Discrete Algorithms*, 2011, pp. 1222–1236.

[56] L. Devroye and P. Morin, "Cuckoo hashing: Further analysis," *Inf. Process. Lett.*, vol. 86, no. 4, pp. 215–219, 2003.

[57] M. Mitzenmacher, "Some open questions related to cuckoo hashing," in *Proc. 17th Annu. Eur. Symp.*, 2009, pp. 1–10.

[58] Q. Li, Y. Hua, W. He, D. Feng, Z. Nie, and Y. Sun, "Necklace: An efficient cuckoo hashing scheme for cloud storage services," in *Proc. 22nd Int. Symp. Quality Service*, 2014, pp. 153–158.

[59] D. Eppstein, M. T. Goodrich, M. Mitzenmacher, and P. Pszona, "Wear minimization for cuckoo hashing: How not to throw a lot of eggs into one basket," in *Proc. 13th Int. Symp. Exp. Algorithms*, 2014, pp. 162–173.

[60] A. Kirsch and M. Mitzenmacher, "The power of one move: Hashing schemes for hardware," *IEEE/ACM Trans. Netw.*, vol. 18, no. 6, pp. 1752–1765, Dec. 2010.

[61] Y. Sun, Y. Hua, D. Feng, L. Yang, P. Zuo, and S. Cao, "MinCounter: An efficient cuckoo hashing scheme for cloud storage systems," in *Proc. 31st Symp. Mass Storage Syst. Technologies*, 2015, pp. 1–7.

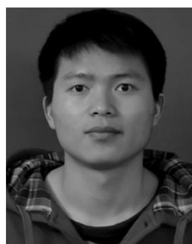
Yuanyuan Sun received the BE degree in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 2014. She is working toward the PhD degree majoring in computer science and technology at HUST. Her current research interests include algorithms of hashing, data analytics, gene data analysis, and energy efficiency in data center networks.

Yu Hua received the BE and PhD degrees in computer science from Wuhan University, China, in 2001 and 2005, respectively. He is currently a professor at Huazhong University of Science and Technology, China. His research interests include computer architecture, cloud computing, and network storage. He has more than 80 papers to his credit in major journals and international conferences including the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, *Proceedings of the IEEE*, USENIX ATC, USENIX FAST, INFOCOM, SC, ICDCS, ICPP, MSST, and MASCOTS. He has been on the organizing and program committees of multiple international conferences, including INFOCOM, ICDCS, ICPP, ICNP, MSST, RTSS, and IWQoS. He is a senior member of the IEEE and CCF, a member of ACM, and USENIX.

Dan Feng received the BE, ME, and PhD degrees in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 1991, 1994, and 1997, respectively. She is a professor and vice dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She is a member of the IEEE and the ACM.

Ling Yang received the BE degree in software engineering from China University of Geosciences, Wuhan, China, in 2014. She is working toward the master's degree majoring in computer science and technology at Huazhong University of Science and Technology, China. Her current interests include power management in datacenters, data analytics, and algorithms of hashing.

Pengfei Zuo received the BE degree in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 2014. He is currently working toward the PhD degree majoring in computer science and technology at HUST. His current research interests include data deduplication, security and privacy issues in cloud storage, key-value store, and content based similarity detection.



Shunde Cao received the BE degree in computer science and technology from Wuhan University of Science and Technology (WUST), China, in 2014. He is working toward the master's degree majoring in computer science and technology at Huazhong University of Science and Technology, China. His research interests include data deduplication, content-based similarity detection, and key-value store.

Yuncheng Guo received the BE degree in information security from Huazhong University of Science and Technology (HUST), China, in 2015. He is working toward the master's degree majoring in computer science and technology at HUST. His current research interests include algorithms of hashing, data analytics, and gene data analysis.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.