MinCounter: An Efficient Cuckoo Hashing Scheme for Cloud Storage Systems

Yuanyuan Sun   Yu Hua   Dan Feng   Ling Yang   Pengfei Zuo   Shunde Cao

Wuhan National Lab for Optoelectronics, School of Computer
Huazhong University of Science and Technology, Wuhan, China
{sunyuanyuan, csyhua, dfeng, yling, pfzuo, csd}@hust.edu.cn
Corresponding Author: Yu Hua (csyhua@hust.edu.cn)

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, which 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 this problem, 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. MinCounter selects the "cold" (infrequently accessed) buckets to handle hash collisions rather than random buckets. MinCounter has the salient features of offering efficient insertion and query services and obtaining performance improvements in 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 two real-world traces. Extensive experimental results demonstrate the efficacy and efficiency of MinCounter.

I. 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 data we create and copy annually will reach 44 ZettaBytes in 2020 [1]. 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.

In order to support real-time queries, hashing-based data structures have been widely used in constructing the index due to constant-scale addressing complexity and thus fast query response. Unfortunately, hashing-based data structures cause low space utilization, as well as high-latency risk of handling hashing collisions. Unlike conventional hash tables, cuckoo hashing [2] addresses hashing collisions via simple "kicking-out" operation (i.e., flat addressing), rather than searching the linked lists (i.e., hierarchical addressing). 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 the items among hash tables. Such scheme ensures a more even distribution of data items among hash tables than using only one hash function. Due to the salient feature of flat addressing with constant-scale complexity, cuckoo hashing needs to probe the hashed buckets only once and obtains the query results. Even in the worst case, the cuckoo hashing guarantees constant-scale query time complexity and constant amortized time for insertion and deletion process. Cuckoo hashing thus improves space utilization without the increase of query latency.

In practice, due to the essential property of hash functions, the cuckoo hashing fails to fully avoid the hash collisions. Existing work to handle hash collisions mainly leverages random-walk approach [3], [4], 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. 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 [5]. The kicking-out operation needs to migrate a selected item to its other candidates 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 only after experiencing 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 endless loops.
- **Space Inefficiency**. When data collisions occur during the insertion process, we cannot predict in advance whether there exists data in the slot we choose randomly. Because of the unpredictable random selection of traditional cuckoo hashing, there is always some small

but practically significant probability that during the data insertion, none of the d buckets are or can easily be made empty to hold the data, causing a insertion failure [6]. In this case, an expensive rehashing of all items in the hash tables or extra space to store insertion failure items is required in conventional cuckoo hashing. Nevertheless, it leads to space inefficiency of hash tables and significantly impact on the average performance.

- **High Insertion Latency**. The cuckoo hashing schemes based on random-walk approach migrate items randomly among their d candidate positions [4]. This exacerbates the unpredictability of hash addressing when all candidates are occupied. The random selection in kicking-out operations may cause repetitions and infinite loops [7], which results in high latency of insertion operations.

In order to address these challenges, we propose a Min-Counter 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. And empty bucket can be chosen to store the item. In order to record kicking-out times occurring at the bucket, we allocate a counter for each bucket. If all buckets are not empty, the item selects the bucket with the minimum counter to kick out the occupied item to reduce or avoid endless loop. The rationale of MinCounter is avoiding busy routes and seeking the empty buckets as quickly as possible. Moreover, in order to reduce the frequency of rehashing, we temporarily store the items with insertion failure into in-memory cache, rather than directly rehash the entire structure.

The rest of this paper is organized as follows. Section II shows the research backgrounds. Section III presents the Min-Counter design and practical operations. Section IV illustrates the performance evaluation and Section V shows the related work. Finally, we conclude our paper in Section VI.

II. BACKGROUNDS

In this section, we present the research backgrounds of the cuckoo hashing scheme.

The cuckoo hashing was described in [8] as a dynamization of a static dictionary. The dictionary leverages two hash tables, T_1 and T_2, instead of only one, and two hash functions h_1, h_2: $U \rightarrow \{0, \ldots, r-1\}$, where r is the length of each hash table. Each item $x \in S$ is stored in one of the buckets $h_1(x)$ in T_1 and the buckets $h_2(x)$ in T_2. For a general lookup, we only check whether the queried item is in one of its candidate buckets. For data insertion, in order to handle hash collisions, cuckoo hashing uses kicking-out operations among the buckets [9]. The cuckoo hashing makes use of $d \geq 2$ hash tables. Each hash table has an independent hash function, and each item has d candidate buckets to alleviate hash collisions.

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 the existing items of 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.

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.

Figure 1 shows an example (i.e., $d = 2$) to illustrate the practical operations of standard cuckoo hashing. We use arrows to show possible destinations for moving items as shown in Figure 1(a). If item x is inserted into hash tables, we first check whether there exists any empty bucket of all 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. Figure 1(b) 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 Figure 1(c), 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 determine the occurrence of endless loops, which causes the entire structure re-construction. The theoretical analysis of MaxLoop has been shown in Section 4.1 of [9]. Moreover, due to essential property of random choice in hash functions, the hash collision can not be fully avoided, but significantly alleviated [4].
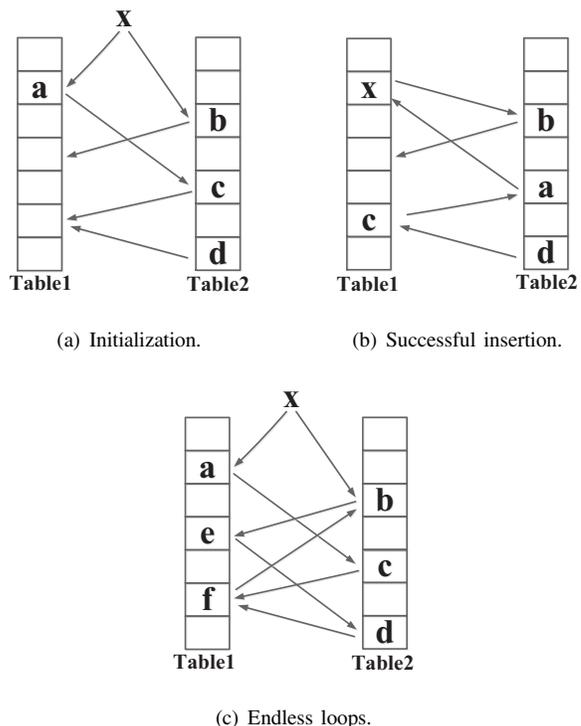

(a) Initialization. (b) Successful insertion.

(c) Endless loops.

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

III. DESIGN AND IMPLEMENTATION DETAILS

In this section, we present a cost-effective insertion scheme, called MinCounter, to insert items in the cuckoo hashing. MinCounter selects the "cold" buckets to alleviate the occurrence of endless loops in the data insertion when hash collisions occur.

It is easy to understand the case of $d = 2$ in the cuckoo hashing. Each bucket contains one item. When an item is kicked out, it has to choose to kick out the item in the other candidate bucket due to avoiding self-kicking-out [9], [10]. In real-world applications, the cases of $d \geq 3$ 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 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 an item x is inserted into the hash table, we look up all of the d candidate buckets in order to observe whether there is an empty one to insert. If no, we have to replace one with the item x. Thus, we choose to use random-walk cuckoo hashing [3], [4] 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 bucket that was replaced 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. Furthermore, we identify that the frequency of kicking-out in each bucket of hash tables is not uniform. Some buckets receives more kicking-out operations than others. 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. We take advantage of the characteristic to propose an effective scheme, called MinCounter, to deal with hash collisions.

A. The Data Structure of MinCounter

MinCounter is a multi-choice hashing scheme to place items as shown in Figure 2. It uses cuckoo hashing to allow each item to have d candidate buckets. An item can choose an empty bucket to locate. The used hash functions are standard (uniform and random). We allocate a counter for each bucket to record the kicking-out times occurring at the bucket. If no empty buckets are available, the item needs to select one with the minimum counter to kick out the occupied item to reduce or avoid endless loop. We temporarily store insertion-failure items into cache rather than rehash the structure immediately to reduce the failure probability of entire item insertion.

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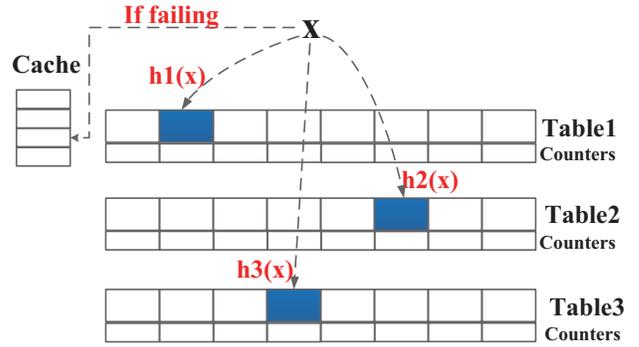

Fig. 2. The data structure of MinCounter.

B. 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 of 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.

As shown in Figure 3, we take $d = 3$ to give an example. When the item x is inserted into hash tables, we first check the buckets of $h_1(x)$, $h_2(x)$, $h_3(x)$ in each hash table respectively to find an empty bucket. Each candidate bucket of x is occupied by a, b, c respectively (Figure 3(a)). 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 (Figure 3(b)). 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.

C. Handling Endless Loops

MinCounter allows items to be inserted into hash tables to improve the storage space efficiency, but fails to fully address hash collisions. Like ChunkStash [6], we leverage an extra space to temporarily store the data that cause hash collisions rather than rehash the structure. For a query, we need to check both the hash tables and the stash to guarantee the query accuracy.

IV. PERFORMANCE EVALUATION

In this section, we evaluate the performance of the designed MinCounter scheme by implementing a prototype under a large-scale cloud computing environment. The evaluation metrics mainly include the utilization ratio of hash tables when insertion failures occur, and total kicking-out times after completing entire item insertion. The utilization ratio of hash tables is interpreted as the proportion of the occupied buckets to all buckets of hash tables when insertion failure occurs.

A. Experimental Setup

We implement the MinCounter scheme in a large-scale cloud computing environment. The prototype is developed un-
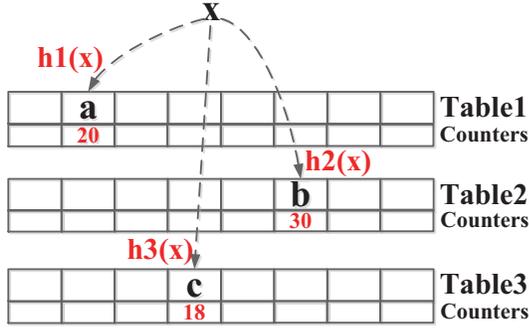
(a) Initial status.
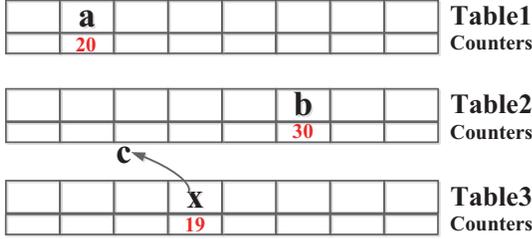
(b) Running status.

Fig. 3. The standard cuckoo hashing table structure.

der the Linux kernel 2.6.18 environment and we implement all functional components of MinCounter in the user space. Each server is equipped with Intel 2.4GHz quad-core CPU, 16GB DRAM, 500GB disk. In order to demonstrate the efficiency and effectiveness of the proposed MinCounter scheme, we use 2 datasets and 2 initial rates, i.e., 1.1 and 2.04, in hash tables. The initial rate means the multiple we set based on sizes of dataset to create hash tables. When the rate is 1.1, hash collisions often occur, and hardly when the rate is 2.04.

1) The Dataset of Randomly Generated Numbers: In order to comprehensively examine the performance of the proposed MinCounter scheme, we first present the theoretical analysis results in terms of randomness by using an open-source random number generator to generate integer datasets as shown in Table I. We further present the performance improvements by using the trace from real-world applications.

TABLE I
THE DATASET OF RANDOMLY GENERATED NUMBERS.

Groups	Range	Size
group1	0-100000000	1314404
group2	0-80000000	2017180
group3	0-100000000	2517415
group4	0-200000000	4960610
group5	0-500000000	7666282
group6	0-500000000	11184784

2) The Real-world Trace: Bag of Words: This trace contains four text collections in the form of bags-of-words. For each text collection, *D* is the number of documents, *W* is the number of words in the vocabulary, and *N* is the total number of words in the collection. The details are shown in Table II. We take advantage of the union of *docID* and *wordID* as keys of items to be inserted into hash tables.

TABLE II
THE BAG OF WORDS TRACES.

Groups	Text collections	D	W	N
group1	KOS blog entries	3430	6906	353160
group2	NIFS full papers	1500	12419	746316
group3	Enron Emails	39861	28102	3710420
group4	NYTimes news articles	300000	102660	4960610

B. 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 several threshold, 50, 80, 100 and 120, to process the following experiments.

C. The Counter Size Settings

First, we need to consider the bits per counter of per bucket in hash tables for space savings. We randomly choose two groups of data from 2 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, Figure 4 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.
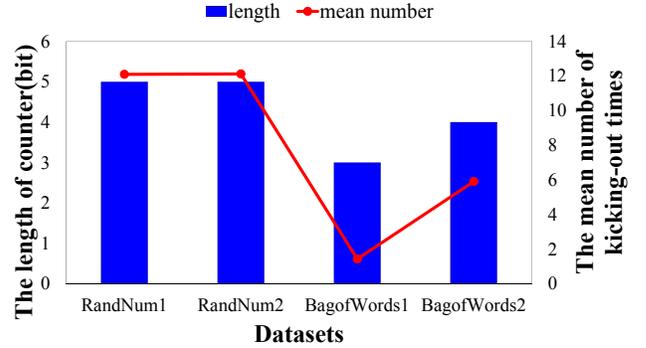

Fig. 4. The distribution of values of counters.

D. Experimental Results

We show advantages of MinCounter over RandomWalk [4] and ChunkStash [7] by comparing their experimental results in terms of utilization ratio of hash tables and total kicking-out times during insertion operations. The thresholds of kicking-out times are 50, 80, 100 and 120. *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 *MT*80 is the MinCounter scheme with the threshold of 80.

1) The Results of using Randomly Generated Numbers: Figure 5 shows the utilization ratio of cuckoo hash tables when insertion failure first occurs by using the trace of randomly generated numbers. We observe that the average utilization ratio of MinCounter is 75% in the trace of randomly generated numbers, which is higher than the percentage of 70% in RandomWalk. Compared with the RandomWalk scheme, MinCounter obtains on average 5% utilization ratio promotion. RandomWalk scheme needs 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 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 systems performance by improving the utilization of hash table and decreasing the rehash probability.

We examine the total kicking-out times of MinCounter and ChunkStash by using the metric of total kicking-out numbers in the trace of randomly generated numbers as shown in Figure 6. When insertion failure occurs, we store the item into a temporary small additional constant-size cache like ChunkStash rather than rehash tables immediately. This may cause slight extra space overhead, but obtain the benefits of reducing the failure probability. Compared with ChunkStash, MinCouter significantly cuts down over 10% total kicking-out numbers in rate = 1.1 (in Figure 7) and on average 37% in rate = 2.04 (in Figure 6(b)). MinCounter enhances the experiences of cloud users through decreasing total kicking-out times.
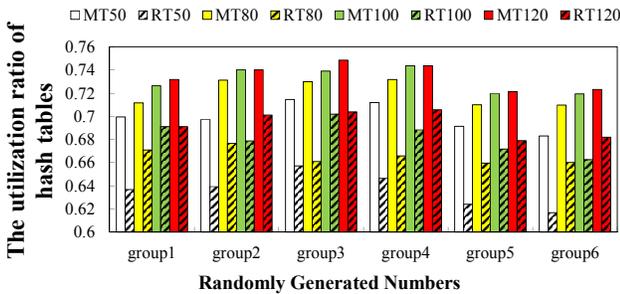

Fig. 5. The utilization ratio of cuckoo hash tables using the trace of randomly generated numbers.

2) The Results of using Bag of Words Trace: Figure 8 illustrates the utilization ratio of cuckoo hash tables when using the bag of words trace. We observe that MinCounter obtains on average 5% utilization improvement, compared with RandomWalk scheme, while the average utilization ratio of MinCounter is 88% in the Bag of Words trace, and 83% in Random-Walk scheme. Figure 9 shows the total kicking-out numbers in the bag of words trace. Compared with
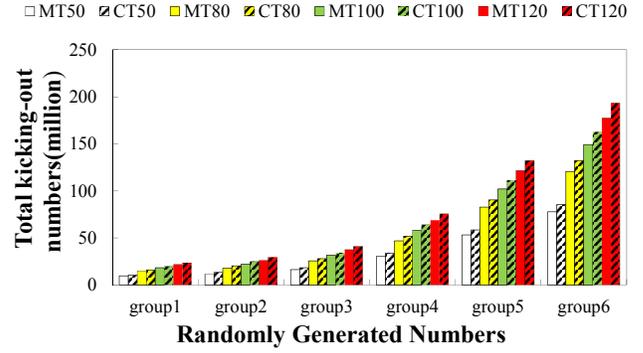
(a) Rate = 1.1.
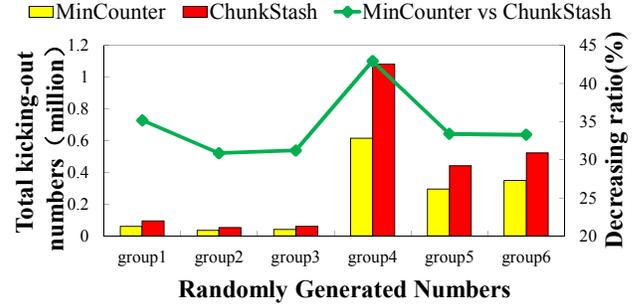
(b) Rate = 2.04.

Fig. 6. The total kicking-out numbers of whole insertion operations using the trace of randomly generated numbers.
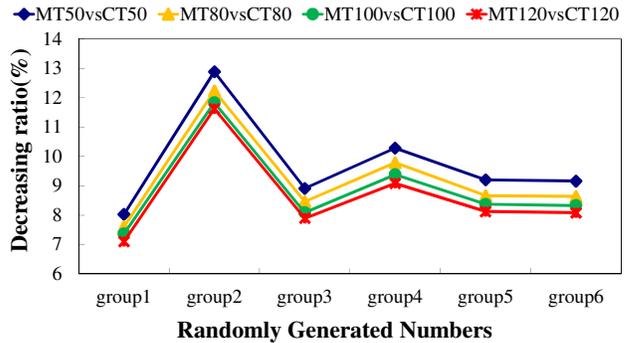

Fig. 7. The decreasing ratio of total kicking-out times of MinCounter using the trace of randomly generated numbers in rate = 1.1.

ChunkStash, MinCouter significantly reduces almost 50% total kicking-out numbers in rate = 1.1 (in Figure 10), and on average 30% in rate = 2.04 (in Figure 9(b)).

E. Summary

Experimental results demonstrate MinCounter has the advantages in terms of the utilization ratio of hash tables and the total kicking-out times. MinCounter can efficiently improve the utilization of cuckoo hash tables and decrease the rehash probability to optimize the cloud computing systems performance. Meanwhile, it enhances experience of cloud users through decreasing the total kicking-out times.

V. RELATED WORK

Cuckoo hashing [9] is an efficient variation of the multi-choice hashing scheme. In the cuckoo hashing scheme, an
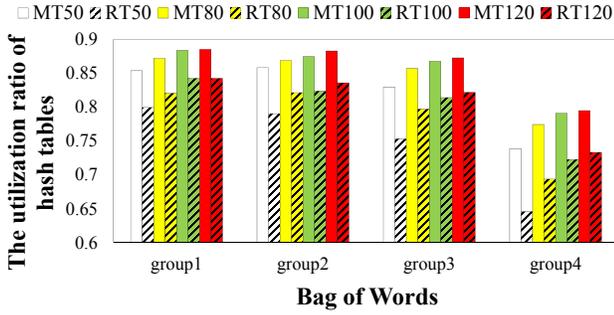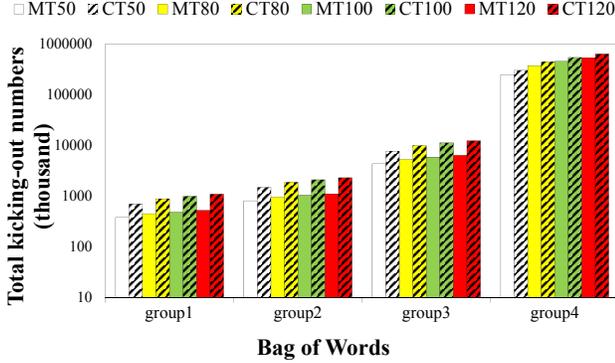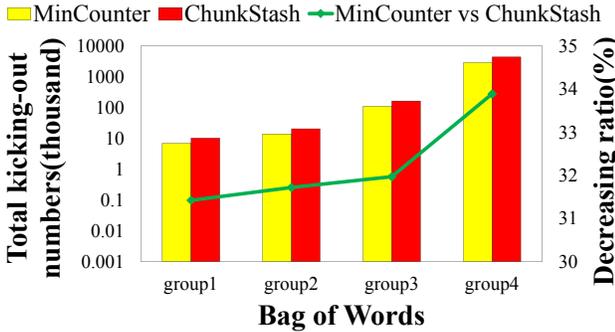
Fig. 8. The utilization ratio of cuckoo hash tables using the trace of Bag of Words.

(a) Rate = 1.1.

(b) Rate = 2.04.

Fig. 9. The total kicking-out numbers of whole insertion operations using the trace of Bag of Words.

Fig. 10. The decreasing ratio of total kicking-out times of MinCounter using the trace of Bag of Words in rate = 1.1.

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 [11], [12].

Existing work about cuckoo hashing [13], [14] presents the theoretical analysis results. Simple properties of branching processes are analyzed in bipartite graph [13]. A study by M. Mitzenmacher judiciously answers the open questions to cuckoo hashing [14].

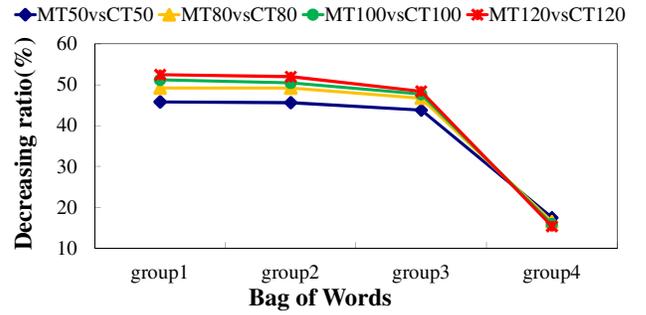Further variations of cuckoo hashing are considered in Re-f. [4], [6], [15]. For handling hash collisions without breadth-first search analysis, the study by A. Frieze et al. presents a more efficient method called random-walk. This method randomly selects one of candidate buckets for the inserted item, if there is no vacancy among its possible locations [4]. 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 analytically and through simulations [6]. Necklace [15] is an efficient variation of cuckoo hashing scheme to mitigate hash collisions in insertion operations.

Cuckoo hashing has been widely used in real-world applications [7], [16], [17]. 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 [16]. ChunkStash [7] improves advantages of a variant of cuckoo hashing to resolve hash collisions, which indexes chunk metadata using an in-memory hash table. NEST [17] leverages cuckoo-driven hashing to achieve load balance.

VI. CONCLUSION

In order to alleviate the occurrence of endless loops, this paper proposed a novel 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.

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] R. Pagh and F. F. Rodler, *Cuckoo hashing*. Springer Berlin Heidelberg, 2001.

[3] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space efficient hash tables with worst case constant access time," *Proc. STACS*, pp. 271–282, 2003.

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

[5] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing.," *Proc. USENIX NSDI*, vol. 13, pp. 385–398, 2013.

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

[7] B. K. Debnath, S. Sengupta, and J. Li, "Chunkstash: Speeding up inline storage deduplication using flash memory.," *Proc. USENIX Annual Technical Conference*, 2010.

[8] R. Pagh, "On the cell probe complexity of membership and perfect hashing," *Proc. ACM symposium on Theory of computing*, pp. 425–432, 2001.

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

[10] R. Kutzelnigg, "Bipartite random graphs and cuckoo hashing," *Proc. DMTCS*, 2006.

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

[12] N. Fountoulakis, M. Khosla, and K. Panagiotou, "The multiple-orientability thresholds for random hypergraphs," *Proc. ACM-SIAM symposium on Discrete Algorithms*, pp. 1222–1236, 2011.

[13] L. Devroye and P. Morin, "Cuckoo hashing: further analysis," *Information Processing Letters*, vol. 86, no. 4, pp. 215–219, 2003.

[14] M. Mitzenmacher, "Some open questions related to cuckoo hashing," *Proc. ESA*, pp. 1–10, 2009.

[15] Q. Li, Y. Hua, W. He, D. Feng, Z. Nie, and Y. Sun, "Necklace: An efficient cuckoo hashing scheme for cloud storage services," *Proceedings of IEEE/ACM International Symposium on Quality of Service(IWQoS)*, pp. 50–55, 2014.

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

[17] Y. Hua, B. Xiao, and X. Liu, "Nest: Locality-aware approximate query service for cloud computing," *Proceedings of the 32nd IEEE International Conference on Computer Communications(INFOCOM)*, pp. 1327–1335, 2013.