SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems

Yuanyuan Sun and Yu Hua, ***Huazhong University of Science and Technology;***
Song Jiang, ***University of Texas, Arlington;*** **Qiuyu Li, Shunde Cao, and Pengfei Zuo,**
Huazhong University of Science and Technology

https://www.usenix.org/conference/atc17/technical-sessions/presentation/sun

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Open access to the Proceedings of the
2017 USENIX Annual Technical Conference
is sponsored by USENIX.

SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems

Yuanyuan Sun, Yu Hua*, Song Jiang†, Qiuyu Li, Shunde Cao, Pengfei Zuo
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
†*University of Texas, Arlington*
**Corresponding Author: Yu Hua (csyhua@hust.edu.cn)*

Abstract

Fast query services are important to improve overall performance of large-scale storage systems when handling a large number of files. Open-addressing cuckoo hash schemes have been widely used to support query services due to the salient features of simplicity and ease of use. Conventional schemes are unfortunately inadequate to address the potential problem of having endless loops during item insertion, which degrades the query performance. To address the problem, we propose a cost-efficient cuckoo hashing scheme, named SmartCuckoo. The idea behind SmartCuckoo is to represent the hashing relationship as a directed pseudoforest and use it to track item placements for accurately predetermining the occurrence of endless loop. SmartCuckoo can efficiently predetermine insertion failures without paying a high cost of carrying out step-by-step probing. We have implemented SmartCuckoo in a large-scale cloud storage system. Extensive evaluations using three real-world traces and the YCSB benchmark demonstrate the efficiency and efficacy of SmartCuckoo. We have released the source code of SmartCuckoo for public use.

1 Introduction

Efficient query services are critical to cloud storage systems at various scales, especially when they process a massive amount of data. According to the report of International Data Corporation (IDC) in 2014, the amount of information created and replicated will reach 44 Zettabytes in 2020 [49], and nearly 50% of cloud-based services will rely on data in storage systems [21]. Moreover, in a recent survey of 1,780 data center managers in 26 countries, over 36% of respondents face two critical challenges, which are efficiently supporting a flood of emerging applications and handling the rapidly increasing data management complexity [2]. This reflects a reality that we are generating and accessing much more data than ever and this trend continues at an accelerated pace. This data volume explosion has imposed great challenge on storage systems, particularly on their support on efficient data query services. In various computing facilities, from small hand-held devices to large-scale data centers, people are collecting and analyzing ever-greater amounts of data. Users routinely generate queries on hundreds of Gigabytes of data stored on their local disks or cloud storage systems. Commercial companies generally handle Terabytes and even Petabytes of data each day [6, 10, 54].

It is becoming increasingly challenging for cloud storage systems to quickly serve queries, which often consumes substantial resources to support query-related operations [51]. Cloud management systems usually demand the support of low-latency and high-throughput queries [7]. In order to address these challenges, query services have received many attentions, such as top-k query processing [23, 34, 37], security model for file system search in multi-user environments [9], metadata query on file systems [26, 38], Web search using multi-cores in mobile computing [27], graph query processing with abstraction refinement [52], energy saving for online search in datacenters [50], efficient querying of compressed network payloads [48], reining the latency in tail queries [22], and scaling search data structures for asynchronized concurrency [12].

An efficient hashing scheme is important for improving performance of query services. A hash table needs to map keys to values and supports constant-time access in a real-time manner. Hash functions are used to locate a key to a unique bucket. While keys may be hashed to the same bucket (the occurrence of hash collisions), lookup latency can become higher with more collisions in a bucket. Cuckoo hashing [43] is a fast and simple hash structure with the constant-time worst-case lookup ($O(\ln \frac{1}{\varepsilon})$) and consumes $(1+\varepsilon)n$ memory consumption, where ε is a small constant. Due to its desirable property of open addressing and its support of low lookup latency, cuckoo hashing has been widely used in real-world cloud applications [13, 24, 28, 35, 46]. Cuckoo hashing uses multiple (usually two in practice) hash functions for resolving hash collisions and recursively kicks items

out of their current positions. Unlike standard hashing schemes that provide only one position for placing an item, cuckoo hashing provides multiple (usually two) possible positions to reduce the probability of hash collisions. To determine the presence of an item, the cuckoo hashing will probe up to two positions, and the worst-case lookup time is a constant.

However, the cuckoo hashing suffers from substantial performance penalty due to the occurrence of endless loops. Currently, the existence of endless loop is detected only after a potentially large number of step-by-step kick-out operations. A search for insertion position in an endless loop turns out to be fruitless effort. In order to deliver high performance and improve lookup efficiency, we need to address two major challenges.

Substantial Resources Consumption. In an endless loop, an insertion failure can only be known after a large number of in-memory operations, and the penalty can substantially compromise the efficiency of cuckoo hashing schemes. When a hash table is substantially occupied, many such loops occur, which can substantially increase insertion costs.

Nondeterministic Performance. Cuckoo hashing essentially takes a random walk to find a vacant bucket for inserting an item since the knowledge on the path for this walk is not obtained in advance [19, 33]. This scheme does not leverage the dependencies among the positions of items. Before walking sufficiently long on the path, one can hardly know if an endless loop exists. Moreover, the cuckoo hashing provides multiple choices of possible positions for item insertion. The kick-out operations need to be completed in an online manner.

Existing schemes have not effectively addressed the two challenges. For example, *MemC*3 [16] uses a large kick-out threshold as its default kick-out upper bound, which possibly leads to excessive memory accesses and reduced performance. *Cuckoo hashing with a stash (CHS)* [29] addresses the problem of endless loops by using an auxiliary data structure as a stash. The items that introduce hash collisions are moved into the stash. For a lookup request, *CHS* has to check both the original hash table and the stash, which increases the lookup latency. Furthermore, *bucketized cuckoo hash table (BCHT)* [15, 44, 45, 55] allocates two to eight slots into a bucket, in which each slot can store an item, to mitigate the chance of endless loops, which however results in poor lookup performance due to multiple probes.

In order to clearly demonstrate the performance impact of endless loops, we measure the loop ratio in *CHS* with three real-world traces (experiment details can be found in Section 4.1). The loop ratio is defined as the percentage of failed insertions due to the existence of endless loops among all item insertions. Figure 1 shows that more than 25% insertions walk into endless loops at a load factor of 0.9 for the hash table, which leads to substantial time and space overheads for carrying out rehashing operations and allocating additional storage space, respectively. The load factor is the ratio of the number of occupancies to that of total buckets in the hash table. To this end, we need to mitigate and even eliminate the occurrence of endless loops to reduce the space and time overheads.

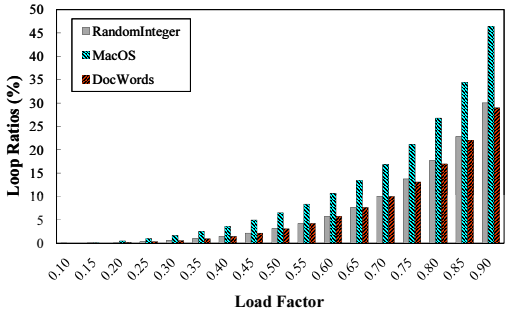


Figure 1: The loop ratios in CHS with three traces.

In this paper, we propose a cost-effective cuckoo hashing scheme, named SmartCuckoo. SmartCuckoo allows flexible configurations and fast item lookup, and achieves much improved insertion performance. Our work aims to answer the following questions: *(1) Is there a vacant bucket available for an item to be inserted before starting kick-outs on a path? (2) How to guarantee efficiency of the insertion and lookup using a space-efficient and lightweight auxiliary structure?*

SmartCuckoo leverages a fast and cost-efficient predetermination operation to help avoid unnecessary kick-out process due to endless loops. This operation runs before item insertion starts by using an auxiliary structure. Moreover, an insertion failure can be identified without any kick-out operations and manual setting of iteration thresholds. SmartCuckoo can avoid the endless loops of cuckoo hashing and deliver high performance. This paper has made the following contributions.

Cost-effective Hashing Scheme. SmartCuckoo retains cuckoo hashing's advantage of space efficiency and constant-time queries via open addressing. In the meantime, SmartCuckoo is able to predetermine insertion failures without the need of carrying out continuous kick-out operations, thus significantly reducing the insertion latency and supporting fast lookup services.

Deterministic Performance. Conventional cuckoo hashing schemes take many kick-out operations in their insertion operations before detecting endless loops and consuming substantial system resources. By categorizing insertions into different cases, SmartCuckoo helps predetermine the result of a new insertion to avoid the endless loop by leveraging the concept of maximal

pseudoforest. SmartCuckoo hence makes insertion performance more predictable.

System Implementations and Public use. We have implemented all the components and algorithms of SmartCuckoo and released the source code for public use[1]. In order to evaluate the performance of SmartCuckoo, we compared it with state-of-the-art schemes, including *CHS* [29] as the evaluation baseline, *libcuckoo* [36], as well as *BCHT* [15].

2 Background

This section presents the research background of the cuckoo hashing and the pseudoforest theory. As a cost-efficient hashing scheme, the cuckoo hashing utilizes open addressing to improve lookup efficiency for large datasets. In the cuckoo hashing, the relationship between items and buckets can be described by a cuckoo graph, where each edge represents a hashed item and its two vertices represent the positions of the hashed item in the hash directory.

Cuckoo hashing does not require dynamic memory allocation, which can be efficiently exploited to provide real-time query services. The cuckoo hashing is able to support fast queries with worst-case constant-scale lookup time due to its addressing open to multiple positions for one item.

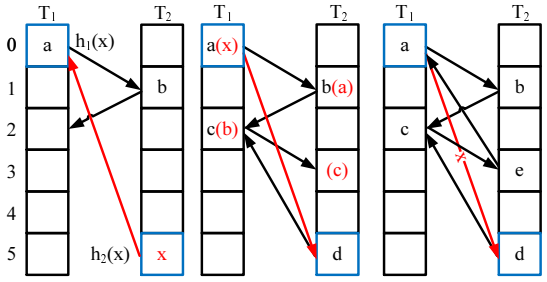
2.1 The Cuckoo Hashing

Cuckoo hashing [42, 43] is a dynamization of a static dictionary. The hashing scheme resolves hash collisions in a multi-hash manner.

Definition 1 ***Conventional Cuckoo Hashing.*** *Let d be the number of hash tables, and S be the set of keys. For the case of $d = 2$, conventional cuckoo hashing uses two hash tables, T_1 and T_2 with a size of n, and two hash functions h_1, h_2: $S \to \{0, \ldots, n-1\}$. A Key $k \in S$ can be inserted in either Slot $h_1(k)$ of T_1 or Slot $h_2(k)$ of T_2, but not in both. The two hash functions h_i (i = 1 or 2) are independent and uniformly distributed.*

As shown in Figure 2, we use an example to illustrate the insertion process in the conventional cuckoo hashing. In the cuckoo graph, the start point of an edge represents the actual storage position of an item and the end point is the backup position. For example, the bucket $T_2[1]$ storing Item b is the backup position of Item a. We intend to insert the item x, which has two candidate positions $T_1[0]$ and $T_2[5]$ (blue buckets). There exist three cases about inserting Item x:

- Two items (a and b) are initially located in the hash tables as shown in Figure 2(a). When inserting Item x, one of x's two candidate positions (i.e., $T_2[5]$) is empty. Item x is then placed in $T_2[5]$ and an edge is added pointing to the backup position ($T_1[0]$).
- Items c and d are inserted into hash tables before Item x, as shown in Figure 2(b). Two candidate positions of Item x are occupied by Items a and d respectively. We have to kick out one of occupied items (e.g., a) to accommodate Item x. The kicked-out item (a) is then inserted into its backup position ($T_2[1]$). This procedure is performed iteratively until a vacant bucket ($T_2[3]$) is found in the hash tables. The kick-out path is $x \to a \to b \to c$.
- Item e is inserted into the hash tables before Item x, as shown in Figure 2(c). There is no vacant bucket available to store Item x even after substantial kick-out operations, which results in an endless loop. The cuckoo hashing has to carry out a rehashing operation [43].



(a) Vacant bucket(s). (b) Finite kicks. (c) An endless loop.

Figure 2: The conventional cuckoo hashing data structure.

A lookup operation probes two candidate positions of an item. Buckets $T_1[0]$ and $T_2[5]$ will be probed for searching Item x, as shown in Figure 2. If the queried item is stored in the hash tables, it must be in one of its two candidate positions.

When all candidate buckets of a newly inserted item have been occupied, the cuckoo hashing needs to iteratively carry out kick-out operations to identify a vacant bucket, which possibly causes an endless loop and an insertion failure, until a kick-out path is tried and a threshold of steps on the path is reached without locating a vacant position.

2.2 Pseudoforest Theory

A pseudoforest is an undirected graph in the graph theory and each of maximally connected components, named **subgraphs**, has at most one cycle [5,20]. In other words,

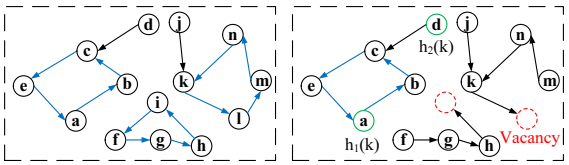
[1] https://github.com/syy804123097/SmartCuckoo.

it is an undirected graph in which each subgraph has no more edges than vertices. In a pseudoforest, two cycles composed of consecutive edges share no vertices with each other, and cannot be linked to each other by a path of consecutive edges.

In order to show the difference of actual and backup positions of items, we take into consideration the direction of kick-out operations. In a directed graph, each edge is directed from one of its endpoints to the other. Each bucket in the hash tables stores at most one item, and thus each vertex in a directed pseudoforest has an outdegree of at most one. If a subgraph contains a vertex whose outdegree is zero, it does not contain a cycle and the vertex corresponds to a vacant slot. Otherwise, it contains a cycle and any insertion into the subgraph will walk into an endless loop [31].

Definition 2 ***Maximal Directed Pseudoforest.*** *A maximal directed pseudoforest is a directed graph in which each vertex has an outdegree of exactly one.*

We name a subgraph whose number of vertices are equal to its number of edges a **maximal subgraph**. A maximal subgraph contains a cycle. Any subgraph in a maximal directed pseudoforest is a maximal subgraph. Figure 3(a) shows an example of a maximal directed pseudoforest. There are three maximal subgraphs in a maximal directed pseudoforest. In contrast, a non-maximal directed pseudoforest has at least one non-maximal subgraph, namely, has at least one vertex whose outdegree is zero. As illustrated in Figure 3(b), the non-maximal directed pseudoforest has three subgraphs, two of which do not have any cycles. It can be transformed to a maximal directed pseudoforest by connecting any vertex whose outdegree is zero (the dotted circles in Figure 3(b)) with any other vertex in the graph by adding a new edge.



(a) Maximal. (b) Non-Maximal.

Figure 3: The Directed Pseudoforest.

We consider the cuckoo graph as a directed pseudoforest. Each vertex of the pseudoforest corresponds to a bucket of the hash tables and each edge corresponds to an item between two candidate positions of the item. An inserted item hence produces an edge. According to the property, a maximal subgraph has no room to admit a new edge, which eventually causes an endless loop when the directed edges are traversed. Such an endless loop will not be encountered in a non-maximal subgraph, which does not contain a cycle.

3 The SmartCuckoo Design

As a cost-efficient variant of cuckoo hashing, SmartCuckoo maintains high lookup efficiency and improves the insertion performance by avoiding unnecessary kick-out operations. It classifies item insertions into three cases and leverages a directed pseudoforest to represent hashing relationship, which is used to track item placements for accurately predicting the occurrence of endless loops. Conventional cuckoo hashing chooses one of the candidate positions for an item's placement without considering whether it would walk into an endless loop. Our design increases insertion efficiency by tracking status of subgraphs to predict the insertion walk outcome. Hence, SmartCuckoo intelligently selects insertion positions for the item to be inserted. In addition, we also illustrate the execution of operations in SmartCuckoo, including item insertion and deletion.

3.1 The Directed Pseudoforest Subgraph

Inserted items in cuckoo hashing form a *cuckoo graph*. We represent the cuckoo graph as a directed pseudoforest, which can reveal the path, consisting of directed edges, of kick-out operations for insertion. Hence, the directed graph can be used to track and tell endless loops in advance to avoid them.

Successful item insertion depends on finding a vacant bucket for storage. To this end, one of candidate buckets of an item to be inserted must belong to a subgraph containing one vertex whose outdegree is zero, corresponding to a vacant slot. Hence, detecting vacancies in a subgraph is crucial in the insertion operation of cuckoo hashing. Knowing the path of a sequence of kick-out operations for an item's insertion before the insertion is carried out will help to identify and avoid an endless loop. In our design, we characterize the cuckoo hashing as a directed graph, in which a bucket is represented as a vertex and an item is represented as an edge between two candidate positions of an item. SmartCuckoo stores at most one item in each bucket, and each item has a unique backup position. Accordingly, each edge has a start point representing the actual storage position of the item and an end point representing the backup one. In the directed graph, each vertex corresponds to a bucket and each edge corresponds to an item. Because items stored in a hash table are always not more than the buckets, the number of vertices is not smaller than that of edges in the directed graph. Therefore, there is at most one cycle existing in a subgraph. Hence, according to the

property of the directed pseudoforest, the directed graph used to characterize item placements in SmartCuckoo is a directed pseudoforest.

Furthermore, we have the following observation. When inserting a new item into a non-maximal directed subgraph of a pseudoforest, it will be stored in one of its candidate buckets, and then one kicked-out item will be stored in the vacant bucket corresponding to the last vertex of a directed cuckoo path. If one attempts to insert the item into a maximal directed pseudoforest, an endless loop will inevitably occur. Each vertex in a directed pseudoforest has an outdegree of one, except those with an outdegree of zero representing vacant buckets located at the ends of the directed paths in the non-maximal subgraphs. In a maximal directed pseudoforest, each vertex has an outdegree of one and no vertex can be the destination on the path of kick-out operations to store the item for insertion. That is, an endless loop is encountered.

The observation inspires us to design a strategy on the selection of a path leading to a vacant position for item insertion. Vertices of ourdegree of zero, which represent vacant positions (buckets) in the directed pseudoforest, are produced by prior item insertions. To reach a vacant vertex in a directed pseudoforest for inserting an item, at least one of the item's candidate buckets must be in a subgraph containing a vacant position. Figure 3(b) illustrates the process of inserting Item *k*. Its two candidate positions are currently occupied by Items *a* and *d* (green vertices) and are in a subgraph without vacant positions. Its insertion would encounter an endless loop and fail, though there exist two vacancies (red vertices) in the pseudoforest. Because only non-maximal subgraphs contain vacant positions, the success of an insertion of an item relies on whether at least one of its candidate positions is in a non-maximal subgraph.

New item insertions can be classified into three cases, i.e., $\mathbf{v+2}$, $\mathbf{v+1}$, and $\mathbf{v+0}$. As each item is represented as an edge in the pseudoforest, different placements of the item will increase the graph's vertex count differently (by two, one, or zero).

3.2 Three Cases of Item Insertions

In the implementation of conventional cuckoo hashing, an insertion failure is not known until a kick-out path is tried and a threshold of steps on the path is reached without locating a vacant position. The lack of a priori knowledge in the traditional implementations often leads to walking into endless loops with substantial time and resources spent on fruitless tries. To obtain the knowledge on endless loops in SmartCuckoo, we classify item insertions according to the number of additional vertices added to the directed pseudoforest.

In a directed pseudoforest, each edge corresponds to an inserted item, and each vertex corresponds to a bucket. Hence, for each item to be inserted into the hash tables, the number of edges is incremented by one. However, the increase of vertex count (v) has three cases, namely, the cases of $v+0$, $v+1$, and $v+2$. In the last two cases, the new item can be successfully inserted, which will be explained. Here we first discuss the status of the directed pseudoforest in the case of $v+0$.

3.2.1 The Case of $v+0$

When inserting an item without increasing vertex count, two vertices corresponding to two candidate buckets of the item should have existed in the directed pseudoforest, which leads to five possible scenarios, as illustrated in Figure 4.

- Two candidate buckets of Item x_1, shown as blue buckets in Figure 4(a), exist in the same non-maximal directed subgraph A. Either bucket can be selected to have a successful insertion as the kick-out operations will always reach a vacant position in the subgraph. As shown in Figure 4(a), Item x_1 is directly inserted into Bucket $T_2[3]$ and creates a new edge from Bucket $T_2[3]$ to Bucket $T_1[0]$, which is the backup position of Item x_1. After the insertion of Item x_1, the original non-maximal directed subgraph A is transformed into a maximal directed subgraph A', which does not have a vacant position to admit a new item.

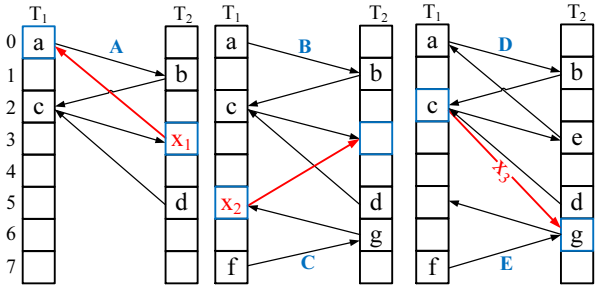
- Two candidate buckets of Item x_2 are in two different non-maximal directed subgraphs B and C, respectively, as shown in Figure 4(b). In this scenario, the insertion operation will also be a success, because each of two non-maximal directed subgraphs offers a vacant bucket. Item x_2 is located in Bucket $T_1[5]$ and constructs a new directed edge from Bucket $T_1[5]$ to Bucket $T_2[3]$ in the directed pseudoforest, which merges the two subgraphs, B and C, into a new non-maximal directly subgraph (BC) with one vacant vertex ($T_2[3]$).

- One candidate bucket of Item x_3 is in the non-maximal directed subgraph E and the other is in the maximal directed subgraph D, as shown in Figure 4(c). If the item enters the hash table from Bucket $T_1[2]$, an endless loop is encountered in the maximal directed subgraph D and unnecessary kick-out operations are carried out. However, if Item x_3 enters the hash table at Bucket $T_2[6]$, the item insertion will be a success after a number of kick-out operations (simply kicking out Item g to Bucket $T_1[5]$ in the example shown in Figure 5(a)).

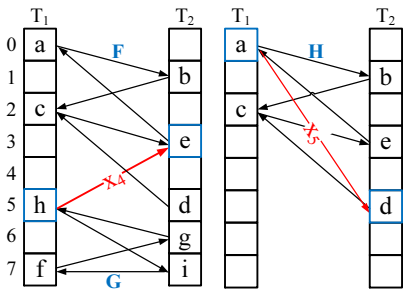
Accordingly, two subgraphs D and E are merged into a new maximal directed subgraph (DE), which does not have any vacant buckets.

- Two candidate buckets of Item x_4 are separated into two maximal directed subgraphs (F and G), as shown in Figure 4(d). Because there doesn't exist any vacant buckets in any of the subgraphs, the insertion of the new item (x_4) will always walk into an endless loop, illustrated in Figure 5(b). This is the worst scenario for an insertion in conventional cuckoo hashing implementations.

- Two candidate buckets of Item x_5 are in the same maximal directed subgraph (H), as shown in Figure 4(e). Similar to the previous scenario, the insertion will turn out to a failure after numerous kick-outs in an endless loop, as shown in Figure 5(c).



(a) One non-maximal. (b) Two non-maximal. (c) One maximal and one non-maximal.

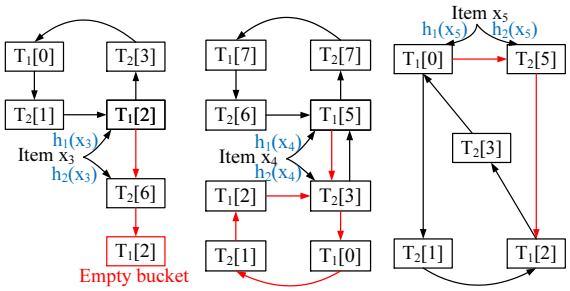


(d) Two maximal. (e) One maximal.

Figure 4: Five scenarios for Case $v+0$.

3.2.2 The Cases of $v+1$ and $v+2$

The $v+1$ represents the case where the number of vertices in the directed pseudoforest is increased by 1 after insertion of an item. As shown in Figure 6(a), in this case one of two candidate positions of Item x_6 corresponds to an existing vertex in the directed pseudoforest. The other will be a new vertex after the item's insertion. That is, this candidate bucket in the

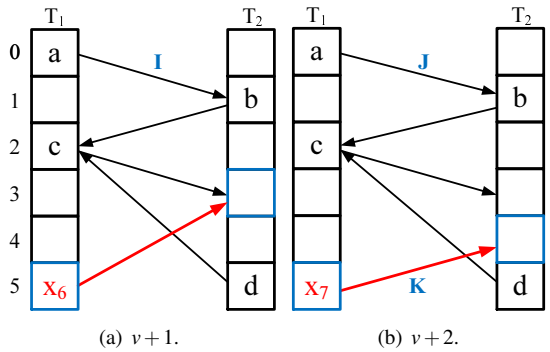


(a) One maximal and one non-maximal. (b) Two maximal. (c) One maximal.

Figure 5: Buckets that are accessed during kick-out operations.

hash table has not been represented by any vertices in the pseudoforest. Item x_6 is then placed in this position, and a new edge connecting the new vertex with the existing vertex is added into the subgraph I of the directed pseudoforest.

In the case of $v+2$, both candidate positions of an item to be inserted have not yet been represented by any vertices in the pseudoforest. Accordingly, they are unoccupied. The item can be inserted in any of the two available positions. Accordingly, two vertices, each corresponding to one of the positions, are added into the pseudoforest. Furthermore, an edge from the vertex corresponding to the position where the item is actually placed to the other corresponding to its backup position is also added. The two vertices and the new edge constitute a new subgraph (K), which is a non-maximal directed one. This case is illustrated in Figure 6(b), where the two vertices are Buckets $T_1[5]$ and $T_2[4]$, and the new edge is from Bucket $T_1[5]$ to Bucket $T_2[4]$ after Item x_7 is inserted at Bucket $T_1[5]$.



(a) $v+1$. (b) $v+2$.

Figure 6: The cases of $v+1$ and $v+2$.

3.3 Predetermination of An Endless Loop

According to Section 3.2, if we know in advance which case an item insertion belongs to, we can predetermine whether any of the item's candidate positions is on an endless loop. This is achieved by tracking status of subgraphs, which is either *maximal directed* or *non-maximal directed*. If a candidate position is in a maximal directed subgraph, it is on an endless loop. Otherwise, it is not on an endless loop.

During an item insertion operation, for each of its candidate positions, we need to find out which one or two subgraphs of a directed pseudoforest it belongs to. To this end, we apply the *Find* operation for a given candidate position to determine the subgraph it belongs to. In addition, if two candidate positions of an inserted item belong to two subgraphs, an edge will be introduced between the subgraphs and the two subgraphs need to be merged. To this end, a *Union* operation is required to merge them. To enable Find and Union operations in SmartCuckoo, we assign each subgraph a unique ID. Each member vertex of the subgraph records the ID in its corresponding bucket. When two subgraphs are merged into a new one, instead of exhaustively searching for member vertices of one or two of the original subgraphs on the hash tables to update their subgraph ID, we introduce trees of the IDs. In a tree for merged subgraphs, the IDs at the buckets representing the subgraphs before the merging are leaf nodes and the ID of the new subgraph is the parent. The new subgraph is likely to be merged again with another subgraph and has its parent. In the end, the ID at the root of the tree represents the subgraph merged from all the previous subgraphs.

In order to determine the status of a subgraph in the pseudoforest, we track its edge count and vertex count. A subgraph is a maximal directed one if its edge count is equal to its vertex count. In this case, the subgraph does not have any room to admit new edges. Otherwise, the edge count is smaller than vertex count, and the subgraph is a non-maximal directed one.

In summary, we can predetermine the outcome of an item insertion based on the statuses of related subgraphs in each of the three cases the insertion belongs to.

- **v + 2:** When the two candidate positions (a and b) of Item x have not yet been represented by any vertices in a directed pseudoforest, the insertion will create a new subgraph, which is non-maximal directed. Therefore, the new Item x can be successfully inserted. Moreover, the vertex count of the subgraph is 2, and the edge count is 1.

- **v + 1:** This case is detected after running $Find(a)$ and $Find(b)$ and finding out that one of the candidate positions corresponds to an existing vertex in a subgraph and the other has not yet been represented by any vertex. In this case, no matter which status the subgraph is on, the insertion will be a success due to the introduction of a new vertex. Both the vertex count and the edge count of the subgraph are increased by 1.

- **v + 0:** In this case, both candidate positions are vertices in subgraphs. To know the outcome of the insertion, we need to determine the status of the subgraphs. Only if at least one of the subgraphs is non-maximal, the insertion is a success. Otherwise, the insertion would fail after walking into an endless loop. The edge count of the corresponding subgraph is increased by 1.

3.4 Implementations of Operations

In the Section, we describe how two common hash table operations, namely insertion and deletion, are supported in SmartCuckoo, as its implementation of lookup operation is essentially the same as that in conventional cuckoo hashing.

3.4.1 Insertion

We use $B[*]$ to represent the item in the bucket. Algorithm 1 describes the steps involved in the insertion of Item x. First, we determine the case the insertion belongs to and increases corresponding vertex count (v), as described in Algorithm 2. The t value indicates one of the three cases ($v+2$, $v+1$, and $v+0$) for the insertion. In the cases of $v+1$ and $v+2$, Item x can be directly inserted, as described in Algorithm 3. If the insertion case is $v+0$, we use Algorithm 4 to determine which of the following five scenarios about corresponding subgraph(s) applies: (1) one non-maximal, (2) two non-maximal (Lines 4-6), (3) one non-maximal and one maximal (Lines 7-13), (4) two maximal, and (5) one maximal. SmartCuckoo avoids walking into a maximal directed subgraph. Due to no loops, SmartCuckoo is able to efficiently reduce the repetitions in one path, thus reducing insertion operation latency.

3.4.2 Deletion

An item can only be stored in one of the candidate positions of the hash tables. During the deletion operation, we only need to probe the candidate positions and, if found at one of the positions, remove it from the position (Lines 3-4). Deleting an item from the hash tables is equivalent to removal of an edge in the corresponding subgraph, which causes the subgraph to be separated into two subgraphs. We assign each

Algorithm 1 Insert(Item x)

```
1: a ← Hash1(x)
2: b ← Hash2(x) /*Two candidate positions of Item x*/
3: t ← Determine-v-add(a,b)
4: if t == v+2 then
5:     Assign a unique ID to the new subgraph
6:     Union(a,b)
7:     DirectInsert(x,a,b)
8:     Return Ture /*Finish the insertion*/
9: else if t == v+1 then
10:    Union(a,b)
11:    DirectInsert(x,a,b)
12:    Return True /*Finish the insertion*/
13: else
14:    InDirectInsert(x,a,b)
15: end if
```

Algorithm 2 Determine-v-add(Hash a, Hash b)

```
1: if neither a nor b have yet existed in the pseudoforest then
2:     Return v+2
3: else if both a and b have existed in the pseudoforest then
4:     Return v+0
5: else
6:     Return v+1
7: end if
```

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

```
1: /*a and b are two candidate positions of Item x*/
2: if B[a] is empty then
3:     B[a] ← x /*Insert Item x into the empty bucket*/
4: else
5:     B[b] ← x
6: end if
```

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

```
1: /*Determine type of the corresponding subgraphs*/
2: if one non-maximal then
3:     Kick-out(x,B[a])
4: else if two non-maximal then
5:     Kick-out(x,B[a])
6:     Union(a,b)
7: else if one non-maximal and one maximal then
8:     if the subgraph containing a is non-maximal then
9:         Kick-out(x,B[a])
10:    else
11:        Kick-out(x,B[b])
12:    end if
13:    Union(a,b)
14: else
15:    Rehash()
16: end if
```

of the two subgraphs a new ID, and update the IDs of each member vertex of the two subgraph in their corresponding buckets (Lines 5-6). In addition, the vertex count and edge count of the two subgraphs are updated. Algorithm 5 describes how the pseudoforest is maintained in the deletion of Item x.

4 Performance Evaluation

4.1 Experimental Setup

The server used in our experiments is equipped with an Intel 2.8GHz 16-core CPU, 12GB DDR3 RAM with a peak bandwidth of 32GB/s, and a 500GB hard disk. The L1 and L2 caches of the CPU are 32KB and 256KB, respectively. We use three traces (RandomInteger [40], MacOS [3, 47], and DocWords [4]), and the YCSB benchmark [11] to run the SmartCuckoo prototype in the Linux kernel 2.6.18 to evaluate its performance. In addition, SmartCuckoo is implemented based on *CHS*.

RandomInteger: We used C++'s STL Mersenne Twister random integer generator [40] to generate items, which are in the full 32-bit unsigned integer range and follow a pseudo-random uniform distribution.

MacOS: The trace was collected on a Mac OS X Snow Leopard server [3,47]. We use fingerprints of files as keys to generate insertion requests. The fingerprints are obtained by applying the MD5 function on the file contents.

DocWords: This trace includes five text collections in the form of bag-of-words [4]. It contains nearly 80 million items in total. We take advantage of the combination of its *DocID* and *WordID* as keys of items to be inserted into hash tables.

We compare SmartCuckoo with CHS (*cuckoo hashing with a stash*) [29] as the *Baseline*, *libcuckoo* [36], and *BCHT* [15] schemes. Specifically, for *BCHT*, we implemented its main components, including four slots in each bucket. For *libcuckoo*, we use its open-source C++ implementation [1], which is optimized to serve write-heavy workloads.

4.2 Results and Analysis

We present evaluation results of SmartCuckoo and compare them with those from the state-of-the-art cuckoo hash tables in terms of insertion throughput, lookup throughput, and the throughput of mixed operations.

4.2.1 Insertion Throughput

Figure 7 shows the insertion throughputs of *Baseline*, *libcuckoo*, *BCHT*, and the proposed SmartCuckoo with the RandomInteger workload. With the increase of the load factor, we observe that SmartCuckoo significantly increases insertion throughput over *Baseline* by 25% to 75%, *libcuckoo* by 65% to 75%, and over *BCHT* by

Algorithm 5 Deletion(Item x)

```
1: a ← Hash1(x)
2: b ← Hash2(x) /*Two candidate positions of Item x*/
3: if x == B[a] or x == B[b] then
4:     Delete x from the corresponding position
5:     Assign two unique IDs to two new subgraphs respectively
6:     Update subgraph ID
7:     Update vertex and edge count
8:     Return True
9: else
10:    Return False
11: end if
```

40% to 50%. Conventional cuckoo hash tables, including *Baseline*, *libcuckoo*, and *BCHT*, essentially take a random walk to find a vacant bucket for inserting an item without a priori knowledge on the path, which leads to unnecessary operations and the extended response time. In particular, in addition to the impact of endless loops, *libcuckoo* suffers from frequent use of locking for consistent synchronization in its support of concurrent accesses. *BCHT* uses multi-slot buckets to mitigate the occurrence of endless loops. However, it requires a search in at least one candidate bucket for an available slot to carry out an insertion, which compromises its insertion throughput. In contrast, SmartCuckoo classifies item insertions into three cases to predetermine outcome of an insertion, so that an insertion failure can be known without actually performing any kick-out operations to significantly save insertion time. This performance advantage is particularly large with a hash table of a high load factor.

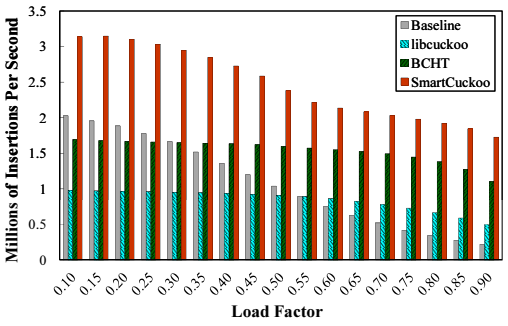


Figure 7: Insertion throughputs with RandomInteger.

Figure 8 shows insertion throughputs of the various cuckoo hash tables with the MacOS workload. Compared with conventional hash tables, SmartCuckoo obtains an average of 90% throughput improvement over *Baseline* at a load factor of 0.9, and 75% over *libcuckoo*, as well as 25% over *BCHT*.

Figure 9 illustrates the insertion throughputs with the

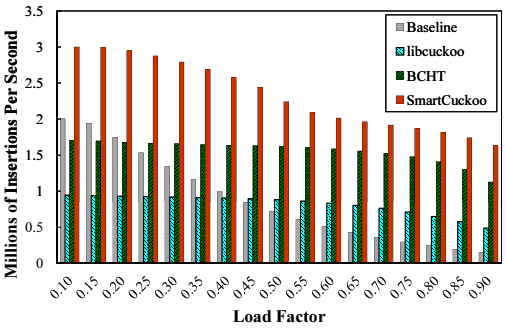


Figure 8: Insertion throughput with MacOS.

DocWords workload. With the increase of the load factor, SmartCuckoo increases insertion throughput over *Baseline* by 33% to 77%, *libcuckoo* by 60% to 75%, and over *BCHT* by 35% to 44%.

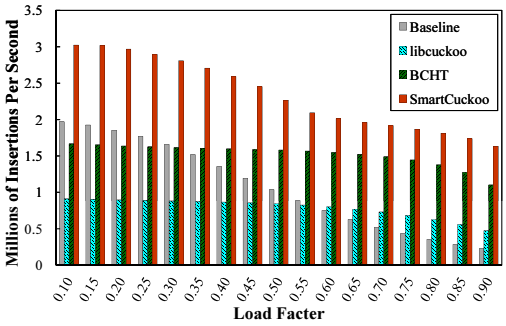


Figure 9: Insertion throughput with DocWords.

4.2.2 Lookup Throughput

In the evaluation of lookup performance of the four hash tables (*Baseline*, *libcuckoo*, *BCHT*, and SmartCuckoo), we generate the workload of all-lookup queries from each of the real-world traces. First, we extract lookup queries from a trace and use the remaining insertion and deletion queries in the trace to populate a hash table. Second, we selectively issue lookup queries, in the order of their appearance in the original trace, to the hash table. For a workload of lookup queries for only existent keys, we skip those for non-existent keys. For a workload of lookup queries for only non-existent keys, we skip those for existent keys. Each workload contains one million queries.

We examine the lookup throughputs of *Baseline*, *libcuckoo*, *BCHT*, and SmartCuckoo with the RandomInteger workload, which are shown in Figure 10. We observe that SmartCuckoo and *Baseline* achieve almost the same lookup throughput due to similar implementation of lookup operation. When all of the keys in the lookup queries are existent in the table,

SmartCuckoo improves the lookup throughputs by 30% and 5% over those of *libcuckoo* and *BCHT*, respectively. When none of the keys are in the table, all candidate positions (slots in *BCHT*) for a key have to be accessed. In particular, *BCHT* searches eight slots (four slots per bucket in the experiment setup) in two candidate buckets for each key, resulting in the reduced throughput.

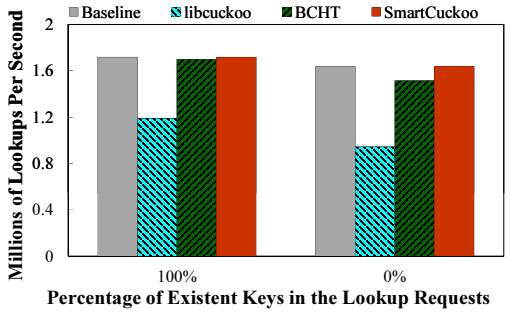


Figure 10: Lookup throughput with RandomInteger.

Figure 11 shows the lookup throughputs with the MacOS workload. Similarly, the throughput of SmartCuckoo is about 30% and 6% higher than that of *libcuckoo* and *BCHT*, respectively, with lookups of only existent keys. If all of the keys are non-existent, the improvements become 45% and 10%, respectively. Figure 12 shows the lookup throughputs with the DocWords workload, revealing similar performance trend.

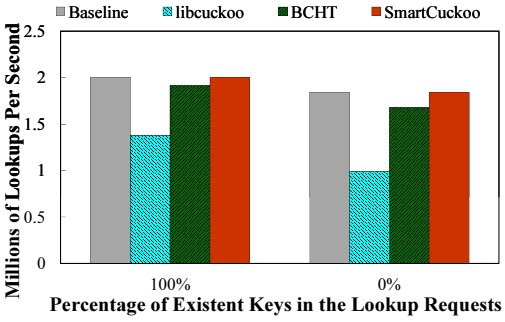


Figure 11: Lookup throughput with MacOS.

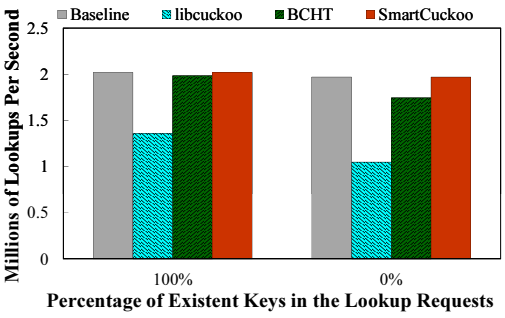


Figure 12: Lookup throughput with DocWords.

4.2.3 Throughput of Workload with Mixed Queries

We use YCSB [11] to generate five workloads, each with ten million key-value pairs, following the zipf distribution. Each key in the workloads is 16 bytes and each value is 32 bytes. The distributions of different types of queries in each workload are shown in Table 1.

Table 1: Distributions of different types of queries in each workload.

Workload	Insert	Lookup	Update
YCSB-1	100	0	0
YCSB-2	75	25	0
YCSB-3	50	50	0
YCSB-4	25	75	0
YCSB-5	0	95	5

Figure 13 shows the throughputs of SmartCuckoo and other hash tables in comparison for running each of the YCSB workloads. With the decrease of the percentage of insertions in the workloads, throughputs of all the cuckoo hash tables increase due to expensive kick-out operations during execution of insertion operations. With the workloads containing insert queries (the first four in Table 1), SmartCuckoo consistently produces higher throughput than the other three cuckoo hash tables, specifically by 25% to 70% than *Baseline*, by 25% to 55% than *libcuckoo*, and by 10% to 50% than *BCHT*. SmartCuckoo takes advantage of its ability of predetermining the occurrence of endless loops to avoid a potentially large number of step-by-step kick-out operations. The fifth workload does not have any insert queries. Instead, it does have a small percentage of update query, whose cost is similar to that of lookup if updating existent keys and is equivalent to that of insert if updating non-existent keys. Because most queries are for existent keys, SmartCuckoo and *Baseline* achieve nearly the same performance due to their similar implementation of lookup operation.

For each of the YCSB workloads that has 10 million key-value pairs with 16B keys and 32B values, the minimal space for holding the data in the cuckoo hash table is $(16+32)*10M = 458MB$. Any space additional to this minimal requirement to hold auxiliary data structure for higher performance is considered as the hash table's space overhead. With its use of a lightweight pseudoforest, SmartCuckoo has a space overhead of about 20% of the minimal requirement in the YCSB workloads. This is in line with that of the other three hash tables (*Baseline*, *libcuckoo*, and *BCHT*).

5 Related Work

Cuckoo Hashing Structures. SmartCuckoo is a variant of the cuckoo hashing, which supports fast and cost-

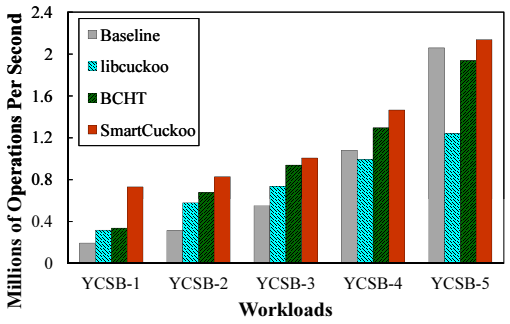


Figure 13: Throughput of mixed operations with YCSB.

efficient lookup operation. Cuckoo hashing [42, 43] is an open-addressing hashing scheme that provides each item with multiple candidate positions in the hash table. Studies of cuckoo hashing via the graph theory provide insightful understanding of cuckoo hashing's advantages and limitations [14, 32]. Cuckoo Graph is proposed to describe the hashing relationship of cuckoo hashing [32]. Cuckoo filter [17] uses the cuckoo hash table to enhance Counting Bloom filter [18] for supporting insertion and deletion operations with improved performance and space efficiency. Horton table [8] is an enhanced bucketized cuckoo hash table to reduce the number of CPU cache lines that are accessed in each lookup. In contrast, we investigate the characteristics of the directed cuckoo graph describing the kick-out behaviors. The proposed SmartCuckoo leverages the directed pseudoforest, a concept in graph theory, to track item placements in the hash table for predetermining occurrence of endless loops.

Content-based Search. NEST [24] uses cuckoo hashing to address the load imbalance issue in the traditional locality-sensitive hashing (LSH) and to support approximate queries. HCTrie [41] is a multi-dimensional structure for file search using scientific metadata in file systems, which supports a large number of dimensions. MinCounter [46] allocates a counter for each bucket to track kick-out times at the bucket, which mitigates the occurrence of endless loops during data insertion by selecting less used kick-out routes. SmartCuckoo aims at avoiding unnecessary kick-out operations due to endless loops in item insertion.

Searchable File Systems. Many efforts have been made to improving performance of large-scale searchable storage systems. Spyglass [34] is a file metadata search system, based on hierarchical partitioning of namespace organization, for high performance and scalability. Smartstore [23] reorganizes file metadata based on file semantic information for next-generation file systems. It provides efficient and scalable complex queries and enhances system scalability and functionality. Glance [25] is a just-in-time sampling-based system to provide accurate answers for aggregate and top-k queries without prior knowledge. Ceph [39, 53] uses dynamic subtree partitioning to support filename-based query as well as to avoid metadata-access hot spots. SmartCuckoo provides fast query services for cloud storage systems.

6 Conclusion and Future Work

Fast and cost-efficient query services are important to cloud storage systems. Due to the salient feature of open addressing, cuckoo hashing supports fast queries. However, it suffers from the problem of potential endless loops during item insertion. We propose a novel cost-efficient hashing scheme, named SmartCuckoo, for tracking item placements in the hash table. By representing the hashing relationship as a directed pseudoforest, SmartCuckoo can accurately predetermine the status of cuckoo operations and endless loops. We further avoid walking into an endless loop, which always belongs to a maximal subgraph in the pseudoforest. We use three real-world traces, i.e., RandomInteger, MacOS, and DocWords, and the YCSB benchmark to evaluate the performance of SmartCuckoo. Extensive experimental results demonstrate the advantages of SmartCuckoo over state-of-the-work schemes, including cuckoo hashing with a stash, *libcuckoo*, and *BCHT*.

SmartCuckoo currently addresses the issue of endless loop for cuckoo hash tables using two hash functions. It is well-recognized that using more than two hash functions would significantly increase operation complexity and is thus less used [13, 56]. A general and well-known approach is to reduce the number of hash functions to two using techniques such as double hashing [30]. As a future work, we plan to apply the approach of SmartCuckoo on hash tables using more than two hash functions. In addition, we will also study the use of SmartCuckoo in cuckoo hash tables with multiple slots in each bucket.

Acknowledgments

This work was supported by National Key Research and Development Program of China under Grant 2016YFB1000202 and State Key Laboratory of Computer Architecture under Grant CARCH201505. Song Jiang was supported by US National Science Foundation under CNS 1527076. The authors are grateful to anonymous reviewers and our shepherd, Rong Chen, for their constructive feedbacks and suggestions.

References

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

[2] Symantec. 2010 State of the Data Center Global Data. *http://www.symantec.com/content/en/us/about/media /pdfs/Symantec_DataCenter10_Report_Global.pdf* (Jan. 2010).

[3] Traces and Snapshots Public Archive. *http://tracer.filesystems.org* (July 2014).

[4] Bag-of-words data set. *http://archive.ics.uci.edu/ml/datasets /Bag+of+Words* (Mar. 2008).

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

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

[7] BELL, G., HEY, T., AND SZALAY, A. Beyond the Data Deluge. *Science 323*, 5919 (2009), 1297–1298.

[8] BRESLOW, A. D., ZHANG, D. P., GREATHOUSE, J. L., JAYASENA, N., TULLSEN, D. M., XU, L., CAVAZOS, J., ALVAREZ, M. A., MORALES, J. A., AGUILERA, P., ET AL. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. USENIX Association, pp. 281–294.

[9] BÜTTCHER, S., AND CLARKE, C. L. A Security Model for Full-Text File System Search in Multi-User Environments. In *Proc. FAST* (2005).

[10] BYKOV, S., GELLER, A., KLIOT, G., LARUS, J. R., PANDYA, R., AND THELIN, J. Orleans: Cloud Computing for Everyone. In *Proc. SOCC* (2011), ACM, p. 16.

[11] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proc. SoCC* (2010), ACM, pp. 143–154.

[12] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. *ACM SIGARCH Computer Architecture News 43*, 1 (2015), 631–644.

[13] DEBNATH, B. K., SENGUPTA, S., AND LI, J. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory. In *Proc. USENIX ATC* (2010).

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

[15] ERLINGSSON, U., MANASSE, M., AND MCSHERRY, F. A Cool and Practical Alternative to Traditional Hash Tables. In *Proc. WDAS* (2006).

[16] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. NSDI* (2013).

[17] FAN, B., ANDERSEN, D. G., KAMINSKY, M., AND MITZENMACHER, M. D. Cuckoo Filter: Practically Better Than Bloom. In *Proc. CoNext* (2014), ACM, pp. 75–88.

[18] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Z. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking (TON) 8*, 3 (2000), 281–293.

[19] FRIEZE, A., MELSTED, P., AND MITZENMACHER, M. An Analysis of Random-Walk Cuckoo Hashing. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 2009, pp. 490–503.

[20] GABOW, H. N., AND WESTERMANN, H. H. Forests, frames, and games: Algorithms for matroid sums and applications. *Algorithmica 7*, 1 (1992), 465–497.

[21] GANTZ, J., AND REINSEL, D. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future 2007* (2012), 1–16.

[22] HSU, C.-H., ZHANG, Y., LAURENZANO, M. A., MEISNER, D., WENISCH, T., MARS, J., TANG, L., AND DRESLINSKI, R. G. Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting. In *Proc. HPCA* (2015), IEEE, pp. 271–282.

[23] HUA, Y., JIANG, H., ZHU, Y., FENG, D., AND TIAN, L. Smartstore: A New Metadata Organization Paradigm with Semantic-Awareness for Next-Generation File Systems. In *Proc. SC* (2009), ACM.

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

[25] HUANG, H. H., ZHANG, N., WANG, W., DAS, G., AND SZALAY, A. S. Just-in-Time Analytics on Large File Systems. *IEEE Transactions on Computers 61*, 11 (2012), 1651–1664.

[26] HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARAYANAN, M., GANGER, G. R., RIEDEL, E., AND AILAMAKI, A. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *Proc. FAST* (2004), pp. 73–86.

[27] JANAPA REDDI, V., LEE, B. C., CHILIMBI, T., AND VAID, K. Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency. In *Proc. ISCA* (2010), pp. 314–325.

[28] KIRSCH, A., AND MITZENMACHER, M. The Power of Qne Move: Hashing Schemes for Hardware. *IEEE/ACM Transactions on Networking 18*, 6 (2010), 1752–1765.

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

[30] KNUTH, D. E. *The Art of Computer Programming: Sorting and Searching*, vol. 3. Pearson Education, 1998.

[31] KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. Efficient parallel algorithms for graph problems. *Algorithmica 5*, 1 (1990), 43–64.

[32] KUTZELNIGG, R. Bipartite Random Graphs and Cuckoo Hashing. In *Fourth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities* (2006), Discrete Mathematics and Theoretical Computer Science, pp. 403–406.

[33] LAM, H., LIU, Z., MITZENMACHER, M., SUN, X., AND WANG, Y. Information Dissemination via Random Walks in d-Dimensional Space. In *Proc. SODA* (2012), SIAM, pp. 1612–1622.

[34] LEUNG, A. W., SHAO, M., BISSON, T., PASUPATHY, S., AND MILLER, E. L. Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems. In *Proc. FAST* (2009), pp. 153–166.

[35] LI, Q., HUA, Y., HE, W., FENG, D., NIE, Z., AND SUN, Y. Necklace: An Efficient Cuckoo Hashing Scheme for Cloud Storage Services. In *Proc. IWQoS* (2014), IEEE, pp. 153–158.

[36] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proc. EuroSys* (2014), ACM.

[37] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZIS, G., AND CAMBLE, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proc. FAST* (2009), pp. 111–123.

[38] LIU, L., XU, L., WU, Y., YANG, G., AND GANGER, G. R. SmartScan: Efficient Metadata Crawl for Storage Management Metadata Querying in Large File Systems. *Parallel Data Laboratory* (2010), 1–17.

[39] MALTZAHN, C., MOLINA-ESTOLANO, E., KHURANA, A., NELSON, A. J., BRANDT, S. A., AND WEIL, S. Ceph as a scalable alternative to the Hadoop Distributed File System. *login: The USENIX Magazine 35*, 4 (2010), 38–49.

[40] MATSUMOTO, M., AND NISHIMURA, T. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8*, 1 (1998), 3–30.

[41] OHARA, Y. HCTrie: A Structure for Indexing Hundreds of Dimensions for Use in File Systems Search. In *Proc. MSST* (2013), IEEE, pp. 1–5.

[42] PAGH, R., AND RODLER, F. Cuckoo hashing. In *Proc. ESA* (2001), Springer, pp. 121–133.

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

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

[45] ROSS, K. A. Efficient Hash Probes on Modern Processors. In *Proc. ICDE* (2007), IEEE, pp. 1297–1301.

[46] SUN, Y., HUA, Y., FENG, D., YANG, L., ZUO, P., AND CAO, S. MinCounter: An Efficient Cuckoo Hashing Scheme for Cloud Storage Systems. In *Proc. MSST* (2015), IEEE.

[47] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating Realistic Datasets for Deduplication Analysis. In *Proc. USENIX ATC* (2012), USENIX Association, pp. 261–272.

[48] TAYLOR, T., COULL, S. E., MONROSE, F., AND MCHUGH, J. Toward Efficient Querying of Compressed Network Payloads. In *Proc. USENIX ATC* (2012), USENIX Association, pp. 113–124.

[49] TURNER, V., GANTZ, J. F., REINSEL, D., AND MINTON, S. The digital universe of opportunities: Rich data and the increasing value of the internet of things. *Framingham (MA): IDC* (2014).

[50] VAMANAN, B., SOHAIL, H. B., HASAN, J., AND VIJAYKUMAR, T. TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *Proc. MICRO* (2015), ACM, pp. 585–597.

[51] WANG, C., REN, K., YU, S., AND URS, K. M. R. Achieving Usable and Privacy-assured Similarity Search over Outsourced Cloud Data. In *Proc. INFOCOM* (2012), IEEE, pp. 451–459.

[52] WANG, K., XU, G., SU, Z., AND LIU, Y. D. GraphQ: Graph Query Processing with Abstraction Refinement-Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *Proc. USENIX ATC* (2015), USENIX Association, pp. 387–401.

[53] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A Scalable, High-Performance Distributed File System. In *Proc. OSDI* (2006), USENIX Association, pp. 307–320.

[54] WU, S., LI, F., MEHROTRA, S., AND OOI, B. C. Query Optimization for Massively Parallel Data Processing. In *Proc. SOCC* (2011), ACM.

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

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