Rectangular Hash Table: Bloom Filter and Bitmap Assisted Hash Table with High Speed

Tong Yang*¶, Binchao Yin†, Hang Li*, Muhammad Shahzad‡, Steve Uhlig§, Bin Cui* Xiaoming Li*
* Peking University, China. ¶ Collaborative Innovation Center of High Performance Computing, NUDT, Changsha, China.
‡ North Carolina State University, USA. § UK & Queen Mary, University of London.
† Beijing University of Posts and Telecommunications, China

***Abstract*—Hash table, a widely used data structure, can achieve an $O(1)$ average lookup speed at the cost of large memory usage. Unfortunately, hash tables suffer from collisions and the rate of collisions is largely determined by the load factor. Broadly speaking, existing research has taken two approaches to improve the performance of hash tables. The first approach trades-off collision rate with memory usage, but only works well under low load. The second approach pursues high load and no hash collisions, but comes with update failures. The goal of this paper is to design a practical and efficient hash table that achieves high load factor, low hash collision rate, fast lookup speed, fast update speed, and zero update failures. To achieve this goal, we take a three-step approach. First, we propose a set of hashing techniques that leverage Bloom filters to significantly reduce hash collision rates. Second, we introduce a novel *kick mechanism* to achieve a high load factor. Last, we develop bitmaps to significantly accelerate the kick mechanism. Theoretical analysis and experimental results show that our hashing schemes significantly outperform the state-of-the-art. Our hash table achieves a high load factor (greater than 95%), a low collision rate (less than 0.56%), and the number of hash buckets almost equals to the number of key-value pairs. Given n key-value pairs, the collision rate is reduced to 0 by either using 1.18 $\times n$ buckets or allowing up to 5 blind kicks. We have released the source code of the implementations of our hash table and of 6 prior hash tables at Github [1].**

I. Introduction

A. Background and Problem Statement

Hash tables are data structures that store key-value pairs. A key-value pair is a set of two linked data items: a *key*, which is a unique identifier for some item of data, and a *value*, which is the data or a pointer to the data that is identified. A hash table is comprised of a finite number of buckets, where each bucket is essentially a certain number of bits in the memory. Depending on how the hash-table is designed, each bucket is used to store one or more key-value pairs. A hash table is associated with three operations: *insertion*, *deletion*, and *query*. The insertion operation stores the value of a given key-value pair at an appropriate location in the hash table. When inserting the value, hash table may experience a collision, *i.e.*, the bucket in the hash table assigned to the new key-value pair is already taken by an existing one. These collisions are called *hash collisions*. The deletion operation deletes a previously inserted value of a given key-value pair from the hash table. The query operation retrieves the previously inserted value corresponding to a given key. In the rest of this paper, at times, we will use the term *update* to refer to the insertion and deletion operations simultaneously. A wide variety of metrics are used to measure the quality of a hash table. These include *load factor, insertion speed, deletion speed*, and *query speed*. Load factor is defined as the ratio of the number of non-empty buckets to the total number of buckets in the hash table. A higher load factor indicates that the hash table utilizes the memory efficiently by minimizing the number of unused buckets. Insertion, deletion, and query speeds refer to the time the hash table takes to insert, delete, and query the value corresponding to a given key, respectively. Clearly, higher speeds are more desirable.

Hash tables are being extensively used in a variety of applications such as key-value stores [11], [9], [22], NLP [26], [32], [13], IP lookups [39], [27], [36], packet classification [14], [31], [30], load balancing [33], [23], [3], [2], intrusion detection [25], [28], and TCP/IP state management [35]. With more and more devices connecting to the Internet and causing an increase in the amount of data at an unprecedented rate, in most real-world applications, the number of key-value pairs are also growing at an unprecedented rate. Unfortunately, the memory available on computing devices to store the hash tables is limited. Therefore, hash tables frequently experience hash collisions when inserting new key-value pairs, which adversely affect all three operations associated with the hash table. For the insertion operation, hash collisions either lead to the failure of insertion or result in an increased time to insert a given key-value pair. For the deletion and query operations, hash collisions result in an increased time to delete or query, respectively, the previously inserted value corresponding to the given key. In this paper, we design a new hash table that is easy to implement in practical hardware devices and has high load factor, high insertion, deletion, and query speeds, low rate of collisions, and zero insertion failures.

B. Limitations of Prior Hashing Paradigms

The majority of the prior hash tables follow one of the following four paradigms: chaining hash table, multi hash table, multi hash table with Bloom filters, or kicking hash table. Next, we describe the insertion, deletion, and query operations on these four categories of hash tables and discuss their limitations. These limitations will motivate the need and design of our proposed scheme.

1) Chaining Hash Table: Chaining hash table (CHT) consists of a single hash table T with m buckets and a hash

function $h(.)$ with uniformly distributed output. Each bucket has chain of units, and each unit has three fields: key, value, and a pointer. The pointer field points to the next unit in the chain, if any. Let $B[i]$ represent the i^{th} bucket in the table T, where $0 \leqslant i \leqslant m$. To insert a key-value pair with key x in table T, CHT evaluates $h(x)$, creates a new unit at the start of the chain of bucket $B[h(x)\%m]$, and inserts the pair into this unit. To delete a key-value pair with key x, CHT traverses the chain of the bucket $B[h(x)\%m]$ looking for the key x. If it find a unit with this key, it deletes the unit, otherwise reports that the key was not found. To query the value for the key x, CHT traverses the chain of the bucket $B[h(x)\%m]$ looking for the key x. If it find a unit with this key, it returns the value, otherwise reports that the key was not found.

Limitations: CHT falls short of three perspectives. First, the use of pointers wastes a lot of memory space. Second, the performance of CHT drops significantly when the load factor is high. Therefore, it must keep many empty buckets to maintain high performance. Third, its query operation is inefficient, because it has to traverse long chains to find the key, which may not even exist in the table. In other words, its worst case query time is not reasonably bounded.

2) Multi Hash Table: Multi hash table (MHT) increases the number of hash tables from 1 to z, where each hash table is called a sub-table. Operation on sub-table is just same as CHT, while the difference is that all operation starts from the first sub-table and continue on the next one if needed, and ends when iterate through all sub-tables.

Limitations: While MHT reduces the number of collisions compared to CHT, it falls short of the perspective of query speed, because it usually has to look into several sub-tables before finding the desired key-value pair. Furthermore, the load factor is also low. Notable schemes following this hashing paradigm include d_random[4] and d_left hashing [34].

3) MHT with Bloom Filters: The limitation of slow query speed of MHT can be overcome by using a Bloom filter F_j for each sub-table T_j. A Bloom filter [5], [38], [37] is a bit array that can be used to quickly check whether an item has been inserted in a sub-table or not. Each Bloom filter consists of an array F_j of m_j^{BF} bits corresponding to its affiliated sub-table. Each Bloom filter F_j is associated with c independent hash functions, represented by $h_{j,k}^{\text{BF}}(.)$, where $0 \leqslant k \leqslant c-1$. We represent MHT with Bloom filters using MHT$^{\text{BF}}$.

Limitations: Although MHT$^{\text{BF}}$ is faster than MHT, for each lookup, it still needs to query all Bloom filters, which requires $z \times c$ hash computations if each Bloom filter uses c hash functions and there are z sub-tables. Furthermore, the load factor of MHT$^{\text{BF}}$ is as low as that of MHT. Notable schemes that follow this hashing paradigm include FHT [29], segment hashing [19], peacock hashing [20], and choice hashing [15].

4) Kicking Hash Table: KHTs can be produced from any of the CHT, MHT, and MHT$^{\text{BF}}$ by introducing slight modifications into their corresponding insertion, deletion, and query operations. While the exact details of implementation for different KHT based schemes are different, the basic underlying principle is that when inserting a key-value, if the bucket to which the key is mapped by the hash-function already contains another key-value pair, kick that existing key-value pair out, insert the new key-value pair in that bucket, and find a new bucket for the kicked-out key-value pair.

Limitations: While KHTs generally have faster query speeds and relatively high load factors, they are severely limited by their slow insertion speeds because they require a large number of hash computations and memory accesses due to kicking and still often end up with insertion failures. Such schemes also need to reconstruct the entire hash tables if they are frequently unable to update the tables, which takes a large amount of time and is hardly acceptable in practical applications. Notable schemes that follow this hashing paradigm include perfect hashing [12], [8], cuckoo hashing [24], the applications of cuckoo hashing [9], [40], and more [17], [21].

C. Proposed Approach

In this paper, we propose a new hash table, called Rectangular Hash Table (RHT). RHT achieves high load factor, high insertion, deletion, and query speeds, low rate of collisions, and zero insertion failures. RHT is an MHT$^{\text{BF}}$ based kicking hash table. Just like MHT$^{\text{BF}}$, RHT has z sub-tables, where we represent the j^{th} sub-table by T_j. Each sub-table T_j has m_j buckets, an independent hash function $h_j(.)$, and a Bloom filter F_j associated with it, where $0 \leqslant j \leqslant z-1$. We represent the i^{th} bucket in the sub-table T_j with $B_j[i]$, where $0 \leqslant i \leqslant m_j$. Each Bloom filter F_j is associated with c independent hash functions with uniformly distributed outputs, represented by $h_{j,k}^{\text{BF}}(.)$, where $0 \leqslant k \leqslant c-1$.

RHT differs from MHT$^{\text{BF}}$ in following four key aspects: kicking mechanism, sub-table sizes, hash chains, and load balancing technique. Regarding kicking mechanism, when inserting a key-value pair with key x, instead of blindly kicking a key-value pair out of one of the z buckets $B_j[h_j(x)\%m_j], \forall j \in [0, z-2]$, RHT first utilizes a bitmap to determine which key-value pair among these $z-1$ buckets will most quickly find a new empty bucket, and then kicks that key-value pair out. For reasons that will become apparent in the next two paragraphs, RHT never kicks out the key-value pairs from the buckets for sub-table T_{z-1}. Our bitmap based kicking scheme overcomes the limitation of slow insertion speeds of conventional KHTs. The insertion speed of RHT is up to 2 times faster than prior hash tables. This bitmap based kicking also enables RHT to achieve a significantly higher load factor compared to prior hash tables because it kicks those key-value pairs that can be inserted into empty buckets quickly with fewest subsequent kicks. The load factor of RHT is up to 1.2 times higher than prior hash tables.

Regarding sub-table sizes, the sizes of sub-tables in RHT follow a decreasing arithmetic progression, *i.e.*, $m_{j+1} = m_j - 1$. We do not keep the sizes of all sub-tables the same because some sub-tables experience fewer collisions compared to others and should have a smaller size to improve the memory efficiency of RHT. These arithmetically decreasing sub-table sizes enable us to combine all z Bloom filters associated with the z sub-tables into a single Bloom filter.

The key intuition is that $\forall j \in [0, z-1]$, $m_j + m_{z-1-j}$ results in the same number, *i.e.*, if we combine sub-tables T_j and T_{z-1-j}, the resulting sub-table will always have the same number of buckets regardless of the value of j. This means that, we can have $\lceil z/2 \rceil$ Bloom filters of the same size for all these $\lceil z/2 \rceil$ combined sub-tables. As long as these $\lceil z/2 \rceil$ Bloom filters use the same c hash functions, we can combine them into a single Bloom filter by placing them adjacently. The results of hash functions now point to $\lceil z/2 \rceil$ consecutive bits instead of a single bit. To set a bit of Bloom filter F_j, where $1 \leqslant j \leqslant \lceil z/2 \rceil$, we take bitwise logical `OR` of the $\lceil z/2 \rceil$ bits with $2^j - 1$. To read the bit of Bloom filter F_j, we take bitwise logical `AND` of the $\lceil z/2 \rceil$ bits with $2^j - 1$. If the result is 0, the bit is 0 otherwise 1.

As RHT combines all Bloom filters into a single Bloom filter, in querying the value for any given key, RHT now computes only c hash functions to retrieve all bits that it needs to identify sub-tables that might contain the key. Thus, by combining the Bloom filters, RHT has reduced the number of hash computations by z times compared to the number of hash computations of MHT$^{\text{BF}}$ to query its z Bloom filters. Our arithmetically decreasing sub-table sizes enable RHT to overcome the limitation of slow query speeds of MHT and MHT$^{\text{BF}}$. The query speed of RHT is up to 5 times faster than prior hash tables, respectively.

Regarding the hash chains, RHT allows hash chains only in the smallest sub-table, *i.e.*, in the sub-table T_{z-1}. When inserting a new key-value pair, if all buckets are occupied, and after consulting the bitmap, RHT determines that neither of the $z-1$ key-value pairs that are stored in buckets $B_j[h_j(x)\%m_j]$, where$0 \leqslant j \leqslant z-2$, will find an empty bucket if kicked out, RHT randomly picks one of these $z-1$ buckets and kicks its key-value pair. RHT then looks at the bitmap again to determine whether any of key-value pair in the $z-2$ buckets that this recently kicked out key-value pair maps to will find an empty bucket if kicked out. This process continues until a kicked out key-value pair finds an empty bucket, or the number of kicks reach a threshold θ. The value of θ is set to ensure that most key-value pairs can find an empty bucket. If the number of kicks reaches the threshold, and there still is a kicked out key-value pair with key y that needs to be inserted, RHT inserts it into the hash chain of bucket $B_{z-1}[h_{z-1}(y)\%m_{z-1}]$. By keeping the hash-chains in a sub-table, RHT overcomes the limitation of insertion failures of KHTs. The motivation behind associating the hash chains with the smallest sub-table is to reduce the memory usage of RHT.

Regarding load-balancing, when inserting a key-value pair, RHT picks the sub-table with the smallest load factor among all sub-tables that have an empty bucket for the key-value pair. To summarize, while RHT is based on the MHT$^{\text{BF}}$ based KHT paradigm, it does not suffer from any of the limitations of either the KHT paradigm (*i.e.*, insertion failure and slow insertion speed) or the MHT$^{\text{BF}}$ paradigm (*i.e.*, slow query and deletion speeds, and low load factor).

D. Key Contributions

1) We propose a novel hash table, namely RHT, that outperforms all prior state-of-the-art hash tables in terms of insertion, deletion, and query speeds, load factor, and rate of collisions.
2) We implemented RHT along with 6 prior state-of-the-art hash tables and extensively evaluated them and performed their side-by-side comparison. Our results show that RHT outperforms all prior state-of-the-art hash tables in terms of insertion speed by 2 times, query speed by 5 times, load factor by 1.2 times, and rate of collisions by 10 times.

II. Rectangular Hash Table

In this section, we describe our rectangular hash table (RHT) scheme. To motivate and justify the design choices we made, we start with the bare bones version of RHT and incrementally build over it to arrive at the final design of RHT. In describing each version of RHT, we will first discuss the design of that version and describe how this design addresses the limitations of the previous version of RHT or of the hash tables previously proposed by researchers. Finally, we will discuss the limitations of that version, which motivate the design of the next version.

A. RHT Version 1: using One Combined Bloom Filter

Recall from Section I-B3 that one of the limitations of MHT$^{\text{BF}}$ is that a large number of hash computations significantly affect the query speed of MHT$^{\text{BF}}$. The objective of version 1 of RHT, represented by RHT^{v1}, is to reduce the number of Bloom filters from z to 1, which will reduce the number of hash computations per query from $z \times c$ to just c, and increase the query speed, deletion speed. RHT^{v1} is essentially the same as MHT$^{\text{BF}}$, except that the number of buckets in each sub-table follows a decreasing arithmetic progression, *i.e.*, $m_{j+1} = m_j - 1$, where $0 \leqslant j \leqslant z-1$. The key intuition is that $\forall j \in [0, z-1]$, $m_j + m_{z-1-j}$ results in the same number. In RHT^{v1}, we logically combine sub-tables T_j and T_{z-1-j} for each $j \in [0, \lceil z/2 \rceil]$, which results in $\lceil z/2 \rceil$ logical sub-tables with equal number of buckets which benefits the combination of Bloom filters thus achieving less number of memory accesses. Figure 1 shows 6 sub-tables with arithmetically decreasing sizes on the left side and 3 combined sub-tables of equal sizes at the right. Note that the logical sub-tables form a rectangular shaped data structure and so we name our scheme *rectangular hash table*.

RHT^{v1} maintains the Bloom filter F_j for each logical sub-table L_j, where $0 \leqslant j \leqslant \lceil z/2 \rceil$. The same number of bits in each Bloom filter F_j, *i.e.*, $\forall j \in [0, \lceil z/2 \rceil]$, m_j^{BF} is a constant value m^{BF}, resulting in same false positive rate of each filter, thus going one step further, instead of keeping the $\lceil z/2 \rceil$ Bloom filters separated, we can combine them by appending their corresponding bits into a *bin* comprising of $\lceil \frac{z}{2} \rceil$ bits as a unit in new combined Bloom filter without influencing each previous filter's performance. Figure 1 also shows the 3 equally sized Bloom filters are appended to one combined

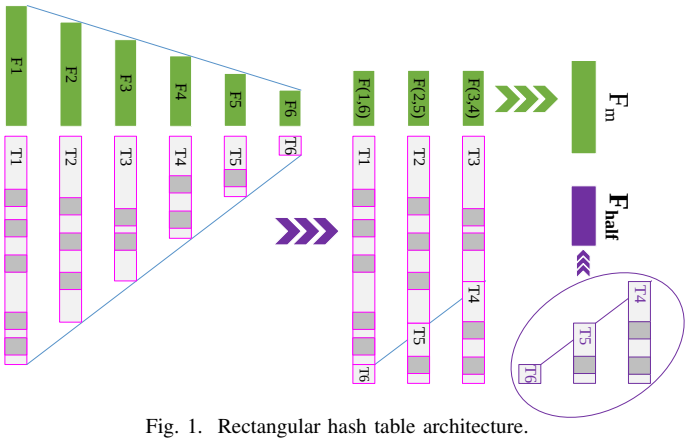


Fig. 1. Rectangular hash table architecture.

filter CF, and $CF[l]$ represents the l^{th} bin of the CF, where $0 \leqslant l \leqslant m^{\text{BF}} - 1$. We use the same set of c hash functions for the combined Bloom filter as previous, where the result of each hash function now points to one bin comprised of $\lceil z/2 \rceil$ bits storing each bit from separated filters. We represent the j^{th} bit in the l^{th} bin by $CF[l][j]$, where $0 \leqslant l \leqslant m^{\text{BF}} - 1$ and $0 \leqslant j \leqslant \lceil z/2 \rceil$. Figure 2 shows a 3-bit Bloom filter which is obtained from three equally sized standard Bloom filters F_1, F_2, and F_3. Unlike sub-tables, which are conceptually combined, we physically combine the $\lceil z/2 \rceil$ Bloom filters to one Bloom filter, which we then place in on-chip memory for fast query processing. To avoid insertion failures due to no space available, RHT^{v1} uses chaining lists as the z^{th} sub-table as it is the shortest among all sub-tables. Thus, the pointers will use the least amount of memory. Next, we describe the insertion, query, and deletion operations of RHT^{v1}.

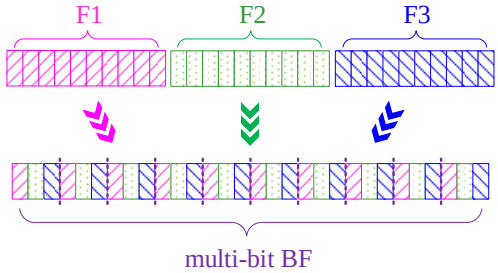


Fig. 2. Structure of the combined Bloom filter.

Insertion: To insert a key-value pair with key x, RHT^{v1} first evaluates $h_j(x)\%m_j$, $j \in [0, z-1]$,and then identifies all empty buckets among $B_j[h_j(x)\%m_j]$. It then inserts this key-value pair into the empty bucket in the sub-table that has the smallest load factor, because of which RHT^{v1} ensures good load balancing across sub-tables. If, however, RHT^{v1} finds that none of the z buckets $B_j[h_j(x)\%m_j]$ are empty, it inserts the key-value pair to the chain of the bucket $B_{z-1}[h_{z-1}(x)\%m_{z-1}]$, which is the last sub-table we assigned as chain list. Note that RHT^{v1} does not use any method of kicking. After inserting the key-value pair in a bucket of a sub-table, in order to record the key-value pair, RHT^{v1} updates the bin in the combined Bloom filter with the associated bit. More specifically, if it adds the key-value pair to sub-table T_j, it either sets $CF[h_k^{\text{BF}}(x)\%m^{\text{BF}}][j]$ to 1 if $0 \leqslant j \leqslant \lceil z/2 \rceil$, or sets $CF[h_k^{\text{BF}}(x)\%m^{\text{BF}}][z-1-j]$ to 1 if $\lceil z/2 \rceil < j \leqslant z-1$. To set the j^{th} bit to 1 in a bin, RHT^{v1} simply takes logical OR to the bin, which is efficient.

Query: To query a key x, RHT^{v1} queries the Bloom filter at every j^{th} bit in all bins, that is the m^{BF} bits $CF[l][j]$, where $0 \leqslant l \leqslant m^{\text{BF}} - 1$. If all c bits $CF[h_k^{\text{BF}}(x)\%m^{\text{BF}}][j]$ are set to 1, it goes to check the buckets $B_j[h_j(x)\%m_j]$ in sub-table T_j and $B_{z-1-j}[h_{z-1-j}(x)\%m_{z-1-j}]$ in T_{z-1-j} to determine whether the key is there. If it finds the key in either bucket, it returns the associated value, otherwise, it increments j by 1 and repeats the entire process until it either finds the key in a sub-table or has iterated all $\lceil z/2 \rceil$ Bloom filters. Note that when querying, RHT^{v1} does not calculate $z \times c$ hash functions which is in MHT$^{\text{BF}}$, but only calculates c hash functions. To read the j^{th} bit in a bin, RHT^{v1} simply takes logical AND to that bin with $2^j - 1$ in binary.

Deletion: To delete a key-value pair with key x, RHT^{v1} first searches the bucket containing the key as described above and then removes the key-value pair from that bucket if the bucket does not belong to the sub-table T_{z-1} which is a chain list. If the bucket belongs to the sub-table T_{z-1}, after traversing the chain of the bucket $B_{z-1}[h_{z-1}(x)\%m_{z-1}]$, it finds the key and copies the value in the pointer field of that unit to the pointer field of the preceding unit, then deletes the unit. If it can not find the key, it declares that the key was not found in the hash table. As the combined Bloom filter CF is generated by combining standard Bloom filters, and standard Bloom filters do not support deletions, RHT^{v1} does not delete the key in the combined Bloom filter after removing the pair from the sub-table.

Limitations: When querying a key x, RHT^{v1} needs to look up buckets in two sub-tables T_j and T_{z-1-j}, which is inefficient. Ideally, we would like our scheme to look up only one bucket in one sub-table. The next version of RHT, represented by RHT^{v2} addresses this limitation.

B. RHT Version 2: using A Second-half Bloom Filter

To reduce the times of looking up from 2 to 1, RHT^{v2} maintains an additional Bloom filter to support it, which we call the second-half Bloom filter and represent by CF_{half}. The number of bins and the number of hash functions in CF_{half} are equal to those in CF. while unlike CF, CF_{half} is not obtained by combining any Bloom filters. CF_{half} is shown at right side in Figure 1. Next, we describe the insertion, query, and deletion operations of RHT^{v2}.

Insertion: To insert a key-value pair with key x, RHT^{v2} follows the same procedure as RHT^{v1} does except that if the sub-table to which it inserts this key-value pair is among the smaller $\lfloor z/2 \rfloor$ sub-tables, then it also updates the Bloom filter CF_{half} in addition to CF. More specifically, if RHT^{v2} inserts this key-value pair into a bucket in the sub-table T_j, where $j \in (\lceil z/2 \rceil, z-1]$, RHT^{v2} also sets the c bits $CF_{\text{half}}[h_k^{\text{BF}}(x)\%m^{\text{BF}}]$ to 1.

Query: To query a key x, RHT^{v2} follows the same procedure as RHT^{v1} does except that if all c bits $CF[h_k^{\text{BF}}(x)\%m^{\text{BF}}][j]$ are set to 1, it then checks the c bits $h_k^{\text{BF}}(x)\%m^{\text{BF}}$ of CF_{half}. If all c bits $CF_{\text{half}}[h_k^{\text{BF}}(x)\%m^{\text{BF}}]$ are not set to 1, RHT^{v2} looks up

the bucket $B_j[h_j(x)\%m_j]$ for key x in sub-table T_j, otherwise it looks up the bucket $B_{z-1-j}[h_{z-1-j}(x)\%m_{z-1-j}]$ in sub-table T_{z-1-j}.

Deletion: To delete a key-value pair with key x, RHT^{v2} follows the same procedure as RHT^{v1} does. Similar to RHT^{v1}, RHT^{v2} does not delete the key from CF and CF_{half} after removing it from the bucket.

Limitations: While RHT^{v2} reduces the number of looking up sub-tables from two to one, it still has a high rate of bucket collisions, *i.e.*, when the number of key-value pairs is large, it is very likely that all candidate buckets are occupied when inserting new key-value pair. RHT^{v1} and RHT^{v2} insert the new key-value pair into the last sub-table which we assigned as a chain list at the bucket $B_{z-1}[h_{z-1}(x)\%m_{z-1}]$. Consequently, this leads to a lower load factor because some buckets in the sub-tables stay unfilled which potentially could offer a position for pairs which are forced to append into the chain list. The next version of RHT, represented by RHT^{v3}, improves upon RHT^{v2} in increasing the load factor.

C. RHT Version 3: using Cuckoo Kicking

To increase the load factor, RHT^{v3} invites kicking method into RHT^{v2}, which is inspired by cuckoo hashing [24]. The key idea behind kicking is that when all z buckets that an incoming key-value pair matches are already occupied, if we kick out one of the exsiting pairs in those buckets and replace it with the incoming pair, there is a chance that the kicked out pair can match to an empty bucket. In this way, the incoming pair as well as the kicked out pair both get placed into buckets and neither of the pairs end up in a chain of sub-table T_{z-1}. This approach results in two advantages. First, more buckets get utilized, which increases the load factor. Second, the sizes of chains in the sub-table T_{z-1} decrease, which accelerates querying the key-value pairs in the chains of the sub-table T_{z-1}. Next, we describe the insertion, query, and deletion operations of RHT^{v3}.

Insertion: To insert a key-value pair with key x, RHT^{v3} follows the same procedure as RHT^{v2} except that if all z buckets $B_j[h_j(x)\%m_j]$, where $0 \leqslant j \leqslant z-1$, are occupied, instead of inserting this new incoming key-value pair into the chain of bucket $B_{z-1}[h_{z-1}(x)\%m_{z-1}]$, it kicks out the key-value pair stored in the bucket $B_{l_x}[h_{l_x}(x)\%m_{l_x}]$, where $l_x \in [0, z-2]$ and replaces it with the incoming key-value pair. Note that our kicking out the pair from the bucket of sub-table T_{l_x} is arbitrary. One could choose any sub-table and achieve the same performance. Suppose, the key in the kicked out key-value pair is x_1. RHT^{v3} executes the same insertion process for this kicked out pair as it did for the new incoming pair except that it does not use sub-table T_{l_x}. It is possible that the $z-1$ buckets $B_j[h_j(x_1)\%m_j]$, where $j \neq l_x$, are again all occupied, in which case RHT^{v3} will again kick out a key-value pair, this time from the bucket $B_{l_{x_1}}[h_{l_{x_1}}(x_1)\%m_{l_{x_1}}]$, where $l_{x_1} \in [0, z-2]-\{l_x\}$, and insert the kicked out key-value pair with key x_1 in that bucket. RHT^{v3} again executes the same insertion process on x_2 and so on. To bound the amount of times that RHT^{v3} spends on kicking, we put a threshold to limit the times key-value pairs can be kicked out. Once the threshold is reached, RHT^{v3} inserts the last kicked out key-value pair into the appropriate bucket of sub-table T_{z-1}.

Query: The query operation of RHT^{v3} is the same as the query operation of RHT^{v2}.

Deletion: The deletion operation of RHT^{v3} is the same as the deletion operation of RHT^{v2}.

Limitations: Kicking method inspired by cuckoo hashing is not time efficient because RHT^{v3} randomly chooses a bucket to kick a key-value pair out without determining whether the kicked out pair will find an empty bucket or not. We call such kicks *blind kicks*. Blind kicks increases the number of kicks before a kicked out pair finds an empty bucket. Consequently, the insertion speed of RHT^{v3} is relatively slow. The next version of RHT, represented by RHT^{v4}, improves upon RHT^{v3} to increase the insertion speed by adopting a new kicking method.

D. RHT Version 4: The Final Design

To reduce the number of kicks, RHT^{v4} replaces blind kicks with *bitmap kicks*. For each sub-table T_j, where $0 \leqslant j \leqslant z-1$, RHT^{v4} maintains a small bitmap M_j in the on-chip memory. Each bitmap M_j is essentially just a set of m_j bits. Recall that m_j represents the number of buckets in sub-table T_j. We represent the i^{th} bit in the bitmap M_j with $M_j[i]$, where $0 \leqslant i \leqslant m_j - 1$. Each bit $M_j[i]$ corresponds to a bucket $B_j[i]$ in sub-table T_j, where $0 \leqslant j \leqslant z-1$ and $0 \leqslant i \leqslant m_j - 1$. The key idea is that each bit in any given bitmap stores the occupancy status of the corresponding bucket in the sub-table, *i.e.*, if bucket $B_j[i]$ is empty, then the bit $M_j[i]$ is 0, otherwise it is set to 1. As bitmaps reside in on-chip memory, RHT^{v4} can quickly access them to determine which buckets are currently empty and which key-value pair, if kicked out from its current bucket, will find an empty bucket. With the help of bitmaps, RHT^{v4} significantly reduces the number of kicks because bitmaps give a holistic view of empty and non-empty buckets in the sub-tables. Next, we describe the insertion, query, and deletion operations of RHT^{v4}.

Insertion: To insert a key-value pair with key x, RHT^{v4} first evaluates the z hash functions $h_j(x)\%m_j$ and then looks up the z bits $M_j[h_j(x)\%m_j]$ in bitmaps, where $0 \leqslant j \leqslant z-1$, to identify the empty buckets. If, it finds some empty buckets, it inserts this key-value pair into the empty bucket of the sub-table that has the smallest load factor, *i.e.*, the sub-table with empty bucket for which $\sum_{i=0}^{m_j-1} M_j[i]/m_j$ is the smallest. It then updates the combined Bloom filters CF and/or CF_{half} to indicate that the key-value pair has been inserted in this sub-table, as described earlier for RHT^{v1} and RHT^{v2}. If, RHT^{v4} does not find an empty bucket, instead of blindly kicking out a key-value pair from $B_{l_x}[h_{l_x}(x)\%m_{l_x}]$, where $l_x \in [0, z-2]$, it sets $l_x = 0$, and checks whether the key-value pair in $B_{l_x}[h_{l_x}(x)\%m_{l_x}]$ will find an empty bucket if kicked out or not. More specifically, let the key of the key-value pair in $B_{l_x}[h_{l_x}(x)\%m_{l_x}]$, where currently $l_x = 0$, be represented with y. RHT^{v4} checks whether any of the bits $M_j[h_j(y)\%m_j]$, where $0 \leqslant j \leqslant z-1$, in the bitmaps are 0. If

there is at least one bit that is 0, it means that there is at least one empty bucket among the buckets $B_j[h_j(y)\%m_j]$, where $j \neq l_x$, in one of the sub-tables to which the key-value pair currently stored in $B_{l_x}[h_{l_x}(x)\%m_{l_x}]$ can be moved. In this case, RHT^{v4} kicks out this key-value pair with key y from the bucket $B_{l_x}[h_{l_x}(x)\%m_{l_x}]$, moves it to the empty bucket of the sub-table with smallest load factor, and inserts the incoming key-value pair with key x into this bucket $B_{l_x}[h_{l_x}(x)\%m_{l_x}]$. If, however, none of the bits $M_j[h_j(y)\%m_j]$ in the bitmaps are 0, this means that none of the other buckets $B_j[h_j(y)\%m_j]$, where $j \neq l_x$, in remaining sub-tables are empty. Therefore, kicking this key-value pair will not do any good and will only lead to another kick. In this case, RHT^{v4} increments l_x by one and checks the bitmaps for the key-value pair stored in the next bucket $B_{l_x}[h_{l_x}(x)\%m_{l_x}]$. It continues this process of looking for an empty bucket for the key-value pair stored in $B_{l_x}[h_{l_x}(x)\%m_{l_x}]$ until either for a certain value of l_x, where $0 \leqslant l_x \leqslant z-2$, RHT^{v4} finds an empty bucket with the help of bitmaps, or all $z-1$ key-value pairs currently stored in the $z-1$ buckets $B_{l_x}[h_{l_x}(x)\%m_{l_x}]$, where $0 \leqslant l_x \leqslant z-2$, have been checked and there is no empty bucket for any of them to be moved to. If in the latter case, RHT^{v4} randomly picks a key-value pair in one of the buckets $B_{l_x}[h_{l_x}(x)\%m_{l_x}]$, where $0 \leqslant l_x \leqslant z-2$, kicks it out (*i.e.*, performs a blind kick as in RHT^{v3}), inserts the incoming pair with key x in its place, and tries to insert the most recently kicked out key-value pair using the same method as it used for the incoming key-value pair with key x. Just like RHT^{v3}, to bound the amount of time RHT^{v4} spends on kicking, we put a threshold θ on the number of blind kicks. Once the threshold is reached, RHT^{v4} inserts the most recently kicked out key-value pair into the chain of the appropriate bucket of sub-table T_{z-1}. By tuning the value of θ, RHT^{v4} can make a trade-off between the load factor of sub-tables and the insertion speed. Finally, note that every time RHT^{v4} inserts a key-value pair into a bucket $B_j[i]$, it always sets the bit $M_j[i]$ to 1, where $0 \leqslant j \leqslant z-1$ and $0 \leqslant i \leqslant m_j$.

Query: The query operation of RHT^{v4} is the same as the query operation of RHT^{v3}.

Deletion: The deletion operation of RHT^{v4} is also the same as the deletion operation of RHT^{v3} except that whenever RHT^{v4} deletes a key-value pair from a bucket $B_j[i]$, and the bucket becomes empty, it always resets the bit $M_j[i]$ to 0, where $0 \leqslant j \leqslant z-1$ and $0 \leqslant i \leqslant m_j$. Note that for sub-table T_{z-1}, it is possible that when a key-value pair is deleted from a bucket, the bucket does not become empty, because the buckets in sub-table T_{z-1} contain chains of key-value pairs, rather than single key-value pairs.

Table I summarizes and compares all hash tables we have discussed until now. We observe from this table that RHT^{v4} outperforms cuckoo hashing as well as other schemes in terms of support for parallelization, number of memory accesses, update speed, and insertion failures. Next, we calculate the false positive rates of the Bloom filters used in RHT^{v4}, which determines the number of buckets that RHT^{v4} will look up when querying the value for any given key or deleting the key-value pair corresponding to that key.

1) False Positive Rate: RHT^{v4} uses two Bloom filters: CF and CF_{half}. Let there be n key-value pairs and z sub-tables regrouped into $\lceil \frac{z}{2} \rceil$ sub-tables. Let the Bloom filter CF has w bins, where each bin has $\lceil \frac{z}{2} \rceil$ bits, corresponding to the $\lceil \frac{z}{2} \rceil$ regrouped sub-tables. Recall from Section II-A, that we use c hash functions for CF. We let $c = ln2\frac{w}{n}$. The false positive rate of CF is equal to that of any of the independent $\lceil \frac{z}{2} \rceil$ Bloom filters F_j. Therefore, the false positive rate of CF, represented by $f(CF)$, is given by the following equation.

$$f(CF) = 1 - (1 - 0.5^c)^{\frac{z}{2}-1} \tag{1}$$

If the number of BFs that report true is $u+1$, the false positive rate is given by the following equation.

$$f(CF, u) = 0.5^{c*u} * (1 - 0.5^{c(z-u-1)}) \tag{2}$$

As we also use c hash functions for CF_{half}, its false positive rate is simply $f(CF_{\text{half}}) = 0.5^c$. Given that a key is present in one of the sub-tables, if CF reports that the key exists only in one sub-table and CF_{half} does not report that the key exists, then no false positives occur and the probability of this event is $(1 - f(CF)) \times (1 - f(CF_{\text{half}}))$. If CF reports that a key exists in $u+1$ sub-tables and CF_{half} does not report that the key exists, u false positives occur and the probability of this event is $f(CF, u) \times (1 - f(CF_{\text{half}}))$. If CF reports that the key exists only in one sub-table and CF_{half} suffers from a false positive, the probability of this event is $(1 - f(CF)) \times f(CF_{\text{half}})$. If CF reports that a key exists in $u+1$ sub-tables, *i.e.*, suffers from u false positives, and CF_{half} also suffers from a false positive, the probability of this event is $f(CF, u) \times f(CF_{\text{half}})$. As an example, when $z = 8$ and $c = 16$, the false positive rate of RHT^{v4} is $1-(1-f(CF))*(1-f(CF_{\text{half}})) \approx 6.1*10^{-5}$, which is extremely small. This shows that our two Bloom filter based approach significantly reduces the number for sub-tables that RHT^{v4} looks up the given key in, which in turn significantly increases its insertion speed.

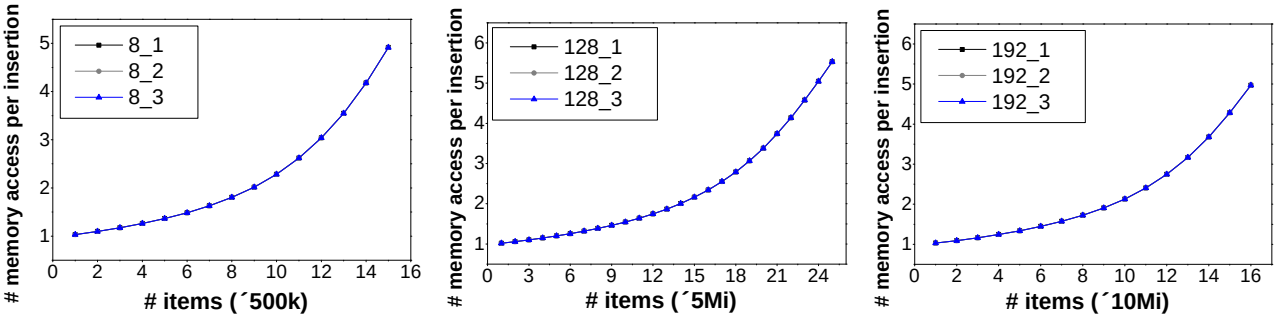
III. Performance Evaluation

A. Experimental setup

For our experiments, we assume that one memory access can read/write one bucket. We use β to represent the ratio of number buckets in all sub-tables to the number of total items. In our experiments, we vary the value of β from 1.05 to 10, *i.e.*, the size of the hash table will be 1.05 to 10 times the number of items. To generate data sets for evaluation, we used synthetic benchmarking data produced by the well known Yahoo Cloud Serving Benchmark (YCSB) [6]. From the YCSB data, we generated a large number of key-value pairs, where the size of *value* in each pair is fixed at 8 bits. Out of all the pairs, the size of *key* in approximately one-third pairs is 8 bits, in another one-third pairs is 16 bits, and in the remaining one-third is 128 bits. Corresponding to each size of the key, we generated three groups of data containing 8 million, 128 million, and 192 million key-value pairs. Consequently, we obtained 9 groups

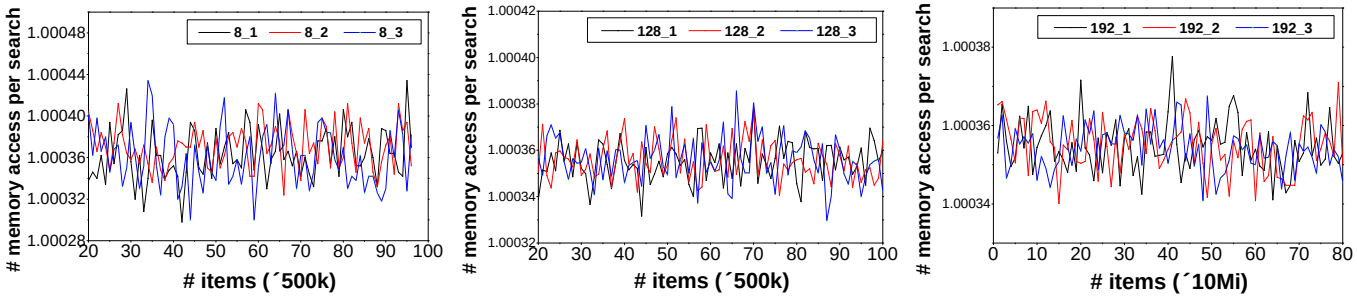
TABLE I
SUMMARY AND COMPARISON OF HASH TABLES

Name	Number of pointers	Load factor	Number of Bloom filters	Number of bitmaps	Query complexity	Update complexity	Update failures
CHT	$\geqslant 2m$	$\approx 50\%$	0	0	$O(1)$	$O(1)$	None
MHT	$\geqslant m$	$\approx 90\%$	0	0	$O(z)$	$O(z)$	None
MHT^{BF}	$\geqslant m$	$\approx 90\%$	z	0	$O(1)$	$O(z)$	None
Cuckoo	0	$\approx 95\%$	0	0	$O(1)$	$O(n)$	Yes
RHT^{v1}	$\approx m/(2z-1)$	$\approx 90\%$	1	0	$O(1)$	$O(z)$	None
RHT^{v2}	$\approx m/(2z-1)$	$\approx 90\%$	2	0	$O(1)$	$O(z)$	None
RHT^{v3}	$\approx m/(2z-1)$	$\geqslant 95\%$	2	0	$O(1)$	$O(z)$	None
RHT^{v4}	$\approx m/(2z-1)$	$\geqslant 95\%$	2	z	$O(1)$	$O(z)$	None



(a) $8Mi$ (b) $128Mi$ (c) $192Mi$

Fig. 3. Memory accesses per insertion during hash table construction ($\beta = 1.05$, $\theta = 1$) for three scales of datasets.



(a) $8Mi$ (b) $128Mi$ (c) $192Mi$

Fig. 4. # memory accesses per query (when $\beta = 1.05$, $\theta = 1$) for three scales of datasets.

of data for this evaluation. We represent the group with X million pairs and Y bit keys as X_Y. Our experiments are conducted on a Thinkstation D30 server with 2 Intel CPUs (Xeon E5-2620, 2.00 GHz, 6 physical cores).

B. Evaluation of RHT^{v4}

Let n represent the number of key-value pairs that need to be inserted into the hash table. We evaluated RHT^{v4} for all three values of n, *i.e.*, $n =$ 8M, 128M, and 192M. When conducting experiment for each value of n, we used 8 sub-tables, where the total size of 8 sub-tables is $\beta \times n$ ($1.05 \leqslant \beta \leqslant 10$). We insert the n items into the $z = 8$ sub-tables using our RHT^{v4} scheme. We define the *collision rate* as the number of items in the chain of the last sub-table to the total number of items. We use $c = 16$ hash functions per Bloom filter. We evaluate RHT^{v4} in terms of load factor, collision rate, insertion speed, query speed, and number of kicks.

1) Load Factor and Collision Rate: For the evaluation of load factor and collision rate, we set $\theta = 1$ and $\beta = 1.05$. Figure 5 plots the load factor of RHT^{v4} for our nine groups. The bars in this figure show the load factor of each subtable, and the line shows the aggregate load factor across all tables. We observe from this figure that the 8 sub-tables are very well balanced, and the overall load factor is 95.21%. RHT^{v4} achieves such a high load factor using just $1.05 \times n$ memory. Figure 6 shows the collision rate of RHT^{v4}. We observe from this figure that the collision rate of RHT^{v4} is just 0.034%.

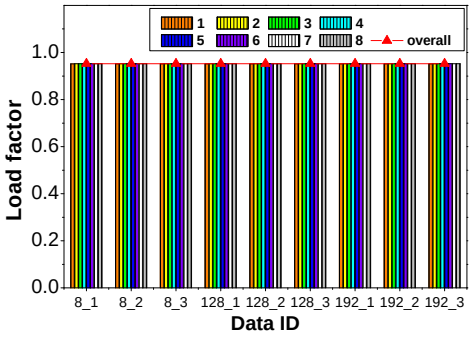


Fig. 5. Load factor of RHT^{v4} on the 9 data groups ($\beta = 1.05$, $\theta = 1$).

2) Insertion and Query Speeds: For the evaluation of insertion and query speeds, we again set $\theta = 1$ and $\beta = 1.05$. We quantify the speed in terms of the number of memory

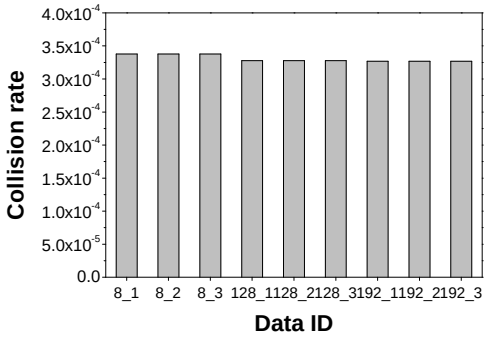


Fig. 6. Collision rate of RHT^{v4} on the 9 data groups ($\beta = 1.05$, $\theta = 1$).

accesses. We insert all the items from each data set into the RHT^{v4} hash table. As more items are inserted, each insertion needs more memory accesses due to *bitmap kicks*. In the worst case, the number of memory accesses is still bounded by $8+1$ accesses per insertion. Figures 3(a), 3(b), and 3(c) show the number of memory accesses per insertion of RHT^{v4} for the 9 groups of data. We observe from these figures that majority of items in all groups require less than 6 memory accesses per insertion. Figure 4(a), 4(b), and 4(c) show the number of memory accesses per query. We observe that the number of memory accesses per query ranges from 1.00033 to 1.00039 with a mean of 1.00036.

3) Bitmap and Blind Kicks: In this experiment, we set $\beta = 1.05$. We observed from our experiments that after one traversal of bitmap kicks, most items find empty buckets, and after 6 blind kicks, there are no items left in the hash chains of the last sub-table. Table II shows the number of items in the hash chains after θ number of kicks. When we set $\theta = 6$, the number of memory accesses per insertion is at most $8 \times 6 + 1 = 49$, and the number of memory accesses per search is at most 8. Even when there is only one blind kick allowed *i.e.*, $\theta = 1$ and when $\beta = 1.05$, only a very small fraction of items (0.035%) lies in the hash chains.

TABLE II
NUMBER OF COLLISION ITEMS BEFORE AND AFTER θ BLIND KICKS.

	before kick	$\theta = 1$	$\theta = 2$	$\theta = 3$	$\theta = 4$	$\theta = 5$	$\theta = 6$
8_1	464883	2702	360	65	11	1	0
128_1	7413615	41936	5991	986	122	7	0
192_1	11114921	62737	8800	1446	151	15	0

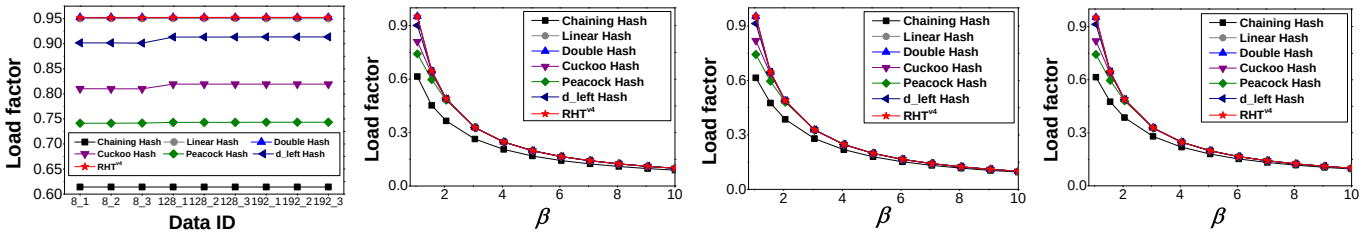
C. Comparison of RHT^{v4} with Prior Art

We compare RHT^{v4} with six prior well-known state of the art hash tables namely 1) hash chaining [7], 2) linear probing [18], 3) double hashing [18], 4) cuckoo hashing [24], 5) d_left hashing [34], and 6) peacock hashing [20]. We use the same data sets as described earlier for this comparative study. Before we present the results from our experiments, let us first define *insertion failures* for prior schemes. Recall that RHT^{v4} does not suffer from insertion failures. In linear probing, double hashing, and cuckoo hashing, whenever a collision happens during insertions, these schemes probe another bucket and this probing is recursively repeated. We set the maximum number of probing recursions to 500, which implies that the maximum number of memory accesses per insertion is 500 for these three schemes. After 500 attempts, if collisions still persist, and the scheme is unable to find an empty bucket for the key-value pair, we declare it as an *insertion failure*. For peacock hashing and RHT^{v4}, we use 16 hash functions for the Bloom filters. For the remaining Bloom filter parameters, we use their optimal values calculated using the equations derived in [5]. To avoid insertion failure in peacock hashing and d_left hashing, we use chaining for the sub-tables to avoid insertion failures. Next, we compare the performance of these hashing schemes in terms of load factor, insertion speed, and query speed.

1) Load Factor: Figure 7(a) shows the load factor of all hashing schemes for all nine groups of data using $\beta = 1.05$. We observe from this figure that the load factor of RHT^{v4} is always the largest. Linear hashing and double hashing achieve similar load factors to RHT^{v4} at the cost of up to 500 memory accesses per key-value pair (due to 500 insertion attempts) and several insertion failures. The load factor of other schemes are much lower, with chaining hash being the lowest. Figure 7(b) 7(c) 7(d) shows the load factor of all hashing schemes for 8_1, 128_1, and 192_1 groups of data, respectively, when β varies from 1.05 to 10. We observe from these figures that the load factor of RHT^{v4} is the largest in almost all cases. Linear hashing and double hashing achieve similar load factors at the expense of much higher memory accesses. Peacock hashing achieves high load factors only when β is large, which means that compared to RHT^{v4}, Peacock hashing requires much larger memory footprint to achieve high load factor.

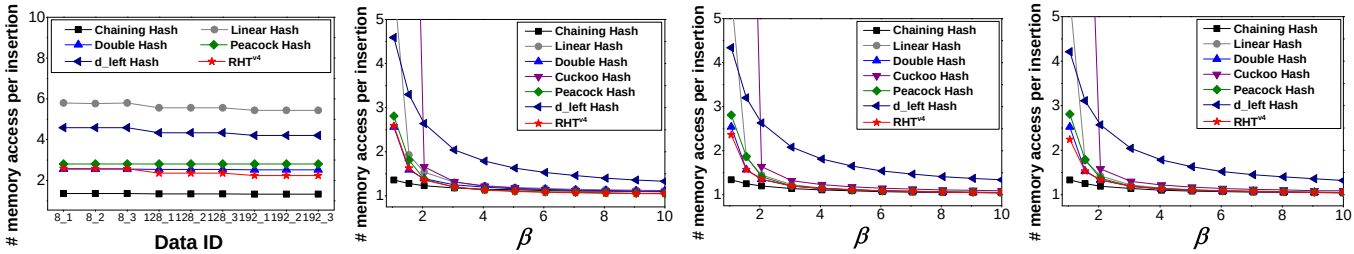
2) Insertion Speed: Figure 8(a) plots the number of memory accesses per insertion during the hash table construction for all hashing schemes when $\beta = 1.05$. We observe from this figure that the chaining hash scheme achieves the fastest insertion speed because each insertion either needs one memory access for empty buckets or two memory accesses for non-empty buckets. RHT^{v4} achieves faster insertion speed compared to all other schemes except chaining hash scheme, owing to the Bloom filters and Bitmaps. The line of cuckoo hashing does not appear in Figure 8(a) because its insertion speed is too slow (around 90 memory accesses per insertion) and lies out of the scale of this figure. Figures8(b), 8(c), and 8(d) plot the number of memory accesses per insertion for 8_1, 128_1, and 192_1 groups of data, respectively, during the hash table construction for all schemes when β varies from 1.05 to 10. We observe from this figure that the chaining hash again achieves the lowest insertion time, followed by RHT^{v4} and double hashing. Cuckoo hashing and linear hashing suffer from very slow insertion speed when β is small.

3) Query Speed: Figure 9(a) plots the number of memory accesses per query for all schemes when $\beta = 1.05$. We observe from this figure that RHT^{v4} achieves the fastest query speed, owing to the high load factor and small false positive rate of Bloom filters. Cuckoo hashing performs very well too, as it only needs at most two memory accesses per search. Figures 9(b), 9(c), and 9(d) plot the number of memory accesses per query for all schemes when β varies from 1.05 to 10. We observe from these figures that RHT^{v4} achieves the fastest query speed, followed by cuckoo hashing, chaining hash, and



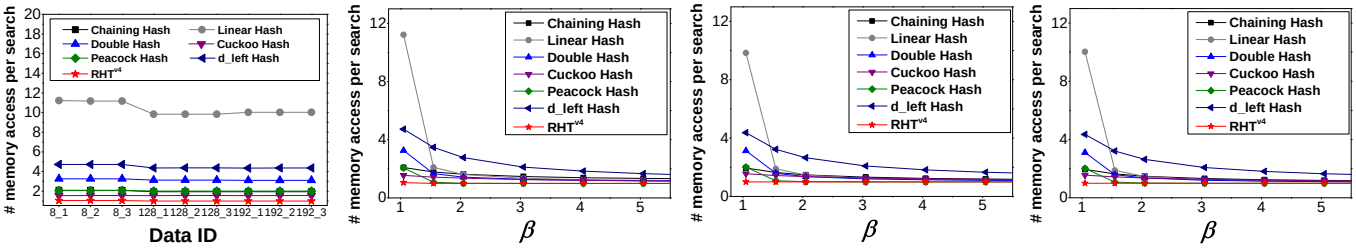
(a) Load factor ($\beta = 1.05$, $\theta = 1$) (b) Load factor ($\theta = 1$, $8Mi$) (c) Load factor ($\theta = 1$, $128Mi$) (d) Load factor ($\theta = 1$, $192Mi$

Fig. 7. Comparison of load factor of RHT^{v4} with prior 6 hashing schemes using three scales of datasets.



(a) Insertion time ($\beta = 1.05$, $\theta = 1$) (b) Insertion time ($\theta = 1$, $8Mi$) (c) Insertion time ($\theta = 1$, $128Mi$) (d) Insertion time ($\theta = 1$, $192Mi$)

Fig. 8. Comparison of insertion time of RHT^{v4} with prior 6 hashing schemes using three scales of datasets.



(a) Search time ($\beta = 1.05$, $\theta = 1$). (b) Search time ($\theta = 1$, $8Mi$) (c) Search time ($\theta = 1$, $128Mi$) (d) Search time ($\theta = 1$, $192Mi$)

Fig. 9. Comparison of search time of RHT^{v4} with prior 6 hashing schemes using three scales of datasets.

double hashing. The search time of linear hashing is very high when β is small.

D. Cost of Using Bloom Filters

Among the seven hashing schemes we have evaluated above, RHT^{v4} and peacock hashing both use Bloom filters. The use of Bloom filters require some extra memory. If we use 16 hash functions for BFs and 8 sub-tables, the false positive rate of RHT^{v4} goes as low as $6.1 * 10^{-5}$ (see section II-D1 for calculation) and each item needs $16/\ln 2 = 22.86$ bits. Peacock hashing requires $z-1$ Bloom filters of different sizes, while RHT^{v4} only requires two Bloom filters.

IV. Related Work

Hash tables have been extensively studied in literature and discussed in Section I-B. Next, we briefly describe only notable hash tables. A more thorough survey of prior hashing schemes is available in [16] by Kirsch *et al.*

A. MHT and MHT^{BF}

Azar *et al.*did the seminal work on multi-choice hashing [4]. They proved a powerful allocation result, which states that if n balls are placed sequentially into $m \geqslant n$ bins and if each ball has $k \geqslant 2$ candidate bins, then with high probability, the maximal load in a bin after all balls are inserted is $(\ln \ln n)/\ln k + \mathcal{O}(1)$. Vöcking presented a notable improvement of Azar's work, namely d_left hashing [34], which uses k equally sized sub-tables, where each table has its own hash function and each bucket in each sub-table supports chaining. While d_left hashing achieves lower hash collision rate and balances the load factor of sub-tables well, it only works efficiently when implemented in pipeline or parallel.

B. KHT

Cuckoo hashing is a notable KHT that utilizes kicking and sustains a high load factor [24]. It is used in applications such as cuckoo filter [10], cuckoo switch [40], and memc3 [9]. To insert a key-value pair, it computes two hash positions using two independent hash functions and inserts the pair in the empty bucket corresponding to either of the two hash positions. If neither of the two buckets are empty, it randomly chooses one of the buckets, kicks its key-value pair out, and replaces it with the new key-value pair. Afterwards, it inserts the kicked out key-value pair into its other candidate bucket, if that bucket is empty. It repeats this kicking process up to 500 times. If after 500 kicks, a key-value pair is still not inserted successfully, it discards the pair, and reports an insertion failure.

V. Conclusion

Hash tables have become indispensable due to their use in a large number of practical applications such as key-value stores, NLP, IP lookups, packet classification, load balancing, TCP/IP state management, and intrusion detection. For this reason, designing efficient hash tables is of paramount importance. In this paper, we propose Rectangular Hash Table (RHT^{v4}), which uses Bloom filters to significantly reduce the hash collision rate, a kick mechanism to achieve high load factor, and bitmaps to significantly accelerate the kick mechanism. Theoretical analysis and experimental results show that our proposed hash tables achieve a very high load factor of up to 96% and a very low hash collision rate, which is much smaller than those of the state-of-the-art hashing schemes. Compared to the state-of-the-art, RHT^{v4} also achieves significantly faster insertion and query speeds without experiencing insertion failures.

ACKNOWLEDGEMENTS

This work is partially supported by Primary Research & Development Plan of China (2016YFB1000304), National Basic Research Program of China (2014CB340400), NSFC (U1536201, 61672061), the Open Project Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences, National Science Foundation, USA under Grant Numbers CNS-1616273 and CNS-1616317.

References

[1] Souce code of RHT and previous 6 hashing schemes. https://github.com/opensrccc/RectangleHash-Table.

[2] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen. Parallel randomized load balancing. In *Proceedings of ACM symposium on Theory of computing*, pages 238–247. ACM, 1995.

[3] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM journal on computing*, 29(1):180–200, 1999.

[4] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM journal on computing*, 29(1):180–200, 1999.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] B. F. Cooper, A. Silberstein, and et al. Benchmarking cloud serving systems with YCSB. In *Proc. ACM SOCC*, 2010.

[7] T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.

[8] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, 1992.

[9] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 385–398, 2013.

[10] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proc. ACM CoNEXT*, 2014.

[11] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[12] M. L. Fredman and J. Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic Discrete Methods*, 5(1):61–68, 1984.

[13] A. Goyal, H. Daumé III, and S. Venkatasubramanian. Streaming for large scale nlp: Language modeling. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 512–520. Association for Computational Linguistics, 2009.

[14] P. Gupta and N. McKeown. Algorithms for packet classification. *Network, IEEE*, 15(2):24–32, 2001.

[15] A. Kirsch and M. Mitzenmacher. Simple summaries for hashing with choices. *Networking, IEEE/ACM Transactions on*, 16(1):218–231, 2008.

[16] A. Kirsch, M. Mitzenmacher, and G. Varghese. Hash-based techniques for high-speed packet processing. In *Algorithms for Next Generation Networks*, pages 181–218. Springer, 2010.

[17] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2009.

[18] D. E. Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[19] S. Kumar and P. Crowley. Segmented hash: an efficient hash table implementation for high performance networking subsystems. In *Proc. ACM ANCS*, pages 91–103, 2005.

[20] S. Kumar, J. Turner, and P. Crowley. Peacock hashing: Deterministic and updatable hashing for high performance networking. In *Proc. IEEE INFOCOM*, 2008.

[21] D. Li, J. Li, and Z. Du. Deterministic and efficient hash table lookup using discriminated vectors. In *Proc. IEEE GLOBECOM*, 2016.

[22] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. *management*, 15(32):36, 2014.

[23] M. Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1094–1104, 2001.

[24] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[25] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[26] D. Ravichandran, P. Pantel, and E. Hovy. Randomized algorithms and nlp: using locality sensitive hash function for high speed noun clustering. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 622–629. Association for Computational Linguistics, 2005.

[27] D. Sarang, K. Praveen, and T. D. E. Longest prefix matching using bloom filters. In *Proc. ACM SIGCOMM*, pages 201–212, 2003.

[28] D. V. Schuehler, J. Moscola, and J. Lockwood. Architecture for a hardware based, tcp/ip content scanning system [intrusion detection system applications]. In *Proc. IEEE HPI*, pages 89–94, 2003.

[29] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. *ACM SIGCOMM Computer Communication Review*, 35(4):181–192, 2005.

[30] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 135–146. ACM, 1999.

[31] D. E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys (CSUR)*, 37(3):238–275, 2005.

[32] B. Van Durme and A. Lall. Online generation of locality sensitive hash signatures. In *Proceedings of the ACL 2010 Conference Short Papers*, pages 231–235. Association for Computational Linguistics, 2010.

[33] B. Vöcking. How asymmetry helps load balancing. *Journal of the ACM (JACM)*, 50(4):568–589, 2003.

[34] B. Vöcking. How asymmetry helps load balancing. *Journal of the ACM (JACM)*, 50(4):568–589, 2003.

[35] G. R. Wright and W. R. Stevens. *TcP/IP Illustrated*, volume 2. Addison-Wesley Professional, 1995.

[36] T. Yang, R. Duan, J. Lu, S. Zhang, H. Dai, and B. Liu. Clue: Achieving fast update over compressed table for parallel lookup with reduced dynamic redundancy. In *IEEE International Conference on Distributed Computing Systems*, pages 678–687, 2012.

[37] T. Yang, A. X. Liu, M. Shahzad, D. Yang, Q. Fu, G. Xie, and X. Li. A shifting framework for set queries. *IEEE/ACM Transactions on Networking*, PP(99):1–16, 2017.

[38] T. Yang, A. X. Liu, M. Shahzad, Y. Zhong, Q. Fu, Z. Li, G. Xie, and X. Li. A shifting bloom filter framework for set queries. *Proceedings of the Vldb Endowment*, 9(5):408–419, 2016.

[39] T. Yang, G. Xie, Y. B. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee ip lookup performance with fib explosion. In *Proc. ACM Sigcomm*, pages 39–50, 2014.

[40] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proc. ACM CoNEXT*, 2013.