Computer Science • 26(2) 2025 https://doi.org/10.7494/csci.2025.26.2.6361

Rajeev Ranjan Kumar Tripathi
Pradeep Kumar Singh
Sarvpal Singh

STABLE AND LOW ASSOCIATIVE LEFT-RIGHT HASHING

Abstract *Hashing is indispensable for efficient search operations, captivating the interest of numerous researchers. Among the diverse array of techniques, Cuckoo Hashing has emerged as particularly effective across a wide range of applications. Nonetheless, Cuckoo Hashing encounters significant challenges, including high insertion latency, inefficient memory usage, and high data migration costs. The concept of Combinatorial Hashing has inspired this research. Our proposed scheme enhances Combinatorial Hashing and introduces an innovative collision resolution technique called Left-Right Random Probing based on Random Probing. This paper introduces two performance indicators, the degree of dexterity and table reference count per key. This paper identifies and quantifies the switching cost as a new challenge in Cuckoo Hashing. The space complexity and insertion latency of proposed scheme is* 1.7 *times and* 1.5 *times better than Cuckoo Hashing respectively. Proposed scheme is* 1.35 *times faster than Cuckoo Hashing and its time complexity is nearly same as Cuckoo Hashing.*

Keywords Cuckoo Hashing, combinatorial hashing, switching cost, degree of dexterity, random probing

Citation Computer Science 26(2) 2025: 151–189

Copyright © 2025 Author(s). This is an open access publication, which can be used, distributed and reproduced in any medium according to the Creative Commons CC-BY 4.0 License.

1. Introduction

In the realm of computing paradigms, all search methodologies can be elegantly classified into three distinct categories: **planning based search**,**data driven search** and **knowledge enhanced search**. From a user's point of view, search operation can be classified into two different ways: **deterministic search** and **abstract search**. This work is dedicated to **data driven search** from **computing paradigm's view** and **deterministic search** from **user's view** [16, 63, 64]. There is a substantial volume of data flow as a result of the ongoing and rapid development in the area of the Internet of Things (IoT), Mobile Computing, Cloud Computing, Distributed Systems, and Social Media. Data is being created at an all-time high rate. Data may be in forms like: *text, audio, video, graphics, images, and animation.* Data centres are instances of this success story since they show how much progress has been made in data storage methods. The common operations that are carried out on stored data are insertion, deletion, update, and search [2, 11, 20, 42]. *A classical problem in Computer Science is to store information in such a way that, on-demand quick lookup could be performed* [10, 18, 43, 62]. This type of search is quite often into data dictionaries, symbol tables maintained by compilers and data-based industries [23, 34, 46].

Hash function $\mathcal{H}$, receives a key, $\mathcal{K}$, as input and produces a hash code, $\mathbf{h}$, as $\mathcal{H}(\mathcal{K})$. The hash code, $\mathbf{h}$ is used as an index in the hash table to store the $\mathcal{K}$. Thus, the mapping of $\mathcal{K}$ to $\mathbf{h}$ is performed and to search a key, a single lookup operation is required in the hash table [28]. Hash function inherently suffers from two challenges: a) **Collision** (when two unique keys, $\mathcal{K}_1$ and $\mathcal{K}_2$ are mapped to the same hash code) and b) **Memory Utilization** (the hash code must be a valid index in the hash table and memory utilization of the hash table should be high) [30, 52]. To deal with the collision, the hash function uses collision resolution techniques like *open addressing (Linear Probing, Quadratic Probing, Random Probing, and Cuckoo Hashing)* and *separate chaining* [19, 22, 26, 27, 35, 46]. In the presence of collision, collision resolution techniques place the key to some other location [51]. A general practice is to use a hush function with a collision resolution technique. Placement of a key to some other location by a collision resolution technique increases *average search length* and reduces the performance of a hashing scheme. Insertion and searching operations are expected to occur in $O(1)$ when the hashing technique is used. The collision resolution technique increases the cost of both operations: *insertion* and *searching.* Under the open addressing scheme, **random probing** or **uniform hashing** is the less addressed collision resolution technique [9, 29, 50, 60].

1.1. Major contributions

The major contributions of this work are as follows:

- Hashing can be applied on both keys: **primary Keys** and **secondary Keys**. Combinatorial hashing was originally proposed for secondary keys. To the best of our knowledge, this is the first work to use a variant of **Combinatorial Hashing** on the **primary key.**

- Instead of using **Chaining** in the **Combinatorial Hashing** as a collision resolution technique, we are the first to use **Left-Right Random Probing** with Combinatorial hashing. *Left-Right Random Probing, is a variant of random probing in which search of an empty location is performed on both sides from the point of collision.* This approach better utilizes the slots of the hash table in contrast with the existing approaches. Authors are the first to use **prime numbers** and **Fibonacci numbers** in *Left-Right Random Probing* as *pseudo random numbers* that a conventional random probing used [40].
- A new performance indicator, **degree of dexterity**(η), is proposed for the hashing schemes. The η is the reciprocal of the sum of *insertion latency* and *average searching time*. Insertion latency is the time consumed in the insertion of all keys in the hash table, whereas average searching time is the time required to search all keys stored in the hash tables.
- This paper identifies, *switching cost*, as a challenge of the Sequential Cuckoo Hashing. The Table Reference Count per Key **(T.R.C./Key)** is another proposed performance indicator that estimates the **switching cost** incurred in searching a key when a hashing technique uses more than one hash table.

2. Literature survey

The most common practice is to use the hashing technique with some collision resolution techniques. The Section 2.1 is committed to conventional hash functions. Section 2.2 is devoted to *Linear Probing* and *Random Probing*. Section 2.3 sheds light on *Quadratic Probing* and *Double Hashing*. Cuckoo Hashing is discussed in detail under Section 2.4 while challenges associated with Cuckoo Hashing are discussed in section 2.5. Section 2.6 describes *Referential Stability* and Section 2.7 deals with *Low Associativity*.

2.1. Conventional hash functions

- *Division Method* is particularly easy because of its simplicity. In this method, the remainder division is performed between key, k, and some values of M. If M is chosen wisely, this method gives better results in comparison to other hash functions. Value of M must be selected as a prime number which is closer to the size of the table,n, and $M \leq n$. Prime numbers uniformly distribute the keys. There are two variants of this method and the selection of the appropriate version is based on the starting address of the table. Prime number, M, uniformly distributes the keys over the addresses available [51, 57].

$$H(k) = \begin{cases} k \mod m & \textbf{if} \quad \text{address starts from 0} \\ \\ (k \; modm) + 1 & \textbf{if} \quad \text{address starts from 1} \end{cases} \tag{1}$$

- *Mid Square Method* squares the key, k and then the same number of digits are discarded from the left and right sides of k^2 and the rest of mid terms are used as address. Digits that are being discarded from both ends must be the same in numbers and it is decided by the fact that how many digits are there in the address offered by the table [37].
- *Folding Method* is dependent on the number of digits, r, offered by the table in its addresses. In this method, the key is partitioned into groups of r digits starting from left to right as k_1,k_2,..k_r. On requirement, padding is performed into the last group from left.Finally all group k_1,k_2,..k_r are added by discarding the last carry [37].

$$H(k) = k_1 + k_2 + ... + k_r \tag{2}$$

- *Universal Family of Hashing*, H, is a set of hash functions,$H_{i \in \mathbb{Z}}$ that map given universe,$\mathbf{U}$, of keys to the addresses in the range $\{1, 2, 3, ...m - 1\}$. For any chosen hash function, $H_i \in \mathbf{H}$ on random basis and any pair of keys, x and y,$\forall\,(x, y) \in \mathbf{U}$, probability of collision, $\mathbf{P_r}\left(\mathbf{H_1}\left(\mathbf{x}\right) = \mathbf{H_1}\left(\mathbf{y}\right)\right)$, is $\frac{1}{m}$.

$$\mathbf{P_r}\left(\mathbf{H_1}\left(\mathbf{x}\right) = \mathbf{H_1}\left(\mathbf{y}\right)\right) = \frac{1}{m} \tag{3}$$

A table with infinite capacity has no issue of collision [8, 25]. In a more general way, collision resolution techniques are divided into three parts: Open Addressing (Linear Probing, Quadratic Probing, Random Probing, Double Hashing, and Cuckoo Hashing), Chaining, and Coalesced Hashing. To measure the performance of the hash function with collision resolution strategies, there are two important parameters: $S(\lambda)$, average number of probes for successful search and $U(\lambda)$, average number of probes for unsuccessful search.

2.2. Linear probing and random probing

In linear probing, a new location of k_2 is searched from H(k_2) in a linear fashion. It is required to move to the right end of the table from H(k_2)+1, H(k_2)+2, ... to search an empty location [14, 21]. If any empty location is found k_2 is stored there else searching is performed again from the start of the table and moves to H(k_2) by increment of 1 in H(k_2). If any vacant position is found,k_2 is placed there, otherwise k_2 is discarded. In linear probing, a search for empty slots linearly takes place and it is time time-consuming task. When load factor(λ) is greater than 0.5, records form clusters. $S(\lambda)$ and $U(\lambda)$ can be expressed in term of load factor (λ) as $S(\lambda) = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)}\right)$ and $U(\lambda) = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$. To minimize this clustering, two variants are proposed and they are: Quadratic Probing and Double Hashing [13]. Similarly, random probing employs a pseudo-random number generator to search for vacant locations. The average number of probes, E, in random probing, as discussed in [40]: $E = -\frac{1}{\lambda} log_e(1 - \lambda)$.

2.3. Quadratic probing and double hashing

Here, the new location of the key,k, is searched in a quadratic manner instead of searching for an empty slot linearly the search is performed as H(k_2)+1, H(k_2)+4, H(k_2)+9, ... If the table offers its size in a prime number, the above sequence accesses half of the locations into the table [41].

Under *Double Hashing*, there is a second hash function, H', which is used to resolve collision. If there is collision for key,k_2 then searching of empty slot will take place in the order, H(k_2) + H'(k_2), H(k_2) + 2H'(k_2), H(k_2) + 3H'(k_2), ... The second hash function, H, should generate a smaller hash code in comparison to the size of the table. So H' should be wisely chosen. If the table offers its size in a prime number, the above sequence accesses all of the locations in the table. Deleting a key into open addressing is a time-consuming task as the key is not at its hash code so before deleting a key from its hash code, a search is always required.$S(\lambda)$ is also known as average search length.$S(\lambda)$ is always 1 in case of the perfect hash function. From a family of hash functions, that hash function is selected for which $S(\lambda)$ is close to 1 when open addressing is concerned. An improvement concept of the bucket is proposed and it further derives the concept of chaining [1].

2.4. Cuckoo Hashing

Randomization is the foundation of the nature-inspired Cuckoo Hashing [47,64]. This hashing scheme uses two tables: denoted as T_1 and T_2 and two hash functions: H_1 and H_2. Cuckoo Hashing can be categorized on the basis of: **way of accessing the hash tables** and **size of hash tables**. If hash tables are accessed in parallel, it is called **Parallel Cuckoo Hashing** as shown in Figure 1.

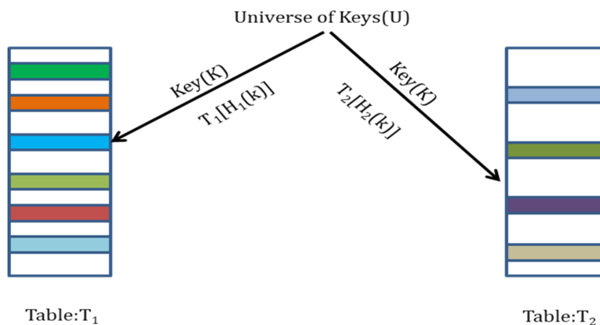


Figure 1. Conceptual view of parallel Cuckoo Hashing with two tables

If hash tables are accessed in sequential manner, it is referred as **Sequential Cuckoo Hashing** as depicted in the Figure 2. In the parallel version of Cuckoo Hashing, both tables are simultaneously consulted in searching for a key while in Sequential Cuckoo Hashing, searching for a key starts from a table and terminates on another table. If both hash tables are of same size, this version of

Cuckoo Hashing is referred as **Symmetric Cuckoo Hashing** else it is known as **Asymmetric Cuckoo Hashing**. Cuckoo Hashing, in its purest form, is **Symmetric Sequential Cuckoo Hashing** [44].

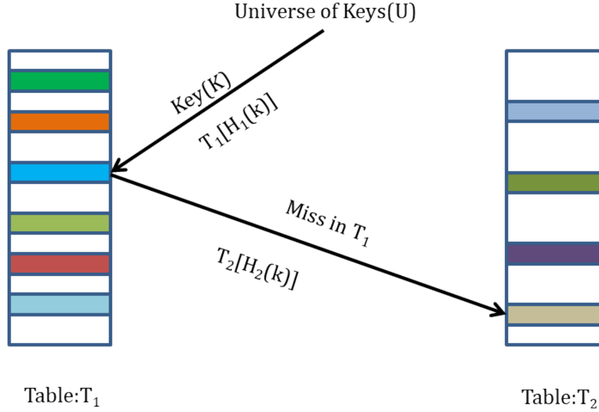


Figure 2. Conceptual view of sequential Cuckoo hashing

A key, K, is stored either at $T_1[H_1(K)]$ or $T_2[H_2(K)]$. Retrieval of K requires a lookup in tables. Pagh and Rodler, inventors of Symmetric Sequential Cuckoo Hashing, suggested the relationship between the size of tables (n) and the keys ($\mathcal{R}$) as, $n \geq (1+\epsilon)\mathcal{R}$, where, ϵ is a constant and $\epsilon \geq 0$ [24, 45]. In case of Asymmetric Sequential Hashing, size of a hash table is more than twice of another hash table. When a collision is observed in a table, to make room for freshly arriving keys, Cuckoo Hashing begins kicking away existing keys from their places. When a predetermined value of MaxLoop(ML) is reached, the kicking-out operation is terminated. If the kicking-out operation proves ineffective at accommodating the key, the rehash procedure is performed. If T_1 and T_2 are almost half full and hash functions, H_1 and H_2, are chosen from Universal family of hash functions of order $(O(1), O(log_n))$, there is a probability of $O(\frac{1}{n})$ that a key (K) is not placed in both, T_1 and T_2 [8]. Cuckoo Hashing with two tables may offer a load factor of $\frac{1}{2}$. Algorithms for lookup and insertion operations are given below.

Algorithm 1: Lookup function

Input: key:K;
Output: K may be in T_1 or T_2, on success it returns K;

```
1 if K = T1[H1(K)] then
2     return K;
3 if K = T2[H2(K)] then
4     return K;
5 end
```

Procedure insert(K)

```
1  if lookup(K) then
2      return
3  loop MaxLoop times
4      if T_1[H_1(K)] = ⊥
5          T_1[H_1(K)] ← K;
6          return;
7      else
8          K ↔ T_1[H_1(K)];
9      if T_2[H_2(K)] = ⊥
10         T_2[H_2(K)] ← K;
11         return;
12     else
13         K ↔ T_2[H_2(K)];
14 end loop
15 rehash(); insert(K);
16 end
```

2.5. Major challenges of Cuckoo Hashing

Cuckoo Hashing exhibits a notable failure rate in insertion operation, which is an unexpected aspect. Despite this, *rehashing may not be suitable for various applications due to its inability to ensure a fixed time for completing insertion operations.* To address this challenge, the utilization of a stash has been proposed as a recommended solution to enhance performance in insertion failure. A stash of size s reduces the probability of insertion failure from $\Theta(\frac{1}{n})$ to $\Theta(\frac{1}{n^{s+1}})$ [39]. Major challenges of Cuckoo Hashing are [15, 48, 58, 59]:

- **Inefficient Memory Usage:** Cuckoo Hashing requires a larger memory footprint than other techniques due to the use of two hash tables.
- **Data Migration:** In case of collision, data migration takes place and it is performed up to the set value of ML. Data migration takes place again in the rehash operation. Thus, the higher the value of ML, the higher will be the data migration.
- **High Insertion Latency:** The way by which data collision is dealt with by the Cuckoo Hashing, demands high insertion latency.

In K-ary Cuckoo Hashing, K hash functions generate K indices to perform lookup operations in K tables. Thus the probability of finding a key in a table is $\frac{1}{K}$. This approach offers worse-case lookup time and space complexity. In past, this issue was addressed by [61]. Another approach encourages to use of a single table (instead of using multiple tables) which is logically partitioned into $\frac{K}{L}$ logical tables and each logical table has a size of L blocks. This modification avoids the additional costs involved in accessing the tables that report unsuccessful lookup operations and includes only the cost involved in key comparisons [12].

2.6. Referential stability

Stability in a hash table refers to the property where the position of an element, once inserted, remains unchanged unless the element is deleted or the table is resized. This predictability is crucial for certain applications where maintaining the order of elements is important. Without stability, the positions of elements might get shuffled during insertion operations or hash table resizing, making it difficult to rely on the order of elements in the table [3, 54]. If a new insertion is performed, stability in Cuckoo Hashing is compromised.

2.7. Low associativity

It is another important property of hashing. The performance of a hashing technique is an exploitation of the association of a key and its corresponding hash code. A perfect hash function guarantees that every key is at its computed hash code and thus shows a strong association. Practically, hashing techniques are suggested to be used with some collision resolution techniques. In this case, a key is assigned to a location (other than its computed hash code); thus, the *average search length* increases. The increase in the average search length negatively affects the search time. Cuckoo Hashing requires a high "MaxLoop" value to ensure stability and high associativity [3].

3. Motivation and problem formulation

Pagh and Rodler presented the idea of Cuckoo Hashing in ESA 2001, with two hash tables [44]. The authors have used an architecture that did not support the parallelization of the two memory accesses involved in lookups. Hence, we can only assess the second hash function after confirming the failure of the initial memory lookup in Table 1 [45]. Originally, the capacity of each bucket was for storing a single key, *i.e.,* one key can be stored in a bucket of either hash table not simultaneous. If a lookup operation fails, the key is searched in the other hash table using the second hash function. It means, *"both hash tables are sequentially accessed, not in parallel fashion"*. While analysing the performance of Cuckoo Hashing, the amortized searching cost is expressed as $O(1)$ as, at most, two memory locations are required to access to declare whether the search is successful or not. Switching from one table to another table consumes switching overhead and this switching overhead can be best described by the given "Two Shops Example" in the next subsection.

3.1. Combinatorial hashing

The space complexity of a hashing technique is dependent on the size of the hash table. The load factor measures the memory utilization of the hash table. While using the probing technique, the size of the hash table is kept greater than the number of keys. This is done so that the maximum number of keys may be stored in the hash table and loss of keys due to collision may be avoided. This work is inspired by the *Combinatorial hashing technique.* In Combinatorial hashing, the hash function $\mathcal{H}$ is

applied on every individual digit/alphabet of a key. Let $\mathcal{H}$ generate $\mathcal{R}$ digit binary codes for each digit/alphabet, then $\mathcal{RN}$ digit long binary code is there for a key of length $\mathcal{N}$. To store this key, the estimated size of the hash table must be $2^{\mathcal{RN}}$. If each cell of the hash table holds only one key, we can store $2^{\mathcal{RN}}$ keys in the best case if no collision is reported. If the universe of keys is $\mathcal{U}$ and $|\mathcal{U}| = \mathcal{L}$ then $2^{\mathcal{RN}}$ must be adjusted in accordance with $\mathcal{L}$ to use the space of hash table efficiently. *The primary motivation of this work is the mechanism employed in Combinatorial hashing to determine the size of the hash table.* Next, if $2^{\mathcal{RN}}$ is much greater than $\mathcal{L}$, then load factor, $\lambda \to 0$ and memory wastage is observed. The hash table space must be efficiently used ($\lambda \to 1$). This improvement in λ demands a reduction in the size of the hash table. A shrinking constant ($\mathcal{C}$) can be proposed and applied on $2^{\mathcal{RN}}$ to resize hash table as $2^{\mathcal{RNC}}$. The $2^{\mathcal{RNC}}$ is still greater than $\mathcal{L}$ and thus probing can be efficiently performed. The conceptual view of Combinatorial hashing is shown in Figure 3. In this work, *the hash function is applied to the entire key rather than the individual digit/alphabet of the key.* In Combinatorial hashing, Chaining is used as a collision resolution technique. Chaining is an expensive collision resolution technique in terms of time and space complexities. Therefore, Chaining is substituted by Left-Right Random Probing in this work. The Left-Right Random Probing in the current work uses Fibonacci and prime numbers. *This work improves the Combinatorial hashing and develops an efficient hashing scheme, which supersedes the performance of Sequential Cuckoo Hashing in terms of various parameters discussed in the experiments.*

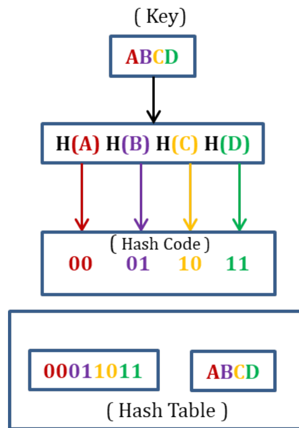


Figure 3. Conceptual view of combinatorial hashing

3.2. Problem formulation

A perfect hash function ensures that each item in a set is uniquely mapped to distinct hash codes by eliminating collisions entirely. Perfect Hashing is memory efficient, and this specialized function is particularly advantageous in scenarios where the set of keys

is known in advance and remains static. It offers fast and efficient lookup operations without the need for collision resolution mechanisms. Conversely, a minimum collision hash function aims to minimize the occurrence of collisions when hashing a set of keys. While it may not entirely eliminate collisions like a perfect hash function, it strategically distributes keys across the hash table to mitigate collisions, optimizing performance by reducing the frequency of collision resolution operations. Both approaches contribute significantly to the efficiency and effectiveness of hash tables, catering to different requirements based on the nature of the keys and the application's demands [4, 31, 33, 36, 55].

Authors decide to develop a stable and low associative hashing scheme based on conventional approaches like: *Combinatorial Hashing*, *Division Methods* and *Random Probing*. We use two asymmetric hash tables where keys are not uniformly distributed among the hash tables and to gauge its performance with best known *High Associative Scheme* and other *Low Associative Hashing Schemes*.

4. Work done: stable and low associative left-right hashing

This section is divided into 7 parts: Section 4.1 sheds light on the *"Theoretical Foundation of the Proposed Work"*. The next section (Section 4.2) describes *"Mathematical Foundation of Proposed Work"*. Section 4.3 describes about the *"Configuration of Low Associative Hashing Scheme* and it gives **Stable and Low Associative Left-Right Hashing"** followed by Section 4.4 that deals with *"Prediction of Collision"*. Switching cost is estimated with the help of a proposed performance indicator called, $\frac{T.R.C.}{Key}$ and it is derived from the " Hit Count and Miss Count". Section 4.5 briefly describes the *"Significance of Hit Count and Miss Count"*. In the last, Section 4.6 and 4.7 measure the performance of algorithms on the basis of *"Space Complexity"* and *"Time Complexity"*, respectively.

4.1. Theoretical foundation of proposed work: two shops example

In a colony, there are two grocery shops run by *Alice* and *Bob*. *Adam* is a customer of both shops and he finds **80**% of his needs in *Alice's* shop (see Fig. 4). The remaining **20**% goods items are only available to the *Bob's* shop. Adam knows these statistics well and he always visits *Alice's* shop first. *Sophia* and *Samuel* are twins in the family of *Adam*. One day, Adam asks *Sophia* and *Samuel* to purchase some grocery items with instruction that, *Sofia* will always start purchasing from *Alice's* shop while *Samuel* from *Bob's* shop and at a time only one item has to be purchased. If an item is not found in the shop then they have to visit another shop. *Adam* provides **10** item-names to both *Sophia* and *Samuel* one by one and starts recording the time consumed in each transaction of *Sophia* and *Samuel*. Undoubtedly, *Sophia* consumes less time than *Samuel* in overall transaction. This is because *Sophia* has visited *Alice's* shop **10** times and only **2** times she has visited *Bob's* shop. Thus, *Sophia* visits both shops for each item, only **1.2** times while *Samuel* consumes **1.8** shop visit per item on average.

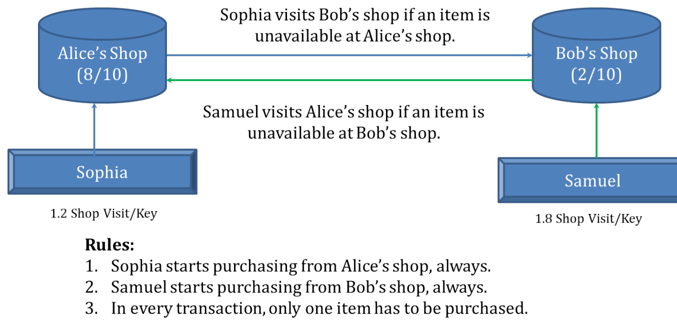


Figure 4. Conceptual view of two shop example

4.2. Mathematical foundation of proposed work

In this section, two hashing schemes are being considered: *High Associative Hashing Scheme* and *Low Associative Hashing Scheme*.

System Model: The system model for the *High Associative Hashing Scheme* and *Low Associative Hashing* is given as:

1. Both hashing schemes are using two hash tables (T_1 and T_2). The size of hash tables in the *High Associative Hashing Scheme* are equal, while unequal sized hash tables are used in the *Low Associative Hashing Scheme*. The larger-sized hash table is referred to as **Primary Table** and the other one as **Backup Table**.
2. In *High Associative Hashing Scheme*, lookup and insertion operations can be initiated either from T_1 or T_2. It means **High Associative Hashing Scheme is unbiased in both operations: lookup and insertion**. While lookup and insertion operations are always initiated from the large-sized hash tables. **Low Associative Hashing Scheme is biased to Primary table in both operations: lookup and insertion.**
3. In *High Associative Hashing Scheme*, all keys are stored at their computed hash codes, while in a low associative hashing scheme, random probing is used to deal with data collision.
4. Both hashing schemes are using two hash functions (H_1 and H_2, for T_1 and T_2 respectively).
5. A key, K, is stored in either a hash table under both hashing schemes. Keys among T_1 and T_2 are not equally distributed. During hashing, both schemes do not offer any data loss.

4.2.1. High associative hashing scheme

Let P_1 and P_2 are the probabilities of finding a key in the tables T_1 and T_2, respectively. According to the **System Model**, $P_1 = (1 - P_2)$. In *High Associative Hashing Scheme* only one probe is required to ensure the presence of the key in a table. Thus

in the worst case, *High Associative Hashing Scheme* requires 2 memory access to ensure that the key is either in T_1 or T_2. Search operation can be initiated either from T_1 or T_2. If cost of this single probe is considered, t, then estimated cost of search, T_{total}, can be computed as $P_1 \times t + P_2 \times (t+t) = P_1 \times t + 2 \times (1-P_1) \times t = t \times (2-P_1)$, when lookup operation starts from T_1. If T_2 is investigated first, $T_{total} = t \times (2-P_2)$. As $P_1 \neq P_2$ thus T_{total} depends on the order of lookup. Table, T_2, is consulted when a *miss* is reported during a search of a key K in the T_1. Transferring of control from T_1 to T_2 involves additional overhead in terms of **switching cost**. If $|\mathcal{U}| = n$, then **T.R.C.** (Total Reference Count) which is the *estimated count access* of both the tables while searching entire $\mathcal{U}$ is $n + n \times (1 - P_1)$ (here, searching starts from T_1). The average *T.R.C.*, denoted as TRC_{avg}, is the ratio of $n + n \times (1 - P_1)$ and $|\mathcal{U}| = n$; hence, TRC_{avg} is $(2 - P_1)$. If $\triangle$ is the switching cost per miss, the estimated switching cost between the tables is $TRC_{avg} \times \triangle$. Let $T_{effective}$ be the overall searching cost incurred in the High Associative Hashing Scheme, which is estimated as $T_{total} + TRC_{avg} \times \triangle = (2 - P_1) \times (t + \triangle)$ for a key if lookup operation starts from T_1.

If the High Associative Hashing Scheme does not provide any prior information about the number of keys stored in the tables, the searching operation may start from any table. Initiation of searching from a table having fewer keys stored leads to worse performance in the High Associative Hashing Scheme.

4.2.2. Low associative hashing scheme

Low Associative Hashing Scheme maintains two tables of different sizes: a primary table and a backup table. To deal with the collision, Random Probing is used in both tables. Average searching length in Random Probing is $\frac{-\mathbf{1}}{\lambda}\mathbf{log_e}(\mathbf{1} - \lambda)$, where λ is the load factor of a hash table.

Next, we estimate the search cost in the Low Associative Hashing Scheme. Let $\mathbf{Pr}_{primary}$ and $\mathbf{Pr}_{backup}$ are the probability of finding a key in the primary and the backup tables, respectively, where $\mathbf{Pr}_{backup} = (1 - \mathbf{Pr}_{primary})$. Let $\lambda_{primary}$ and λ_{backup} be the load factors of the primary and backup tables, respectively. The estimated searching time, denoted as $T_{proposed}$, of a key in the *Low Associative Hashing Scheme*, is $\mathbf{Pr}_{primary} \times (\frac{-1}{\lambda_{primary}} log_e(1 - \lambda_{primary})) \times t + \mathbf{Pr}_{backup} \times (\frac{-1}{\lambda_{backup}} log_e(1 - \lambda_{backup})) \times 2t$. The **T.R.C.**(Total Reference Count) in the *Low Associative Hashing Scheme* for entire $\mathcal{U}$ is $n + n \times (1 - \mathbf{Pr}_{primary})$. The average *T.R.C.* for the *Low Associative Hashing Scheme*, TRC_{pavg}, is the ratio of $n + n \times (1 - \mathbf{Pr}_{primary})$ and $|\mathcal{U}| = n$. Thus, TRC_{pavg} is $(2 - \mathbf{Pr}_{primary})$. The effective searching time for a key in the *Low Associative Hashing Scheme*, denoted as **EST**, is the sum of $T_{proposed}$ and $TRC_{pavg} \times \triangle$, *i.e.*, $\mathbf{Pr}_{primary} \times (\frac{-1}{\lambda_{primary}} log_e(1 - \lambda_{primary})) \times t + \mathbf{Pr}_{backup} \times (\frac{-1}{\lambda_{backup}} log_e(1 - \lambda_{backup})) \times 2t + (2 - \mathbf{Pr}_{primary}) \times \triangle$.

The *Low Associative Hashing Scheme* appears potentially better than the *High Associative Hashing Scheme* under the assumption of $\mathbf{Pr}_{primary} > P_1$ and

$\mathbf{Pr}_{primary} > P_2$ with restricted average search length in both the primary table and the backup table.

4.3. Configuring low associative hashing scheme: stable and low associative left-right hashing

1. **Universe of Keys:** Authors have considered universe of keys ($\mathcal{U}$) as $|\mathcal{U}| = 10^6$. The size of a key is 15 digits long. If $\mathcal{R} = 2$ is considered then hash code of a key is 30 bits long and it covers total (2^{30}) $1,07,37,41,824$ memory locations, which is 1074 times of the $\mathcal{U}$.
2. **Defining the sizes of hash tables**: *Stable and Low Associative Left-Right Hashing* uses, two hash tables. To use memory efficiently, the original idea of Combinatorial Hashing has been revised, as discussed in Section 3. The hash function **H** spreads the $\mathcal{U}$ over $2^{\mathcal{RNC}}$ bit long hash code, where $\mathcal{C}$ is a constant called, **shrinking constant**, whose value is adjusted according to $\mathcal{U}$. This revision in the Combinatorial Hashing has resemblance as proposed by Rivest [53] and A.E. Brouwer [7]. The $\mathbf{Prime}(\mathcal{X})$, returns a prime number which is the closest to and greater than $\mathcal{X}$. Size of primary table is kept as $\mathbf{Prime}(2^{\mathcal{RNC}})$ while size of backup table is maintained, as $\mathbf{Prime}(\frac{1}{f}2^{\mathcal{RNC}})$ as stated in Section 3. Stable and Low Associative Left-Right Hashing first consults with the *primary table* in case of both: *insertion* and *search*. If the lookup operation in the primary table fails, *backup table* is consulted. In this manner, Stable and Low Associative Left-Right Hashing reduces the involved switching cost. As the size of the primary table is much more than the backup table, it responds more than 95% lookup operations positively. Thus, the probability of finding a key in the primary table is higher.
3. **Left-Right Random Probing:** Searching time is directly proportional to the average search length. To maintain the small average search length, authors have used a small set of prime numbers and Fibonacci numbers, **P** and **F** respectively. *In random probing, the search of an empty location is performed in a single direction unless one end is met.* In the proposed scheme, search for an empty location is performed on both sides: the left side and the right side. From the point of collision, searching for an empty slot takes place on the left side and the right side (if the previous one is found unsuccessful). The number of probes consumed in searching empty slots are at most $2 \times |\mathbf{P}|$ or $2 \times |\mathbf{F}|$ (depends on the set used) in the Left-Right Random Probing.
4. **Hash Functions Used:** Stable and Low Associative Stable Left-Right Hashing uses two hash functions (one hash function for each hash table). These hash functions are based on *Division Method*. The hash function $K\%\mathbf{Prime}(\mathbf{2}^{\mathcal{RNC}})$ is used for primary table and $K\%\mathbf{Prime}(\frac{\mathbf{1}}{\mathbf{f}}\mathbf{2}^{\mathcal{RNC}})$ is used for backup table.

The proposed scheme uses Left-Right Random Probing as a collision resolution technique; thus, it is "low associative" in nature. Once a key is stored in a table, it is never migrated; thus, it is "stable" in nature too. On the basis of 1) stability, 2) low

associative nature, and 3) probing technique used, the proposed scheme is referred to as "Stable and Low Associative Left-Right Hashing". The conceptual view of Stable and Low Associative Left-Right Hashing is shown in Figure 5. Algorithm 2 summarizes different steps involved in the proposed scheme.

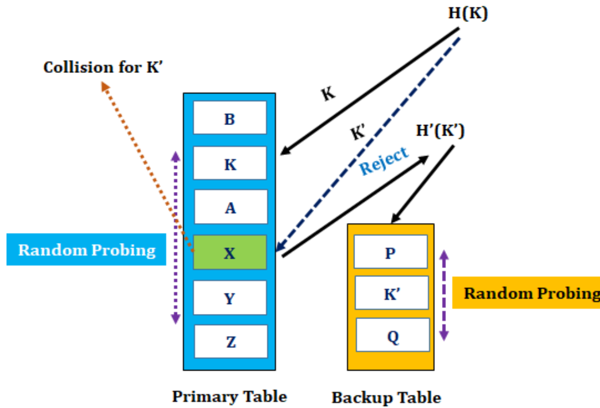


Figure 5. Conceptual view of stable and low associative left-right hashing

Algorithm 2: Stable and Low Associative Left-Right Hashing

```
1  Initialize: PT_SIZE, BT_SIZE, P_DIVISOR, & B_DIVISOR;
2  Function Table_Initialization( )
3    for i = 0 to PT_SIZE do
4        Primary_Table[i] = −99;
5    for i=0 to BT_SIZE do
6        Backup_Table[i] = −99;
7  Function Primary_Hash_Code(key) return (key mod P_DIVISOR);
8  Function Backup_Hash_Code(key) return (key mod B_DIVISOR);
9  Function Insert_Backup_Table(key, bhashCode);
10   if Backup_Table[bhashCode]= −99 then
11       Backup_Table[bhashCode]=key;
12   else
13       Perform random probing to store the key;
14 Function Insert_Primary_Table(key)
15   phashCode = primary_Hash_Code(key)
16   if Primary_Table[phashCode]= −99 then
17       Primary_Table[phashCode]= key;
18   else
19       if Random Probing is successful then
20           key is stored.
21       else
22           Insert_Backup_Table(key, bhashCode);
```

Algorithm 2: (continued)

```
24 Function Search_Backup_Table(key, bhashCode)
25     if Backup_Table[bhashCode] = key then
26         return 1;
27 end
28 else
29         Perform random probing to search the key;
30 end
31 Function Search_Primary_Table(key)
32     phashCode = primary_Hash_Code(key)
33     if Primary_Table[phashCode] = key then
34         return 1;
35 end
36 else
37         if random probing is successful then
38                 return1
39     end
40         else
41         bhashCode=backup_Hash_Code(key)    search_Backup_Table(key,bhashCode)
42     end
43 end
```

4.4. Prediction of collision

Bin and Ball model can be used to estimate an average number of collisions. If there are $\mathcal{X}$ balls and $\mathcal{Y}$ bins and $\mathcal{Y} > \mathcal{X}$. The $\mathcal{X}$ balls are randomly thrown into the $\mathcal{Y}$ bins. The first ball is placed in a bin, and the remaining ($\mathcal{X}$-1) balls have an equal probability of falling into the same bin is $\frac{1}{\mathcal{X}}$. Thus, the average number of collisions with the first ball will be $\frac{(\mathcal{X}-1)}{\mathcal{Y}}$. The second ball falls into the same bin, and the remaining ($\mathcal{X}$-2) balls have an equal probability of falling into the same bin is $\frac{1}{\mathcal{X}}$. Thus, average number of collisions, denoted as Avg_{cols}, with the second ball will be $\frac{(\mathcal{X}-2)}{\mathcal{Y}}$.

$$Avg_{cols} = \sum_{i=1}^{(\mathcal{X}-1)} \frac{i}{\mathcal{Y}} = \frac{\mathcal{X}(\mathcal{X}-1)}{2\mathcal{Y}} \tag{4}$$

4.5. Significance of hit count and miss count

Sequential Cuckoo Hashing distributes keys in between the tables equally likely. If the lookup operation in a table fails, Sequential Cuckoo Hashing computes hash code using other hash functions to perform the lookup operation in the second table. The probability of finding a key in a table is $\frac{1}{2}$ in Cuckoo Hashing [49]. In the proposed scheme, the probability of finding a key in the primary table is much higher than the backup table. As keys are unlikely distributed between the primary table and backup table, the backup table is only consulted when a failure either in search operation or insertion operation is observed. Frequent computation of hash codes

and context switches in between the tables, downgrade the searching performance when a large number of keys are searched. To include these computational overheads in determining the performance of both searching cost and insertion latency hit counts and miss counts of tables must be considered. Hit ratio is the probability of success in searching a key while miss ratio shows the chance of an unsuccessful search in a table. For a given $\mathcal{U}$, we can compute **Table Reference Count (T.R.C.)** while adding the hit counts and miss counts of the individual hash tables participating in a hashing scheme. *The* $\frac{T.R.C.}{key}$ *is the ratio of T.R.C. to the total number of keys.*The $\frac{T.R.C.}{key}$ tells *"how many reference counts of tables are required to search a key on average basis"*. Universal Hash Functions report the loss of a key with a probability of $\frac{1}{m}$ when $|\mathcal{U}|$ is m. Sequential Cuckoo Hashing uses Universal Hash Functions thus, **miss counts of Table$_1$ $\neq$ (m-hit counts of Table$_2$)**.

4.6. Space complexity

The size of a hash table in the High Associative Hashing Scheme is marked as standard to estimate the space complexity of the proposed scheme. If the size of a hash table in the High Associative Hashing Scheme is $\mathcal{S}$, the space complexity of the High Associative Hashing Scheme is $O(2\mathcal{S})$. The size of the primary table and backup table in Left-Right Hashing is $C_1\mathcal{S}$ and $C_2\mathcal{S}$ respectively where C_1 and C_2 are constants with values, 1.048 and 0.13. Thus, the space complexity of Left-Right Hashing is O($\mathcal{CS}$) with constant $\mathcal{C} = 1.18$. **Space complexity of Left-Right Hashing outperforms High Associative Hashing Scheme by a factor of 1.7**.
Lemma 1. *A high Associative Hashing Scheme offers a load factor less than* $\frac{1}{2}$.

Proof. Let the total number of keys be n. The size of a hash table is $n + x$; that is, the hash table is larger than the total number of keys. Load factor (λ) is the "ratio of total number of keys stored to the total number of locations offered in the hash table(s)." The load factor $\lambda = f(n)$. High Associative Hashing Scheme uses two hash tables thus $f(n) = \frac{n}{2\times(n+x)}$, As, $x \neq 0$, $f(n) < \frac{1}{2}$. □

4.7. Time complexity

Theorem 1. *Left-Right Hashing with Random Probing using Prime Numbers and Fibonacci Numbers performs searching in constant time.*

Proof. Let the probability of finding a key in the primary table and backup table be $\mathbf{Pr}_{primary}$ and $\mathbf{Pr}_{backup}$ respectively. The proposed scheme first attempts to accommodate the key in the primary table, if insertion is unsuccessful in the primary table then the key is accommodated in the backup table. In the proposed scheme, $\forall Key(k) \in \mathcal{U}$ is successfully stored and thus $\mathbf{Pr}_{backup} = (1 - \mathbf{Pr}_{primary})$. Let $\lambda_{primary}$ and λ_{backup} be the load factors of the primary table and backup table respectively. The $T_{proposed}$, estimated searching time of a key in the proposed scheme, is $\mathbf{Pr}_{primary} \times (\frac{-1}{\lambda_{primary}} log_e(1-\lambda_{primary})) \times t + \mathbf{Pr}_{backup} \times (\frac{-1}{\lambda_{backup}} log_e(1-\lambda_{backup})) \times 2t$. The **T.R.C.**(Total Reference Count) in the proposed scheme for entire $\mathcal{U}$ is $n + n \times (1 - \mathbf{Pr}_{primary})$. The average $T.R.C.$ for the proposed scheme, $TRC_{pavg} = \frac{T.R.C.}{|\mathcal{U}|}$,

is $\frac{n+n\times(1-\mathbf{Pr}_{primary})}{n}$ as $|\mathcal{U}| = n$. Thus TRC_{pavg} is $(2 - \mathbf{Pr}_{primary})$. The effective searching time for a key, **EST**, is the sum of $T_{proposed}$ and $TRC_{pavg} \times \triangle$.

$$\mathbf{EST} = \mathbf{Pr}_{primary} \times \frac{-1}{\lambda_{primary}} log_e(1-\lambda_{primary}) \times t + \mathbf{Pr}_{backup} \times (\frac{-1}{\lambda_{backup}} log_e(1-\lambda_{backup})) \times 2t + (2 - P_{primary}) \times \triangle \quad (5)$$

In the Equation 5, $\mathbf{Pr}_{primary}, \mathbf{Pr}_{backup}, \lambda_{primary}$, and λ_{backup} are all constants for given $\mathcal{U}$ in the proposed scheme. The cost of a single probe (t) and switching cost ($\triangle$) are constant for a given machine. Thus the proposed scheme performs searching in constant time. The experimental proof is given in Figure 14. □

Lemma 2. *High Associative Hashing Scheme offers two different* $\frac{T.R.C.}{key}$ *for* $\mathcal{U}$.

Proof. Let $\mathbf{P}_1$ and $\mathbf{P}_2$ be the probabilities of finding a key in the tables, $\mathbf{Table}_1$ and $\mathbf{Table}_2$ in High Associative Hashing Scheme. Searching may start either from $\mathbf{Table}_1$ or $\mathbf{Table}_2$. Let, Sequential Cuckoo Hashing offers two different $\frac{T.R.C.}{|\mathcal{U}|}$ for $|\mathcal{U}| = n$ as $\mathbf{T.R.C}_{avg1}$ and $\mathbf{T.R.C}_{avg2}$ when searching starts from $\mathbf{Table}_1$ and $\mathbf{Table}_2$, respectively. The $\mathbf{T.R.C.}_1$ is, $n + n \times \mathbf{P}_2$ and The $\mathbf{T.R.C.}_2$ is $n + n \times \mathbf{P}_1$. Thus $\mathbf{T.R.C}_{avg1}$ and $\mathbf{T.R.C}_{avg2}$ are $(1+\mathbf{P}_2)$ and $(1+\mathbf{P}_1)$. As $P_1 \neq P_2$, thus $\mathbf{T.R.C}_{avg1} \neq \mathbf{T.R.C}_{avg2}$. Hence, High Associative Hashing Scheme offers two different $\frac{T.R.C.}{key}$ for given $\mathcal{U}$. The experimental proof is shown in Figure 11. □

Theorem 2. *For given* $\mathcal{U}$*, High Associative Hashing Scheme offers two searching times.*

Proof. Let $\mathbf{P}_1$ and $\mathbf{P}_2$ be the probabilities of finding a key in tables, $\mathbf{Table}_1$ and $\mathbf{Table}_2$, and $\mathbf{P}_1 \neq \mathbf{P}_2$. If searching starts from $\mathbf{Table}_1$, the searching time $\mathbf{T}_1$ is $\mathbf{P}_1 \times t + \mathbf{P}_2 \times (2t) = \mathbf{P}_1 \times t + (1-\mathbf{P}_1) \times (2t) = (2-\mathbf{P}_1) \times t$. If searching starts from $\mathbf{Table}_2$, the searching time $\mathbf{T}_2$ is $\mathbf{P}_2 \times t + \mathbf{P}_1 \times (2t) = \mathbf{P}_2 \times t + (1-\mathbf{P}_2) \times (2t) = (2-\mathbf{P}_2) \times t$. Now, switching cost is considered and it has to be added with $\mathbf{T}_1$ and $\mathbf{T}_2$ to estimate the effective searching time, $\mathbf{EST}_1$ and $\mathbf{EST}_2$. From lemma 2, $\mathbf{T.R.C.}_{avg1}$ and $\mathbf{T.R.C.}_{avg2}$ are $(1+\mathbf{P}_1)$ and $(1+\mathbf{P}_2)$.Thus $\mathbf{EST}_1$ is $\mathbf{T}_1 + \mathbf{T.R.C.}_{avg1} \times \triangle$ and $\mathbf{EST}_2$ is $\mathbf{T}_2 + \mathbf{T.R.C.}_{avg2} \times \triangle$, where $\triangle$ is the switching cost. As $\mathbf{P}_1 \neq \mathbf{P}_2$ thus $\mathbf{EST}_1 \neq \mathbf{EST}_2$ under the assumption that High Associative Hashing Scheme does not report any data loss. High Associative Hashing Scheme shows *worse performance* when searching starts from $\mathbf{Table}_{i,1\leq i\leq 2}$ that owns lesser number keys. Thus for given $\mathcal{U}$, High Associative Hashing Scheme offers two searching times. The experimental proof is shown in Figure 11. □

Lemma 3. *Proposed scheme has unique* $\frac{T.R.C.}{key}$ *for given* $\mathcal{U}$.

Proof. Let $\mathbf{Pr}_{primary}$ and $\mathbf{Pr}_{backup}$ be the probabilities of finding a key in the primary table and backup table, respectively. In the proposed scheme, insertion and search always start from the primary table. For given, $|\mathcal{U}| = n, \frac{T.R.C.}{key}$ is $\frac{n\times(1+\mathbf{Pr}_{backup})}{n} = (1+\mathbf{Pr}_{backup}) = (2-\mathbf{Pr}_{primary})$. When a failure is observed either in insertion or

in searching, the backup table is consulted. Proposed scheme is free from data loss thus, $\mathbf{Pr}_{primary} = (1 - \mathbf{Pr}_{backup})$. Hence, the proposed scheme offers unique $\frac{T.R.C.}{key}$. The experimental proof is shown in Figure 11. □

Theorem 3. *Left-Right Hashing using prime numbers or Fibonacci numbers is faster than the worse case performance of the High Associative Hashing Scheme in terms of searching for a given $\mathcal{U}$.*

Proof. Let **EST** be the estimated searching time of a Left-Right Hashing. Let $\mathbf{EST}_1$ and $\mathbf{EST}_2$ are the estimated searching times of the High Associative Hashing Scheme. From Theorem 2:

$$\mathbf{EST}_1 = \mathbf{T}_1 + \mathbf{T.R.C.}_{avg1} \times \triangle \tag{6}$$

$$\mathbf{EST}_2 = \mathbf{T}_2 + \mathbf{T.R.C.}_{avg2} \times \triangle \tag{7}$$

$\mathbf{EST}_1$ and $\mathbf{EST}_2$, in terms of $\mathbf{P}_1$ and $\mathbf{P}_2$, are defined as:

$$\mathbf{EST}_1 = (2 - \mathbf{P}_1) \times t + \mathbf{T.R.C.}_{avg1} \times \triangle \tag{8}$$

$$\mathbf{EST}_2 = (2 - \mathbf{P}_2) \times t + \mathbf{T.R.C.}_{avg2} \times \triangle \tag{9}$$

From Lemma 2, $\mathbf{T.R.C.}_{avg1}$ and $\mathbf{T.R.C.}_{avg2}$ can be also expressed as $\mathbf{P}_1$ and $\mathbf{P}_2$ so $\mathbf{EST}_1$ and $\mathbf{EST}_2$ are:

$$\mathbf{EST}_1 = (2 - P_1) \times (t + \triangle) \tag{10}$$

$$\mathbf{EST}_2 = (2 - P_2) \times (t + \triangle) \tag{11}$$

From Theorem 1, the **EST** of the scheme is given as:

$$\begin{aligned}\mathbf{EST} = &\mathbf{Pr}_{primary} \times (\frac{-1}{\lambda_{primary}} log_e(1 - \lambda_{primary})) \times t \\ &+ \mathbf{Pr}_{backup} \times (\frac{-1}{\lambda_{backup}} log_e(1 - \lambda_{backup})) \times 2t \\ &+ (2 - P_{primary}) \times \triangle \end{aligned} \tag{12}$$

If $\mathbf{P}_1 < \mathbf{P}_2$, $\mathbf{EST}_1$ is the worse performance of High Associative Hashing Scheme. As $\mathbf{Pr}_{primary} >> \mathbf{P}_1$, **EST** of the proposed scheme is better than $\mathbf{EST}_1$, $\mathbf{EST} < \mathbf{EST}_1$. Similarly, $\mathbf{EST} < \mathbf{EST}_2$ holds if $\mathbf{P}_2 < \mathbf{P}_1$ as $\mathbf{Pr}_{primary} >> \mathbf{P}_2$. In worse performance, the High Associative Hashing Scheme pays more switching costs than the proposed scheme. Hence "Left-Right Hashing with prime numbers or Fibonacci numbers is faster than the worse-case performance of the High Associative Hashing Scheme in terms of searching for a given $\mathcal{U}$." The experimental proof is shown in Figure 26. □

5. Results and discussions

This section describes the experimental findings of the proposed work. Initially, Section 5.1 describes the experimental setup and used data structure, followed by the performance comparison of the proposed work in Section 5.2. During the comparison, we consider Sequential Symmetric Cuckoo Hashing, Random Probing and Quadratic Probing approaches, where we use existing as well as proposed performance indicators (*Degree of Dexterity*). In addition, the other proposed performance indicator (T.R.C./key) is used while comparing with Sequential Symmetric Cuckoo Hashing.

Section 5.3 deals with the performance estimation of the proposed scheme, *Stable and Low Associative Left-Right Hashing.* Section 5.4 sheds light on *Sequential Symmetric Cuckoo Hashing Scheme.* Experimental results on Random Probing and Quadratic Probing are tabulated in Section 5.5. Finally, Section 5.6 compares the performance of *Stable and Low Associative Left-Right Hashing* with Sequential Cuckoo Hashing, Random Probing and Quadratic Probing.

5.1. Experimental setup and data structure used

During the evaluation, this work considered a pure random dataset from Kaggle (10^6 in numbers, for every scheme) [6] that possesses a key length of 15 digits. Prototype of Stable and Low Associative Left-Right Hashing, Cuckoo Hashing, Random Probing and Quadratic Probing are implemented under GCC Compiler, 11.3.0 on $i7$ processor of 3.40 GHz with 8 GB DRAM at Ubuntu Linux 22.04.1. Additionally, *clock()* of *time.h* is used to record consumed time. During the experiment, two arrays of the same size (1000033) are implemented for Cuckoo Hashing with $\epsilon = 0.000033$. Two hash functions ($(a*key+b)\%M$ and $(key/M)\%M$), where M is a prime number and a and b are constants) from the Family of Universal Hash Functions are used to compute the hash code. The size of the hash table in Random Probing and Quadratic Probing is kept 1000033. The size of the Primary table and backup table are 1048583 and 131101, respectively. The hash function used in Random Probing, Quadratic Probing, Primary Table and Backup Table are based on the division method with the form $K\%m$ ($Kmodm$), where K is key and m is a prime number.

5.2. Comparison of proposed work with existing schemes and involved performance indicators

Authors use Cuckoo Hashing as High Associative Hashing Scheme. **Left-Right Hashing** is an acronym to the proposed *Stable and Low Associative Left-Right Hashing.* This section shows the effectiveness of the proposed Left-Right Hashing scheme over Cuckoo Hashing, Random Probing and Quadratic Probing. In performance evaluation, we engage different performance indicators, including a) *memory utilization*, b) *data loss*, c) *collision rate*, d) *hit count and miss count*, e) $\frac{T.R.C.}{Key}$, f) *average search length*, g) *insertion latency*, h) *average searching time*, and i) *degree of dexterity* (η).

5.3. Stable and low associative left-right hashing

Authors have considered two versions of Left-Right Hashing based on prime numbers and Fibonacci numbers as **LRHP** and **LRHF** respectively. As Left-Right Hashing uses two tables, it is referred to as **LRH2T**. To introduce randomness authors have used two sets of prime numbers (**P5** and **P8**) and Fibonacci numbers (**F5** and **F8**) with respective sizes of 5 and 8. When **P5** and **F5** are used in **LRH2T**, it is referred as **LRH2TP5** and **LRH2TF5** respectively. Similarly, when **P8** and **F8** are used in **LRH2T**, we have **LRH2TP8**, and **LRH2TF8**, respectively. Thus under **LRHP** there are two variants as: **LRH2TP5** & **LRH2TP8** and **LRH2TF5** & **LRH2TF8** similarly under **LRHF**.

5.3.1. Memory utilization

A hashing technique must efficiently use the memory. *Memory utilization is the ratio of the total number of keys stored to the total number of memory allocated.* The **LRH2TP5** and **LRH2TF5** store **935810** and **928300** keys out of 10^6 in the hash table of size **1048583**, respectively. The **LRH2T** variants of Left-Right Hashing handle the data loss of the primary table by providing accommodations in the backup tables. The unequal-sized hash tables and performance of the used hash function with Left-Right Random Probing collectively produce better space utilization with 0 data loss in **LRH2TF8** and **LRH2TP8**. Memory utilization (in %) of Left-Right Hashing is depicted in Figure 6.

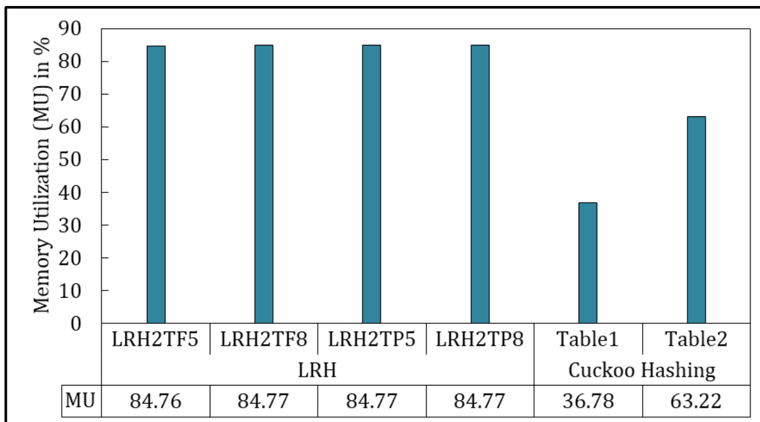


Figure 6. Memory utilization in Left-Right Hashing and Cuckoo Hashing at ML = 200

5.3.2. Data loss

The collision resolution technique is used with the hash function to prevent data loss due to collision. The Left-Right Random Probing shows that data loss is decreasing

and it finally reports 0 data loss when the backup table is introduced and **P5** and **F5** are replaced with **P8** and **F8**. Data loss is shown in Figure 7.

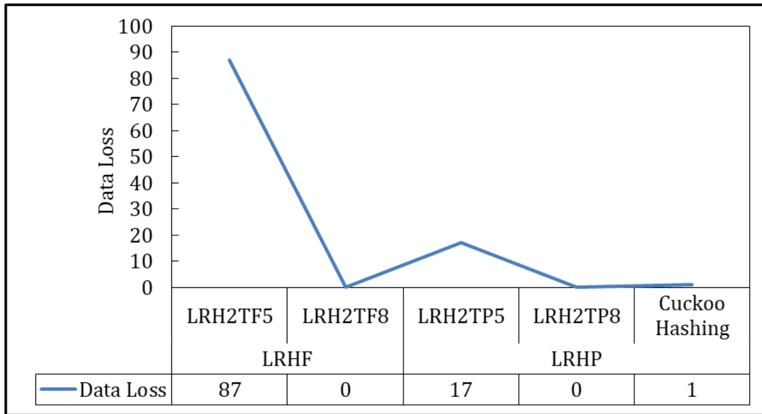


Figure 7. Data loss in Left-Right Hashing and Cuckoo Hashing at ML = 200

5.3.3. Collision rate

The model described in Section 4.4 gives the upper bound of the number of data collisions. *Collision rate is a ratio of the total number of collisions that occurred to the total number of keys stored in the hash table.* Figure 8 shows the number of data collisions computed by the model given in Section 4.4.

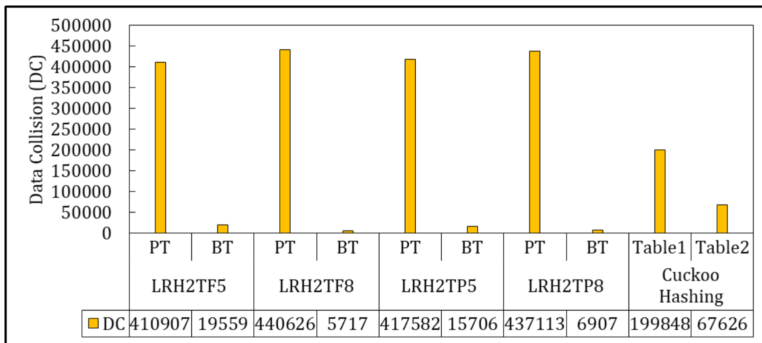


Figure 8. Data Collision in Left-Right Hashing and Cuckoo Hashing at ML = 200

The collision rate in primary tables of both **LRH2TP8** and **LRH2TF8** are higher than the collision rate in backup tables. This is because primary tables hold more keys than backup tables in **LRH2TP8** and **LRH2TF8**, as shown in Figure 9.

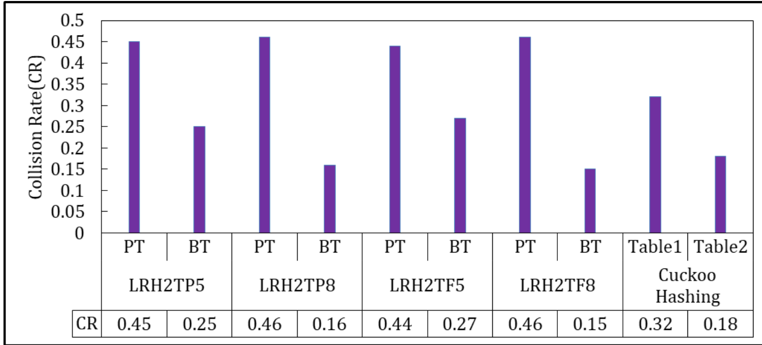


Figure 9. Collision Rate in Left-Right Hashing and Cuckoo Hashing at ML = 200

5.3.4. Hit count and miss count

In Left-Right Hashing, the backup table is only consulted when one of the following occurs: either **a)** when the search operation is unsuccessful in the primary table or, **b)** when insertion failure occurs in the primary table. This restriction avoids unlike consultation of the backup table and avoids switching overhead. The hit and miss count of both tables in **LRH2TP8** and **LRH2TF8** are shown in Figure 10. The miss count, 0, in the backup tables of **LRH2TP8** and **LRH2TF8** indicate that no data loss is reported in these versions of Left-Right Hashing.

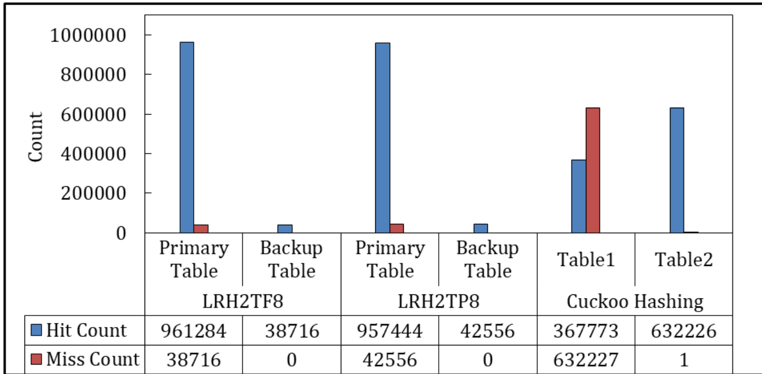


Figure 10. Hit Count and Miss Count in Left-Right Hashing and Cuckoo Hashing at ML = 200

The proposed indicator, $\frac{\textbf{T.R.C.}}{\textbf{Key}}$ *is the ratio of the sum of hit counts and miss counts of a primary table and its associated backup table to the size of* $\mathcal{U}$. The $\frac{\textbf{T.R.C.}}{\textbf{Key}}$ estimates the switching overhead when a hashing technique uses more than one hash table. The $\frac{\textbf{T.R.C.}}{\textbf{Key}}$ of **LRH2TP8** and **LRH2TF8** are shown in the Figure 11.

Fibonacci numbers are superseding the prime numbers. The $\frac{\mathbf{T.R.C.}}{\mathbf{Key}}$ of **LRH2TF8** is lower than **LRH2TP8**. This is because **LRH2TF8** stores 961284 keys in its primary table and **LRH2TP8** holds only 957444 keys in its primary table. Switching overhead in **LRH2TF8** is less than **LRH2TP8**. **LRH2TP8** and **LRH2TF8** offer unique $\frac{T.R.C.}{Key}$ for a given universe of keys, $\mathcal{U}$ and this is the *experimental proof of the Lemma 3*.

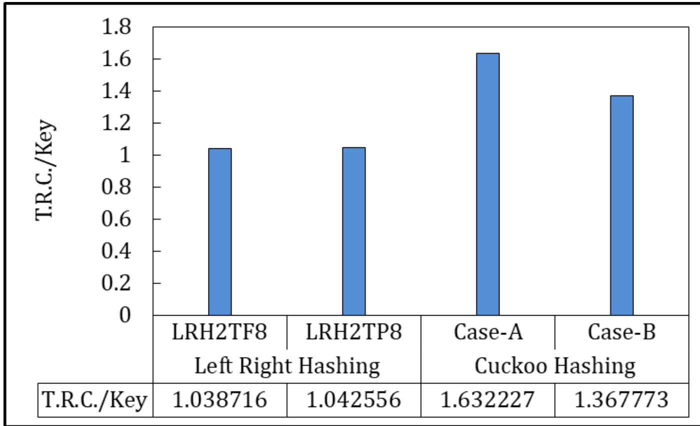


Figure 11. T.R.C./Key in Left-Right Hashing and Cuckoo Hashing at ML = 200

5.3.5. Average search length

When the collision resolution technique is used with a hash function, average search length plays a vital role. Searching time is directly proportional to the average search length. The average search length in Left-Right Hashing is shown in Figure 12.

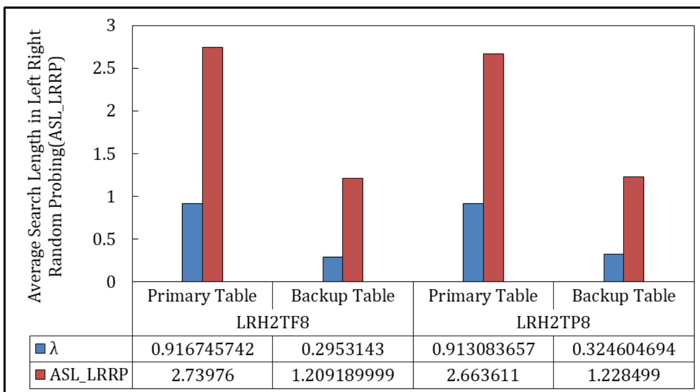


Figure 12. Load Factor and Average Search Length in Left-Right Hashing

Prime numbers are efficiently operating against Fibonacci numbers in the proposed Left-Right Random Probing. The average search length and load factors of **LRH2TP8** and **LRH2TF8** are shown in Figure 12 and this result endorses the claim made by [17]. Maximum 2 and 3 probes are required to search a key in the primary and backup tables, respectively, in **LRH2TP8** and **LRH2TF8**.

5.3.6. Insertion latency

Insertion latency is the total time consumed in the insertion of keys into the hash table. *If the total inserted keys increase then the insertion latency increases.* Ideally, the hashing technique should execute an insertion operation in $O(1)$. The **LRH2TP8** and **LRH2TF8** successfully store all the keys *i.e.,* $\mathbf{10^6}$. Insertion latency is shown in Figure 13.

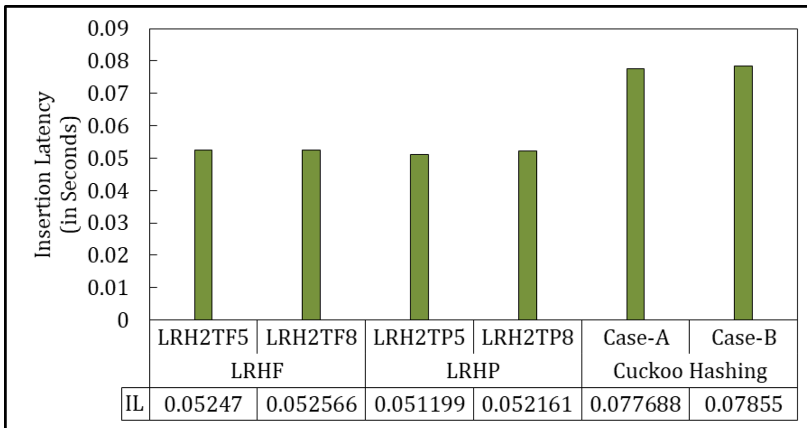


Figure 13. Insertion Latency in Left-Right Hashing and Sequential Cuckoo Hashing at ML = 200

5.3.7. Average search time

Searching time depends on two factors: **number of keys searched** and **average search length**. Searching time increases with an increase in the number of keys searched and average search length.

Average searching time is shown in the Figure 14. Comparatively, **LRH2TP5** and **LRH2TF5** have less keys in its hash tables than **LRH2TP8** and **LRH2TF8** (see Figure 7). During the search, all keys stored in the above variants are fetched in **LRH2TP5**, **LRH2TF5**, **LRH2TP8** and **LRH2TF8**. **LRH2TP8** is 1.12 times faster than **LRH2TF8** in term of searching time. The average search length in **LRH2TP8** (2.663611) is lesser than **LRH2TF8** (2.73976), as shown in Figure 12 and the average search time is depicted in the Figure 14.

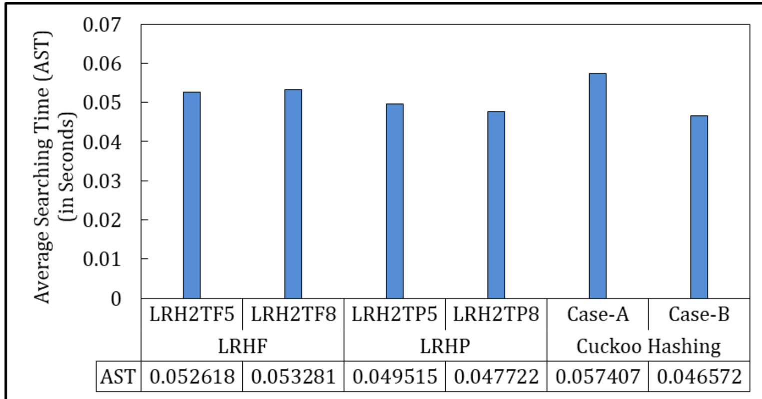


Figure 14. Average Searching Time in Left-Right Hashing Cuckoo Hashing at ML = 200

5.3.8. Degree of Dexterity (η)

It measures quickness of a hashing technique. This performance indicator is a function of insertion latency and average searching time. This performance indicator estimates the *cost of first use* of a hash table. The hash tables may be either static or dynamic in nature. Static hash is created once while dynamic hash tables are recreated after a fixed time interval [3, 5, 32]. During the construction of a hash table, insertion latency is more important. A hash table can be used after the completion of the insertion operation. The η of each variant of Left-Right Hashing is shown in Figure 15. **LRH2TP8** shows best performance among **LRH2TP5**,**LRH2TF**, and **LRH2TF8** in term of η.

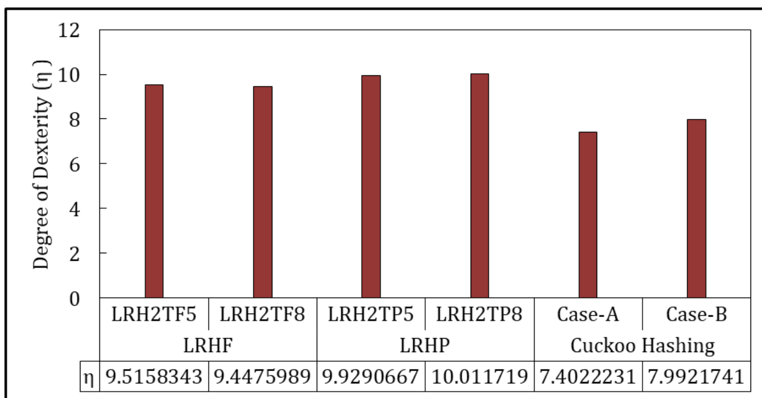


Figure 15. Degree of dexterity in Left-Right Hashing and Cuckoo Hashing at ML = 200

5.4. Sequential symmetric Cuckoo Hashing as High Associative Hashing Scheme

This section presents the results of rigorous experiments done with Cuckoo Hashing. During the trials, a variety of ML has been taken into consideration. The ML range begins at **ML = 10** to **ML = 220** with the interval of **10**. The performance indicators like: *Average Searching Time*, *Insertion Latency*, *Memory Utilization*, *Data Loss*, and *Degree of Dexterity* have been taken into consideration during the experimental trials. As discussed previously, two cases have been considered: **Case-A** and **Case-B**, in the analysis of Sequential Cuckoo Hashing. In **Case-A**, the table with the minimum number of keys is referred first to initiate search and insertion operations and in **Case-B**, the table with the maximum number of keys is referred first to initiate search and insertion operations. Thus, under **Case-A**, insertion and lookup operation is initiated from **Table1** that holds 367773 keys and under **Case-B**, insertion and lookup operation are started from **Table2** that holds 632226 keys. *Average Searching Time, Insertion Latency, and Degree of Dexterity* are **case-dependent** performance indicators and the remaining performance indicators are **case-independent**. Notable observations are highlighted as:

1. At ML = 200, Cuckoo Hashing shows data loss of only one key. This is the experimental evidence that *"Universal Hash Functions report data loss of single key"* [8] (results shown in the Figure 16, experimentally verifies this claim).

2. Even at ML = 220, 50% memory utilization is not obtained. The size of a hash table in Cuckoo Hashing is kept at 1000033, and the total number of keys is 10^6 to be stored (Cuckoo Hashing shows data loss of 1 key). Thus, load factor (λ) is $\frac{\mathbf{(10^6)-1}}{\mathbf{2\times(10^6+33)}} = \mathbf{0.499}$ (*Experimental proof of Lemma 1*). When λ is multiplied by 100, memory utilization in % is computed. **Thus,** 50% **memory utilization in Cuckoo Hashing appears as a theoretical upper bound, and this result endorses the claim made by [45] about Cuckoo Hashing in terms of load factor**.

3. Data loss in Cuckoo Hashing is influenced by ML. With a large value of ML, Cuckoo Hashing exhibits minimal data loss. So, a small-size stash can be introduced to deal with this situation [38] (results shown in Figure 16 experimentally verifies this claim).

4. "The probability of finding a key in a table is $\frac{1}{2}$ in Cuckoo Hashing" claimed by [49] is not observed.

5. Universal Hash Functions excel in minimizing collisions. The collision rate is shown in the Figure 9.

6. ML count does not impact searching time in Cuckoo Hashing. Results are shown in the Figures 17 and 18.

7. The average searching time is **0.05740655** seconds and **0.0465724** seconds under **Case-A** and **Case-B** respectively as depicted in the Figure 14 (This is the

experimental proof of the Theorem 2). *Switching cost influences average searching time.* In **Case-A**, $\frac{T.R.C.}{Key}$ is **1.632227** while $\frac{T.R.C.}{Key}$ is **1.367773** under **Case-B** as presented in the Figure 11 (This is experimental proof of Lemma 2). Average searching time can be improved if the search operation originates from the table that holds more keys as it saves involved switching costs.

8. In experiments, a non-consistent relation is observed between ML and insertion latency. Insertion latency in **Table1** and **Table2** are almost equal with a difference of **0.000862** seconds. Results are shown in the Figures 19 and 20.

9. Proposed performance indicator, η, is also independent of ML. The value of η is **7.40222311** in Case-A and **7.992174063** in Case-B.Thus **Case-B** is **1.08** times more quicker than **Case-A**. Results are shown in the Figures 21 and 22.

10. At a high value of ML, Cuckoo Hashing reports data collision of the single key while the authors in [56] claim that the maximum collision that is allowed in most versions of Cuckoo Hashing is 8 keys only. *Costly rehash operation can be avoided by setting a high value of ML and the single unpositioned key can be simply discarded or stored in a separate node. The authors are the first to address this issue in the context of Cuckoo Hashing to the best of their knowledge.*

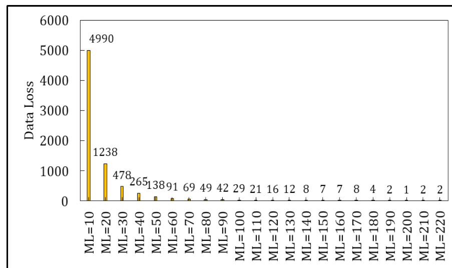


Figure 16. Data Loss in Cuckoo Hashing (from ML = 10 to ML = 220)

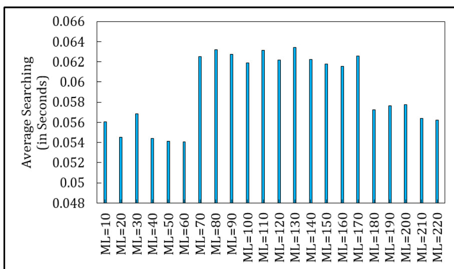


Figure 17. Average searching time in Cuckoo Hashing under Case-A

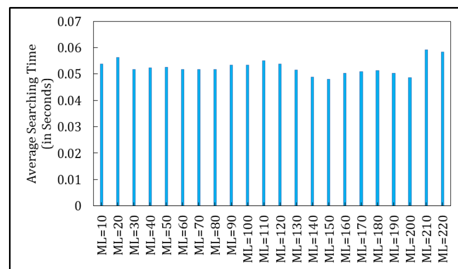


Figure 18. Average searching time in Cuckoo Hashing under Case-B

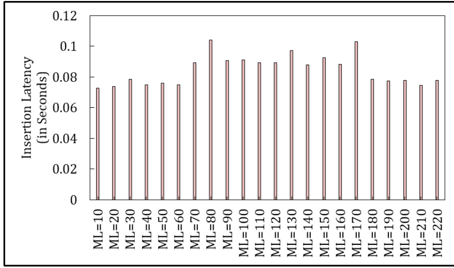


Figure 19. Insertion Latency in Cuckoo Hashing under Case-A

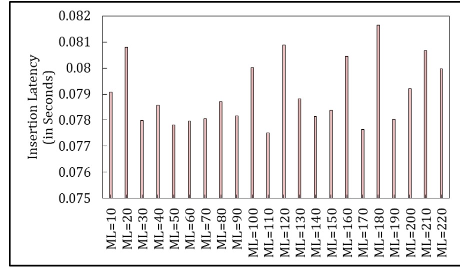


Figure 20. Insertion Latency in Cuckoo Hashing under Case-B

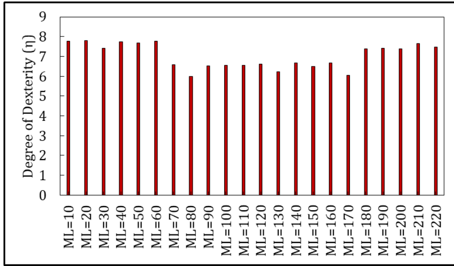


Figure 21. Degree of dexterity in Cuckoo Hashing under Case-A

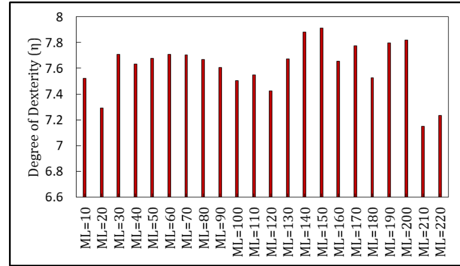


Figure 22. Degree of Dexterity in Cuckoo Hashing under Case-B

5.5. Random probing and quadratic probing

Performance of Random Probing and Quadratic Probing are tabulated in Table 1.

Table 1
Performance Analysis of Random Probing and Quadratic Probing

Parameter	Scheme	
	Random Probing	Quadratic Probing
Data Loss	35195	367520
Collision Rate	0.460054	0.301589
Load Factor	0.92011	0.60318
Average Searching Time (in seconds)	0.061441	0.031863
Insertion Latency (in seconds)	0.028302	0.041207
Degree of Dexterity (η)	11.14289	13.68557

During the evaluation of the schemes, the following performance indicators are considered: *Data Loss*, *Collision Rate*, *Load Factor*, *Average Searching Time*, *Insertion Latency*, and *Degree of Dexterity.* Notable observations are enumerated as:

1. Data loss in Quadratic Probing is higher than in Random Probing.
2. Collision Rate in Random Probing is greater than Quadratic Probing as Random Probing accommodates more keys than Quadratic Probing in its hash table.
3. Random Probing supports a load factor (λ) greater than Quadratic Probing. When is multiplied by 100, we can compute memory utilization in percent.

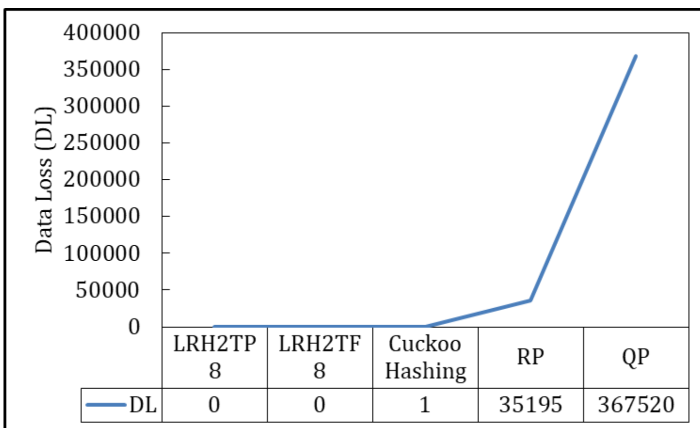
4. Quadratic Probing is faster than Random Probing in terms of Average Searching Time.
5. In terms of insertion latency, the performance of Random Probing is better than Quadratic Probing.
6. In term of η, Quadratic Probing performs better than Random Probing.

5.6. Performance Comparison of Left-Right Hashing, Sequential Cuckoo Hashing, Random Probing, and Quadratic Probing

In this section, the performance of **LRH2TF8**, **LRH2TP8**, Sequential Cuckoo Hashing at **ML = 200** (at ML = 200, data loss is 1), Random Probing (RP) and Quadratic Probing (QP) are evaluated. Performance indicators: *Average Searching Time, Insertion Latency, Memory Utilization, and Degree of Dexterity* are used to measure the effectiveness of the proposed schemes against Sequential Cuckoo Hashing, RP and QP.

5.6.1. Data Loss

In terms of data loss, Left-Right Hashing excels the existing scheme. Stability is absent in Cuckoo Hashing; thus, data migration exists only in Cuckoo Hashing. Primary table holds 957444 keys in **LRH2TP8** while 961284 are stored in the primary table under **LRH2TF8** scheme. It indicates that the set **P8** and **F8** are working well with the proposed *Left-Right Random Probing*. The modification made in the **Combinatorial Hashing** to estimate the table sizes appears as a worthy choice. Size of primary table is $\mathbf{Prime}(2^{\mathcal{RNC}})$ while size of backup table is maintained, as $\mathbf{Prime}(\frac{1}{f}2^{\mathcal{RNC}})$, where $\mathcal{C}$ is 0.67 and f is 8. The size of the hash table in Random Probing and Quadratic Probing is equal. The performance of Random Probing is better than Quadratic Probing in terms of data loss, as shown in Figure 23.



	LRH2TP 8	LRH2TF 8	Cuckoo Hashing	RP	QP
DL	0	0	1	35195	367520

Figure 23. Data Loss in Left-Right Hashing, Cuckoo Hashing, Random Probing and Quadratic Probing

5.6.2. Collision Rate

Data collision rate in the primary table of **LRH2TF8** and **LRH2TP8** are the same i.e. **0.46**. The **LRH2TF8** performs better than **LRH2TP8** when the data collision rate of the backup table is considered. However, the data collision rate of Table2 (in Cuckoo Hashing) is more than the data collision rate of backup tables of **LRH2TF8** and **LRH2TF8**. Primary tables in **LRH2TP8** and **LRH2TF8** store **1.5** times more keys than Table1 (in Cuckoo Hashing) in the Sequential Cuckoo Hashing and that's why the collision rate in the primary tables of **LRH2TF8** and **LRH2TP8** are **1.44** times more than the collision rate of Table1 (in Cuckoo Hashing). The collision rate in Table2 of Sequential Cuckoo Hashing is **1.125** times and **1.2** times more than the backup tables of **LRH2TP8** and **LRH2TF8**. The collision rate is shown in Figure 24.

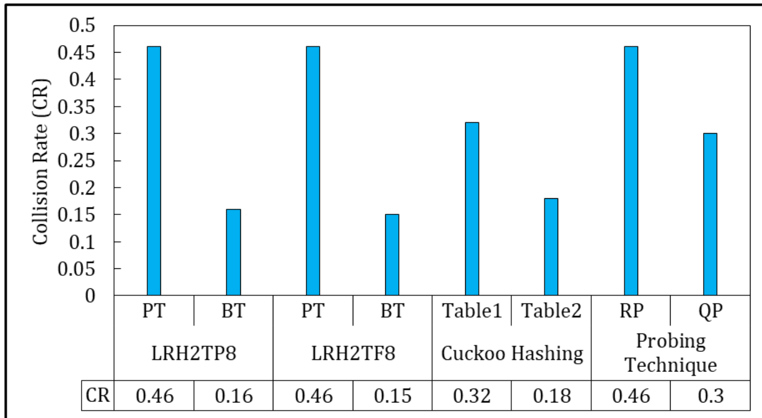


Figure 24. Collision Rate in Left-Right Hashing, Cuckoo Hashing, Random Probing and Quadratic Probing

5.6.3. Memory Utilization

The number of keys in the primary table and backup table of **LRH2TF8** and **LRH2TP8** are **961284** and **38716**, **957444** and **42556** respectively. In Sequential Cuckoo Hashing, the number of keys held by **Table1** and **Table2** are **367773** and **632226** respectively. The number of keys is 964805 (with a loss of 35195 keys) in Random Probing while 632480 (with a loss of 367520 keys) in Quadratic Probing. The size of the primary table and the backup table are 1048583 and 131101, respectively. While Cuckoo Hashing maintains two hash tables of size 1000033 each. Random Probing and Quadratic Probing have the same-sized hash table, 1000033. Memory utilization in the proposed schemes are **84.77**% and ≈ **50**% in the Sequential Cuckoo Hashing. Thus, the proposed schemes are **1.7** times better than the Sequential Cuckoo Hashing in terms of memory utilization. Thus, memory utilization of the right Hashing Scheme is best among all if 0 data loss is taken into account. Otherwise,

Random Probing shows the memory utilization of 92% at the cost of 35195 data loss. Memory utilization is shown in the Figure 25.

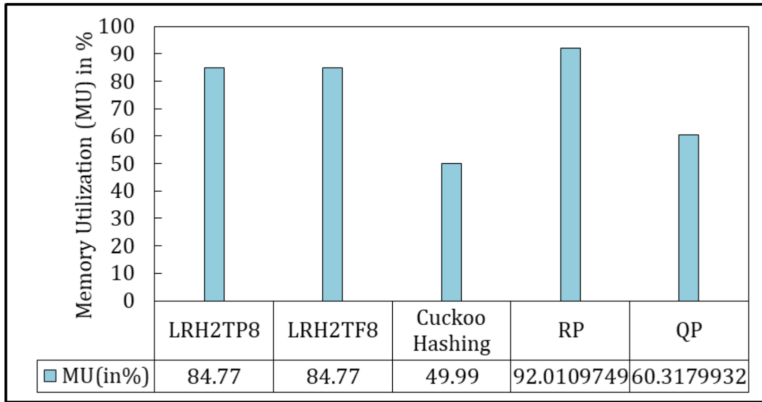


Figure 25. Memory Utilization in Left-Right Hashing, Cuckoo Hashing, Random Probing and Quadratic Probing

5.6.4. Average Searching Time

The **LRH2TF8** and **LRH2TP8** are **1.08** times and **1.2** times faster than the worse case performance of Sequential Cuckoo Hashing (under **Case-A**). The **LRH2TP8** is only **0.001** second behind the Sequential Cuckoo Hashing in the average searching time under **Case-B**. Thus, the performance of **LRH2TP8** is much better than **LRH2TF8**. The average searching time of **LRH2TF8**, **LRH2TP8**, Sequential Cuckoo Hashing, Random Probing and Quadratic Probing are shown in the Figure 26.

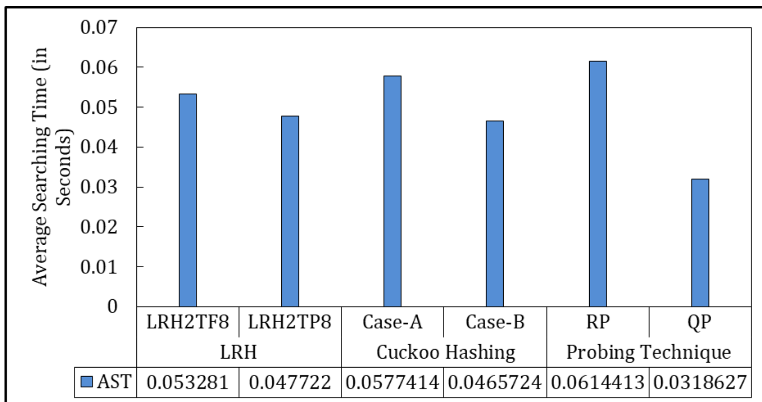


Figure 26. Average Searching Time in Left-Right Hashing, Cuckoo Hashing, Random Probing and Quadratic Probing

The secret behind the performance of **LRH2TP8** is its average search length, as shown in Figure 12. The smaller average search length of **LRH2TP8** successfully neutralizes the edge of switching cost that occurred in **LRH2TF8** over **LRH2TP8**. In terms of average searching time, Quadratic Probing is the fastest hashing scheme with the highest data loss.

5.6.5. Insertion Latency

Insertion latency of Sequential Cuckoo Hashing is **0.077688** and **0.07855** seconds under **Case-A** and **Case-B** respectively. The value of insertion latency is **0.052566** and **0.052161** seconds in **LRH2TF8** and **LRH2TP8** respectively. Thus, **LRH2TF8** and **LRH2TP8** are **1.5** times more efficient than Sequential Cuckoo Hashing in terms of insertion latency. *Thus the proposed schemes effectively overcome the challenge of insertion latency in Sequential Cuckoo Hashing.* Insertion latency of LRH2TF8, LRH2TP8, Cuckoo Hashing, Random Probing and Quadratic Probing are shown in Figure 27. We drew a wonderful outcome from this experiment that the Left-Right Hashing Scheme performs better than other schemes in terms of insertion latency if 0 data loss is emphasized.

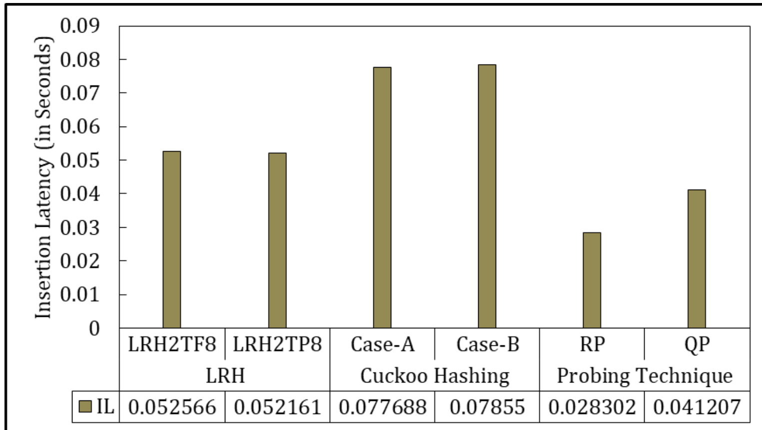


Figure 27. Insertion Latency in Left-Right Hashing, Cuckoo Hashing, Random Probing and Quadratic Probing

5.6.6. Degree of Dexterity (η)

Value of this proposed performance indicator is **9.447599** and **10.01172** for **LRH2TF8** and **LRH2TP8**, respectively. The value of η under **Case-A** and **Case-B** of Sequential Cuckoo Hashing are **7.402223** and **7.992174**, respectively. Thus, **LRH2TF8** is **1.27** times and **LRH2TP8** is **1.35** times quicker than Sequential Cuckoo Hashing under **Case-A**. Figure 28 shows the performance of the right Hashing Scheme, Cuckoo Hashing, Random Probing and Quadratic Probing in terms of η. Without data loss, Left-Right Hashing is the best scheme among all if η is considered.

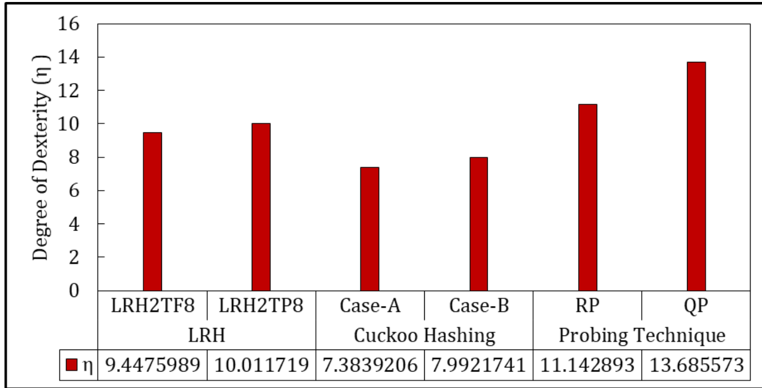


Figure 28. Degree of Dexterity in Left-Right Hashing, Cuckoo Hashing, Random Probing and Quadratic Probing

6. Conclusion and future scope

This endeavor unveils a pioneering framework for a "High Associative Hashing Scheme", characterized by the utilization of dual hash tables. The formidable hurdle of **switching cost** emerges as an ancillary challenge intertwined with the realm of the "High Associative Hashing Scheme". In this work, Cuckoo Hashing stands and is acknowledged as a prominent exemplar of the "High Associative Hashing Scheme." Thus, switching cost emerges as an additional challenge in Sequential Cuckoo Hashing, alongside **high insertion latency**, **inefficient memory usage**, and **data migration**. In addition to Cuckoo Hashing, Random Probing and Quadratic Probing are also considered in the performance evaluation. The proposed Left-Right Hashing scheme refines the **Combinatorial Hashing** to estimate the size of the asymmetric hash tables for non-uniform key distribution. Hash functions based on **Division Methods** distribute the keys over the primary and backup tables. *Left-Right Random Probing*, a variant of **Random Probing**, deals with the collision by using first 8 prime numbers and Fibonacci numbers.

Proposed schemes are **1.5** times more efficient than Sequential Cuckoo Hashing in terms of insertion latency. The distribution of keys is unequal in both Stable and Low Associative Left-Right Hashing and Sequential Cuckoo Hashing. High Associative Hashing Scheme like Sequential Cuckoo Hashing always offers $\lambda < \frac{1}{2}$. While dealing with memory utilization, proposed schemes are **1.7** more efficient than Sequential Cuckoo Hashing. If data loss is discarded, *Random Probing* is the best in utilizing the hash table. In the case of average searching time, the performance of **LRH2TP8** and Sequential Cuckoo Hashing are almost equal. In the proposed schemes, keys are never removed from their locations, and hence the proposed scheme is *stable* in nature, there is no data migration cost. The η for the proposed schemes are greater than Sequential Cuckoo Hashing. In the case of η, the performance of Quadratic Probing is the best on

the cost of data loss. In the case of dynamic hash tables, **LRH2TP8** is **1.35** faster than the Sequential Cuckoo Hashing. The proposed collision resolution technique, Left-Right Random Probing, is performing well with the prime numbers and Fibonacci numbers to maintain a smaller average search length and reports **0** data loss.

The proposed schemes successfully address the major challenges of Sequential Cuckoo Hashing and provide two solutions, ***LRH2TF8*** *and* ***LRH2TP8****, to overcome the challenges associated with Sequential Cuckoo Hashing.* Still, Cuckoo Hashing excels in terms of average searching time because of its high associative nature. Authors acknowledge Cuckoo Hashing as the best example of **time and space trade-off** in the area of the hashing technique. The proposed scheme is sequential in nature, and in the near future, attempt may be made to present a parallel version of it. Cuckoo Hashing has been utilized across various domains to enhance solution optimization. The primary areas where cuckoo Hashing is implemented include *Image Processing, Pattern Recognition, Software Testing, Data Mining, Cyber Security, Cloud Computing,* and the *Internet of Things.* It is a challenging task for the researchers to evaluate the performance of "Left-Right Hashing" against Cuckoo Hashing in such application areas.

References

[1] Awad M.A., Ashkiani S., Porumbescu S.D., Farach-Colton M., Owens J.D.: Analyzing and implementing GPU hash tables. In: *2023 Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pp. 33–50, SIAM, 2023. doi: 10.1137/1.9781611977578.ch3.

[2] Balkesen C., Teubner J., Alonso G., Özsu M.T.: Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 362–373, IEEE Computer Society, Los Alamitos, CA, USA, 2013. doi: 10.1109/ICDE.2013.6544839.

[3] Bender M.A., Conway A., Farach-Colton M., Kuszmaul W., Tagliavini G.: Iceberg Hashing: Optimizing Many Hash-Table Criteria at Once, *Journal of the ACM*, vol. 70(6), 2023. doi: 10.1145/3625817.

[4] Bender M.A., Farach-Colton M., Kuszmaul J., Kuszmaul W.: Modern Hashing Made Simple. In: *2024 Symposium on Simplicity in Algorithms (SOSA)*, pp. 363–373, SIAM, 2024. doi: 10.1137/1.9781611977936.33.

[5] Bender M.A., Farach-Colton M., Kuszmaul J., Kuszmaul W., Liu M.: On the optimal time/space tradeoff for hash tables. In: *STOC 2022: Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 1284–1297, Association for Computing Machinery, New York, NY, USA, 2022. doi: 10.1145/3519935.3519969.

[6] Bozsolik T.: Random numbers, 2019. doi: 10.34740/KAGGLE/DSV/816507.

[7] Brouwer A.E.: An associative block design ABD (8, 5), *SIAM Journal on Computing*, vol. 28(6), pp. 1970–1971, 1999. doi: 10.1137/s0097539797316622.

[8] Carter J.L., Wegman M.N.: Universal classes of hash functions, *Journal of Computer and System Sciences*, vol. 18(2), pp. 143–154, 1979. doi: 10.1016/0022-0000(79)90044-8.

[9] Celis P., Larson P.A., Munro J.I.: Robin hood hashing. In: *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pp. 281–288, 1985. doi: 10.1109/SFCS.1985.48.

[10] Chen W.C., Vitter J.S.: Analysis of new variants of coalesced hashing, *ACM Transactions on Database Systems*, vol. 9(4), pp. 616–645, 1984. doi: 10.1145/1994.2205.

[11] Chen Z., He X., Sun J., Chen H., He L.: Concurrent hash tables on multicore machines: Comparison, evaluation and implications, *Future Generation Computer Systems*, vol. 82, pp. 127–141, 2018. doi: https://doi.org/10.1016/j.future.2017.12.054.

[12] Devroye L., Morin P.: Cuckoo hashing: Further analysis, *Information Processing Letters*, vol. 86(4), pp. 215–219, 2003. doi: https://doi.org/10.1016/S0020-0190(02)00500-8.

[13] Dolev S., Lahiani L., Haviv Y.: Unique permutation hashing, *Theoretical Computer Science*, vol. 475, pp. 59–65, 2013. doi: 10.1016/j.tcs.2012.12.047.

[14] Flajolet P., Poblete P., Viola A.: On the analysis of linear probing hashing, *Algorithmica*, vol. 22(4), pp. 490–515, 1998. doi: 10.1007/pl00009236.

[15] Frieze A., Johansson T.: On the insertion time of random walk cuckoo hashing, *Random Structures and Algorithms*, vol. 54(4), pp. 721–729, 2019. doi: 10.1002/rsa.20808.

[16] Gao J., Tao X., Cai S.: Towards more efficient local search algorithms for constrained clustering, *Information Sciences*, vol. 621, pp. 287–307, 2023. doi: 10.1016/j.ins.2022.11.107.

[17] Gonnet G.H.: Expected length of the longest probe sequence in hash code searching, *Journal of the ACM*, vol. 28(2), p. 289–304, 1981. doi: 10.1145/322248.322254.

[18] Greene D.H., Knuth D.E.: *Mathematics for the Analysis of Algorithms*, Springer Science & Business Media, 2007.

[19] Halatsis C., Philokyprou G.: Pseudochaining in hash tables, *Communications of the ACM*, vol. 21(7), p. 554–557, 1978. doi: 10.1145/359545.359560.

[20] Han Z., Li Y., Li J.: A novel routing algorithm for IoT cloud based on hash offset tree, *Future Generation Computer Systems*, vol. 86, pp. 456–463, 2018. doi: 10.1016/j.future.2018.02.047.

[21] Janson S.: Asymptotic distribution for the cost of linear probing hashing, *Random Structures and Algorithms*, vol. 19(3-4), pp. 438–471, 2001. doi: 10.1002/rsa.10009.

[22] Janson S., Viola A.: A unified approach to linear probing hashing with buckets, *Algorithmica*, vol. 75(4), pp. 724–781, 2016. doi: 10.1007/s00453-015-0111-x.

[23] Jensen M.S., Pagh R.: Optimality in external memory hashing, *Algorithmica*, vol. 52(3), pp. 403–411, 2008. doi: 10.1007/s00453-007-9155-x.

[24] Jiang J., Yan Y., Zhang M., Yin B., Jiang Y., Yang T., Li X., Wang T.: Shifting hash table: An efficient hash table with delicate summary. In: *2019 IEEE Globecom Workshops (GC Wkshps)*, IEEE, 2019. doi: 10.1109/gcwkshps45667.2019.9024392.

[25] Karp R.M., Luby M., Heide auf der F.M.: Efficient PRAM simulation on a distributed memory machine, *Algorithmica*, vol. 16(4), pp. 517–542, 1996. doi: 10.1007/bf01940878.

[26] Knott G.D., Torre de la P.: Hash table collision resolution with direct chaining, *Journal of Algorithms*, vol. 10(1), pp. 20–34, 1989. doi: 10.1016/0196-6774(89)90021-7.

[27] Köppl D., Puglisi S.J., Raman R.: Fast and simple compact hashing via bucketing, *Algorithmica*, vol. 84(9), pp. 2735–2766, 2022. doi: 10.1007/s00453-022-00996-y.

[28] Kurpicz F., Lehmann H.P., Sanders P.: *PaCHash: packed and compressed hash tables*, pp. 162–175. doi: 10.1137/1.9781611977561.ch14.

[29] Larson P.: Analysis of uniform hashing, *Journal of the ACM*, vol. 30(4), pp. 805–819, 1983. doi: 10.1145/2157.322407.

[30] Lehmann H.P., Sanders P., Walzer S.: SicHash – small irregular cuckoo tables for perfect hashing. In: *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 176–189, SIAM, 2023. doi: 10.1137/1.9781611977561.ch15.

[31] Lehmann H.P., Sanders P., Walzer S.: ShockHash: Towards optimal-space minimal perfect hashing beyond brute-force. In: *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 194–206, SIAM, 2024. doi: 10.1137/1.9781611977929.15.

[32] Li Y., Zhu Q., Lyu Z., Huang Z., Sun J.: DyCuckoo: Dynamic hash tables on GPUs. In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 744–755, 2021. doi: 10.1109/ICDE51399.2021.00070.

[33] López-Valdivieso J., Cumplido R.: Design and implementation of hardware-software architecture based on hashes for SPHINCS+, *ACM Transactions on Reconfigurable Technology and Systems*, vol. 17(4), 54, 2024. doi: 10.1145/3653459.

[34] Luo W.: Hashing via finite field, *Information Sciences*, vol. 176(17), pp. 2553–2566, 2006. doi: 10.1016/j.ins.2005.12.001.

[35] Ma Z., Sha E.H.M., Zhuge Q., Jiang W., Zhang R., Gu S.: Towards the design of efficient hash-based indexing scheme for growing databases on non-volatile memory, *Future Generation Computer Systems*, vol. 105, pp. 1–12, 2020. doi: 10.1016/j.future.2019.07.035.

[36] Majewski B.S., Wormald N.C., Havas G., Czech Z.J.: A family of perfect hashing methods, *The Computer Journal*, vol. 39(6), pp. 547–554, 1996. doi: 10.1093/comjnl/39.6.547.

[37] Manohar S., Vignesh M., Prabhu G.M.: Sensitive data transaction using RDS in AWS, *Advances in Science and Technology*, vol. 124, pp. 782–788, 2023. doi: 10.4028/p-3z1665.

[38] Minaud B., Papamanthou C.: Generalized cuckoo hashing with a stash, revisited, *Information Processing Letters*, vol. 181, 106356, 2023. doi: 10.1016/j.ipl.2022.106356.

[39] Mitzenmacher M.: Some open questions related to Cuckoo Hashing. In: A. Fiat, P. Sanders (eds.), *Algorithms – ESA 2009*, Springer, Berlin–Heidelberg, 2009. doi: 10.1007/978-3-642-04128-0_1.

[40] Morris R.: Scatter storage techniques, *Communications of the ACM*, vol. 11(1), pp. 38–44, 1968. doi: 10.1145/362851.362882.

[41] Mugher R.A., Alhammadi N.A.M.: Performance evaluation of quadratic probing and random probing algorithms in modeling hashing technique, *Journal of Soft Computing and Data Mining*, vol. 3(2), pp. 52–59, 2022. doi: 10.30880/jscdm.2022.03.02.006.

[42] Narayanan V., Detweiler D., Huang T., Burtsev A.: DRAMHiT: A hash table architected for the speed of DRAM. In: *EuroSys '23: Proceedings of the Eighteenth European Conference on Computer Systems*, pp. 817–834, Association for Computing Machinery, New York, NY, USA, 2023. doi: 10.1145/3552326.3587457.

[43] Oussous A., Benjelloun F.Z., Ait Lahcen A., Belfkih S.: Big Data technologies: A survey, *Journal of King Saud University – Computer and Information Sciences*, vol. 30(4), pp. 431–448, 2018. doi: 10.1016/j.jksuci.2017.06.001.

[44] Pagh R., Rodler F.F.: Cuckoo hashing. In: F.M. Heide auf der (ed.), *Algorithms – ESA 2001*, pp. 121–133, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. doi: 10.1007/3-540-44676-1.

[45] Pagh R., Rodler F.F.: Cuckoo hashing, *Journal of Algorithms*, vol. 51(2), pp. 122–144, 2004. doi: 10.1016/j.jalgor.2003.12.002.

[46] Pătraşcu M., Thorup M.: The power of simple tabulation hashing, *Journal of the ACM*, vol. 59(3), 2012. doi: 10.1145/2220357.2220361.

[47] Pontarelli S., Reviriego P., Mitzenmacher M.: EMOMA: Exact match in one memory access, *IEEE Transactions on Knowledge and Data Engineering*, vol. 30(11), pp. 2120–2133, 2018. doi: 10.1109/tkde.2018.2818716.

[48] Porat E., Shalem B.: A cuckoo hashing variant with improved memory utilization and insertion time. In: J.A. Storer, M.W. Marcellin (eds.), *2012 Data Compression Conference, Snowbird, UT, USA, April 10–12, 2012*, pp. 347–356, IEEE Computer Society, 2012. doi: 10.1109/DCC.2012.41.

[49] Rajwar K., Deep K., Das S.: An exhaustive review of the metaheuristic algorithms for search and optimization: taxonomy, applications, and open challenges, *Artificial Intelligence Review*, vol. 56, pp. 13187–13257, 2023. doi: 10.1007/s10462-023-10470-y.

[50] Ramakrishna M.: Analysis of random probing hashing, *Information Processing Letters*, vol. 31(2), pp. 83–90, 1989. doi: 10.1016/0020-0190(89)90073-2.

[51] Ramakrishna M.V.: Hashing practice: analysis of hashing and universal hashing, *ACM SIGMOD Record*, vol. 17(3), pp. 191–199, 1988. doi: 10.1145/50202.50223.

[52] Regassa D., Sung D., Kim S., Yeom H.Y., Son Y.: EHS: An efficient hashing scheme for persistent memory. In: *SAC '23: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pp. 301–304, Association for Computing Machinery, New York, NY, USA, 2023. doi: 10.1145/3555776.3577850.

[53] Rivest R.L.: On hash-coding algorithms for partial-match retrieval. In: *15th Annual Symposium on Switching and Automata Theory (swat 1974)*, pp. 95–103, IEEE, 1974. doi: 10.1109/swat.1974.17.

[54] Sanders P.: Hashing with linear probing and referential integrity, 2018. doi: 10.48550/arXiv.1808.04602.

[55] Shahbazi N., Sintos S., Asudeh A.: FairHash: A fair and memory/time-efficient hashmap, *Proceedings of the ACM on Management of Data*, vol. 2(3), 136, 2024. doi: 10.1145/3654939.

[56] Shi S., Qian C.: Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems, *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4(2), pp. 1–32, 2020. doi: 10.1145/3392140.

[57] Sprugnoli R.: Perfect hashing functions: a single probe retrieving method for static sets, *Communications of the ACM*, vol. 20(11), pp. 841–850, 1977. doi: 10.1145/359863.359887.

[58] Sun Y., Hua Y., Chen Z., Guo Y.: Mitigating asymmetric read and write costs in cuckoo hashing for storage systems. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 329–344, USENIX Association, Renton, WA, 2019. https://www.usenix.org/conference/atc19/presentation/sun.

[59] Sun Y., Hua Y., Feng D., Yang L., Zuo P., Cao S., Guo Y.: A collision-mitigation cuckoo hashing scheme for large-scale storage systems, *IEEE Transactions on Parallel and Distributed Systems*, vol. 28(3), pp. 619–632, 2017. doi: 10.1109/TPDS.2016.2594763.

[60] Thorup M.: Fast and powerful hashing using tabulation, *Communications of the ACM*, vol. 60(7), p. 94–101, 2017. doi: 10.1145/3068772.

[61] Walzer S.: Load thresholds for cuckoo hashing with overlapping blocks, *ACM Transactions on Algorithms*, vol. 19(3), 24, 2023. doi: 10.1145/3589558.

[62] Wiederhold G.: *Database design*, vol. 1077, McGraw-Hill New York, 1983.

[63] Wu X., Zhu X., Wu M.: The evolution of search: Three computing paradigms, *ACM Transactions on Management Information Systems (TMIS)*, vol. 13(2), 20, 2022. doi: 10.1145/3495214.

[64] Yang Q., Huang H., Zhang J., Gao H., Liu P.: A collaborative cuckoo search algorithm with modified operation mode, *Engineering Applications of Artificial Intelligence*, vol. 121, 106006, 2023. doi: 10.1016/j.engappai.2023.106006.

Affiliations

Rajeev Ranjan Kumar Tripathi
Madan Mohan Malaviya University of Technology, Department of Computer Science and Engineering, Gorakhpur, Uttar Pradesh, India, rajeevranjankumartripathi@gmail.com (Corresponding author)

Pradeep Kumar Singh
Madan Mohan Malaviya University of Technology, Department of Computer Science and Engineering, Gorakhpur, Uttar Pradesh, India, topksingh@gmail.com

Sarvpal Singh
Madan Mohan Malaviya University of Technology, Department of Information Technology and Computer Application, Gorakhpur, Uttar Pradesh, India, singhsarvpal@gmail.com

Received: 14.06.2024
Revised: 30.07.2024
Accepted: 31.07.2024