Collision Resolution Techniques in Hash Table: A Review

Ahmed Dalhatu Yusuf[1], Saleh Abdullahi[2], Moussa Mahamat Boukar[3], Salisu Ibrahim Yusuf[4]
Nigerian Communications Commission[1]
Department of Computer Science, Nile University of Nigeria, Abuja, Nigeria[2,3,4]

***Abstract*—One of the major challenges of hashing is achieving constant access time $O(1)$ with an efficient memory space at a high load factor environment when various keys generate the same hash value or address. This problem causes a collision in the hash table, to resolve the collision and achieve constant access time $O(1)$ researchers have proposed several methods of handling collision most of which introduce a non-constant access time complexity at a worst-case scenario. In this study, the worst case of several proposed hashing collision resolution techniques are analyzed based on their time complexity at a high load factor environment, it was found that almost all the existing techniques have a non-constant access time complexity. However, they all require an additional computation for rehashing keys in a hash table some of which is as a result of deadlock while iterating to insert a key. It was also found out that there are wasted slots in a hah table in all the reviewed techniques. Therefore, this work, provides an in-depth understanding of collision resolution techniques which can serve as an avenue for further research work in the field.**

Keywords*—*Hashing; collision resolution; hash table; hash function; slot

I. Introduction

Hashing is a data structure for searching an element from a collection with the primary goal of achieving a constant time complexity$O(1)$ [6], [7], [1]. It uses a hash function $h(key)$ to generate an address or a hash value of an element in a hash table. A hash table is a collection of slots in memory defined for storing a set of keys. A number of hashing techniques exist, all of them use a hash function to identify an address of a key in order to achieve constant access time both for insertion and searching a key. However, In a situation where a hash function $h(key)$ generates the same address/hash value for more than one key that introduces a collision in the hash table. The constraint of a hash table is only one element or key can be placed in a single slot. Therefore, to resolve the collision researchers have proposed several collision resolution techniques such as probing techniques, double hashing, separate chaining, cuckoo hashing etcetera for handling the colliding keys in the hash table. A number of these techniques introduce a non-constant time complexity in a high load factor environment.

Existing work on the hashing techniques generally focus on security oriented hashing [23], applications of hashing [9], [20], [8], and general comprehensive knowledge of hashing techniques. However, in this work we focused on runtime complexity of the existing technique and identifying the problems of the existing techniques so that a research gap will be provided which can serve as an avenue to improve upon collision resolution techniques. This is due to the importance hashing demonstrated in insertion, searching, and matching in many areas of computer [9], [8], [3]. In cryptography such as password verification, message digest, and Rabin-Karp algorithm. Therefore, a development in the area of hashing will advance the efficiency of several applications across many areas of computing.

In this work relevant proposed collision resolution techniques in hash table were reviewed. Highlighting the hash function employed in each method, how key is hashed into a hash table, key retrieval strategies and costs based on worse-case runtime complexity, alongside problems associated with each existing technique and we also provided a research gap in hashing technique for researchers to improve on.

II. Related Work

The research effort of Lianhua *et al* [9] proffered the main ideas on the prevailing hashing techniques for various "data and applications", it investigated the similarities and robustness of each technique in the two categories demonstrated in their work: data-situated and security-situated hashing and also present a brief application domain that requires hashing. The work was conducted due to an increase in the volume of data generated every day from diverse areas such as social network activities, daily transactions in the business domain, data from IoT applications, and other numerous domains. This increment of data has led to significant issues in analyzing and processing data and hashing strategy has been an efficient approach for fast data access for decennaries. The techniques and applications reviewed in their work have shown the positive impact hashing has on the performance of various applications such as networking, image classification, text classification and thus makes it an interesting area of study in order to make real-world applications very efficient.

The work of Tom [15] Indicated that several algorithms are implemented using dictionary complex type that most high-level programming comes with. This dictionary type could be implemented in several ways with a different data structure. However, a study has shown that for better lookup performance it should be implemented with a hash table. Python dictionary takes advantage of that, it uses a hash table with open addressing [30]. But the problem with that is whenever a collision occurs probing method has to happen. Therefore, In an environment where the collision is high then the lookup performance reduces. The trivial method is to use chainning for dictionary implementation.

Abhay [20] presented a technique that minimized the memory space used in implementing the hash table by compressing the key for data-item in the hash table. The hash value $h(key)$ can be generated by any hash function and subsequently, the compressed value is generated to mapped the input key.

Sailesh *et al* [8] proposed a technique that try to achieve constant time for retrieving keys at high load factor environment and memory minify bandwidth in high-performance networking subsystem. The method uses a hash table with many multiple logical chunks given a $key\ n$ likely slot in memory. An item will be mapped with $h(k)$, which inserted the key in the search space U in the scope of the chunked subscripts of the hash tables, i.e. *h:* $U = \{0, 1, ..., |U| - 1\}$, where $|U|$ is the length of the chunked hash table. An item can be inserted in a bucket $h(k)$ in any of the chunked hash tables. In the event where all the chunked buckets for *h(k)* are not empty then a collision is inevitable.

Yuanyuan *et al* [4] improved upon open address cuckoo hashing by overcoming the problem of an infinite loop at a time of insertion, which reduces the efficiency of query processing. They proposed a better technique called SmartCuckoo, which presents the relationship of hashing using directed pseudo forest and uses it subsequently to indict element placement for the correct determination of the existence of an infinite loop. SmartCuckoo can also predict insertion failure without going through some probing steps. However, in some environment the prediction might not be accurate. Therefore, there is still a change for an infinite loop. The work has been implemented on a “cloud storage system” the source code has been released for public users.

Peter *et al* [12] presented an approach for resolving collision in one-dimensional array. The technique concatenated (dot (.)) the key and the *h(k)* and insert it in the first empty bucket in the hash table. For example, in a hash table of *size* 7, *h(23) = 23* mod *7*, will be placed 2.23 at the first available cell in the hash table. Searching an element in this technique is linear *O*(n), hence the *h(k)* will not help in locating the key from the hash table.

Randomized hashing was proposed by Shai *et al* [11] as a process that takes a message and returns a hash value of the message that can be used in digital signature without any modification in traditional hash function such as SHA. The objective of their work is to free “digital signature schemes” from their dependence on collision contention.

A. Collision Resolution Strategies

The circumstance where a hash value calculated using a hash function matches with another hash value that is already involved within the hash table is named as collision [24]. To place all the colliding keys in the hash table that could be achieved using a method called collision resolution. Collision resolution refers to a situation when two items hash to the same slot, and a systematic method must be used to insert the second item in another slot in the hash table. An example of collision is described in Fig. 1.

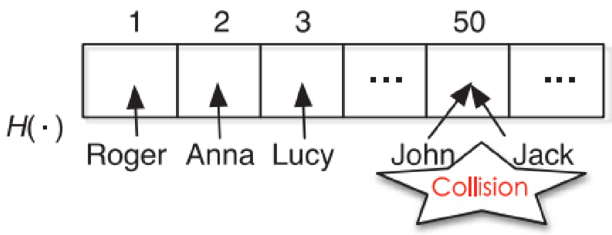

Fig. 1. Example of Collision [9].

1) Open Addressing: is a technique that resolve colliding keys in the hash table by looking for an empty slot using some sequence of probing techniques to find a new slot for an element that caused the collision [28]. This hash table has a probe sequence which is usually in the form: *(h(k) = [h(k) + c(i)* mod *n, for i=0,1,..,n-1)* where *h* is the hash function and *n* is the size of the hash table. The function *c(i)* is required to iterate through *n-1*.

The probe sequence method include the following:

- Linear probing, in this method if a collision occurs it resolve it by finding the next empty slot in the hash table and place a key. To search a key say y, the approach is to look for it in a hash table starting with an index of $h(y)$ and continue to the next slot in the hash table i.e. $h(y)+1, h(y)+2, ...$, until an empty slot is reached or a slot whose content is y. If an empty slot is reached then the key is not in the hash table. The problem with this method is clustering, (likelihood of one collision causing neighbouring bucket collision) at a high load factor which degrades its performance [25]. The method was founded by “Gene, Elaine and Samuel” [26].
- Quadratic probing, is another open addressing collision resolution method in which the interval to place a key if collision occur is quadratic i.e $h(key) + 1^2, h(key) + 2^2, ..., h(key) + n^2$. This technique considers better than open addressing with linear probing since it keeps away from the problem of clustering, even though it is not resistant to it [19]. The major problem with this method is finding an empty bucket is challenging when the hash table is $> 59\%$ full [27].
- Double hashing, this method resolve a collision by using another hash function to determine the interval to insert a key. For instance given, two different hash functions h_x and h_y, the position i of key in the hash table of size $|n|$ slots is; $h_x(key) + i * h_y(key)\%|n| for\ i = 1, ..., n-1$, where h_x and $h_y \in \cup = \{h_a, h_b, ..., h_z\}$ [29].

The work of Benjamin *et al.* [10] demonstrated the usage and efficiency of open addressing with quadratic probing to handle collision in communication between applications that use different communication design, computation, and data structures. For example, data can be distributed to many processes but each process will carry different tasks independently on the data. The work offered a Berkeley Container Library BCL; a cross-platform data structure library for a “one-sided communication” environment for parallel applications. The BCL is composed in C++ programming and its data structures phase are intended to be sans coordination, utilizing one-sided communication primitives that can be executed utilizing RDMA equipment while not requiring coordination with remote CPUs. Along these lines, BCL is steady with the soul of Partitioned Global Address Space GAS language, however profferer efficient add and search operations in the hash table, instead of reading and read operation of PGAS languages. BCL provides a central data structure for all the processes and can be used by each process in a parallel program.

2) Separate Chaining: This strategy uses a collection of nodes known as a linked list or list data structure to resolve the colliding keys in the hash table whenever a collision occurs as described in Fig. 2. In a high load factor environment this method provide a non-constant time complexity of $O(n)$ for inserting and retrieving a key from the hash table and tends to cause problem of tracking linked list [5]. However, another method invented by Dhar *et al.* [16] provide better performance from $O(1+n)$ to $O(logn)$. The technique uses a binary search tree to chain a collide key rather than using a list or linked list which reduces the time for searching a key. The problem of this method is an additional cost of balancing BST when the inserted keys cause a skewed binary tree.

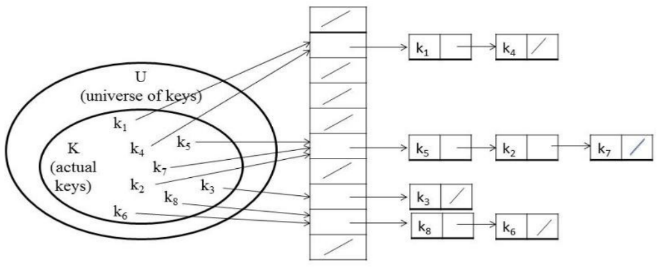

Fig. 2. Separate Chaining using Linked list [12].

3) Coalesced Hashing: is approach takes advantage of two different collision resolution techniques to handle collision in a hash table, an open addressing and chaining. It uses similar insertion procedure as open addressing to insert an element in the hash table using $h(k)\ mod\ n$. When a collision occurs at *i* position in the hash table, coalesced hashing resolve it similar to separate chaining by inserting the key that causes the collision in the first empty slot from the bottom of the hash table i.e. *for (i ¡ n, i ¡=0, i–)* where *i* is an index and *n* is the size of the hash table. It then chains the colliding key original hash value to hash value the colliding key is inserted using a pointer instead of creating a linked list like separated chaining. It minimizes space usage but constant time lookup is not achievable at a high load factor like an open addressing and separate chaining [2]. Any open addressing method can be used to identify a position to insert a key that collides in coalesced hashing.

4) Cuckoo Hashing: This is another open addressing technique that was first introduced in 2004 by “Flemming and Rasmus” [13]. The method is ubiquitous and uses in an array of real-life applications [14], [17], [18], [21]. It uses two or more hash functions to insert key to slot in a hash table, which means any key in *U* = {*k1, k2... kn*} can be in more than one slot. Any key can also be relocated to another slot in the hash table. Insert a key has a number of hash function options say, *h1(k)* and *h2(k)*. Relocation of a key can be done if *h1(k)* and *h2(k)*. are not free for insertion. This problem can be overcome by relocating an existing key to a new slot using another *h(k)* and supersede the new key into relocating key position hashing. If the relocating key *h(k)* is not empty, then repeat relocating key supersede another key until the method gets a free slot. In a situation it iterates through the hash table without resolving the problem, all the keys will be rehashed with different *h(k)*. *N* number of rehashes might be conducted in order for cuckcoo to achieve. However, “MinCounter” technique presented in [22] reduced the number of rehashing by superseding a new key with a rarely accessed key to address collision in a hash table instead of superseding any random key. Each slot in a MinCounter method has a counter variable that keeps track of the number relocation that occurs at a slot. To insert a new key it checks the counter variable and inserts it into a slot with a minimum value rather than iterating through the hash tables for an empty slot to place a key. In a situation of insertion failure, a key is placed in the “memory cache” to avoid rehashing. “MinCounter” provides better performance for inserting and query response in cloud services. The structure of this algorithm is described in Fig. 3.

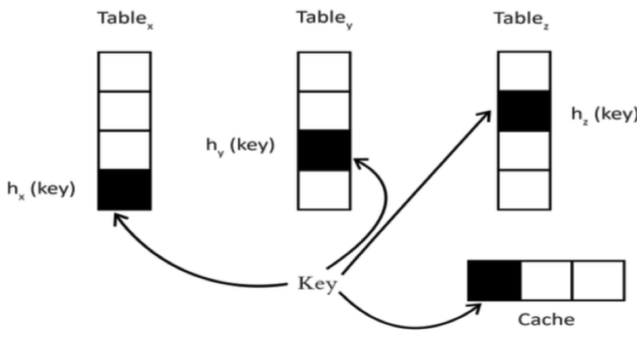

Fig. 3. Example of MinCounter Hashing Technique [22].

Jeyaraj *et al* [14] research effort improve the performance of eliminating duplicate fingerprint date in data duplication method for a backup system using cuckoo hashing. The approach in the work is implemented with two tables T_i and T_j. To insert a new fingerprint. We use fp to denote fingerprint, It will check; **if** $T_i[h(fp)] = fp$ then it will skip the insertion. Else it checks; **if** $T_i[h_i(fp)]$ = NULL then; $T_i[h_i(fp)] = fp$ else; **if** $T_i[h_i(fp)] \neq$ NULL then check; **if** $T_j[h_j(fp)]$ = NULL then; $T_j[h_j(fp)] = fp$ It continue with normal cuckoo process until all the available fingerprint are distributed into the two hash table depending on the when cuckoo succeed. The research work also support parallel insertion into the hash tables which gives the technique a better throughput and minimises memory space compare to [5] but add cost for inserting a key similar to techniques in [28], [10], [2], [13].

III. RESEARCH GAP

Hashing is indeed a very important algorithm that depicted interest from many areas of storage systems. It must be efficient for retrieving an element at all times and the memory space should be utilized efficiently due to the limited nature of the storage system. The worse-case time complexity, mapping method, key retrieval approach and problem of most of the important techniques is shown in Table I. Reviewed research efforts above tried to use hashing in various domains to

TABLE I. SUMMARY OF SOME OF THE IMPORTANT REVIEWED HASH COLLISION RESOLUTION TECHNIQUES

Technique	Mapping	Lookup	Problem	Time complexity of lookup & mapping (worst-case)
Linear probing [26]	$x \leftarrow h(key)\ mod\ n,\ i \leftarrow 0$ **while** $(x \neq NULL)\{\ i \leftarrow i{+}1,\ x \leftarrow x{+}i\ mod\ n\}$ insert 'key' at x position. Where n is the size of the hash table.	$x \leftarrow h(key)\ mod\ n,\ i \leftarrow 0$ **while** $(x < n)\{$**if**$(x = key)$ print 'Key found' **break loop** $i \leftarrow i+1,\ x \leftarrow x + i\ mod\ n$ **if**$(x = NULL)$ print 'Key doesn't exist' **break loop**$\}$.	• Clustering problem • Wasted slot(s) • Rehashing	$O(n),\ O(n)$
Quadratic probing [19]	This works similar to linear probing but move in quadratic form to resolve a collision. $x \leftarrow h(key)\ mod\ n,\ i \leftarrow 0$ **while** $(x \neq NULL)\{\ i \leftarrow i+1,\ x \leftarrow x + i^2\ mod\ n\}$ insert 'key' at x position. Where n is the size of the hash table.	$x \leftarrow h(key)\ mod\ n,\ i \leftarrow 0$ **while** $(x < n)\{$**if**$(x = key)$ print 'Key found' **break loop** $i \leftarrow i+1,\ x \leftarrow x + i^2\ mod\ n$ **if**$(x = NULL)$ print 'Key doesn't exist' **break loop**$\}$.	• This technique keeps away from clustering problem although is not resistant to it. • Wasted slot(s) • Rehashing • Finding empty slot if hash table is $> 59\%$ occupied is challenging.	$O(n),\ O(n)$
Double hashing [29]	This technique uses two different hash functions h_x and h_y, the position i of key in the hash table of size $\|n\|$ slots is; $h_x(key) + i * h_y(key)\%\|n\|$**for** $i = 1, ..., n-1$, where h_x and $h_y \in \cup = \{h_a, h_b, ..., h_z\}$	$i \leftarrow 0,\ x \leftarrow h_x(key),\ y \leftarrow h_y(key)$ **while**$(T[(x{+}i{*}y)\ mod\ n] \neq key)\{$ **if**$(T[(x{+}i{*}y)\ mod\ n] = -1)\{$ print "key does not exist" **break**$\}$ $i \leftarrow i+1$ $\}$ print "Key found"	• Wasted slot(s) • Rehashing	$O(n),\ O(n)$
Coalesced hashing [2]	$x \leftarrow h(key)\ mod\ n,\ i \leftarrow n$ **while** $(x \neq NULL\ AND\ i > -1)\{\ i \leftarrow i-1,\ x \leftarrow x + i\ mod\ n\}$ insert 'key' at x position then set a pointer from the colliding position. Where n is the size of the hash table.	$x \leftarrow h(key)\ mod\ n,\ i \leftarrow n$ **while** $(x < n\ AND\ i > -1)\{$**if**$(x = key)$ print 'Key found' **break loop** $i \leftarrow i-1,\ x \leftarrow x + i\ mod\ n$ **if**$(x = NULL)$ print 'Key doesn't exist' **break loop**$\}$.	• Wasted slot(s) • Rehashing	$O(n),\ O(n)$
Cuckoo hashing [13]	This method uses two hash tables (T_1 & T_2) and two hash functions $h_1(key)$ & $h_2(key)$). $h_1(key) = key\ mod\ n$ and $h_2(key) = (key/n)\ mod\ n$. **if** $(T_1[h_1(key) = key\ OR\ T_2[h_2(key) = key]])$ print "Key already exist" **else while**(true)**{** key swap $T_1[h_1(key)]$ **if** $key = NULL\ key$ swap $T_2[h_2(key)]$ **if** $key = NULL$**}** rehash all keys then try inserting the key.	Similarly, retrieve a key uses the two hash tables (T_1 & T_2) and two hash functions $h_1(key)$ & $h_2(key)$). $h_1(key) = key\ mod\ n$ and $h_2(key) = (key/n)\ mod\ n$. **if** $(T_1[h_1(key) = key\ OR\ T_2[h_2(key) = key]])$ print "Key already exist" **else** print "Key does not exist"	• Wasted slots • Rehashing • High cost of insertion which could lead to deadlock	$O(1),\ O(n)$
Separate chaining with linked list hashing [5]	This method uses $h(key)\ mod\ n$ to insert a key like linear probing. But resolve the colliding keys by using linked list.	**if** $(h(key)\ mod\ n)$ = key print "Key found" **else**$\{\ p \leftarrow head$ **while**$(p \neq NULL$ AND $p.info \neq key)\{$ access $p.info\quad p = p.link\ \}$	• Wasted slots • Rehashing	$O(n),\ O(n)$
Separate chaining with binary search tree hashing [16]	This works similar to separate chaining with a linked list but it resolve collision using a binary search tree. The time complexity for searching a key from a binary search tree is $O(log\ n)$.	$node \leftarrow start$ **while** $(node \neq NULL)\{$ **if**$(key[node] = key)$ return y else if $key[node] < key$ then $node \leftarrow$ right[node] else $node \leftarrow$ left[node] $\}$ print "Key not found". Where $start$ is the root node.	• Wasted slots • Rehashing • Computation for balancing skew tree	$O(log\ n),\ O(log\ n)$

improve the performance of retrieving, matching, and inserting data. However, these existing collision resolution techniques both have their pros and cons. In this work we identified three major issues of the current techniques mention below:

Issues I, All the existing technique mentioned reviewed in this work suggested using prime number as a size of a hash table $|T|$, usually in form:

$$s = (x \times |key|)$$
$$|T| => p(s)$$

where $> p$, means next prime number of s and $x > 1.0$. The problem with this method it creates a wasted slot in the hash table and in that case it does not achieve the goal of the hashing that is primarily designed to utiliaze minimum amount of memory to store data, which is less than the amount to store the actual data [26].

Theorem: For n arbitrary set of keys K to hash into a hash table T

$$|T| => p\{x \times |K|\} \Rightarrow \exists_{wastedSlot} \in T$$

Proof: Consider a set of positive integers $K = \{k_1, k_2, ..., k_n\}$ to hash into a hash table

$$|T| => p(x \times |K|)$$
$$Let\ I_{i=1}^{|K|} = \begin{pmatrix} i{+}{+} & if\ h(k_i) \to T \\ i{=}i{+}0 & else \end{pmatrix}$$
$$h(k_i) = k_i\ mod\ |T|$$
$$|T| > |K|$$

So, after mapping k_n into $T,\ I$ will be equal to $|K|$
Therefore, $wastedSlot = |T| - I$

Issues II, Rehashing is a problem with all the existing techniques, which has to be done whenever there is an additional element to hash as a result of determining the size of hash table described in *issues I* and also when a certain threshold is reached which was considered to be set for some hashing technique like, double hashing technique [29]. For any arbitrary key is map into T

$$K = \{k_1, k_2, ..., k_n\} \to T[0, 1, ..., m-1]$$

Every ith location of a key is determine with:

$$h(key)\ mod\ |T|$$

Therefore, any change in $|T|$ all the element need to be remapped into T:

$$h(key) \to T\ using\ h(key)\ mod\ |T_{new}|$$

Issues III, The better performance technique among the reviewed works is cuckoo hashing, which has a time complexity of $O(1)$ to search for an element but has a deadlock problem which is a result of a high number of relocations before inserting an element. To resolve this problem of deadlock entire rehashing of the keys has to be done which is quite time-consuming and not efficient. It also has a high amount of wasted slots compare to other reviewed technique this because it uses two hash tables.

IV. Conclusion

In conclusion, this work reviewed a number of different collision resolution techniques in the hash table. These techniques were employed in many areas of computer science such as IP address lookup, job balancing, security, etc. The time complexity for inserting and retrieving an element of all the collision resolution techniques was identified. However, we found that achieving constant access time with a good insertion performance is still challenging with all the current collision resolution techniques. This work provided runtime complexity and the major problems associated with the existing collision resolution techniques which can serve as an avenue for further research in the field. Here, the analysis was based on worse-case time complexity of the respective algorithms. However, other future research should consider other different aspect of algorithm analysis such as space complexity and most suitable conditions with respect to input size through mathematical notation or simulation.

Acknowledgment

I will like to express my gratitude to my parents for their advice and support. Many thanks goes to Nile University of Nigeria for the 90% financial support opportunity provided to carryout this work.

References

[1] Black, Paul E. ”DADS: The On-Line Dictionary of Algorithms and Data Structures,” NIST: Gaithersburg, MD, USA, 2020.

[2] Sriram, Ranjena, et al. ”Efficient Data Cleaning Algorithm and Swift Unique User Identification Algorithm Using Coalesced Hashing and Binary Search Techniques for Web Usage Mining,” International Journal of Pure and Applied Mathematics 118.18, 2018.

[3] Brenton Lessley and Hank Childs. ”Data-Parallel Hashing Techniques for GPU Architectures,” IEEE Transactions on Parallel and Distributed Systems Volume: 31, Issue: 1, 2020.

[4] Sun, Yuanyuan, et al. ”SmartCuckoo: a fast and cost-efficient hashing index scheme for cloud storage systems.” 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17). 2017.

[5] Brad Miller, David Ranum. ”Problem Solving with Algorithms and Data Structures,” 2013

[6] Necaise, Rance D. ”Data structures and algorithms using Python,” Wiley Publishing, 2010.

[7] Cormen, Thomas H., et al. Introduction to algorithms. MIT press, 2009.

[8] Sailesh Kumar, Patrick Crowley. ”Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems,” 2005 Symposium on Architectures for Networking and Communications Systems (ANCS), 2005.

[9] Lianhua Chi, Xingquan Zhu. ”Hashing techniques: A survey and taxonomy,” ACM Computing Surveys, Vol. 50, No. 1, Article 11, 2017.

[10] Brock, Benjamin, Ayd?n Buluc, and Katherine Yelick. ”BCL: A cross-platform distributed data structures library,” Proceedings of the 48th International Conference on Parallel Processing. 2019.

[11] Halevi, Shai, and Hugo Krawczyk. ”Strengthening digital signatures via randomized hashing,” Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 2006.

[12] Nimbe, Peter, Samuel Ofori Frimpong, and Michael Opoku. ”An efficient strategy for collision resolution in hash tables,” International Journal of Computer Applications 99.10, 2014.

[13] Pagh, Rasmus, Flemming Friche Rodler. ”Cuckoo hashing,” Journal of Algorithms 51.2, 2004.

[14] Jane Rubel A. Jeyaraj, Sundarakantham Kambaraj and Velmurugan Dharmarajan. ”High-speed data deduplication using Parallelized Cuckoo Hashing,” Turkish Journal of Electrical Engineering & Computer Sciences 2018

[15] Van Dijk, Tom. ”Analysing and improving hash table performance,” 10th Twente Student Conference on IT. University of Twente, Faculty of Electrical Engineering and Computer Science, 2009.

[16] Dhar, Siddharth, et al. ”A tree based approach to improve traditional collision avoidance mechanisms of hashing.” 2017 International Conference on Inventive Computing and Informatics (ICICI). IEEE, 2017.

[17] Debnath, Biplob K., Sudipta Sengupta, and Jin Li. ”ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory,” USENIX annual technical conference, 2010.

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

[19] Konheim, Alan G. ”Hashing in computer science: Fifty years of slicing and dicing,” John Wiley & Sons, 2010.

[20] Abhay Kulkarni. ”Efficient Hash Table Key Storage,” Avago Technologies International Sales Pte . Limited, Singapore (SG), 2019.

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

[22] Sun, Yuanyuan, et al. ”MinCounter: An efficient cuckoo hashing scheme for cloud storage systems,” 2015 31st Symposium on Mass Storage Systems and Technologies (MSST). IEEE, 2015.

[23] Arvind K. Sharma ; S.K. Mittal. ”Cryptography & Network Security Hash Function Applications, Attacks and Advances: A Review,”. 2019 Third International Conference on Inventive Systems and Control (ICISC), 2019

[24] Joux, Antoine. ”Multicollisions in iterated hash functions. Application to cascaded constructions.” Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 2004.

[25] Goodrich, Michael T., and Roberto Tamassia. ”Algorithm design and applications,” Hoboken: Wiley, 2015.

[26] Knuth, Donald E. ”Sorting and searching (6. printing, newly updated and rev. ed.). Boston [ua].” 2000

[27] Weiss, Mark Allen, ”Data Structures and Algorithm Analysis in C++,” Pearson Education. ISBN 978-81-317-1474-4, 2009.

[28] Agrawal, Anand, Sriram Bhyravarapu, and Nuthalapati Venkata Krishna Chaitanya. ”Matrix hashing with two level of collision resolution.” 2018 8th International Conference on Cloud Computing, Data Science & Engineering (Confluence). IEEE, 2018.

[29] Phillip G.Bradford and Michael N. Katehakis, ”A Probabilistic Study on Combinatorial Expanders and Hashing”, SIAM Journal on Computing, 37 (1): 83-111, doi:10.1137/S009753970444630X, 2017

[30] Kumar, Arun, and Supriya P. Panda. ”A survey: how python pitches in IT-world.” 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon). IEEE, 2019.