ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 4, Issue 1, pp: (91-98), Month: January - March 2016, Available at: www.researchpublish.com

Cuckoo Hashing and Drawbacks

Nitesh Gupta[1], Veeresh Erched[2]

[12]Amadeus Labs, Bangalore, India

Abstract: **We will present dictionary and hashing and then a simple concept of cuckoo hashing, its need and how it is better than existing know hashing algorithm. We will also try to analyze the drawback of this algorithm and we will see if we can solve that problem of not. We will also try to find the complexity of the algorithm and will compare it with existing hashing algorithm's complexity.**

Keywords: **Hashing, Collisions, Cuckoo Hashing, Memory Management.**

I. INTRODUCTION

Dictionary is an abstract data type composed of a collection of pairs (key, value), such possible keys appear once in the collection.

We can find a lot of examples related to dictionary in our daily life. Take an example of book. It contains index page which we can treat as dictionary for the book. Now here chapter number can be treated as the KEY and actual chapter can be treated as the VALUE.

Now why do we need dictionary as a data structure is the question that always comes in our mind. But the answer is as simple as the question. We can think of why we need Index page in a book (you can think of an encyclopaedia as the book). It helps us in searching a chapter or topic in really big books. Same concept is for the dictionary, in huge memory if you want to find some element it will be very difficult and time consuming. Hence, dictionary comes handy.

Now the most efficient dictionaries are based on hashing techniques. Hashing is the transformation of keys into hashes that are used for keys and values.

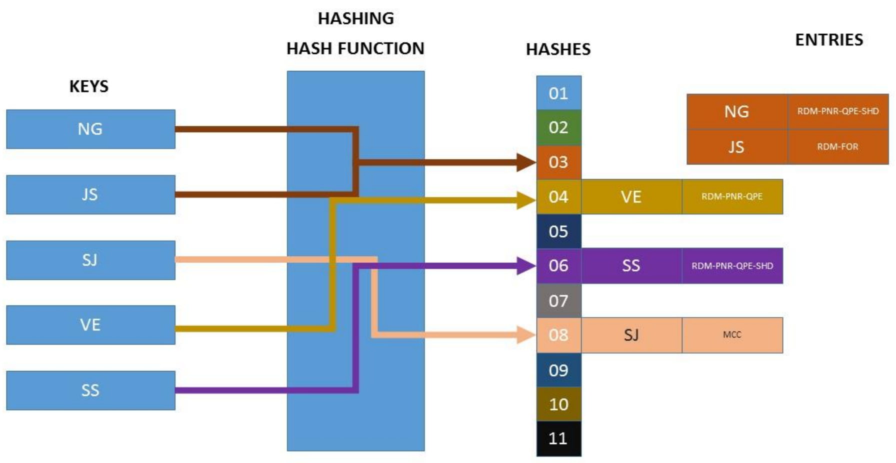

Fig. 1

Here you can see that we are converting KEYS into HASHES which you can treat as memory locations.

II. COLLISIONS

But as you can see two keys (NG & JS) are pointing to same hash. Now this is a problem known as COLLISIONS as you can store only one key in a location. To solve these problems we have many different solutions. Few famous and basic solutions are Bucketed Hashing & Linear Probing.

A. Bucketed Hashing:

In Bucketed Hashing also known as Separate Chaining, each bucket is independent and has some sort of list (mostly in form of linked list) of entries with the same index.

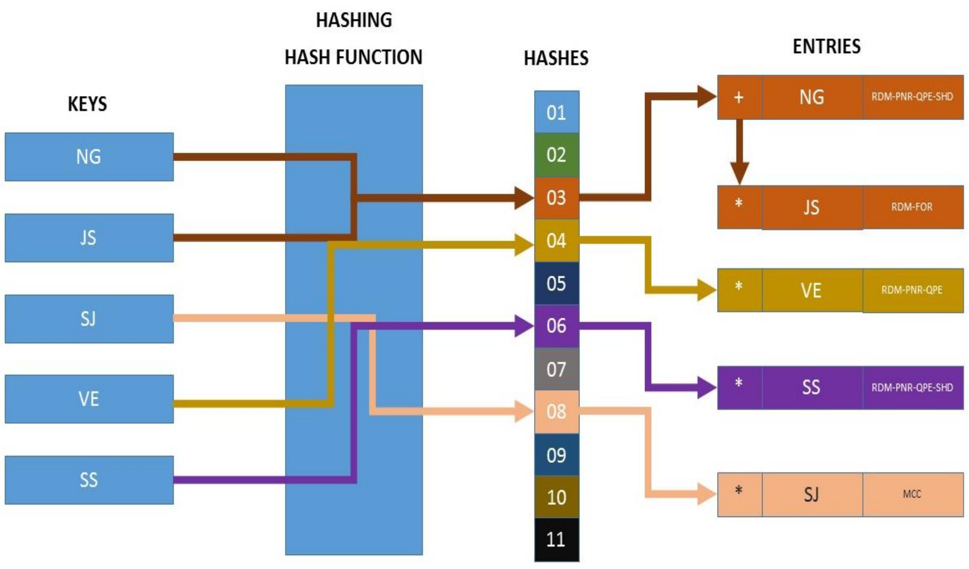

Fig. 2

B. Linear Probing:

In Linear Probing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in same sequence until an unoccupied slot is found.

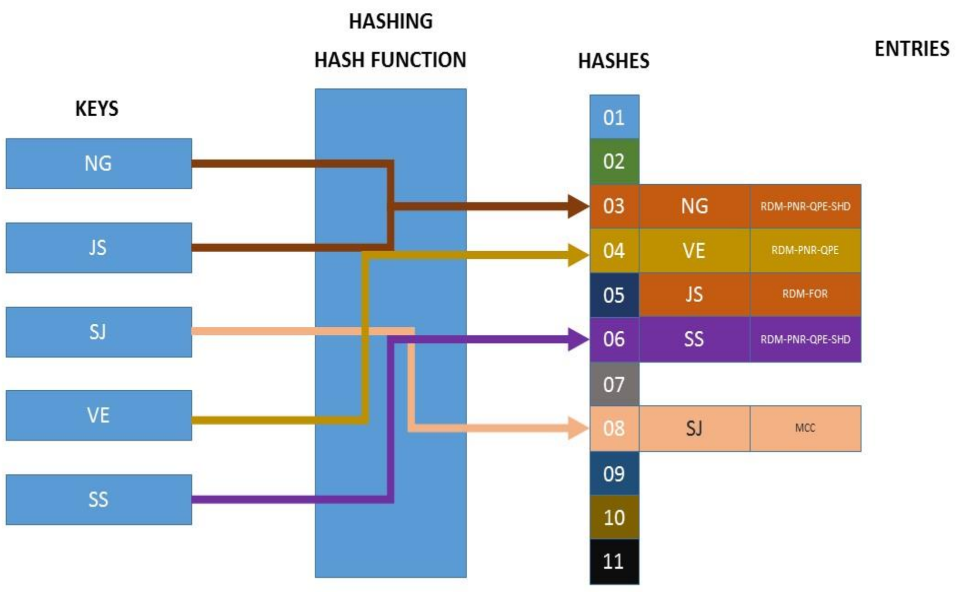

Fig. 3

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 4, Issue 1, pp: (91-98), Month: January - March 2016, Available at: www.researchpublish.com

III. CUCKOO HASHING

Cuckoo Hashing is an algorithm for resolving hash collisions of the values of the hash functions in the table and enhance the worst case lookup time.

The name derives from the behaviour of the bird CUCKOO where the bird lay eggs in another bird's nest and when these eggs are about to hatch, bird pushes other eggs and young out of the nest.

Algorithm behaves in the same way. If there is a collision then the existing element present in the location is pushed out and the new element take the place. But then the question comes as where will this existing element go. Then algorithm states that you should have another hash function which takes this element and find a new location to store the key. This process will repeat until every element is placed in an empty location.

Algorithm:

1. Let F (n) used for hashing is a hash function and initial assignment is F (1).
2. Hash key (HK) with Function F (n).
3. Check if the place is empty or not.
4. If place is not empty push the existing entry into TEMP.
5. Place HK into the location.
6. Check which function is used to place TEMP into the location.
7. Assign TEMP back to HK.
8. Assign alternate hash function into F (n).
9. Repeat from step 2 until there is no collision.

Now let's see how this algorithm solve the above collision problem. Let's say first all the keys are placed in empty locations (except JS). Now key JS came for the entry. It is hashed with function 1 and location is derived as 03. Now when the location is checked if it is free or not, it says that there is one entry stored at that location. Then entry (NG) is pushed out of the location and JS is stored at that location. Then this NG is hashed again and a new location is derived as 11. Now this new location is checked if it is free or not. Since location 11 is free, NG is placed at this location.

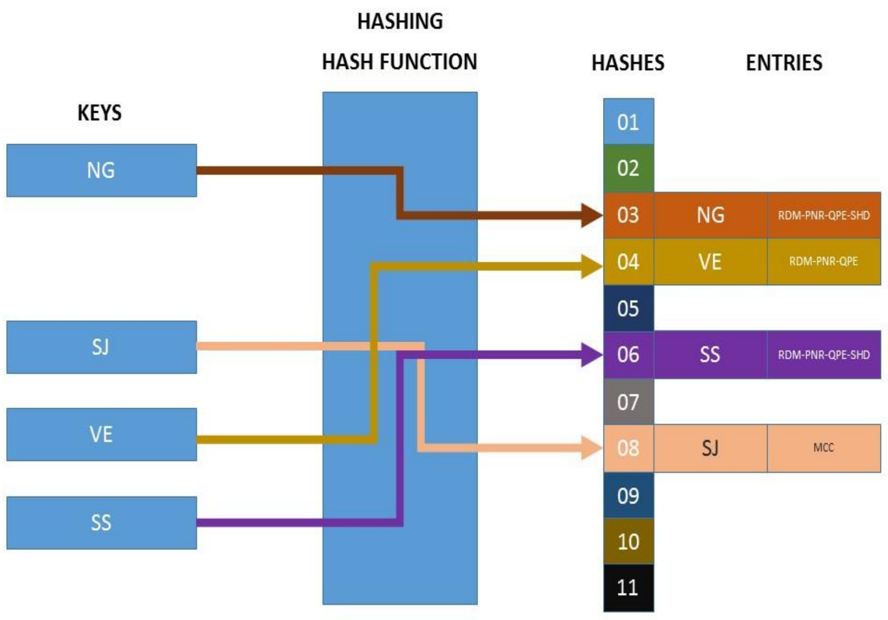

Fig. 4

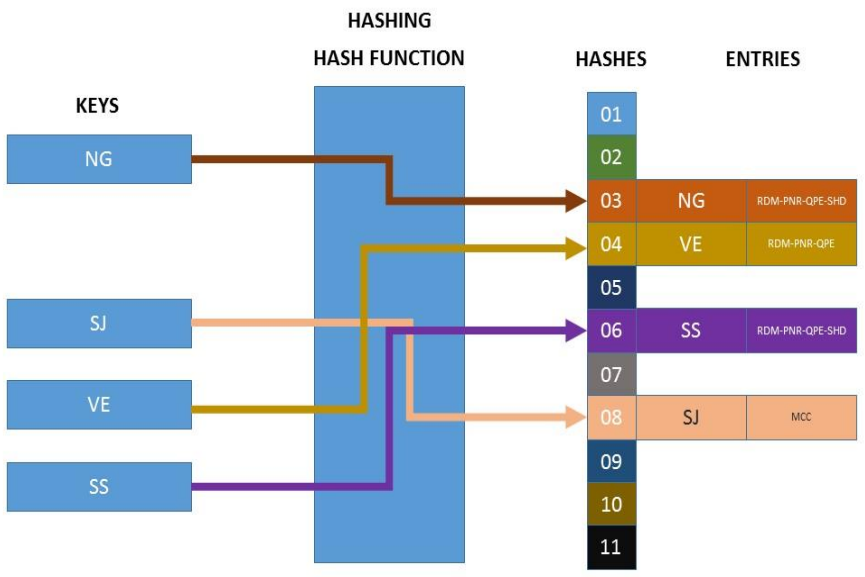

Fig. 5

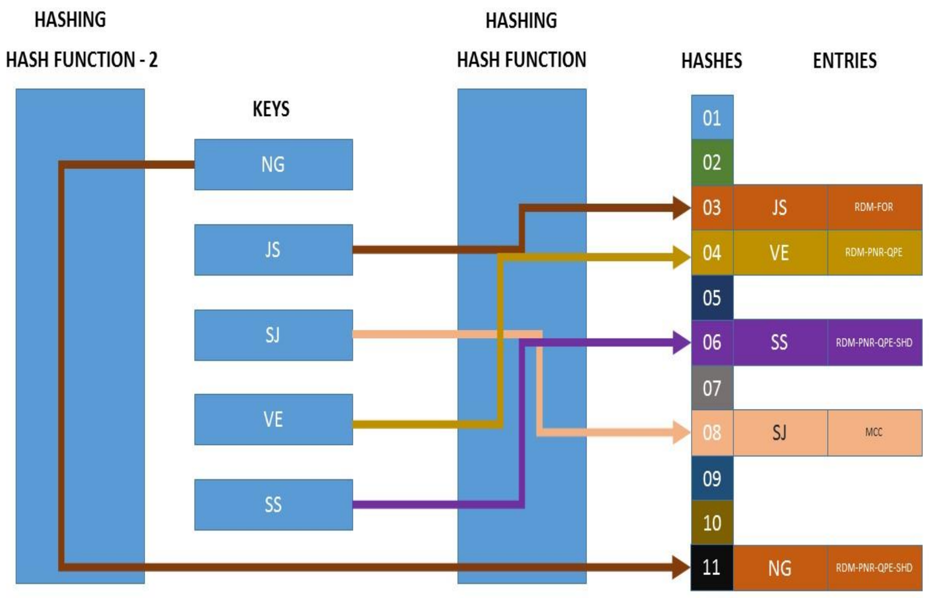

Fig. 6

IV. COMPLEXITY

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of string representing the input. It is commonly expressed using big O notation which excludes constants, coefficients and lower order terms. In other words, time taken by an algorithm to finish its execution is known as Time Complexity.

We usually deal with WROST CASE TIME COMPLEXITY, which defines the maximum time that algorithm can take for a particular set of input.

All the operations that we perform on Hash Table can be formed using two main operations:

1. INSERT
2. LOOKUP / SEARCH

Now let's see the Worst Case Time Complexity for all the algorithms that we have discussed above.

A. Bucketed Hashing:

Insert:

As we know when a Key needs to be inserted into Hash Table, it is hashed using Hash Function and a location is derived. Then we check if location is empty or not. If it is empty it is placed at that location. If not, it is inserted in the linked list.

Let's say every time a collision occurs we are inserting the new element at the beginning of the list. Hence, the algorithm will look like

a. Create a linked list node (X) with the Key.

b. Assign the existing pointer of the linked list in X.

c. Change the pointer to X.

This means for every insert there can be maximum of these 3 steps to be performed which is a constant value. Hence, the Time Complexity for this algorithm is big O (1).

Lookup / Search:

When we have to search a key in the Hash Table, it is hashed using the same hash function and with the location derived, we will get the value. But if there is a conflict then we traverse the linked list attached to the location. Hence, the algorithm will look like

a. Hash the key (K) using Hash Function and derive the location X.

b. Check if the key is present at location X.

c. If not then go to the next element in the list associated with the location and check for the element.

d. Perform step (c) until the key is found or end of the linked list.

This means in worst-case scenario, where every key is having a collision, for every search we need to traverse complete list. Hence, the Time Complexity for this algorithm is big O (n).

B. Linear Probing:

Insert:

In Linear Probing, if the derived location is empty it is placed at that location. If not then we check for the next adjacent location. This will go on until we find an empty location for the key. Hence, the algorithm will look like

a. If location is not empty location = location + 1.

b. Repeat step (a) until location is empty.

c. Place the Key in location.

This means, in worst-case scenario, for every insert, n number of locations are looked before any insertion. Hence, the Time Complexity for this algorithm is big O (n).

Lookup / Search:

In this algorithm, location is derived by hashing the Key, if Key is not present on the location we check for the next location until we find the Key or we reach the end of the locations. Hence, the algorithm will look like

a. Check if the key is present at location.

b. If not then location = location + 1.

c. Perform step (a) until the key is found or end of the locations.

This means in worst case scenario, where key is not present, we need to traverse complete list. Hence, the Time Complexity for this algorithm is big O (n).

C. Cuckoo Hashing:

Insert:

In Cuckoo Hashing, if the derived location is empty it is placed at that location. If not then the Key present at the location is removed and the new Key is placed at the location. The old Key is then hashed using another hash function to find the alternate location and repeated. This will go on until we find an empty location for the key. Hence, the algorithm will look like

a. Location = Hash of Key using Hash function Fn.

b. If location is not free, take out old Key and place new Key.

c. Key = old Key and Hash function Fn = alternate function.

d. Repeat step (a) until location is empty.

This means, in worst case scenario, for every insert there can be n number of locations are looked up before any insertion. Hence, the Time Complexity for this algorithm is big O (n). But the probability of occurrence of this situation is least as compared to other algorithms.

Lookup / Search:

In this algorithm, location is derived by hashing the Key using two Hash functions as a result Key can be found on exactly two locations.

a. Location = Hash of Key using Fn (1).

b. If Key if not present at location then Location = Hash of Key using Fn (2)

This means even in worst case scenario, Key must be present at one of the two locations. Hence, the Time Complexity for this algorithm is big O (1).

	Insert	**Lookup / Search**
Bucketed Hashing	O (1)	O (n)
Linear Probing	O (n)	O (n)
Cuckoo Hashing	O (n)	O (1)

V. PROBLEM WITH CUCKOO HASHING

As we have discussed earlier, cuckoo hashing needs two hash functions for avoiding collision. Now there is a possibility that collision occurs in such a way the algorithm enters into an infinite loop. This is one of the biggest problem with Cuckoo Hashing as these kind of problem can come with a probability of 20%.

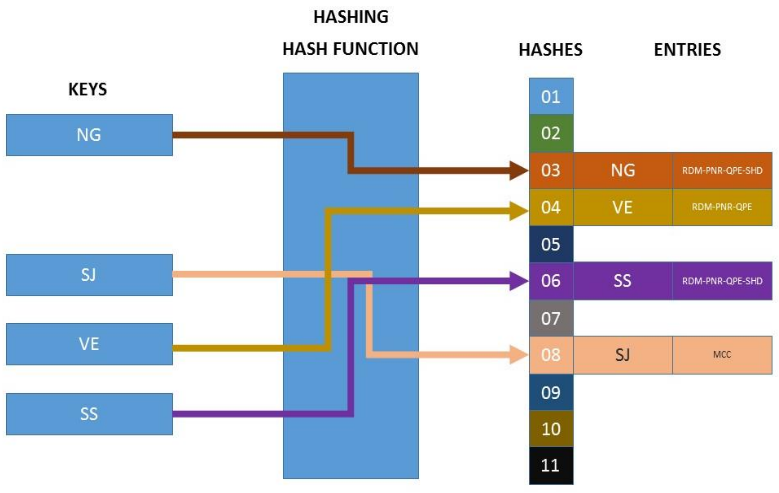

Fig. 7

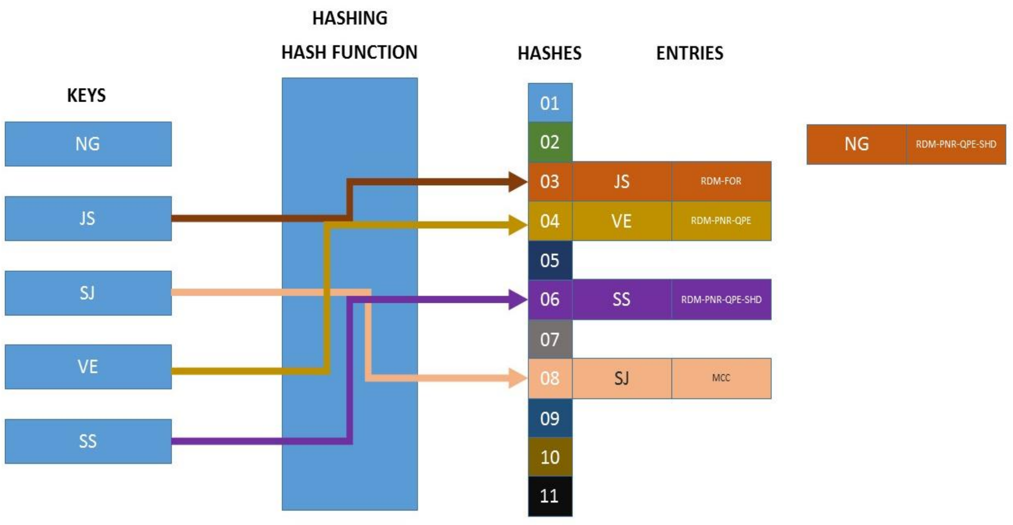

Fig. 8

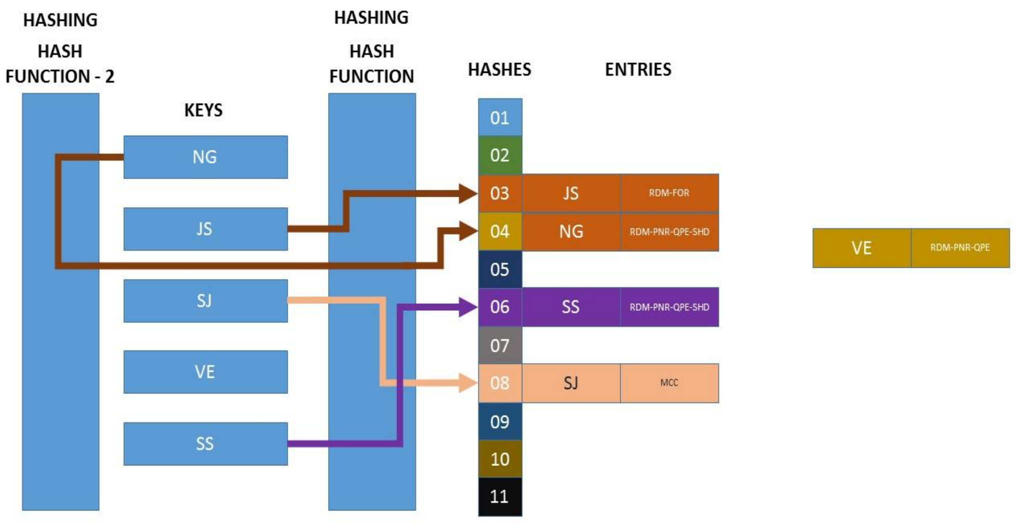

Fig. 9

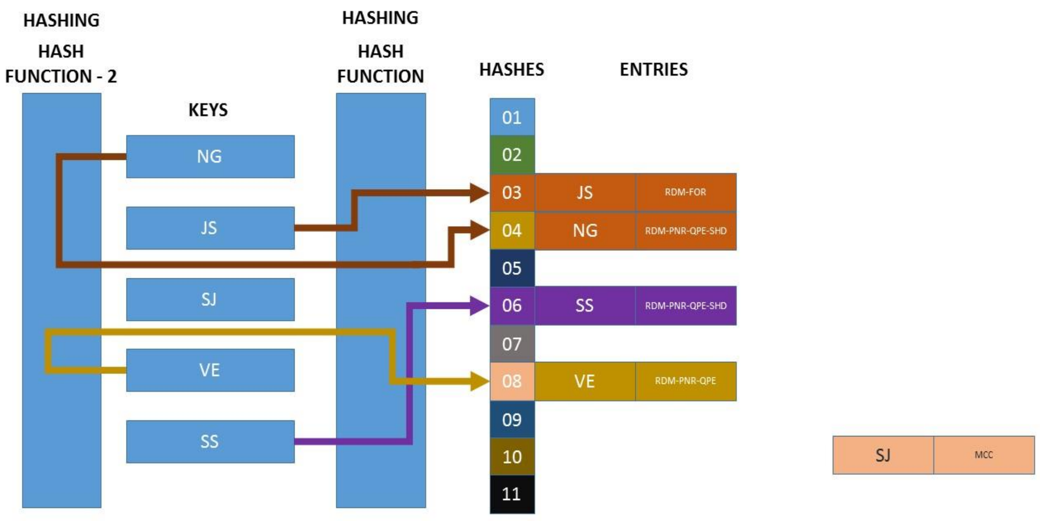

Fig. 10

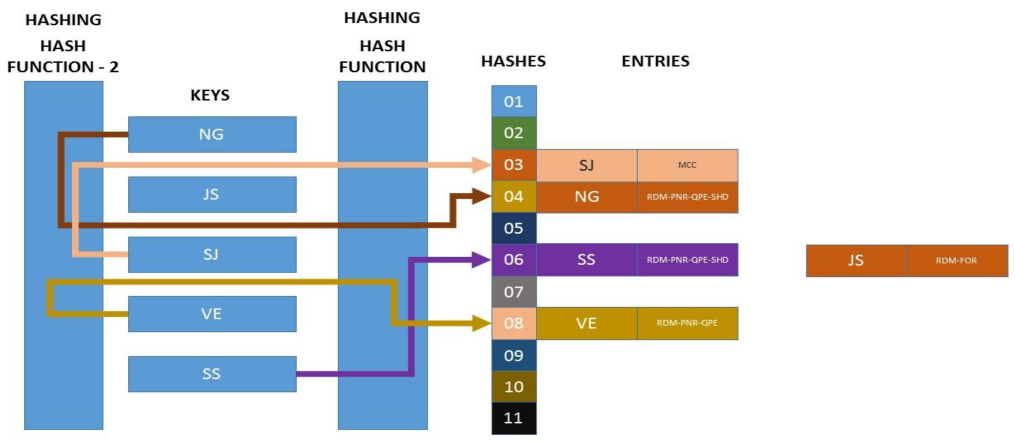

Fig. 11

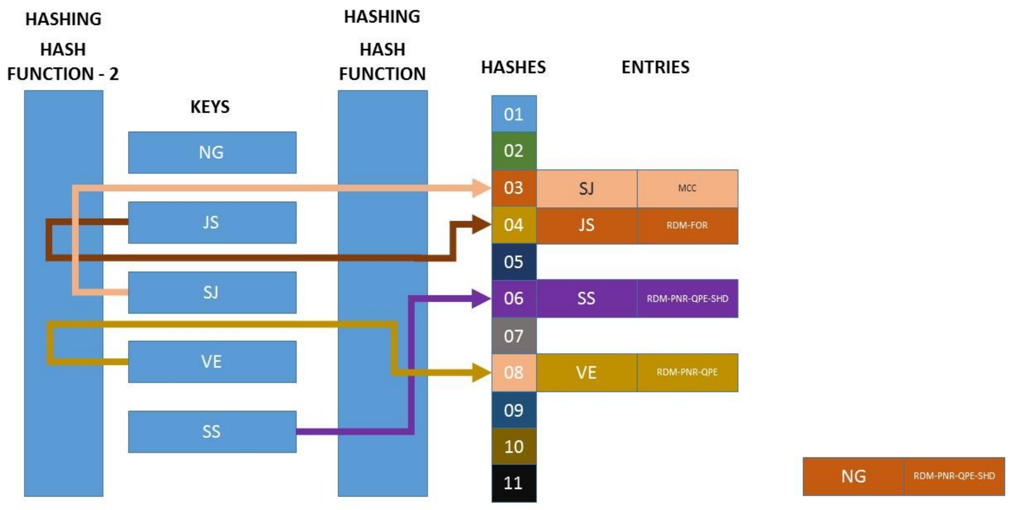

Fig. 12

VI. CONCLUSION

After analysing all the possible scenarios we can conclude that Hashing is the best way to store and retrieve content, but it still need improvement. There are lot of algorithms that solve the problem of collision but every algorithm has its own drawback. Out of the algorithms discussed above we can state that the best of all is Cuckoo Hashing. But it can go into an infinite loop hence cannot be trusted with less memory and big data coming into the memory. However if somehow we can manage to improve this algorithm then it will be the best algorithm among all to solve the problem of collision.

REFERENCES

[1] Pagh, Rasmus and Rodler, Flemming Friche (2001). "Cuckoo Hashing". Algorithms — ESA 2001. Lecture Notes in Computer Science 2161. pp. 121–133. doi:10.1007/3-540-44676-1_10. ISBN 978-3-540-42493-2.

[2] Knuth, Donald (1998). 'The Art of Computer Programming'. 3: Sorting and Searching (2nd ed.). Addison-Wesley. pp. 513–558.

[3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009). Introduction to Algorithms (3rd ed.). Massachusetts Institute of Technology. pp. 253–280. ISBN 978-0-262-03384-8.

[4] Cuckoo Hashing, Theory and Practice (Part 1, Part 2 and Part 3), Michael Mitzenmacher, 2007.

[5] Algorithmic Improvements for Fast Concurrent Cuckoo Hashing, X. Li, D. Andersen, M. Kaminsky, M. Freedman. EuroSys 2014.