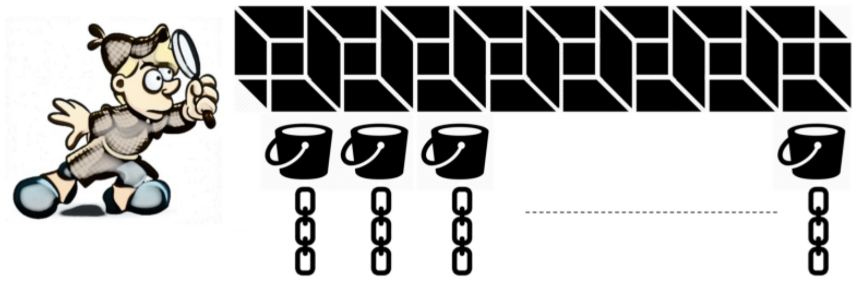
Curious case of ConcurrentHashMap

Romil Jain Follow
Apr 9 · 7 min read



In most of the interviews, Interviewer asks questions on Data Structure. If you are appearing for any product based company then these are must ask topics. No Interview can go without touching any thing from java.util.* package and Maps.

Recently It happened with me also while giving an interview in Silicon Valley. At this point, I must admit Silicon Valley loves Data Structure :). Discussion started with util package and ended at ConcurrentHashMap. Though it didn't go well as It seems Interviewer was having the vast knowledge on Map especially ConcurrentHashMap :) .

So I thought to dig more in to it and write about what so curious about ConcurrentHashMap. What is the need of it when there are Hashtable and HashMap already? Does it really efficient in terms of performance and thread safety? How does it handle rehashing ?

Today, I will deep dive in to ConcurrentHashMap and try to answer all these boiling questions. Lets start with first,

Need of ConcurrentHashMap when there are Hashtable and HashMap already.

I am assuming, you must be knowing Hashtable and HashMap before reading this article.

***So Lets consider a scenario where there is a need of frequent Read and less Writes and at the same time without compromising on Thread Safety*.**

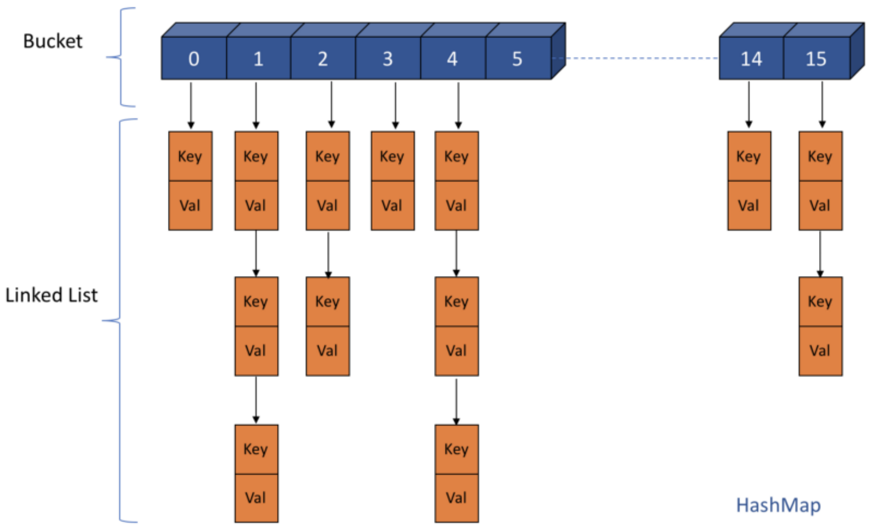
Is Hashtable a viable solution ?—So if one knows how Hashtable works, will immediately say NO. Though Hashtable is thread safe but give poor performance in multi-threading scenario because all methods including get() method of Hashtable is synchronized and due to which invocation to any method has to wait until any other thread working on hashtable complete its operation(get, put etc).

Is HashMap a viable solution ?—Hashmap can solve performance issue by giving parallel access to multiple threads reading hashmap simultaneously.
But Hashmap is not thread safe, so what will happen if one thread tries to put data and requires Rehashing and at same time other thread tries to read data from Hashmap, It will go in infinite loop.

I think because of above reasons Java introduced ConcurrentHashMap in version 5.0. As it takes the good of both and serves the purpose.

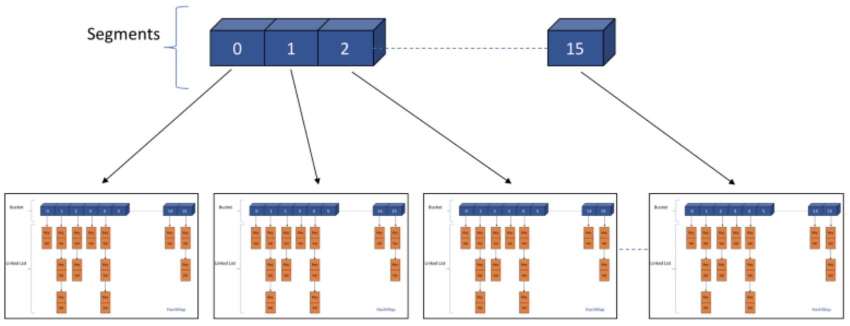
. . .

Now lets dive in to understanding the underlying data-structure used by all 3. Hashtable and HashMap both uses array and linkedlist as the data structure to store the data.



HashMap/Hashtable

ConcurrentHashMap creates an array on the top of it and each index of this array represents a HashMap. (In Java 8, It is a tree structure instead of linked-list to further enhace the performance)



ConcurrentHashMap

Read, Write and Delete Operation in ConcurrentHashMap

a) If one observes the above diagram, It is clear that weather it is Insertion or Read operation, one has to first identify the index of the segment where Insert/Read operation suppose to happen.

b) Once that is identified then one has to identify the internal bucket/array of the hashmap to find the exact position for insertion/read.

c) After identifying the bucket, iterate over the linked-list to check the key value pair.

> *i) In case of insertion if key matches replace the value with the new one otherwise insert the key with value at the end of the linked-list.*

> *ii) In case of read wherever key matches retrieve the value and return that value. if no match then return null.*

> *iii) In case of delete if the key matches delete the link corresponding to that key.*

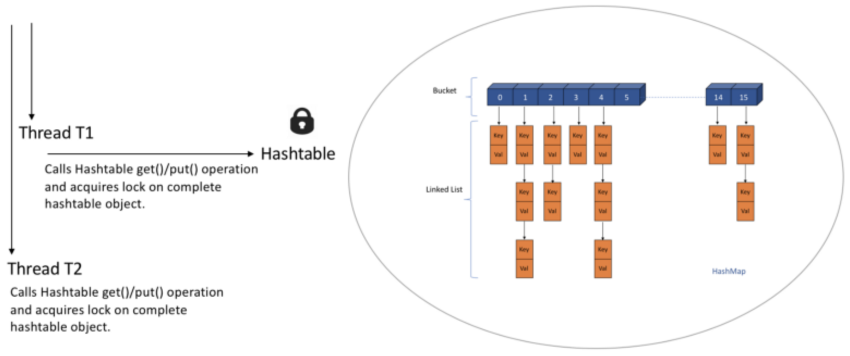
Performance and Thread Safety in ConcurrentHashMap over HashMap/Hashtable

HashMap is not synchronized and so doesn't provide any thread safety also. In contrast Hashtable is synchronized and provides thread safety but on the stake of performance. Hastable write operation uses map wide lock which means it locks the complete map object.

So if 2 Threads tries to do get() or put() operation on Hashtable,

Thread T1 calls get()/put() operation on Hashtable and acquires the lock on complete hashtable object.

Now if Thread T2 calls get()/put() operation, It has to wait till T1 finishes get()/put() operation and releases the lock on the object as depicted below.


Thread T1
Hashtable
Calls Hashtable get()/put() operation and acquires lock on complete hashtable object.
Thread T2
Calls Hashtable get()/put() operation and acquires lock on complete hashtable object.
Bucket
Linked List
HashMap

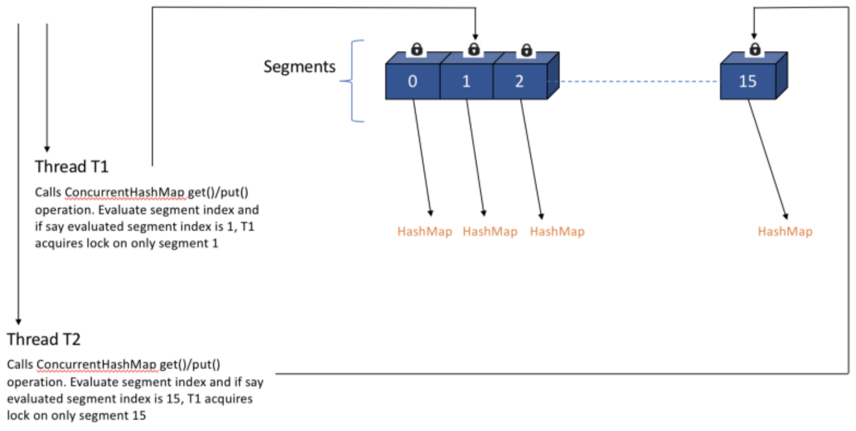
Threads acquiring lock on Hashtable

This makes Hashtable inefficient and that is why Java came up with ConcurrentHashMap.

ConcurrentHashMap works bit different as it acquires lock per Segment which means instead of single map wide lock, it has multiple Segment level locks.

So 2 Threads operating on different segments can acquire lock on those segments without interfering each other and can proceed simultaneously as they are working on separate Segment locks.

Thread T1 calls concurrentHashMap.put(key, value), It acquires lock on say Segment 1and invokes put method.
Thread T2 calls concurrentHashMap.put(key, value), It acquires lock on say Segment 15 and invokes put method as depicted below.


Segments
0
1
2
15
Thread T1
Calls ConcurrentHashMap get()/put() operation. Evaluate segment index and if say evaluated segment index is 1, T1 acquires lock on only segment 1
HashMap
HashMap
HashMap
HashMap
Thread T2
Calls ConcurrentHashMap get()/put() operation. Evaluate segment index and if say evaluated segment index is 15, T1 acquires lock on only segment 15

Threads acquiring lock on ConcurrentHashMap

This is how ConcurrentHashMap improves Performance and provides Thread safety as well.

. . .

Simultaneous Read and Write operations by Multiple Threads on same or different segments of ConcurrentHashMap

Read/Get Operation :- Two Threads T1 and T2 **can read data from same or different segment** of ConcurrentHashMap at the same time without blocking each other.

Write/Put Operation :- Two Threads T1 and T2 **can write data on different segment** at the same time without blocking the other.

But Two threads **can't write data on same segments** at the same time. One has to wait for other to complete the operation.

Read-Write Operation :- Two threads **can read and write data** on **different segments** at the same time without blocking each other. In general, Retrieval operations do not block, so may overlap with write (put/remove) operations. Latest updated value will be returned by get operation which is most recently updated value by write operation (including put/remove).

. . .

Size of Segments and concurrency level of ConcurrentHashMap

By default ConcurrentHashMap has segment array size as 16 so simultaneously 16 Threads can put data in map considering each thread is working on separate Segment array index.

ConcurrentHashMap has 3 arguments constructor which helps in tuning segment array size by defining concurrencyLevel.

```
ConcurrentHashMap map = new ConcurrentHashMap(int
initialCapacity, float loadFactor, int concurrencyLevel)
```

For Example :-

```
ConcurrentHashMap map = new ConcurrentHashMap(100, 0.75f,
10)
```

Initial capacity of map is 100 which means ConcurrentHashMap will make sure it has space for adding 100 key-value pairs after creation.

LoadFactor is 0.75f which means when average number of elements per map exceeds 75 (initial capacity * load factor = 100 * 0.75 = 75) at that time map size will be increased and existing items in map are rehashed to put in new larger size map.

Concurrency level is 10, it means at any given point of time Segment array size will be 10 or greater than 10, so that 10 threads can able to write to a map in parallel.

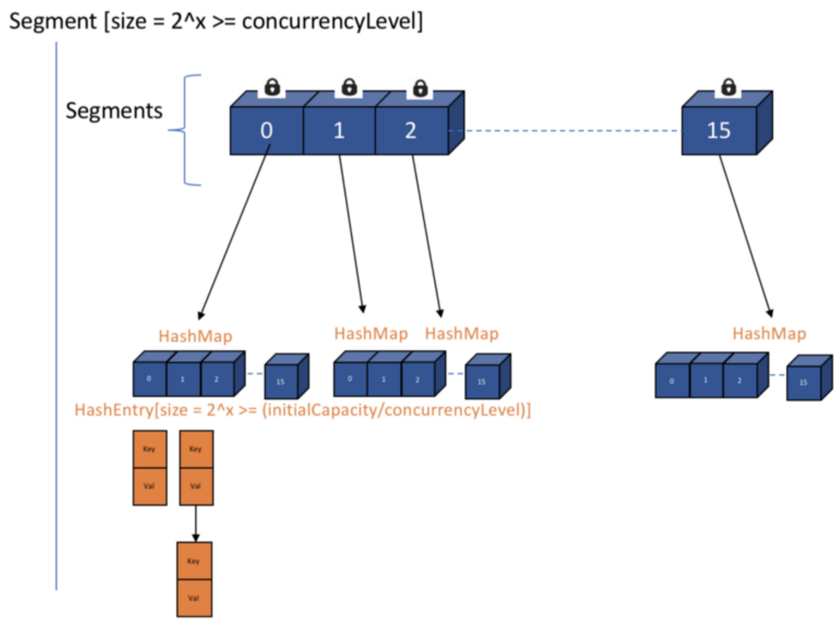
But how to calculate segment array size —

Segment array size = 2 to the power x, where result should be ≥ concurrencyLevel (in our case it is 10)

Segment array size = 2 ^ 4 = 16 ≥ 10 which means Segment array size should be 16.

Another Example:-
concurrenyLevel = 8 then Segment array size = ?
Find 2 ^ x ≥ 8
2 ^ 3 ≥ 8
Segment array size will be 8.

Now when one knows how to calculate segment array size the immediate question can come what would be the size of internal hashmap in a segment.


Segment [size = 2^x >= concurrencyLevel]
Segments
0
1
2
15
HashMap
HashMap
HashMap
HashMap
HashEntry[size = 2^x >= (initialCapacity/concurrencyLevel)]
Key
Val

Segments and HashEntry[]

As shown in diagram above each index of Segment array is a HashMap and In a HashMap the bucket/array is of class Entry[] and in ConcurrentHashMap the array is of class HashEntry[] (In Java 8, It is a tree structure instead of linked-list). So the size of HashEntry[] is

HashEntry[] array size = 2 ^ x ≥ (initialCapacity / concurrencyLevel)

For Example —

```
ConcurrentHashMap map = new ConcurrentHashMap(64, 0.75f, 8)
```

HashEntry[] array size = 2 ^ x ≥ 8 (64/8)

Find 2 ^ x ≥ 8

2 ^ 3 ≥ 8 ≥ 8
HashEntry[] array size will be 8.

It means there will always be capacity of 8 key-value pairs each segment will have in ConcurrentHashMap after its creation.

LoadFactor and Rehashing

ConcurrentHashMap has loadFactor which decides when exactly to increase the capacity of ConcurrentHashMap by calculating threshold (initialCapacity*loadFactor) and accordingly rehashing the map.

Basically, *Rehashing* is the process of re-calculating the hashcode of already stored entries (Key-Value pairs), to move them to another bigger size map when Load factor threshold is reached. Also It is not only done to distribute items across the new length map, but also when there are too many key collisions which increases entries in one bucket so that get and put operation time complexity remains O(1).

In ConcurrentHashMap, Every segment is separately rehashed so there is no collision between Thread 1 writing to Segment index 1 and Thread 2 writing to Segment index 4.

For Example:- If say Thread 1 is putting data in Segment[] array index 3 and finds that HashEntry[] array needs to be rehashed due to exceed Load factor capacity then it will rehash HashEntry[] array present at Segment[] array index 3 only. HashEntry[] array at other Segment indexes will still be intact, unaffected and continue to serve put and get request in parallel.

I believe this should help in understanding the implementation and internal working of ConcurrentHashMap.

. . .

Please share your thoughts, feedback, comments and clarifications. It will help me to improvise.

If you liked this post, you can help me share it by recommending it below ❤

Follow me on Linkedin and Twitter