Performance Improvement for HashMap in Java 8

SEE MORE

JS Jayant Sahu (https://www.linkedin.com/in/jayant-sahu-73626116)

16 DEC 2014

Java Modernization (https://www.nagarro.com/en/blog/tag/java-modernization)

Energy & Utilities (https://www.nagarro.com/en/blog/tag/energy-utilities)

4 min read

In our last post, we discussed the top five momentum builders for adoption of Java 8 in enterprises (/us/en/perspectives/post/20/Top-5-items-that-make-adoption-of-Java-8-exciting-for-Enterprises). In this post, we will dive deeper into JDK 8's new strategy for dealing with HashMap collisions. Hash collision degrades the performance of HashMap significantly. With this new approach, existing applications can expect performance improvements in case they are using HashMaps having large number of elements by simply upgrading to Java 8.

Hash collisions have negative impact on the lookup time of HashMap. When multiple keys end up in the same bucket, then values along with their keys are placed in a linked list. In case of retrieval, linked list has to be traversed to get the entry. In worst case scenario, when all keys are mapped to the same bucket, the lookup time

of HashMap increases from O(1) to O(n).

Java 8 has come with the following **improvements/changes** of HashMap objects in case of high collisions.

- The alternative String hash function added in Java 7 has been removed.
- Buckets containing a large number of colliding keys will store their entries in a balanced tree instead of a linked list after certain threshold is reached.

Above changes ensure performance of O(log(n)) in worst case scenarios (hash function is not distributing keys properly) and O(1) with proper **hashCode()**.

How linked list is replaced with binary tree?

In Java 8, HashMap replaces linked list with a binary tree when the number of elements in a bucket reaches certain threshold. While converting the list to binary tree, hashcode is used as a branching variable. If there are two different hashcodes in the same bucket, one is considered bigger and goes to the right of the tree and other one to the left. But when both the hashcodes are equal, HashMap assumes that the keys are comparable, and compares the key to determine the direction so that some order can be maintained. It is a good practice to make the keys of HashMap comparable.

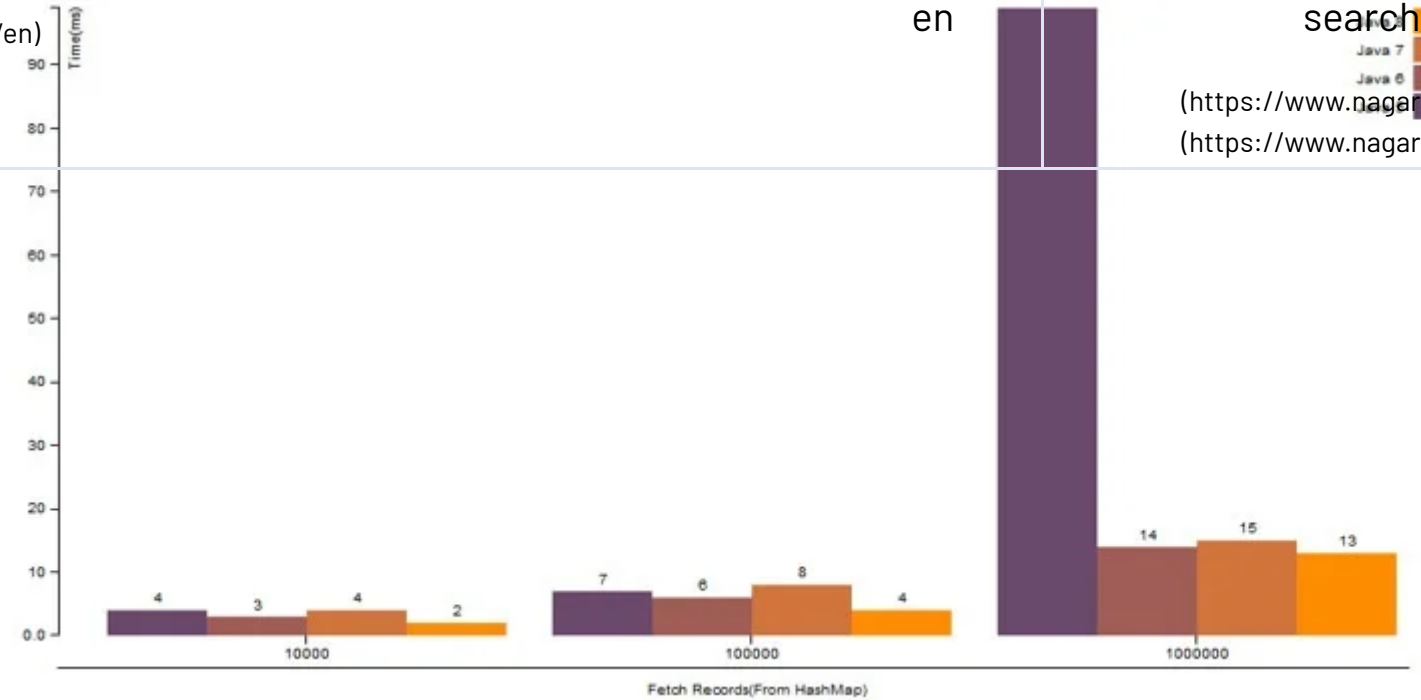
This JDK 8 change applies only to **HashMap, LinkedHashMap** and **ConcurrentHashMap**.

Based on a simple experiment of creating HashMaps of different sizes and performing put and get operations by key, the following results have been recorded.

1. HashMap.get() operation with proper hashCode() logic

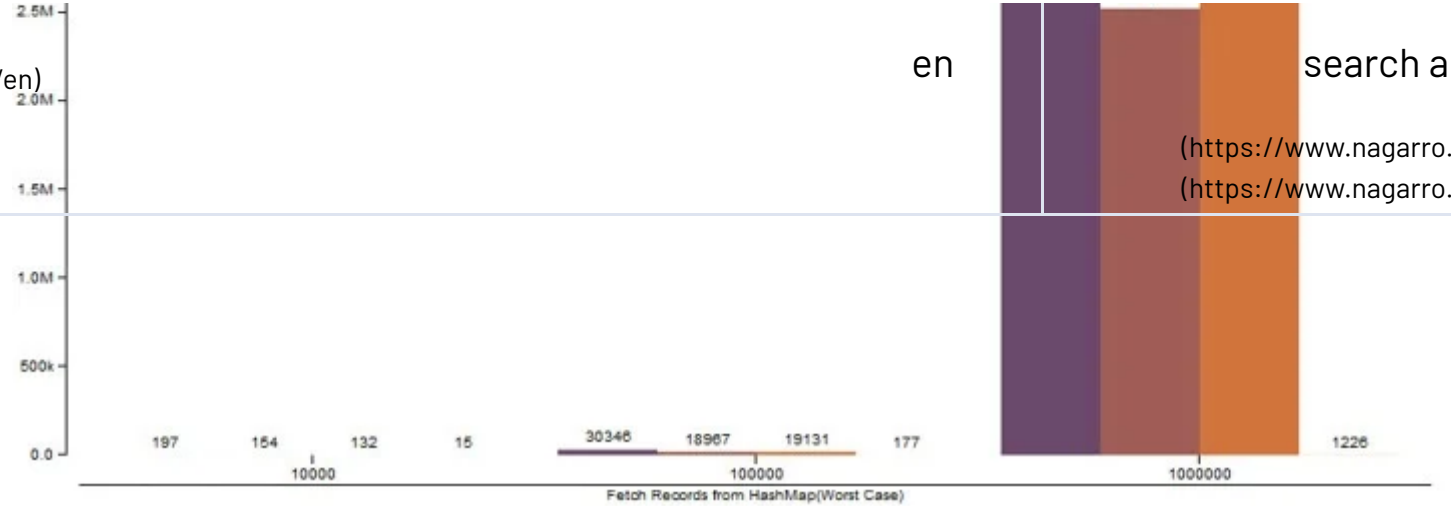
Number Of Records	Java 5	Java 6	Java 7	Java 8
10,000	4 ms	3 ms	4 ms	2 ms
100,000	7 ms	6 ms	8 ms	4 ms
1,000,000	99 ms	15 ms	14 ms	13 ms

(https://www.nagarro.com/en/search-menu)



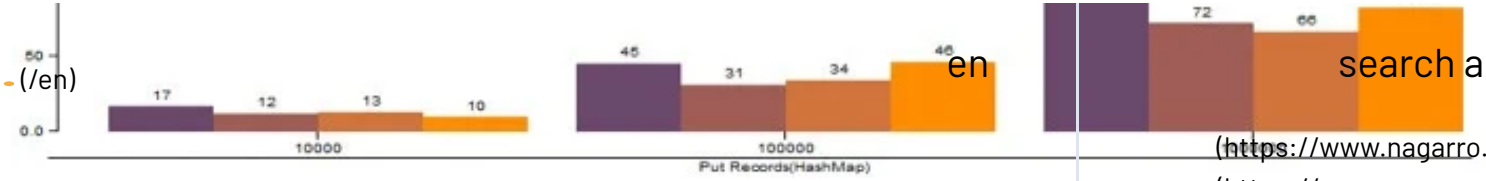
2. HashMap.get() operation with broken (hashCode is same for all Keys) hashCode() logic

Number Of Records	Java 5	Java 6	Java 7	Java 8
10,000	197 ms	154 ms	132 ms	15 ms
100,000	30346 ms	18967 ms	19131 ms	177 ms
1,000,000	3716886 ms	2518356 ms	2902987 ms	1226 ms
10,000,000	OOM	OOM	OOM	5775 ms



3. HashMap.put() operation with proper hashCode() logic

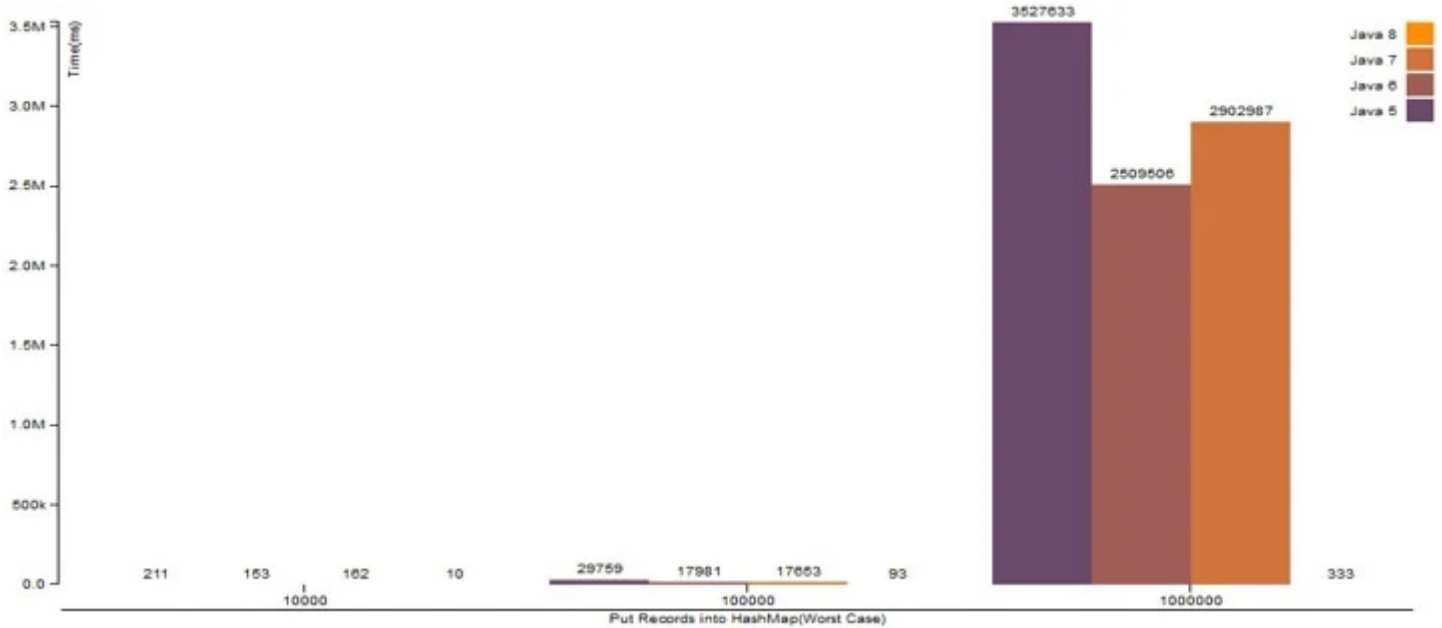
Number Of Records	Java 5	Java 6	Java 7	Java 8
10,000	17 ms	12 ms	13 ms	10 ms
100,000	45 ms	31 ms	34 ms	46 ms
1,000,000	384 ms	72 ms	66 ms	82 ms
10,000,000	4731 ms	944 ms	1024 ms	99 ms



(https://www.nagarro.com/en/search-menu)

4. HashMap.put() operation with broken (hashCode is same for all Keys) hashCode() logic

Number Of Records	Java 5	Java 6	Java 7	Java 8
10,000	211 ms	153 ms	162 ms	10 ms
100,000	29759 ms	17981 ms	17653 ms	93 ms
1,000,000	3527633 ms	2509506 ms	2902987 ms	333 ms
10,000,000	OOM	OOM	OOM	3970 ms



The above experiment demonstrates the power of this changed approach in improving the performance of existing Java applications. While this is an internal detail, simply upgrading to JDK 8 from older JDK versions will allow certain applications to see performance improvements if they use HashMaps heavily and have a large

amount of entries in the HashMaps.

In our next post, we will discuss the use of lambda expressions in Java 8 for applying functional constructs.

(https://www.nagarro.com/en/search-menu)

D177-4365-B378-BB62141EA769&PLACEMENT_GUID=29825AB4-B761-4A02-8DE7-G%2FPOST%2F24%2FPERFORMANCE-IMPROVEMENT-FOR-HASHMAP-IN-JAVA-8&REDIRECT_URL=APEFJPGP0XQ6TFJ8CC-ZIZLTRWLQAHJNAT9EKNHVCPXSLLGQYBAL-RY3XRPADI_5DKITCVJDRX4_JUZIBE_W5I54JX_5ANTKRSNL2FLSBTQQU0N9WXX-XCANGGASBWDMFZ5DHO5Y6VPMKH5LCFJD4SC7IDYA&CLICK=F0CCA9F8-AE49-4BA6-8E2B-RBKCHSXFHJEMG&UTM_REFERRER=HTTPS%3A%2F%2FWWW.GOOGLE.COM%2F&PAGEID=6034608096&CONTENTTYPE=BLOG-2159.1588413692159.1588413692159.1&__HSSC=251543577.1.1588413692159&__HSFP=3299747788)