CPSC 438 - Big Data Systems
Project 1

Amirmohammad Nazari

Memory and locking structure

The key-value store in this implementation consists of N (NUM_BINS) buckets, each holding key-value pairs that share a common hash value modulo N. When a user makes a query, the key is hashed using the default hasher provided by Rust, and based on its hash value, it is placed into or retrieved from the corresponding bucket. Moreover, each bucket keeps the pairs sorted by their keys, to find them later faster using binary search.

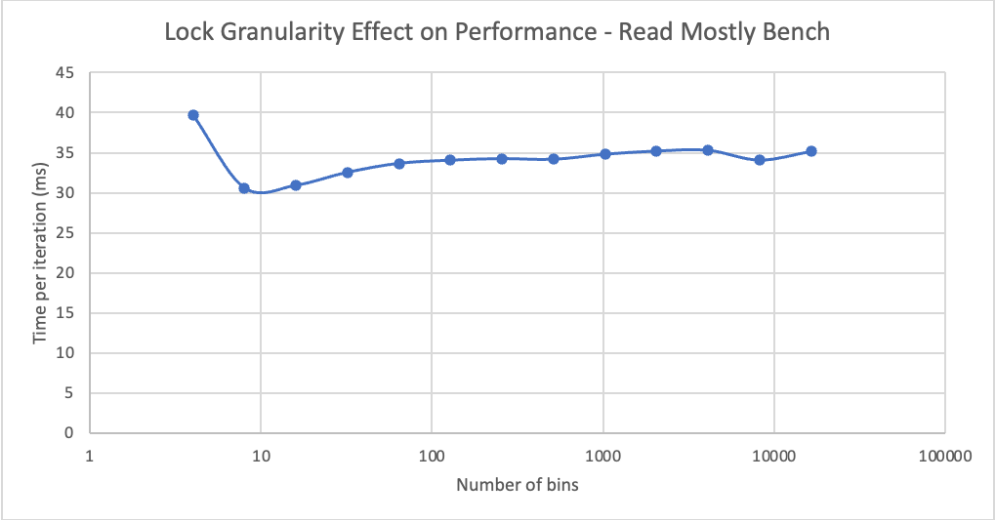
Each key-value pair is stored within a Pair struct, which offers functions for retrieving the key (get_key()), getting the value (get_value()), and updating the value (update_value()). These pairs are organized within synchronized vectors of type Pair (RwLock<Vec<Pair>>) for each bucket. These vectors can accommodate any number of key-value pairs. To specifically answer the second question, each lock protects all key-value pairs that share a common key hash value modulo N. In other words, changing the number of buckets will impact the portion of key-value pairs that are protected by one key and that is how we can modify the locking granularity. The more buckets we have, the finer granularity of locks we can achieve.

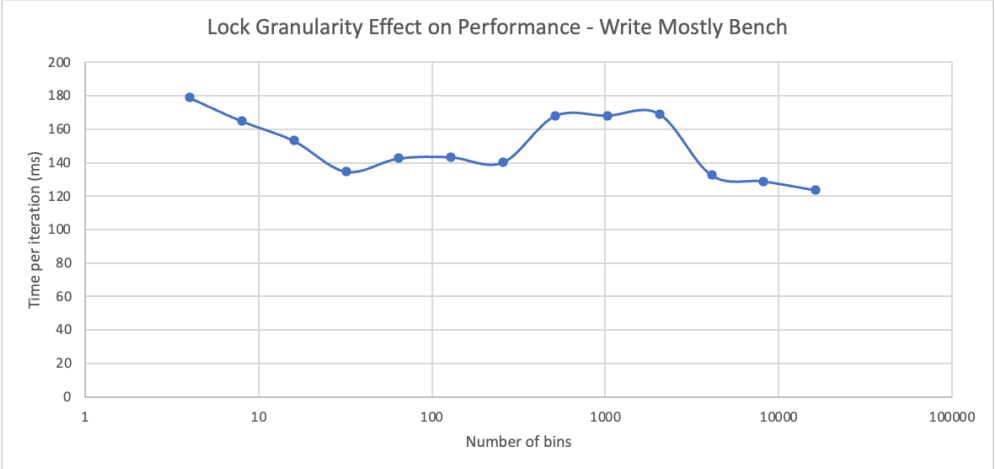
A single global lock is employed to safeguard all the buckets. Although it could potentially be used to increase the number of buckets, this particular feature has not yet been implemented. All functions mentioned in the instructions need to acquire this lock through a read() operation before proceeding with their tasks. Importantly, this locking mechanism does not adversely affect their performance unless a resizing operation is initiated.

Evaluation

Lock Granularity

As mentioned in the previous section, the locking granularity changes as the number of bins changes. Therefore, we can increase the granularity of our locks by increasing the number of buckets. The impact of varying lock granularity on the iteration times can be seen in the following figures:





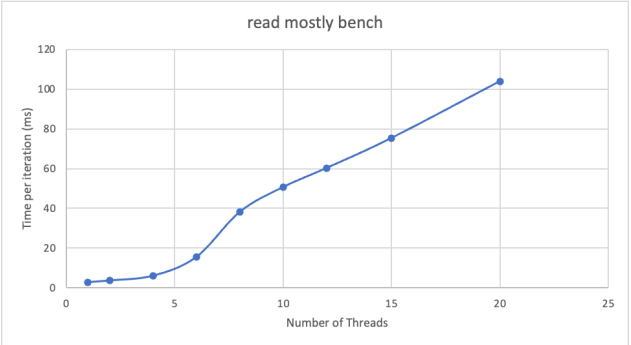
We can see that as the granularity of the locking system increases, the performance of the write-mostly benchmark improves. This is because the number of keys protected by a single lock reduces and more “put()” functions can work in parallel. However, for the read-most benchmark,

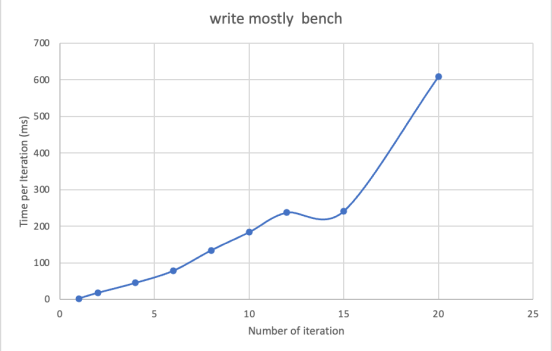
the “read()” functions can already work in parallel in the situation where only one lock protects all the keys. Therefore, increasing the lock granularity won’t improve their performance.

Another thing that affects the performance of benchmarks is the number of pairs that is stored in the key-value store. In the read-mostly benchmark around 5% percent of the time we add keys, therefore, the key-value store is less crowded and performs faster compared to the other benchmark.

Number of Thread

For these tests we choose the number of buckets to be 1000. The results are shown as below:





The time it takes for a single iteration to finish increases almost linearly as for both benchmarks. The reason is that when the number of threads rises, more lock contentions will occur, leading to more conflicting put requests on different threads. This means more read queries have to wait until another thread is done with a put query.

Read Ratio

We make this test using 10000 keys for each iteration and 8 threads. For the global lock 1 hash bucket and for finer-grained locking 4 hash buckets have been used. Here we only show the benchmark performance for the read-most benchmark. The same effect is also true for the write-most benchmark.

We can see that until around 0.7 read ratio the time taken per iteration decreases linearly. However, after that point, the rate at which each iteration time decreases is a lot more. This is because read requests can perform simultaneously but when one write operation occurs, all

proceeding read requests would have to wait for it to finish. Therefore the lock contention increases, leading to higher thread idleness and therefore higher iteration times.

One other thing to note is that this effect becomes less visible when the key-value store uses fine-grained locks rather than a global one. The reason for this is again less occurrence of lock conflict over a pair. However, they all converge to the same point where the read ratio is one, because no lock conflict happens in that case.