4. DELETION HELP HEURISTICS.

In the rotating skip list implementation we first mark a node as logically deleted (to minimize contention) before physically remove it from the list. The background thread is the main thread that is responsible for physically remove nodes but other threads can also help him remove nodes.

4.1 Thread-Num-Size Rate

In the original implementation, there is a heuristic that helps deciding whether the delete should help removing. This happens only when the ratio of logically deleted nodes over non-logically deleted nodes (as communicated by the background thread) reaches 3 after what it pays off. We thought that the decision to whether help remove or not should also be taken according to the number of threads. As the number of the threads is bigger the background has much more work to do in order to balance the skip-list structure and maintain the index index levels so help-remove sounds like a good idea. On the other hand, if the number of threads is small, we would rather that there will be more operations happening (add, remove, contains) and the background thread will handle the deletions by itself.

We benchmarked the original heuristic agains ours, (where we take the log of number of threads in consideration when deciding whether the delete should help remove), with initial size of 256 (and 1024), key range of 512 (and 2048) and update operations rate (delete / insert) of 70 (and 90) percent.

4.1.1 Evaluation As We can be seen on figure 4, there are really minor changes between the two heuristics, it seems that as long as the update rate gets higher the performance of our new heuristic that takes in count the number of threads is a bit better than the original, probably because there are more add/delete operations so in case of many threads that are doing many changing list operations (add/remove) it's more critical to enable the help-removing, but anyway, it's not look like something that have significant improvement.

5. BACKGROUND THREAD SLEEP HEURISTICS

The background thread executes a continous loop (until bg\_stop() is called) to maintain index levels.

In each iteration the background thread:

- traverses the nodes/indexes and for each level from bottom to top it raises each node in the middle of three consecutive non-deleted nodes of the same height.

- physically delete logically deleted nodes

- remove an entire bottom level of the skip list according to some heuristic.

The background thread also sleeps for 50 microseconds (a heuristic that is used in the original implementation).

We thought it could be interesting to test different heuristics related to sleeping time of the background thread.

5.1 Thread-Num Rate

Instead of sleep for a constant time (50 microseconds) we thought that the time to sleep should be set according to the number of threads. The reason behind is that as long as we have more threads, there are more operation that is happening, changing the skip list structure, which means, that in order to get good performance we must rebalance the skip-list structure as soon as possible, that's why we decided to decrease the time the background thread sleep (in proportion to the original sleeping time) when the number of thread increases. So when we have a small number of thread, we have less work so the background thread can sleep some more.

5.2 Sleep double time, Don't sleep at all

A different heuristic we thought is to make the background thread sleep for double time - 100 ms instead of 50 ms and then canceling the sleep time of the background thread so it won't sleep at all in every second iteration.

We wanted to test if the fact that sometimes the background thread doesn't sleep at all make it better because the skip-list structure should be more balanced.

We benchmarked the original sleep time against the heuristics described in sections 5.1 and 5.2, with initial size of 256 (and 1024), key range of 512 (and 2048) and update operations rate (delete / insert) of 20 (and 50) percent.

5.3 Evaluation As we can be seen on figure 5, the performance of both of our new heuristics was similar to the one of the original heuristic. Our new heuristics always performed better but it seems to be minor changes. We tested it with different parameters and we got to the conclusion that the sleeping time of the background thread as long as it's not too extreme doesn't affect the performance significantly.

6. Multiple Skip Lists

The original implementation of the rotating skip list uses one background thread and some worker threads that execute operations like add, remove and contains.

The implementation use only **one** skiplist that's everybody is working on, along the years the researchers of the skip-list performance tried to diminish contention to get better results.

We thought to go with the same direction, diminish contention, so we decided to use more than one skiplists. We know that as long as there are many threads, there is still contention, many threads interfering on the same shared data.

6.1 Dynamic Number of Skip List

We decided that the number of skip lists will be depend on the number of threads because more threads means more contention so our heuristic is to take the number of the skip lists to be half of the number of threads.

We benchmarked the original sleep time against the the multiple skip lists heuristic, with initial size of 128, key range of 256 and update operations rate (delete / insert) of 30 (and 60) percent.

6.1.1 Evaluation - As can be seen on figure 6, the performance of the multiple skip lists was very similar to the original implementation as long as the number of threads is smaller than 16, what matches our thoughts that our heuristic is good for high contention, many threads interfering to each other. Indeed, when the number of threads increases to 32 and especially 64, we can see that our implementation almost twice as good as the original implementation, which is a significant improvement.