The Lock-free k-LSM Relaxed Priority Queue (Re-implementation)

Michael Jimenez, Chintip Winn

April 19th 2019

Abstract

A priority queue is a data structure that stores elements that are ordered by priority and are implemented in many modern data structures, due to their efficient access to minimal (and maximal) keys. Here we present a re-implementation of a data structure that has the efficient lookup of minimal elements, while being able to scale into large read and write applications. This is a re-implementation of the k-LSM Relaxed Priority Queue, based on the 2013 paper by Martin Wimmer, et al.[Mar15] The LSM tree, which stands for Log-Structured Merge Tree, is commonly written using the B-Tree or 2-4 Tree and the Bloom Filter data structures which allow for logarithmic look up times as well as fast data storage. Our implementation of the k-LSM Relaxed Priority Queue is written in C++ and is a combination of a Skiplist (N.Shavit,I.Lotan et al.)[Nir00] as our Priority Queue and a B-Tree which stores data on the disk. We later add concurrency to our lock-free LSM (P.O'Neil,E.Cheng)[E C00] by allowing it to support k threads, each with their own respective skiplist, which are allowed to remove any of the $p + 1$ smallest keys, where p is a configurable parameter. We manage the synchronization of threads by adding a thread ID field to each node to determine which thread has modified a particular block. Finally, we test our implementation of the k-LSM relaxed priority queue by showing our implementations scaling and performance using multiple threads.

1 Introduction

The use of a concurrent skiplist based queue, or skipqueue, has many advantages over other notable priority queue implementations like heaps and trees. Some of the advantages are that: all locking is distributed and that there is no locking of a root or counter like heaps or B-Trees which greatly reduces contention and in addition, there is no need to halt operation to re-size like trees and heap data structures.

Unlike heaps, the delete min operation is evenly distributed throughout the list, and skipqueues allocate memory as the size of the structure grows unlike tree and heap data structures that require a static size before runtime. Compared to other data structures such as state of the art concurrent min/max heaps and B-Trees, skipqueues scale better with a growing number of threads due to less contention, and they also do not have to halt operations to resize. As with any data structure, there are disadvantages of the skiplist based queue in that developing a skipqueue is quite complicated and adjusting it to work concurrently will take a lot of additional effort and troubleshooting.

Initially, the paper mentioned above, was unclear in that the diagrams were relatively unhelpful and that many of the functions required to implement the project were not provided in the pseudocode. Figuring out how the k-LSM works took an extensive amount of time and was difficult to comprehend as a result of the lack of detail. In addition, our implementation is vastly different from the one presented in the paper. Furthermore, since this is our first experience dealing with research-level papers, a complex data structure and concurrency of this scale, we have a lot of hurdles to overcome.

2 Related Works

Throughout the process of re-implementing the Lock-free k-LSM relaxed priority queue, we have found a few papers and articles useful in our approach. The first scholarly paper that helped us get started with our implementation was (M.Wimmer,P.Tsigas,J.Larsson) [Mar15] which was the extended version of the paper that was initially given to us for this project. This paper was useful in that it provided some examples and pseudocode which were essential for understanding the basis for this assignment. Unfortunately, the examples and pseudocode provided are very basic and do not fully explain how most of the data structure functions. Thankfully, we found other high level research papers like [Nir00] that explain large portions of the LSM that others have missed and (M.Herlihy,N.Shavit et al.)[Mau] which clearly explains how to construct a lock free skiplist.

Nir Shavit, author of the art of multiprocessor programming and concurrency pioneer, helped us understand the idea of using skiplist based priority queues for the implementation of the k-LSM through his works. Finally, we also found (P.O'Neil,E.Cheng et al.)[E C00], which is based on the LSM, we found useful in that it helped us better understand how log structured merge trees work. The diagrams in this paper were very poorly drawn and do not give the reader a very good grasp on how merging to the disk works. We are still currently researching on how many parts of our data structure work and on how to implement it.

3 Approach Overview

Since the name of our concurrent data structure is "The Lock-free k-LSM Relaxed Priority Queue" we can surmise that our data structure provides a lock free progress guarantee in that some thread calling a method eventually will return. This is an important property to have for scalability when using large amounts of threads. Our data structure is also deadlock free.

Since we use a lock free skiplist that was inspired from (N.Shavit et al.)[Mau] the same progress guarantee holds. Our k-LSM consists of a distributed LSM and a shared LSM where each respective container functions as a stand alone priority queue. The find min operations of both of these structures are wait free since they function off of thread local states which do not interleave with other thread operations. Regardless of if the delete min operation takes more than one attempt in deleting an item, clean up from the find node function ensures that a node is eventually found with the exception of the skiplist being empty. This is a result of logically deleted nodes staying in the list until they are pruned out in the find node function. Other threads may increase the number of attempts at deleting a node by performing operations of their own, but this implies that some other thread is making progress and thus delete min is lock free. Our version of this data structure is deadlock free because a thread will always acquire the lock of a node with a greater key first. This can be seen in our remove and add functions where we acquire locks from the bottom level upwards. Our contains method is wait free in that it does not acquire any locks and traverses the list only

once. Each of the skiplists in the distributed LSM are essentially sequential making them wait free. We can extent this to our shared LSM concluding that our data structure is lock free.

It is important to make sure that when working with concurrent data structures that your operations are correct. This can be seen through linearization points or the instant where every operation takes effect throughout the moment of invocation to termination. The main functions that our lock-free skiplist support are add, remove and contains. In addition, our data structure supports the deletion of the p + 1 smallest keys as a result of p relaxation.

We will first discuss the linearization points for the primary functions of our data structure. In the add function, the linearization point is once the newly created node is linked into the skiplist and once its fully linked flag is set to true. This ensures that once a thread properly inserts a node into the shared LSM that it will be seen with respect to all other threads ensuring that ordering properties and logarithmic look up is maintained. In the remove function, the linearization point is once a nodes flag field has been set to true implying that it has been logically deleted. After this point, all other threads will know that the current node has been deleted but is still present in the skiplist. This ensues that other threads will not disrupt the logarithmic structure and ordering of the elements in our concurrent container. The contains functions linearization point is the moment that the find node function returns a Boolean value indicating that the current node is not logically deleted, fully linked and has the desired key. This ensures that our container is able to accurately determine if an element is present in our skiplist regardless of concurrent operations by multiple threads.

Our data structure supports the deletion of the p + 1 minimum keys. To support local ordering semantics, each thread deletes the minimum key that it inserted into the shared LSM or the minimum item if none of its nodes are still present. To accomplish this, each node is given a threadID field. Since this value is never modified, it does not effect correctness. We ensure that each node is not able to interfere with each other through the use of locks, which are present in each node. This is standard in concurrent skiplists and ensures our correctness conditions are upheld.

The paper given by (M.Wimmer,J.Larsson et al.)[Mar15] did not explicitly specify which data structures that they were using in their implementation, so it was up to us to determine which combination of data structures were the most appropriate. Our agreed upon data structure consists of three primary containers: a skip queue supporting global ordering semantics (shared LSM), an array of wait free skiplists that support local ordering semantics (distributed LSM) and a B-Tree in memory. We chose these containers because they uphold the logarithmic runtime requirement and the fact that serialization and deserialization of data to and from the disk is beyond the scope of this course. The authors of the paper use a linked list of block arrays where the log base 2 of the size of each block array symbolizes the height of each respective block. In our implementation we use a skiplist for reasons mentioned in the introduction. Our shared LSM was inspired from (N.Shavit et al.)[Mau] while the distributed LSM consists of an entirely separate skiplist container that we designed to support constant re-sizing to allow for logarithmic runtimes of smaller amounts of data. To avoid the use of a bloom filter when determining which thread inserted a node, we simply added a field called threadID in each block. This allows each thread to identify which node it inserted. This is important in that this is how we are able to determine which of the p + 1 nodes a thread can skip considering a thread cannot ignore a node that it inserted into the shared LSM.

The transactional data structure provides marginally better performance than the concurrent data structure does, in many cases. We suspect that this is due to the removal of the locks, which in many cases lead to better performance from less waiting.

As a note, the gnu-gcc transactional memory model does not provide the means to explicitly adjust transaction size. Here we wrap various parts of the code that we deem necessary with the transactional memory model as an atomic transaction.

The modified implementation should follow the same informal proof of correctness we discussed prior in the concurrent section, since there are no changes in the control of the logic used, rather, the changes come in the form of the removal of locks, and wrapping certain sections of the code in atomic transaction blocks.

References

[E C00] P. O'Neil E. Cheng D. Gawlick. *The Log-Structured Merge-Tree (LSM-Tree)*. 2000. URL: https://www.cs.umb.edu/~poneil/lsmtree.pdf.

[Nir00] Itay Lotan Nir Shavit. *Skiplist-Based Concurrent Priority Queues*. 2000. URL: http://people.csail.mit.edu/shanir/publications/Priority_Queues.pdf.

[Mar15] Jesper Larsson Traff Martin Wimmer Jakob Gruber. *The Lock-free k-LSM Relaxed Priority Queue*. 2015. URL: https://www.researchgate.net/publication/273788017_The_Lock-free_k-LSM_Relaxed_Priority_Queue.pdf.

[Mau] Yossi Lev Maurice Herlihy Nir Shavit. *A Provably Correct Scalable Concurrent Skip List*. URL: https://www.cs.umb.edu/~poneil/lsmtree.pdf.