The Lock-free k-LSM Relaxed Priority Queue

Michael Jimenez, Chintip Winn

March 3rd 2018

Abstract

Priority Queues are data structures that store elements that are ordered by priority and are implemented in many modern data structures, due to their efficient access to minimal (maximal) keys. Here we present a data structure that has the efficient look up of minimal elements, while being able to scale into large read and write applications. This is an implementation of the k-LSM Relaxed Priority Queue List, based on the 2013 paper[Mar15] by Martin Wimmer, Jakob Gruber, Jesper Larson Traff, and Philippas Tsigas. The LSM tree, which stands for Log-Structured Merge Tree, is commonly written using the B-Tree (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 List is written in C++ and is a combination of a Skiplist[Nir00] as our Priority Queue and a B-Tree which stores data on the disk. We later add concurrency to our lock-free LSM[E C00] by allowing it to support k threads 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 using a bloom filter 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

1. What are the advantages and disadvantages of this data structure compared to its alternatives?

   The use of a concurrent skiplist based queue, or SkipQueue, has many advantages over other notable priority queue implementations like heaps and trees in that:

   - all locking is distributed and there is no locking of a root or counter like heaps or B-Trees

   - there is no need to halt operation to re-size like trees and heaps data structures

   - unlike heaps, the delete min operation is evenly distributed throughout the list

   - SkipQueues allocate memory as the size of the structure grows unlike tree and heap data structures that require a static size before run time.

   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

2. Are there any specific use cases where this design would be more beneficial than the-state-of-the-art alternative container?

   The SkipQueue are more beneficial than the state of the art concurrent min/max heaps and B-Trees in that they:

-scale better with a growing number of threads due less lock contention

-not having to halt operations to re-size.

3. Can you think of ways to improve the design of the data structure? If so, please attempt them and compare your design and the original re-implementation of the algorithm.

   No, the Skiplist based queue is probabilistic in that it averages O(logn) runtime on its operations.

4. What are the biggest obstacles you encountered in your implementation?

   Initially the paper was relatively unclear, in that the diagrams were relatively unhelpful, and many of the functions required to implement the project were not provided in the pseudocode– so figuring that out took an extensive amount of time and was difficult. Furthermore, since this is our first experience dealing with research-level papers and a complex data structure and concurrency of this scale, we have a lot of hurdles to overcome.

2 Related Works

Throughout the process of 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 [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 a 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. Nir Shavit, author of the textbook used in class and concurrency pioneer, helped us understand the idea of using skiplist based priority queues for the implementation of the k-LSM. Finally, we also found [E C00], which is on the LSM, 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

1. What is the progress guarantee that your data structure provides?

   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 when using more threads.

2. Include an informal proof of why the data structure meets the specified progress guarantee.

   We will provide this informal proof specifying our data structures progress guarantee once we implement it concurrently.

3. What is the correctness condition that your data structure provides?

   We will provide the correctness condition specifying our data structure once we implement it concurrently.

4. Include an informal proof of why the data structure meets the specified correctness condition.

   We will provide this informal proof specifying our data structures correctness conditions once we implement it concurrently.

5. What are the key synchronization techniques that allow this design to meet the described correctness and progress guarantees?

   We will provide the key synchronization techniques specifying our data structures once we implement it concurrently.

6. What did you have to change from the design described in the research paper in your implementation of the algorithm?

   The paper given [Mar15] did not explicitly specify which data structures that they were using, so it was up to us to determine which combination of data structures were the most appropriate. Our agreed upon data structure consists of two primary containers, one entirely in memory and one entirely on the disk. The container in memory will be a SkipQueue while the container on disk will be a B-Tree. We chose these containers because they uphold the logarithmic runtime requirement.

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`.