The Effects of Rotating Skiplist on LevelDB

Jesse Galloway and Sam Paleologopoulos

Github Repository: https://github.com/jsngalloway/leveldb_rotating_skiplist

Topic

The aim of this project is focused around LevelDB (https://github.com/google/leveldb), an open-source, in-memory database designed and used by Google. Our project is an attempt to replace the underlying skiplist data structure it employs as an index with a rotating skiplist, as outlined in the paper "A Skip List For Multicore", by Dick, Fekete, and Gramoli.

Background

LevelDB is a fast key-value storage that provides an ordered mapping of string keys to string pairs. Basic operations include Put(key, value), Get(key), Delete(key).

Rotating Skiplist is a type of skiplist that claims to be the fastest concurrent skiplist to date. It aims to improve throughput performance by rotating the head of the skiplist for minimizing contention, using wheels instead of towers in the internal structure, and no locks. According to the paper, *A Skip List For Multicore*, the Rotating skiplist outperforms all other novel logarithmic skiplists/trees (Fraser's skiplist, Crain's no hotspot skiplist, Optimistic skiplist, Transaction-based skiplist, Speculation-Friendly BST, Citrus Tree, Non-blocking BST) across 0%, 10%, 30%, and 90% update transactions in terms of throughput.

Motivation

LevelDB is based on concepts created by Google for their Bigtable database system while remaining open source. The key-value lookup aims to be fast, and achieves this through data compression, write batching, and it's own memory allocator. LevelDB also aims to be simple: allowing forward and backward iterators, no indexes, and no queries. LevelDB itself can handle multiple threads operating on it, however the skiplist that stores the data internally cannot. If this

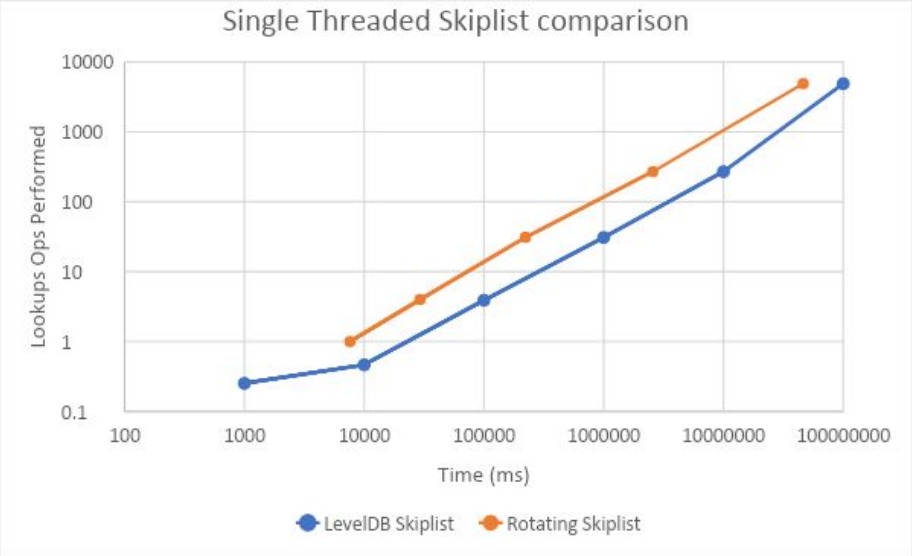
skiplist were replaced with one which could handle concurrent operations, such as the rotating skiplist, we predict to see an overall performance improvement of LevelDB.

Major Challenges

The first major challenge we encountered in this project was getting an instance of LevelDB to build. Initially, we worked with a Windows port of LevelDB with limited documentation. After little success with this, and acknowledging that a system built on Linux would have more replicability we switched to a Ubuntu Docker environment. Container specifics can be found in our Github repository. In this development container we were able to successfully built LevelDB and Synchrobench in the same environment. Once running on the same machine we were able to modify the source and compare the performance of the two codebases.

Our first test intended to measure the performance of LevelDB's existing skiplist against Synchrobench's implementation of the Rotating skiplist. This proved to be difficult, however, as Synchrobench's results were not originally comparable to our custom LevelDB tests. The test structure that Synchrobench uses in its test files involves a pre-populated skiplist and then only executes updates and lookups – for a set duration. In contrast the pre-built tests for LevelDB populated the skiplist and performed only lookups in its testing script.

After producing a replicable, maintainable version of LevelDB, we wrote tests measuring LevelDB's performance in accordance with the tests described in "A Skiplist for Multicore". We tested monotonically increasing keys/values, across single and multithreaded instances, using different proportions of updates. In order to compare the performance of the Rotating Skiplist and the LevelDB Skiplist, we created custom tests which could operate on the skiplists of both codebases performing lookups and updates for configurable durations. The results of these performance assessments is shown below.

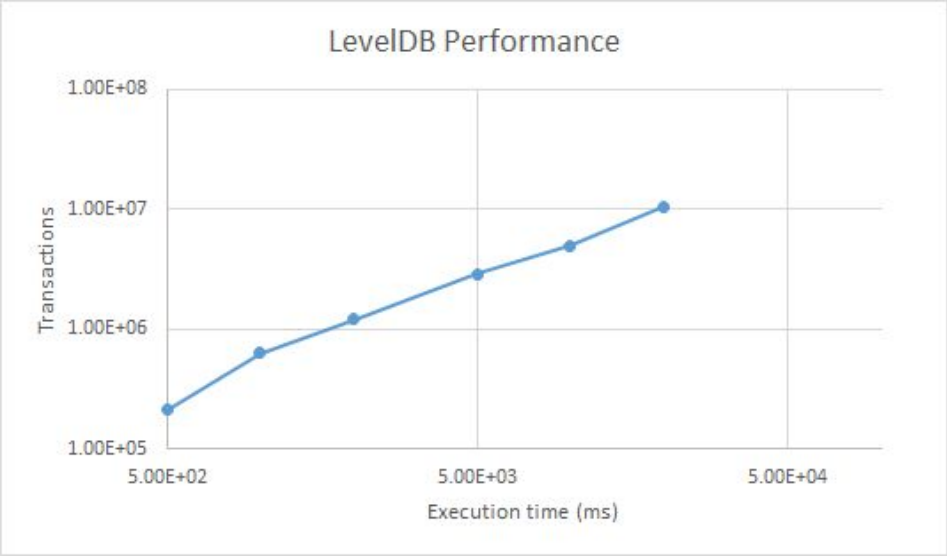


It is important to note that because the LevelDB skiplist must be externally synchronized we only performed these performance tests using a single thread. From the concurrent tests packaged with LevelDB and the rotating skiplist's documented performance is likely that that Rotating Skiplist will continue to outperform the LevelDB skiplist by an increasing margin with the further addition of threads.

LevelDB's iterator methods are also different from the rotating skiplist implementation in Synchrobench, where the original skiplist implementation has custom findlessthan() and findlast() commands. Some of the code for these methods was convoluted, and used method from header files that did not provide any insight as to how they operated, or what needed to be replaced for a rotating skiplist to be implemented.

LevelDB Skiplist Structure

LevelDB itself is a complex system which focuses on wrapping a skiplist for datastore-like transactions. To determine the latency of the system under various workloads we conducted initial tests on LevelDB. The results of a performance assessment of 70% lookups, 20% inserts, and 10% deletes is shown below.



We are not focused on the entire levelDB system, just the skiplist within. The remainder of this section focuses on our work with that skiplist.

Instantiating a skiplist in LevelDB requires two arguments, an *Arena* pointer and a comparator. Since values in skiplists must have comparable values associated with the skiplist takes in a comparator which is used to determine ordering in the list. More notably for our application, however, is the *Arena* pointer. Arena is a memory allocating utility also created by Google which claims to have 'better allocation performance'. All skiplist memory management is handled through Arena. It is also notable that the LevelDB skiplist never performs physical deletes. All deletes are logical, which may be cleaned up later.

In contrast to the rotating skiplist, the skiplist that ships with LevelDB is not designed for concurrency. While multiple threads can access the structure all locking and synchronization happens externally, and is not controlled by the skiplist. For integration with the rotating skiplist this must be addressed. While it is not necessary to replace for functionality, it is detrimental to performance.

Rotating Skiplist Structure

The implementation of rotating skiplist in Synchrobench has the same logical structure as the skiplist in LevelDB, but has more sophisticated maintenance. For instance, a node as proposed in the paper can be marked as logically deleted or physically deleted. Depending on whether a node

is logically or physically deleted, the thread performing the current request pauses its operation, and cleans up the deleted nodes, and retries.

The rotating skiplist also uses a background thread to conduct maintenance on the skiplist. This maintenance thread removes unnecessary (repetitive) levels in the skiplist, base level cleanup, full skiplist level lowering, and more helper functions. Some of these helper functions use method calls from atomic_ops.h, which defines a number of compile-time variables and functions, most of which we couldn't understand.

Rotating Skiplist in LevelDB

Implementing the rotating skiplist in place of the existing skiplist in LevelDB was frustrating, as the Synchrobench version of the rotating skiplist had compatibility issues with the LevelDB framework. Even so much as getting the new header files, custom functions, and file hierarchy to correspond to levelDB's cmake methods was difficult. After achieving build attempts, many issues arose in trying to modify the rotating skiplist such that the methods would correspond with what LevelDB needed. Other LevelDB files needed to be modified for the rotating skiplist to perform correctly (in theory), such as db_impl. Many files, such as db_impl, have convoluted calls that access or modify the skiplists, some of which are not obvious, and lead to frustration while debugging.

To try and produce some results, we attempted to implement a scaled-down, less robust version of the rotating skiplist. This included moving the maintenance from the background helper thread, attempting to use locks instead of a non-blocking version (which would have likely been slower anyways), and only implementing the absolutely necessary methods for the rotating skiplist to be compatible with LevelDB. However, between the file structure, convoluted method calls, complexity of how skiplist is designed and built in synchrobench, and LevelDB architecture, we were not able to produce any meaningful results of a rotating skiplist implementation in LevelDB.

Results

While we did not create a version of LevelDB with an embedded rotating skiplist our analysis of the two skiplists currently in play give some insight into performance in such a system. The results, we theorize, would be highly dependent on workload. On even a single threaded implementation for workloads with a mixed load of reads and writes (90%, 10%) the native skiplist in LevelDB outperformed the rotating skiplist. However, as the workload progressed to 100% reads the rotating version performed impressively. This leads us to theorize two points:

First, simply replacing the current skiplist with the rotating skiplist would probably not result in the expected performance improvement. Additional subsystems must be considered. The process of batching inside LevelDB should be reassessed and the synchronization and locking elements removed. Additionally, a cleanup portion must be added to the end of each transaction or a cleanup process must be constantly active if deletes are to be considered. Secondly, it is possible that the rotating skiplist's approach of atomic transactions may underperform the traditional skiplist. While we initially did not expect this result our tests have shown that this is plausible. Further research must be done before reaching a definite conclusion.

References

I. Dick, A. Fekete, V. Gramoli. A Skip List for Multicore. Concurrency and Computation: Practice and Experience. 2016.

V. Gramoli. More than You Ever Wanted to Know about Synchronization. PPoPP 2015

K. Fraser. Practical lock freedom. PhD thesis, Cambridge University, 2003.

T. Crain, V. Gramoli and M. Raynal. No hot spot non-blocking skip list. In ICDCS, 2013.

M. Herlihy, Y. Lev, V. Luchangco and N. Shavit. A Simple Optimistic Skiplist Algorithm. In SIROCCO, p.124-138, 2007.

T. Crain, V. Gramoli and M. Raynal. A speculation-friendly search tree. In PPoPP, p.161–170, 2012.

M. Arbel and H. Attiya. Concurrent updates with RCU: Search tree as an example. In PODC, 2014.

P. Felber, V. Gramoli and R. Guerraoui. Elastic Transactions. In DISC 2009.