# Systems Project: Final Report

IAN KELK, Harvard CS265, Spring 2023

## 1 DESIGN DESCRIPTION

### 1.1 Introduction

Log-Structured Merge (LSM) trees are a data structure designed for managing large data volumes, particularly in write-intensive workloads. They consist of multiple sorted key-value pair levels, with the first level being an in-memory buffer called a Memtable and the rest being immutable sorted files or "runs" on disk. LSM trees excel in high write performance, optimized read performance, and storage efficiency. They use Bloom filters to speed up read operations and have different compaction strategies to maintain efficiency. However, they can experience write and space amplification and slower read performance in some cases compared to B-Tree based systems. Popular databases like Apache Cassandra, RocksDB, and LevelDB utilize LSM trees, which are ideal for frequent write workloads like time-series data, event logging, and wide-column databases.

This project implements a LSM tree using a multi-level tiered structure, where runs are periodically merged between levels to maintain overall data organization. On the test MacBook Pro it was able to reach sustained write speeds of 1.2M per second and read speeds of 40k per second on databases containing over 1.3 billion entries comprising 10 GiB. As well, it has level-granular concurrency safeguards and can be simultaneously accessed by numerous clients depending on the hardware; on my laptop it was tested with 1-10 simultaneous clients performing mixed read and write tasks, and was found to run 2.5x times faster with four clients sending an equal mix of GET and PUT workloads.

### 1.2 Technical Description

*1.2.1 Choice of Programming Language and Data Structures.* I chose C++ for both its Standard Library (STD) which I use extensively, and it's object oriented nature. I make use of smart pointers, maps, vectors, and shared mutexes which are made much easier with the C++ STD. Additionally, since the project is highly dependent on experimentation, having classes to encapsulate the logic of the different sections makes it much easier to try different approaches. I've implemented every knob for experimentation as command-line arguments, and have run experiments through Jupyter notebooks to capture the data.

The available knobs that can be modified to tune the database, accessed through the command line when starting it up, are:

(1) **Error rate,** $e$ This option sets the Bloom filter error rate. The Bloom filter is a probabilistic data structure used to test whether an element is a member of a set. The error rate represents the probability of a false positive, i.e., the Bloom filter indicating that an element is in the set when it is not. The default error rate is 10%.
(2) **Level fanout,** $f$, denoted as 'f', determines the size ratio between the various levels in an LSM tree. Each level can have a maximum of f runs, and the maximum run size increases exponentially with f. A higher fanout results in a broader and shallower tree structure. The first level has runs of size $bf$, the second level has runs of size $bf^2$, the third $bf^3$. The level fanout plays a crucial role in balancing read and write throughput. Its default value is set to 10.
(3) **Number of pages in buffer,** $n$**:** This option configures the size of the buffer by specifying the number of disk pages. The buffer is an in-memory component that temporarily holds the data before it is written to disk. The default size is 128 disk pages, which on the test laptop

results in a 2MiB buffer. When the server is started, it prints the size of the disk page size to allow users to adjust this parameter according to their machines.

(4) **Level compaction policy, $l$,** determines the compaction policy used by the LSM tree. Compaction is a process that merges data from multiple sorted runs to reduce the amount of overlapping data and improve read performance. The default policy is TIERED. Other available policies are LEVELED, LAZY_LEVELED, and PARTIAL.

(5) **Compaction percentage, $c$,** sets the compaction percentage used only for the PARTIAL compaction policy. The value represents the fraction of data that should be compacted in each run, which helps balance compaction work and write amplification. The default compaction percentage is 0.2, or 20%.

(6) **Number of threads, $t$,** specifies the number of threads created in the thread pool for handling RANGE queries and compaction. The thread pool helps distribute the workload among multiple threads to improve performance and responsiveness. The default number of threads is 10.

Additional command-line arguments include $p$ to specify a port to run the server on, $v$ to display thread-specific progress messages from clients, $s$ to display the throughput of the entire database for all connected clients combined, and $d$ to choose the database directory where the data will be persisted.

## 2 DESIGN DESCRIPTION

### 2.1 Memtable (buffer)

I chose a `std::map` for the **Memtable (buffer)** as it is a sorted key-value store which enforces key-uniqueness and simplifies the process of merging sorted runs when flushing the buffer to disk or when compacting levels. When data is flushed to disk in sorted order, it obviates the need for an additional sort process to create sorted runs, which are crucial for efficient reads and writes in an LSM tree. `std::map` is typically implemented as a balanced binary search tree, such as a Red-Black tree or an AVL tree. This guarantees that the height of the tree remains logarithmic, ensuring that insertions, deletions, and searches are performed with a time complexity of $O(\log B)$, where B is the number of entries in the map. This makes it an efficient data structure for managing key-value pairs in memory.

`std::map` also enforces key uniqueness, which simplifies the process of updating and deleting keys in the `Memtable`. When a new key-value pair is inserted into the `Memtable`, if the key already exists, the value is simply updated. This behavior aligns well with LSM tree requirements, as newer versions of keys should overwrite older versions when the `Memtable` is merged with on-disk data. This would improve the speed of skewed PUT requests where the same entries are inserted many times, as they would be handled in memory and simply change the value in the `Memtable`.

Finally, since `std::map` allows for efficient retrieval of key-value pairs within a specified key range. The iterator interface provided by std::map enables easy traversal of keys in sorted order.

The trade-off of using `std::map` is that as it grows very large is that the time complexity of insertions, deletions, and searches is $O(\log B)$. As the number of entries increases, these operations will take longer to complete. For large buffers, an alternative structure such as the *sparsehash* hash map library would be a better choice, although it would require sorting when flushed.

### 2.2 Levels

The **Levels** are stored in a `std::vector` of `std::unique_pointers` to `Level` objects, and each level has a `std::deque` double-ended queue of `std::unique_pointers` pointing to `Run` objects. I chose to keep a vector of pointers to `Level` objects to ensure that level locking for concurrency is

more secure. If the vector were of the `Level` objects themselves, a new level could lose its mutex lock when copied into the vector, or it could be instantiated partly before the mutex was locked if emplaced directly. I also chose to keep a queue of pointers to minimize the copying of the `Run` objects in memory, which can get quite large since they contain the Bloom filters and fence pointers. The main advantage of using a `std::deque` for `Runs` is that I can add new `Runs` to the *front* of the `std::deque`, thereby making the most recent Runs at the front and obviating the need to iterate through the level backwards when searching for the newest appearance of a key. In this way, the runs are stored in a reverse-chronological level, where each level has newer data than lower levels, and each `Run` in the `Level` is in order from newest to oldest.

*2.2.1 Run Files.* The on-disk data layout for the **Run** class is a binary file where each key-value pair is stored as a continuous sequence of bytes. The keys and values are stored as a pair of 32-bit integers, with the key first followed by the value. They are abstracted in the program as a `struct` of a key-value pair of `KEY_t` key and `VAL_t` value, the same as they are defined in the provided `generator.c` file.

The file is created when the `Run` object is constructed, and data is written to the file using the C++ `std::ofstream` file abstraction. I had initially used the C `write()` system call, but found that `std::ofstream` for writing and `std::ifstream` for reading doubled the speed of the file access. Because the runs are flushed from the `Memtable`, they are automatically already sorted, which allows for efficient range queries as the data can be read from the file in a sequential manner.

*2.2.2 Fence Pointers.* The `Run` object also maintains a vector of **fence pointers** marking the beginning of each disk page. For each disk page, a fence pointer is created, which stores the minimum key and maximum key present in that page. This allows the system to determine bother whether and where a sought key might be present in that page without having to search through the entire file. Binary search is then used on that page of the `Run` file to find the key.

The fence pointers are stored in memory as a `std::vector` and not written to the files. Along with the fence pointers, the `Run` also stores the maximum key value in the file, since fence pointers are only added when every multiple of the disk page size has been written. The maximum key serves as an upper bound for searching the very largest keys in the `Run`. The fence pointers are very simple in their implementation, as they are just a straight `std::vector` of `KEY_t` keys.

*2.2.3 Bloom Filters.* In addition to the key-value pairs, the Run object also maintains a **Bloom filter** implemented with a `boost::dynamic_bitset`. The Bloom Filter is initialized with a given capacity and a tunable error rate. It calculates the number of bits $m$ and the number of hash functions $k$ required to achieve the desired error rate with the following formulas:

$$m = \lceil -\frac{n \cdot \ln p}{(\ln 2)^2} \rceil$$

$$k = \lceil m \cdot \frac{\ln 2}{n} \rceil$$

where $n$ is the capacity of the Bloom Filter and $p$ is the desired error rate. The Bloom filter computes two 64-bit hash values (*hash1* and *hash2*) using the XXH3 hash function. I chose the XXH3 specifically because it has high performance on ARM64 architectures such as my M1 MacBook. The filter then iteratively computes and sets the corresponding bit positions in the bitset for each of the $k$ hash functions.

I initially wrote my own dynamic bitset implementation, but found the `boost::dynamic_bitset` library was faster and could guarantee that bits were used instead of bytes.

### 2.3 Serialization and Deserialization for Data Persistence

In order to have a persistent database store that can resume after it is shut down, I implemented **serialization** and **deserialization** with JSON. While this is effective and works perfectly for my purposes in this project, it has proven to be a *very* poor method of saving the state of the database. JSON requires a complete rewrite every time it is written to disk, and cannot be easily updated due to merges. Thus it cannot be constantly updated, and can only be written when shutting down the server, making it unusable for recovering from crashes. It is also written in plain text, and thus creates massive JSON files. Changing this system to an appropriate one that can handle crashes would be an excellent future goal for the project.

### 2.4 Compaction Policies

This project's LSM tree supports four types of compaction: **tiered**, **leveled**, **lazy leveled**, and **partial**.

*2.4.1 Tiered Compaction.* Tiered compaction organizes data into levels with multiple ordered Runs. The policy aims to keep the size of Runs close on the same level, although that's not guaranteed because runs can become smaller than their maximum size when merged with other runs, and duplicates are removed. When the data size of one level reaches the `Level`'s limit, the entire level is merged and flushed to the next level to become a larger Run.

The advantage of tiered compaction is its low write amplification, making it suitable for write-intensive workloads. However, the drawbacks include higher read amplification and space amplification compared to leveled compaction.

*2.4.2 Leveled Compaction.* Leveled compaction merges all the Runs on all `Levels` any time a new Run is added. This means that a merge is triggered every time the buffer is flushed, and that each `Level` has at most a single Run. This method reduces read amplification and space amplification while providing fine-grained task splitting and control.

Advantages of leveled compaction include reduced space amplification and read amplification compared to size-tiered compaction. This makes it more suitable for scenarios with a higher number of reads and fewer writes. However, leveled compaction has higher write amplification, which can be problematic in scenarios with many random writes.

*2.4.3 Lazy Leveled Compaction.* Lazy leveled compaction is a hybrid approach that combines aspects of both tiered and leveled compaction policies. In this approach, the largest level operates in leveling mode while the other levels work in tiering mode. This policy is well-suited for mixed workloads involving updates, point lookups, and long-range lookups.

The advantages of lazy leveled compaction include lower space amplification and read amplification compared to tiered compaction, as well as lower write amplification compared to leveled compaction.

*2.4.4 Partial Compaction.* Partial compaction works by merging a subset of sorted data files from one level of the LSM tree with the corresponding files in the next level. This process reduces the number of files that need to be read when searching for a specific piece of data, as well as the number of files that need to be written when updating or deleting data. In this project, I've used a simple heuristic to determine which Runs are most likely to have the smallest ranges of keys, and therefore have the greatest number of duplicates that can be removed. A compaction percentage, $c$, is a parameter of the LSM tree and ranges between 0 and 1. It is multiplied by the number of Runs in the `Level` and uses either the integer ceiling of that result or the number 2, whichever is

greater. It then iterates through the `Level` and finds the group of `Runs` with the smallest combined differences between their first and last keys.

Partial compaction requires less memory than the other policies, as it doesn't require loading all the `Runs` of a `Level` into RAM, which might have insufficient space. The trade-off is that this results in many more `Run` files than with policies that compact entire `Levels`, and therefore results in slower reads.

### 2.5 PUT Queries

The `put()` function of `LSMTree` first attempts to insert the key-value pair into the `Memtable` buffer. If the buffer is full after the insertion, the function creates a sorted vector containing all the key-value pairs from the buffer. It then empties the buffer and inserts the new key-value pair so that the buffer is left with a single new entry; this is all done under the protection of a buffer mutex to make the buffer modification thread-safe.

`put()` then checks the first level of the tree to see if there is room to flush the buffer to the level. If there is no space for the buffer, the function moves the `Run` objects in memory until space is created. At this point it does not touch the underlying files, but it does create a "compaction plan", which is a map of `Level` numbers and segment bounds to compact later. Since everything is done in memory at this stage, it's very fast.

A new `Run` is constructed and a pointer to it stored into the first level. Then, the contents of the vector copy of the buffer are flushed into that run. If the compaction plan is not empty, the function locks the appropriate `Level` mutexes, executes the compaction plan, and then clears the compaction plan.

The worst case time complexity for PUT is $O(B+\log B+L)$ where $B$ is the size of the buffer and $L$ is the number of levels. Note that this analysis does **not** take the time complexity of compaction into account, as we will consider that a separate event despite being triggered by `put()`. The algorithm and time complexity calculation for PUT queries are shown in Algorithm 1.

### 2.6 DELETE Queries

DELETE queries are handled identically to PUT queries, but insert a special value called a TOMBSTONE to indicate the key has been deleted. I noticed from Test 0 that the possible values are restricted to KEY_MAX of 2147483647 and KEY_MIN of -2147483647, leaving the value -2147483648 available to use as the TOMBSTONE. TOMBSTONE values are kept in the LSM tree until they reach the final level, where they are removed upon compaction.

### 2.7 LOAD Queries

LOAD queries are very simple, and send a binary stream of key-value pairs to be inserted as PUTS. They have no special algorithm and their time complexity is identical to PUT. Their advantage is their small size since each key-value pair is exactly 8 bytes, and their disadvantage is that they are inflexible and limited to just PUT queries.

### 2.8 `LSMTree::moveRuns()`

In the `LSMTree::moveRuns()` function, the goal is to make sure there is enough space in the current level to accommodate runs before compaction. The function takes an integer, `currentLevelNum`, as an argument. The function first checks if the current level has enough space; if it does, there is no need to move any runs and the function returns.

If there isn't enough space in the current level, and it is not the last level, the function proceeds to lock the next level and checks if there is enough space to move runs from the current level. If the next

level also does not have enough space, the function recursively calls itself with `currentLevelNum+1` as the argument, thereby recursively moving the whole process to the next level.

If the current level is the last level, that means that all the levels that currently exist are full. The function creates a new level, locks it, and moves runs to it.

The `LSMTree::moveRuns()` function behaves differently based on the level compaction policy. If the policy is TIERED or LAZY_LEVELED and the current level is not the last level, the function moves the whole current level to the next level. If the level policy is LEVELED or LAZY_LEVELED and the current level is the last level, the function appends the current level to the next level. When the level policy is PARTIAL, the function moves the best segment runs to the next level.

The way that the runs are selected is using The `LSMTree::findBestSegmentToCompact()` function, which is responsible for finding the best segment of runs to compact within a particular level based on the sum of key differences from the first key in the first run of the segment to the last key in the last run.

First, the function calculates the number of runs that should be merged in a segment. It computes this value based on the LSM tree's compaction percentage and the level's total runs. `LSMTree::findBestSegmentToCompact()` returns the start and end indices corresponding to the segment with the lowest sum of key differences. This segment represents the one that is best suited for compaction in the current level according to this simple heuristic.

The key part of `LSMTree::moveRuns()` is that it is only moving `Run` objects around in memory and creating a compaction plan. It runs extremely quickly since it doesn't make any changes to the underlying file structure, and as a result is the only major function that has a mutex blocking it completely from more than a single thread. Once `LSMTree::moveRuns()` completes, all the `Run` objects are in the correct levels, and a plan is set to compact the correct segments of each level. The worst-case time complexity of `LSMTree::moveRuns()` is $O(N*\log(K))$ where N is the total number of key-value pairs in the segment and K is the number of runs in the segment. The algorithm and time complexity calculation for `LSMTree::moveRuns()` are shown in Algorithm 2.

### 2.9 `LSMTree::executeCompactionPlan()`

Once the `Runs` have been moved and a compaction plan set, the `LSMTree::executeCompactionPlan()` function locks the levels to have compaction operations performed on them, and then enqueues the compaction tasks to a thread pool. This way, only the levels currently being compacted are locked to other threads, which means that GET queries coming in that find their key on a higher level that is not being compacted can successfully retrieve their key-value pair. Later on, in Experiment 3, it's revealed that running 4 concurrent clients with GET and PUT queries is significantly faster than running a single client with the same combined workload.

### 2.10 `Level::compactSegment()`

The Level::compactSegment() function is used to compact runs within a specified segment in a level according to a provided pair of values representing the boundaries of the segment that need to be compacted.

It then iterates through the runs in the specified segment, retrieves their corresponding vectors of key-value pairs, and adds the first element of each run to the priority queue. The priority queue is necessary since more recent runs take priority if there are duplicate keys, so the final accumulated vector needs to only take the most recent key-value pair available. It also removes any deleted items indicated by a TOMBSTONE if the level being compacted is the last one in the tree, since these values are no longer needed and only clutter up the largest level.

Once the merging process has finished, the compacted key-value pairs are flushed into the a newly created `Run` object and returned, where it replaces the `Runs` it compacted in the level,

and the old files are deleted. The worst-case time complexity of `Level::compactSegment()` is $O(N * \log(N))$ where $N$ is the total number of key-value pairs in segment runs. The algorithm and time complexity calculation for `Level::compactSegment()` are shown in Algorithm 3.

### 2.11 GET Queries

A GET query first searches for the key in the memtable (buffer). If the key is not found in the buffer, it then iterates through the levels, locking each one with a shared lock as it goes, and then the runs of the levels, using the Bloom filters and fence pointers to optimize the search process. If the key is found in either the buffer or the runs, the corresponding value is returned; otherwise, a null pointer is returned, indicating that the key is not in the LSM tree.

I spent some time trying to make a concurrent GET query using the thread pool, but in the end it was getting overly complex and didn't work very well. The idea was to have multiple threads searching the Runs, and if one found the key, it would check if all the earlier Runs had completed their search yet. If they had, and none of them had found the key, it would signal all the other threads to stop searching immediately and would then return the key. It sounded like a promising idea in theory, but even had it been successful, that could be a lot of extra work done to find a key even if the signalling worked properly. In the end, I scrapped the idea and kept GET single threaded.

The GET query has a worst-case time complexity of $O(\log B + L * R_i * \log P)$ where $B$ is the number of entries in the buffer, $L$ is the number of levels, $R_i$ is the number of runs in a level $i$, and $P$ is the number of key-value pairs in a page. The algorithm and time complexity calculation for GET queries are shown in Algorithm 4.

### 2.12 RANGE Queries

The RANGE query is used to retrieve a vector containing all the key-value pairs within a specified key range. The range is inclusive of the start key and exclusive of the end key. Because a RANGE query almost always has to search every Run (except in the case where it finds the maximum possible keys in the range before searching all the Runs), it's an excellent candidate for parallelization.

The function begins by validating the start and end keys to ensure they are within the acceptable key range, distinct, and in the correct order. It then searches for key-value pairs within the specified range in the buffer in memory, and adds the found pairs to a priority queue. If the buffer search doesn't yield all possible key-value pairs within the range, the function proceeds to search the levels on disk for the remaining pairs using multi-threading to improve efficiency. Afterward, the found key-value pairs are added to the priority queue. The function then merges all key-value pairs from the buffer and levels in the priority queue into the result vector, removing duplicate entries during the process. It also removes any tombstones, or deleted keys, from the result vector. Ultimately, the function returns the result vector containing the key-value pairs within the specified range.

The worst-case time complexity of the RANGE query is $O(\log(B) + L * R * (K + log(P)))$ where $B$ is the buffer size, $K$ is the number of keys in the range, $L$ is the number of levels, $R$ is the maximum number of runs per level, and $P$ is the number of pages per run. The algorithm and time complexity calculation for RANGE queries are shown in Algorithm 5.

### 2.13 Parallelization

The LSM Tree implements a concurrent server that listens for incoming client connections and processes their commands. It's main functions are:

- `main()` sets up the signal handler, parses command-line options, creates a server instance, sets up the LSM Tree, and starts the server by running the `run()` method. It also creates a

separate thread to listen to standard input, enabling the user to interact with the server while it's running.
- `run()` listens for incoming connections, and for each new client, it creates a new thread that handles the client's requests.
- `handleClient()` runs in a separate thread for each connected client. It reads commands from the client, parses them, and handles them accordingly. The method uses a **`std::shared_mutex`** to implement concurrency control, ensuring exclusive or shared access to the LSM Tree based on the operation being performed.
- `handleCommand()` processes the client's command by executing the requested operation on the LSM Tree. Concurrent access to the database is managed at a per-level granularity. The entire database is only momentarily locked when the `LSMTree::moveRuns()` function is triggered, but it is a memory only function and unlocks the unaffected levels before the disk operations are begun.

## 3 OPTIMIZATIONS AND ENHANCEMENTS

### 3.1 Optimizations

- Implemented all 3 options of tiering, leveling, and lazy leveling.
- Implemented partial compaction with percentage threshold as a control.
- Added internal multi-threading for RANGE and compaction operations.
- External concurrency for multiple clients to connect with level-granular level locking.
- Binary search with fence pointers for GET and RANGE. If start key is not in the run, it selects the next largest.
- Priority queue for fast accumulation of results from merging and RANGE queries.
- xxHash super-fast hashing for Bloom filters.
- MONKEY Bloom filter optimization

### 3.2 Usability Enhancements

- Allow selection of data directory when creating multiple databases.
- Both client-specific and global counters printing out current operation counts with selectible frequency.
- Allow user to select the port to run multiple servers at the same time.
- Printable Bloom filter summaries including theoretical and measured false positive rates
- Printable report of how many GET and RANGE hits and misses have occurred
- Quiet mode for client to not print everything when running test workloads
- "Info" command to get a full summary of the tree, including number of runs and keys and the current layout
- Added optional integer for "stats" command to prevent computer from going insane printing 1 billion key-value pairs
- Added I/O summary including simulating disk types with time penalties for slower disks to estimate real-world latency

## 4 CHALLENGES

- I'm a pretty seasoned programmed, but I hadn't written C++ in more than 20 years. There was quite a bit of ramp-up time and at this point I've likely put more than 300 hours into the project. Since the midway check-in, I've rewritten significant parts of the compaction logic to make it work for concurrency.

- Initially I had Run objects delete their files when they were destroyed. At that point I was also adding them to a Level's **`std::deque`** which *copies* them in and then destroys the original. This resulted in files seemingly randomly deleted until I found the problem.
- My Bloom filters were performing terribly when I was initially using the **`std::hash`** function. I tried double hashing but it remained awful with a 90% plus FPR. I switched to the non-linear and much faster XXH3 hash function and the improvement was immediate and impressive. Workloads with a million GETS went from 4 minutes to 16 seconds.
- I had a lot of difficulty debugging some functions, and most notably the recursive `moveRuns()` function. I eventually just improved my debugging ability and realized that just because the debugger displays a ?? symbol as a value, that doesn't always mean the value wasn't set.
- For lazy leveling, I was originally setting and tracking a boolean flag to mark the last level. This was faulty and prone to errors, until I realized that I could move the logic up to the LSMTree itself which could just check the `levels` **`std::vector`** to see if it was at the end.
- I had an interesting problem where my database stopped being able to open Run files at a given point. What was even stranger was that if I ran it through the debugger, it didn't crash. I eventually figured out it was opening too many file descriptors, and the OS had a limit of 250 but the debugger seemed to have no limit at all. I fixed it by ensuring file descriptors were closed when no longer needed.
- Concurrency was a major challenge I hadn't anticipated. I spent days working on PUT and merges, thinking that once that was finished, GET would be a cakewalk. However GET was even worse, and somehow RANGE continued that tradition. I liken the idea of concurrency to trying to control an angry swarm of robot bees you've accidentally hired to work in your factory. Also the bees fly at the speed of light and if given the chance, will crash into each other at every opportunity.
- Not having access to a dedicated Linux computer for performance profiling. I ended up using a combination of `cachegrind` which is extremely slow, and Activity Monitor on the Mac that I literally had to film using screen recording software to then match up the reported numbers with the information being printed by the database.

## 5 EXPERIMENTATION

The experiments in this section were mostly performed on my 2021 MacBook Pro. It is a 2021 model with an M1 Max CPU, 64GB LPDDR5 RAM, 10 CPU cores 10 (8 performance and 2 efficiency), 32 GPU cores, and an 8TB NVMe SSD. It was running the current Mac OS 13.3.1 Ventura.

The other machine used for getting cache misses is a Microsoft Azure instance. It's description is that it is a standard B4ms with 4 vCPUs and 16GiB RAM. It was running Ubuntu 20_04-lts-gen2.

**Note:** I use the term "sliding window throughput" in a few of my experimental plots. A sliding window is a technique used to analyze and process data points in a continuous data stream, in this case the very rapid activities of a database workload. The sliding window represents a fixed-size subsequence of consecutive data points from the larger data stream. The window "slides" along the data stream one data point at a time, and processes the data within the window. This allows for continuous analysis of the data stream as it evolves over time. For example, in my plots where I log the throughput of 1B PUTs, I have a sliding window size of 100M. This allows a more granular look at what happens during the sequence rather than just the overall average.

By using a sliding window, we can notice short-term changes or trends in the data, such as seeing when a major compaction took place, while still considering the overall context of the data stream.

### 5.1 Comparison of Compaction Policies

In the first experiment, the performance and efficiency of various compaction policies were evaluated. These policies include Tiered compaction, Leveled compaction, and Lazy Leveled compaction, which applies Tiered compaction on every level except for the last one, where Leveled compaction is used. Additionally, Partial compaction with a specified compaction percentage was introduced. The compaction percentage, denoted as $c$, ranges from 0 to 1 and is used to determine the number of Runs to compact in a Level.

The primary objective was to assess the behavior of the system when utilizing a considerably larger buffer size than previously tested: 256 MiB. The goal was to compare the various compaction policies under this configuration and analyze their performance with a large dataset of 10 GiB, equivalent to 1,342,177,280 key-value pairs.

Furthermore, the potential performance bottlenecks arising from using a `std::map` as the LSM tree buffer were investigated, and the throughput and overall efficiency of the implemented compaction policies were determined. Figure 1 shows the results.

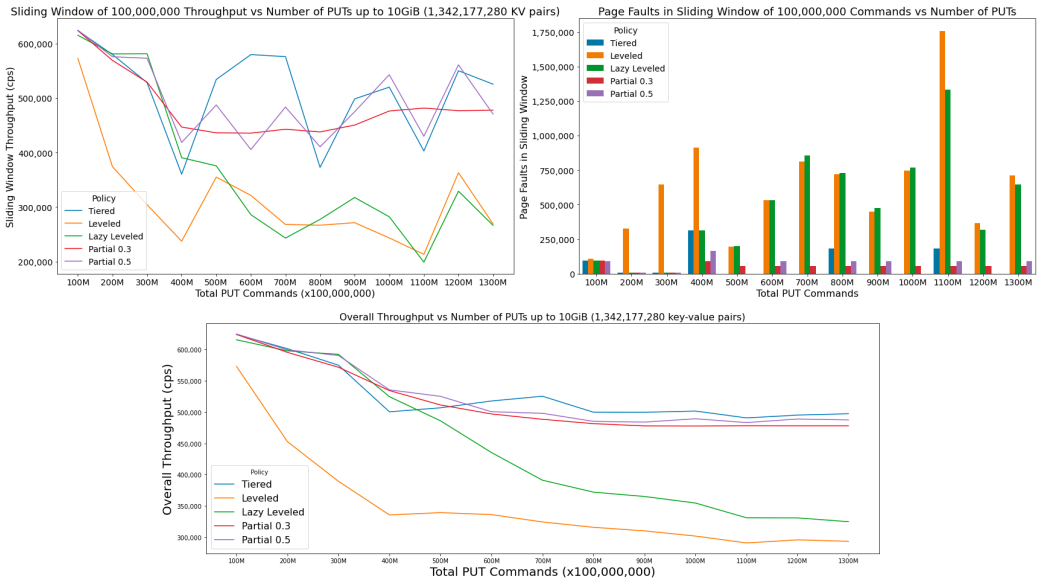


Fig. 1. A comparison of the performance and efficiency of various compaction policies—Tiered, Leveled, and Lazy Leveled—alongside Partial compaction with different compaction percentages, using a 256 MiB buffer and a 10 GiB dataset. The investigation focuses on throughput and the impact of using a std::map as the LSM tree buffer. Using such a large buffer was indeed a bottleneck, as is seen in the plots for Experiment 2 which show much higher throughput with a comparatively tiny buffer.

### 5.2 Discovering Speedup with Small Buffers

While previously I'd had it in my head that a large buffer would result in faster throughput, I decided to use the same 2MB buffer I had worked with throughout the design and construction of the database. I realized that the use of a `std::map` might actually have the effect that I could reach great speeds by allowing the Memtable to sort a smaller set of data. Smaller buffer sizes result in better cache locality, as they are more likely to fit into the CPU cache. This might lead to fewer

cache misses and faster access to data in the `Memtable`. As well, when the `Memtable` is flushed to disk, the amount of data that needs to be written is proportional to the buffer size. Smaller buffer sizes could result in less data being written to disk, thus reducing write amplification.

I was concerned that smaller buffer sizes could result in more frequent flushing of the buffer to disk, which would increase the overall I/O overhead and slow down the system. This frequent flushing could slow down the write throughput of the LSM tree. However, I found the system ran about 2.5x faster than with the large buffer. This experiment was especially exciting to me, since my midway check-in report had a 1 billion PUTs time of 96 min 28 seconds, so lowering this to under 15 minutes is a massive improvement in performance. See Figure 2.

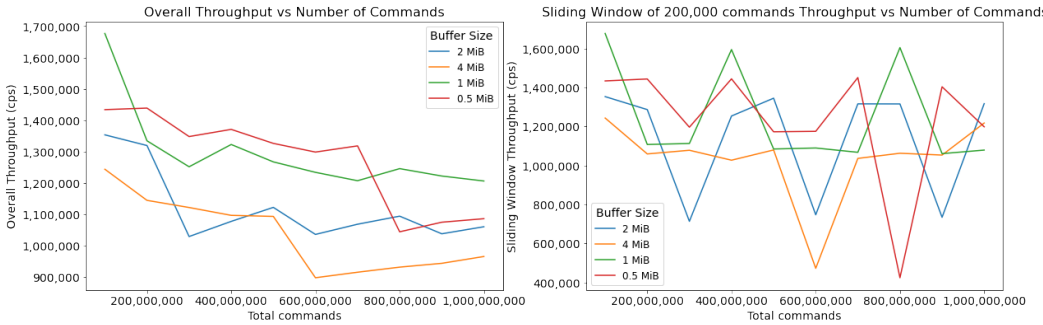


Fig. 2. I switched to a much smaller buffer and the overall speed increase was significant. The fastest for insertions was using a 1MiB buffer - 256x less than what I had used for my initial experiment. Over the insertion of 1 billion keys, the quickest experiment completed them in 13 minutes, 49.51 seconds, and even the slowest in 17 minutes, 16.74 seconds. All except the 4 MiB completed the 1 billion PUTs with a throughput over 1 million CPS, and not counting the initial burst of speed, the 1 MiB reached 1,266,611 CPS between 400 million and 500 million PUTs. It's interesting to see the detail in the sliding window plot, where it's clear that 0.5 MiB had a major compaction at 800,00,000, and 4 MiB had a major compaction at 600,000. The 1 MiB seems to avoid having compactions with such latency.

### 5.3 Testing concurrent throughput with multiple clients

The LSM tree database is thread-safe and supports concurrent access by multiple clients. However, I have observed different performance characteristics depending on the number of clients accessing the database. I've noticed that with 10 clients on my test MacBook, the performance is equivalent to single threaded speed. I assume this is because when multiple clients access the database simultaneously, they compete for shared resources such as CPU, memory, or disk I/O. As the number of clients increases, the contention for these resources can become more significant, which may lead to performance degradation. With enough clients, the contention might be high enough that the overall performance drops to single-threaded equivalent. My MacBook only has 10 cores after all; there is a limit to how much performance can be improved through parallelism and concurrency.

The LSM tree uses level-granular locking, which reduces lock contention among multiple clients (I had originally locked the entire database when performing writes at the time of the midway check-in). However, lock management can still introduce some overhead, especially when the number of clients increases. As more clients compete for the same lock, the overhead associated with lock acquisition and release may grow, leading to performance degradation.

The performance of the database can also be influenced by factors specific to the LSM tree structure, such as compaction and `Memtable` flushing. For example, when the first level is being

flushed, clients might experience slower GET and RANGE queries due to the need to search the remaining levels. Additionally, compaction can introduce performance overhead as it merges Runs and reorganizes the database. The impact of these factors on performance might vary depending on the number of clients accessing the database. By testing a few numbers of clients, I discovered the optimal client count on my test machine is four clients. See Figure 3.

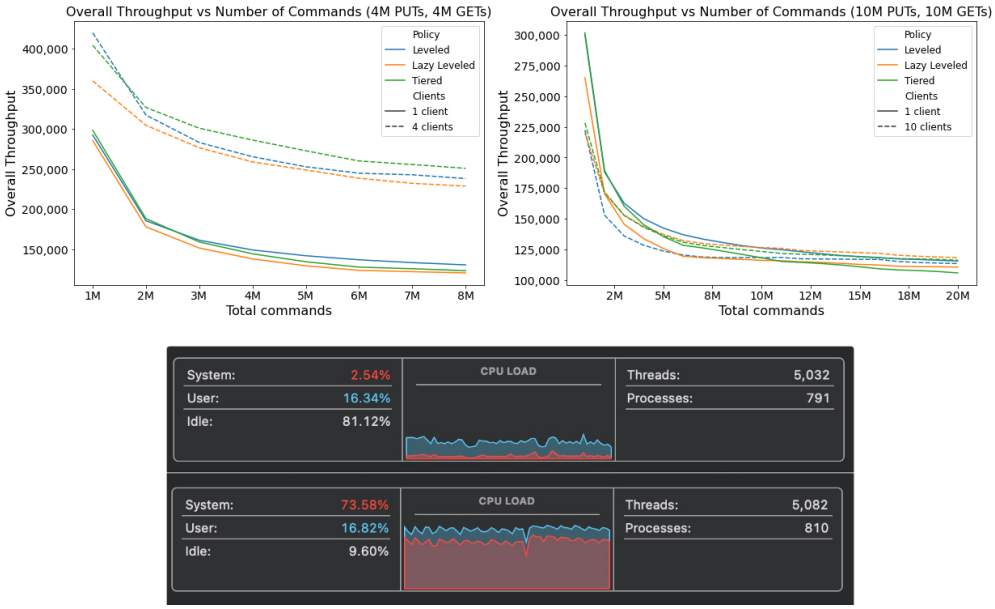


Fig. 3. In this experiment, I wanted to compare different numbers of simultaneous clients accessing the database with a mixture of PUT and GET queries. Each of the files created for the individual clients contains 1M PUTs and 1M GETs. This was to avoid the generator's tendency to frontload all of its GET and RANGE queries when creating PUTs. For the single client comparison, a workload of the equivalent number of PUTs and GETs was used. Thus on the left there are 4 clients totalling 4M PUTs and 4M GETs against a single workload with 4M PUTs and 4M GETs, while on the right there are 10 clients totalling 10M PUTs and 10M GETs against a single file with the same numbers. The effect is that the left plot is "zoomed in" a bit, but it's still very clear that there is a major difference in performance when using 4 clients on the test MacBook. Similarly, the CPU usage plots show the difference in CPU utilization of 2.54% for a single client to 73.58% for 10 clients while the experiment on the right was running.

### 5.4 Range queries using internal multi-threading with variable-sized thread pool

I've noticed some unexpected performance characteristics in the LSM tree with different numbers of available threads in the thread pool. It seems that finding the optimal number of threads for any given workload is not straightforward, due to the interplay between a few factors.

As in the previous experiment, when using multiple threads there can be synchronization overhead due to the locking mechanisms used to ensure thread safety. This overhead can increase as the number of threads grows, especially if the threads need to access shared data structures or contend for shared resources. As well, the likelihood of cache contention grows with more threads. This occurs when multiple threads access different data that map to the same cache line, causing cache line invalidations and cache misses. Threads are forced to fetch data from higher-latency

memory, resulting in performance degredation. The specific number of threads that cause this slowdown due to cache contention can vary depending on the system's hardware and the workload characteristics.

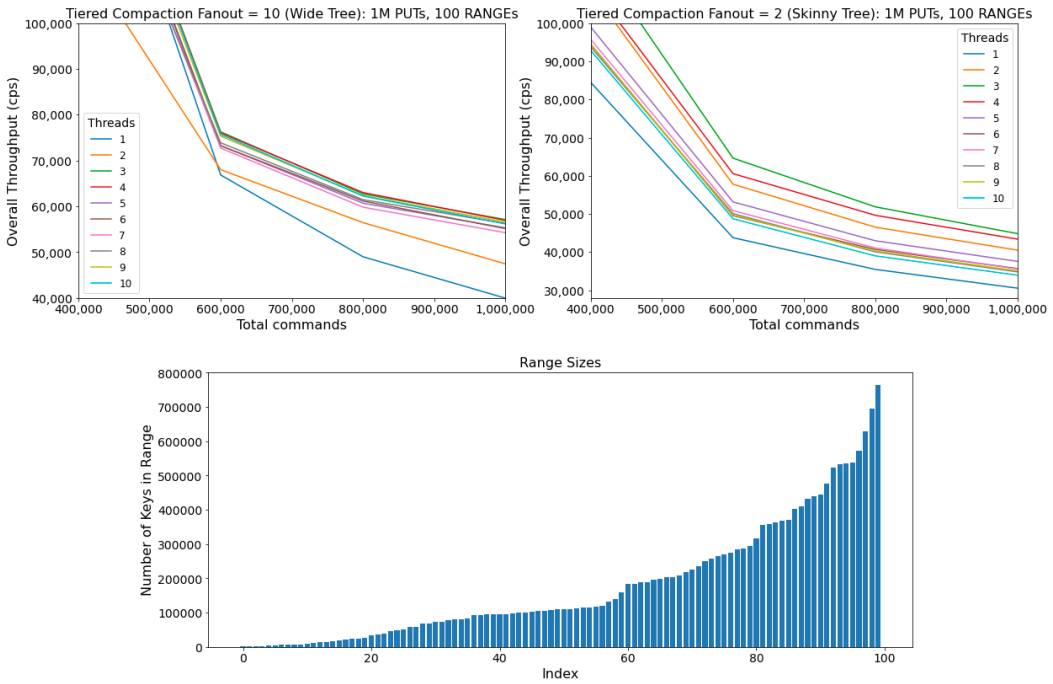


Fig. 4. A comparison of internal multi-threading for RANGE queries with 1M PUTs and 100 RANGEs. The generator provided has a limitation where it places all the RANGE queries at the beginning of the workload, creating a trivial RANGE list because at the point they are inserted, the database is nearly empty. Conversely, if you PUT a large number of entries, the ranges become enormous, with sizes in the millions. My solution here was to generate 1M PUTs and 100 RANGEs, then shuffle the workload. The result is that 75% of the ranges contain more than 50,000 entries, and in fact summing the 100 ranges up reaches a total of 17,989,964 entries to return. Since RANGE queries are multi-threaded, I used this method to see the number of threads that is ideal for a single client to use when performing RANGE queries. Surprisingly, on my test machine the number is three.

I mentioned earlier that my test MacBook has 10 cores; using more threads than the number of available CPU cores can lead to context switching overhead, which might degrade performance. In some cases, using fewer threads than the number of cores can result in better performance, as it allows the system to better balance the workload among the available cores. As well, the distribution of RANGE queries and compaction tasks among the different levels of the LSM tree can affect the performance of the thread pool. If the workload is unevenly distributed, having more threads might not result in a proportionate improvement in performance. If one level requires significantly more processing time than the others, having more threads may not improve the overall performance, as some threads will be waiting for the slowest level to complete.

The setup of the experiment was to use identical LSM trees with the smallest possible buffer (1 page size, or 16,384 bytes on my MacBook) except for the fanout ratio. On the left the fanout is 10,

and on the right the fanout is 2. This resulted in the "fat" tree having 3 levels, whereas the "skinny" tree had 8 levels. This way I could directly measure the effects of the internal multi-threading, which is level-granular. The more levels, the more multi-threading will have an effect.

The multi-threading had an expectedly higher effect on the skinny tree, and the number of threads was a little surprising. The fastest was 3, followed by 4, 2, and 5, then pretty much everything else followed by 2 and 1. See Figure 4.

### 5.5 Skewed GET queries

Skewed data can cause hotspots, where a small subset of keys is frequently accessed, leading to an increased cache miss rate. This is because the hot keys are spread across multiple levels, making it difficult to efficiently cache them. Leveled compaction is generally faster because it reduces read amplification by maintaining a smaller number of runs per level. However, the continuous merging in leveled compaction increases the chances of cache misses due to the constant movement of data between levels.

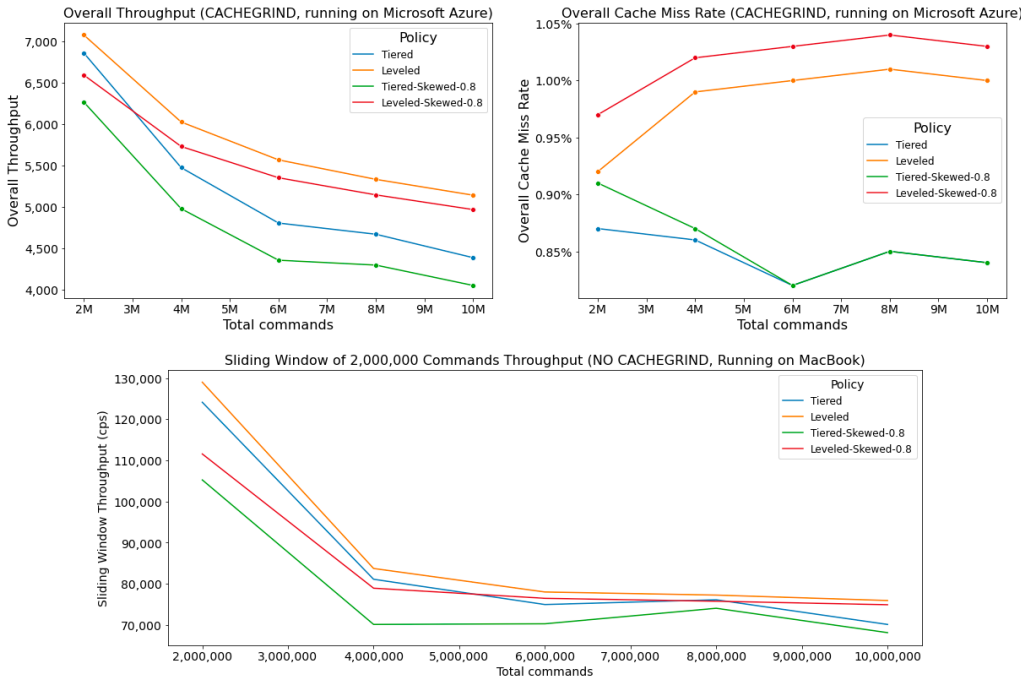


Fig. 5. In this experiment, the skewed data is causing the leveled policy to have more cache misses due to the aggressive merging and movement of data between levels.

In this experiment I wanted to compare how GET queries respond to skewness in a workload of GETs. Again I see a setup of equal amounts of PUTs and GETs with a gets-misses ratio of 0.5 to avoid the generator front-loading all the GETs. This was my first experience using `cachegrind` on Microsoft Azure, which creates roughly a 50x increase in runtimes, so you'll notice that in the top right the throughput is an order of magnitude slower.

The tiered policy, although slower, experiences fewer cache misses because the data is spread across multiple levels and not frequently moved. This might be improved by partition the data

based on access patterns to better handle hotspots and improve cache efficiency, or designing a compaction policy that adapts to the data's characteristics, combining the advantages of both tiered and leveled policies. Performance for skewed GETs can depend on the order in which keys are PUT and later retrieved with GET, particularly when there are no specific caching mechanisms for frequently accessed keys. This is because the tree organizes data by inserting newer data in the higher-level runs, and older data is gradually merged into lower-level runs. So if the skew is centered around an older key, it can slow throughput, whereas if it's a higher key it might improve it. LSM trees can still take advantage of the operating system's page cache to store frequently accessed disk pages in memory, which can help improve the performance of GET operations on frequently accessed keys. However, since the skewness slowed the throughput, it's worth looking at the cache misses. What's notable is that despite being faster, the leveled policy had many more cache misses.

### 5.6 MONKEY Bloom Filter Optimization

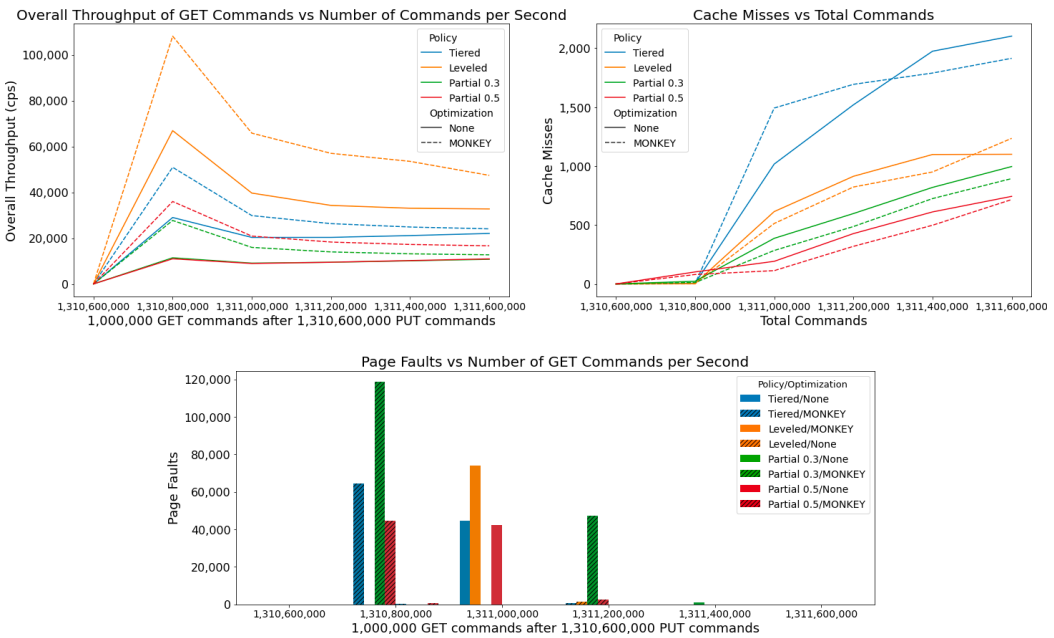


Fig. 6. I believe that the true benefits of MONKEY become apparent with much larger databases than I tested with, however it did have an notable effect with the 1 GiB (1,310,600,000 entry) datbase with the large 256 MiB buffer.

Here I wanted to test the effect of MONKEY key optimization using the same large buffer from Experiment 1 on large data. I used the algorithm for optimizing Bloom filter bitset sizes from the paper "Monkey: Optimal Navigable Key-Value Store". MONKEY's core insight is that the worst-case lookup cost is proportional to the total false positive rates of Bloom filters across all LSM-tree levels. Unlike traditional key-value stores that assign a fixed number of bits-per-element to all Bloom filters, Monkey allocates memory to filters across levels to minimize this sum. Analytical results show that Monkey reduces the worst-case lookup I/O cost's asymptotic complexity, while

empirical tests on LevelDB demonstrate that Monkey lowers lookup latency by 50% to 80% as data volume increases.

There is an initial burst that occurs when the workloads are started, and that can be attributed to caching, as the number of cache misses rises dramatically as the workloads run. As well, as can be seen below, there are a larger number of RAM page faults that were created by the partial compaction methods, and some for the leveled policy where performance also falls. This was probably the trickiest part of the experiments, as I had to manually gather the page fault data by screen capturing the Mac's Activity Monitor and entering the page fault values manually. For the cache misses, I ran the experiments using `cachegrind` on Azure, where it took nearly a week of running tests as each run was about 12 hours. Not having access to a dedicated Linux server was a hindrance for these experiments, as virtualized instances do not allow access to the underlying CPU counters for utilities such as `perf`.

The page faults plot still show spikes in RAM page faults where the initial burstiness occurred. At the beginning of the workload, the system might have had sufficient resources available to efficiently process the tasks. This would result in an initial burst of speed as the LSM tree quickly handled the incoming requests. As the workload progressed, the system's memory demand increased due to the growing size of the LSM tree and the data structures associated with it. When the system ran out of available physical memory, it started swapping memory pages between RAM and disk to free up space. This process caused page faults, as the system needed to retrieve the required data from disk when it wasn't available in memory. This introduced significant latency, as disk access are orders of magnitude slower than RAM.

As more data was swapped between RAM and disk, the system spent more time handling page faults, which reduced the time available for processing the LSM tree tasks. As the workload continued and memory pressure grew, the CPU cache might have also become more contested. This could have led to cache evictions and cache misses, as the system struggled to keep the most frequently accessed data in the cache. This is all speculation, but it's what I can draw from the plots.

## 6 CONCLUSION AND FUTURE WORK

Creating this LSM tree and finding little ways to eke out performance increases here and there has become a mild obsession over the past few months. I'm proud that I managed to get it's write throughput over 1.2M writes per second and read throughput over 50k reads per second, as early on it was several times slower than that. As I mentioned previously, one improvement it could definitely use is a much better way to continuously save its state (such as a write ahead log) rather than use the JSON method it does currently. I would like to try a hash map option to see if it could work for a larger buffer, and how that would compete with the small buffer performance I have now. I could also look into using libraries that have already been made highly performant for Bloom filters or the thread pool. I would also like to see if my work can be guided towards a research goal, since it's now a fairly solid baseline, and I would also like to gain more experience on profiling using tools like `perf` or Xcode's Instruments in order to make faster software.

## A APPENDIX: ALGORITHMS

**Algorithm 1** Put function in LSMTree class

```
procedure PUT(key, val)
    BUFFER(key, val)
    if buffer is full then
        bufferVector ← buffer
        CLEAR(buffer)
        BUFFER(key, val)
        if levels.front() has no space then
            MOVERUNS(1)                    ▷ Runs recursively, potentially through all levels
        end if
        Create new run and add it to levels.front()
        Flush bufferVector to new Run, adding Bloom filters and fence pointers
    end if
    if Compaction plan exists then
        Acquire locks for involved levels
        Execute compaction plan and clear it
    end if
end procedure
```

### Time Complexity Analysis

**`std::map`** is typically implemented as a balanced binary search tree with $O(\log B)$, so the time complexity is:

**Best-case time complexity:** $O(B)$

- Attempting to insert a key-value pair into the buffer: $O(\log B)$
- Copying the buffer to a vector: None (buffer doesn't become full in the best case)
- Moving runs: None (no merge across levels in the best case)

**Average-case time complexity:** $O(B + \log B + \log L)$

- Attempting to insert a key-value pair into the buffer: $O(\log B)$
- Copying the buffer to a vector: $O(B)$
- Moving runs: $O(\log L)$ (assuming partial cascading, but not in every `put()` operation)

**Worst-case time complexity:** $O(B + \log B + L)$

- Attempting to insert a key-value pair into the buffer: $O(\log B)$
- Copying the buffer to a vector: $O(B)$
- Moving runs: $O(L)$ (cascade effect goes through all $L$ levels)

These complexities have $B$ as the size of the buffer, and $L$ as the number of levels. An important note is that this analysis does **not** take the time complexity of compaction into account, as we will consider that a separate event despite being triggered by `put()`.

**Algorithm 2** moveRuns in LSMTree class

```
1:  procedure moveRuns(currentLevelNum)
2:      Lock the level if it's not the first level since the first level is already locked
3:      Set it to the iterator pointing to the current level
4:      if the current level has space for another run then
5:          return
6:      else
7:          if The next level is not the last level not exist then
8:              Exclusively lock the next level
9:              if The buffer won't fit in the next level then
10:                 Recursively call moveRuns(currentLevelNum+1)
11:             end if
12:         else
13:             Exclusively lock the vector of levels so we can add to it
14:             Create a new level, exclusively lock it
15:             Add a pointer to it to the levels vector
16:         end if
17:     end if
18:     if Policy is not PARTIAL then
19:         if Policy is TIERED or (Policy is LAZY_LEVELED and not the last level) then
20:             Add the current level's runs to the compaction plan for the next level
21:         end if
22:         if Policy is LEVELED or (Policy is LAZY_LEVELED and the last level) then
23:             Add the current runs and next level's runs to compaction plan for the next level
24:         end if
25:         Move the runs to the next level
26:         Erase all the current level's runs
27:     else
28:         Find the best segment of 2 or more runs (depending on compaction percentage)
29:         if The buffer won't fit in the next level then
30:             Add the segment to the compaction plan starting at the beginning of the next level
31:             Move the runs to the beginning of the next level
32:             Erase the selected runs from the current level
33:         else
34:             Add the segment to the compaction plan in the same level it's currently at
35:         end if
36:     end if
37: end procedure
```

## Time Complexity Analysis

The main steps in compacting the runs are merging the sorted runs and writing them to disk.

- **Best-case complexity:** $O(N)$ one run in the segment, iterate through all key-value pairs $N$
- **Average-case complexity:** $O(N * \log(K))$ priority queue for sorting adds $\log(K)$
- **Worst-case complexity:** $O(N * \log(K))$

Where N is the total number of key-value pairs in the segment and K is the number of runs in the segment.

**Algorithm 3** compactSegment Function in Level class

```
1:  procedure COMPACTSEGMENT(errorRate, segmentBounds, isLastLevel)
2:      Initialize empty priority queue pq
3:      Initialize new vector compactedKvPairs
4:      Initialize newMaxKvPairs to 0
5:      for all runs in the segment do
6:          Retrieve each run's key-value pairs vector, and add the first element to pq
7:          Add the run's maxKvPairs to newMaxKvPairs
8:      end for
9:      while pq is not empty do
10:         Pop top (smallest key) element from pq
11:         if not last level or the value is not TOMBSTONE then
12:             if the current key is not a duplicate then
13:                 Add current key-value pair to compactedKvPairs
14:             end if
15:         end if
16:         Add the next element from the top run to the pq
17:     end while
18:     Create new Run object compactedRun
19:     Flush compactedKvPairs to the new Run object (compactedRun)
20:     Replace the runs in the segment with the compactedRun
21: end procedure
```

## Time Complexity Analysis

- **Best case:** $O(N * \log(K))$ the input segment runs are optimally sorted, and every pop and push operation for the priority queue takes $O(\log(K))$ *time*
- **Worst case:** $O(N * \log(N))$ there is only one run in the segment ($K = 1$)
- **Average case:** $O(N * \log(K))$

Where $N$ is the total number of key-value pairs in segment runs, and $K$ is the number of runs in the segment.

**Algorithm 4** Get function in LSMTree class

```
1:  procedure GET(key)
2:      if key is not within valid range then
3:          return nullptr
4:      end if
5:      val ← search buffer for key
6:      if val is not nullptr then
7:          increment hit counter
8:          if val is TOMBSTONE then
9:              return nullptr
10:         end if
11:         return val
12:     end if
13:     copy levels to localLevelsCopy
14:     for all levels in localLevelsCopy do
15:         lock level with shared lock
16:         for all runs in level do
17:             if key is not in Bloom filter or key is outside fence pointers range then
18:                 return a null pointer
19:             end if
20:             Find the page index using fence pointers
21:             Search the page with binary search for the key and return a value
22:             if value is not a null pointer then
23:                 increment hit counter
24:                 if value is not TOMBSTONE then
25:                     return value
26:                 else
27:                     return a null pointer
28:                 end if
29:             end if
30:         end for
31:     end for
32:     increment miss counter
33:     return nullptr
34: end procedure
```

## Time Complexity Analysis

- **Searching the buffer** - $O(\log B)$, where $B$ is the number of entries in the buffer. This is also the best case time complexity.
- **Searching the levels** - The LSM Tree has $L$ levels. For each level, there could be multiple runs $R$. In the worst-case, we need to search all runs in all levels.
- **Searching the runs** - The runs are searched with binary search with a time complexity of $O(\log P)$ where $P$ is the number of key-value pairs in a page.

Thus the GET query has a time complexity of $O(\log B) + O(L * R_i * \log P)$ which can be simplified to $O(\log B + L * R_i * \log P)$ where $R_i$ is the number of runs in a level $i$. The best case is when the key is found in the buffer and has a complexity of $O(\log B)$.

**Algorithm 5** Range Function in LSMTree class

```
1:  procedure RANGE(start, end)
2:      if start or end is not within valid range then
3:          return empty vector
4:      end if
5:      if start > end then
6:          Swap start and end
7:      end if
8:      if start = end then
9:          return empty vector
10:     end if
11:     Initialize rangeResult, priorityQueue pq, mostRecentKey
12:     Search buffer for key range and add results to pq
13:     if all keys in range found in buffer then
14:         searchLevels ← false
15:     else
16:         searchLevels ← true
17:     end if
18:     if searchLevels then
19:         copy levels to localLevelsCopy
20:         for all levels in localLevelsCopy do
21:             lock level with shared lock
22:             for all runs in level do
23:                 Enqueue task for searching in the run
24:                 Each run is searched for the start key with binary search
25:                 If the key is not found, the next largest key is returned
26:                 The run is searched linearly until the end key
27:                 The range is returned as a vector
28:             end for
29:         end for
30:         Wait for tasks to finish and add results to pq
31:     end if
32:     Merge sorted key-value pairs using pq
33:     Remove TOMBSTONES from rangeResult
34:     Update range hits and misses
35:     return rangeResult
36: end procedure
```

## Time Complexity Analysis

- **Best case:** $O(\log(B) + K)$ – The entire range is found in the buffer (memtable), where B is the buffer size and K is the number of keys within the range.
- **Worst case:** $O(\log(B) + L * R * (K + log(P)))$ – The entire range is distributed across all runs within all levels, where $L$ is the number of levels, $R$ is the maximum number of runs per level, and $P$ is the number of pages per run.
- **Average case:** $O(\log(B) + L * R * (K + \log(P)))$ – The range could be found anywhere within the levels and runs, or not found at all. On average, the time complexity would be similar to the worst case.

# B APPENDIX: CLASS DIAGRAMS

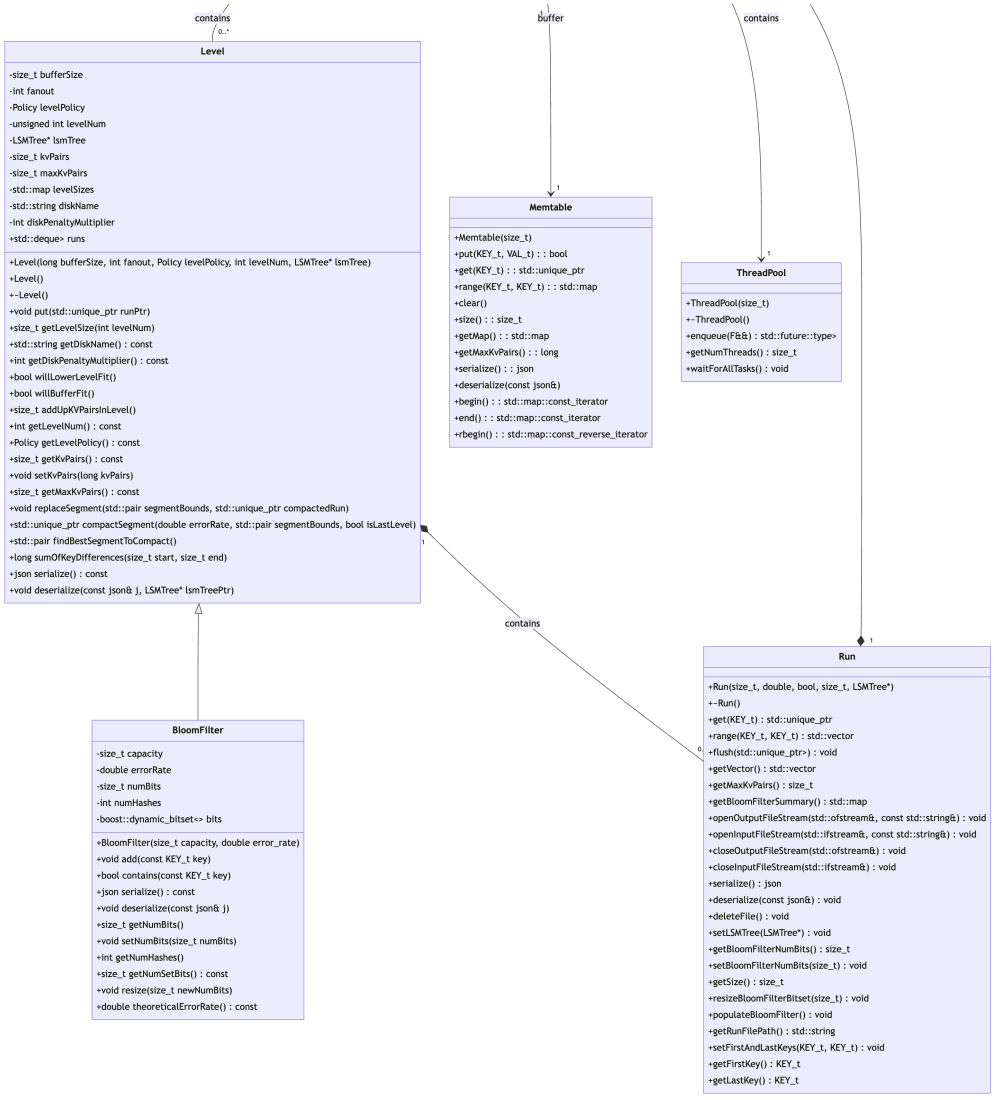


Fig. 7. Class diagram showing the Level, Memtable, BloomFilter, ThreadPool, and Run classes. All the classes have serialize and deserialize methods that roll up to a main function in the LSMTree main class.

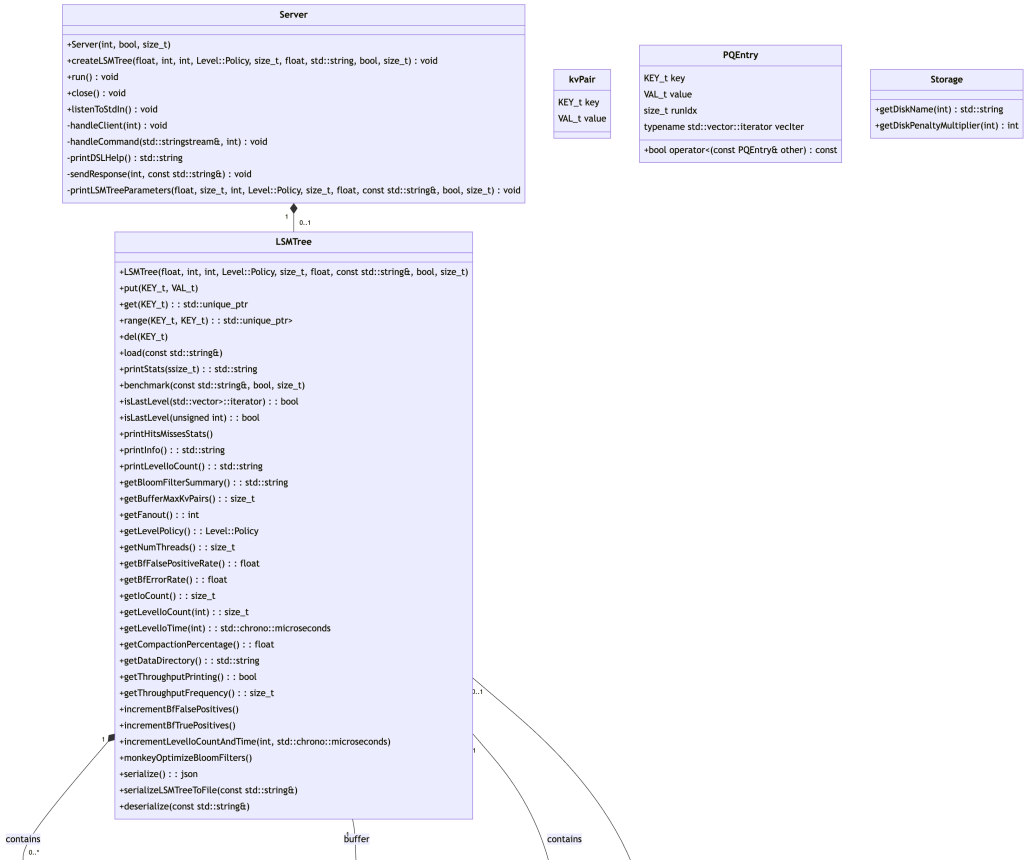


Fig. 8. Class diagram of the system showing the LSMTree, Server, kvPair, PQEntry, and Storage classes.