CSC443/CSC2525H

Database System Technology Project Report

Weizhou Wang - 1004421262
Dechen Han - 1006095636
Shi Tang - 1005930619

Project Status

All the required features, as well as bonus features (such as Handling Sequential Flooding, Dostoevsky, Min-heap, Blocked Bloom Filters, and Monkey), have been implemented and thoroughly tested. Additionally, we have successfully run benchmarks with 1GB of data. The details of these implementations will be discussed in the `Description of Design Elements` section.

Testing

We created both a unit test for functionality assurance and a benchmark test for performance evaluation. The unit test consists of two parts: small workload and big workload. The small workload part tests the basic functionality of the system on ‘put’, ‘get’, and ‘scan’ under memtables of size less than 10 entries, which serves as a sanity check during new-feature implementations. On the other hand, the big workload test consists of a 32MB data volume, together with another 1MB updates and 2MB deletions. The updates and deletions are applied to both memtable and SSTs, and we carefully designed the deletion operations so that the tombstones should reach both lower and higher levels of the LSM tree.

We also allocate memories to record all inserted and deleted key-value pairs to check the correctness of later-on ‘get’ and ‘scan’ test cases. For the ‘get’ test on a big workload, we tested the point queries on (1) entries on memtable, (2) entries on SSTs, and (3) entries that do not exist or have been deleted. Meanwhile, for the ‘scan’ test, we test the range queries over (1) both key1 and key2 are within the bounds of entries in the DB and (2) some of key1 or key2 are out of bounds of the DB’s data. Moreover, we also designed a test case to test the correctness of the Sequential Flooding feature using a long-range query.

Lastly, another test case tests the correctness of “openDB” and “closeDB” by first closing the current DB followed by an open operation. Then, we issue point queries to test if the entries that were originally in the DB are still reachable and if the values are up-to-date.

For the benchmarking performance test, please see the Experiment sections in each step.

Compilation & Running Instructions

Note: Please ensure the `data/` folder exists before starting any experiment (you can easily create this folder by compiling the codebase using either `make test` or `make db`)
Note: Please remember to issue `make clean` between different builds and experiments to clean up the data/ folder and to use different optimization flags.

1. **Codebase structure (the `main` branch contains our final codebase)**
   include/: Stores all .h files
   src/: Stores all .cpp files
   bin/: Stores executables after compilation
   data/: Stores SSTs and Bloom Filters (**please make sure this folder exists before running any executables**)

2. **Unit tests**
   Please use the `make test` command followed by `./bin/tests`.
   For convenience, we saved the executables for the unit tests for step 3 under CSC443-FinalProject/: `./unittest_step3`

3. **Benchmark performance tests**
   We saved the executables for the benchmark tests for step 3 under CSC443-FinalProject/:
   - To run benchmark with Dostoevsky enabled (T=4): `./benchmark_step3 <path_to_csv>.csv`
   - To run benchmark with Basic LSM Tree (T=2): `./benchmark_step3_without_Dostoevsky <path_to_csv>.csv`

   For other steps:
   For the step1: please first check out to `**step1_benchmark**` branch
   For the step2: please check out to `**step2_benchmark**` branch
   For the step3: please use the `**main**` branch (this is our final codebase)
   On each branch, please use the `make db` command followed by `./bin/db <path_to_csv>.csv` to execute benchmarks.

   Since we have implemented both Monkey and Dostoevsky, our benchmark script is initially configured to run with an LSM-Tree size ratio (T) of 4, in order to support Dostoevsky. To execute the Benchmark without using Dostoevsky, please modify the following parameters in the `constants.h` file under the main branch:
   LSMT_SIZE_RATIO = 2; LAST_LEVEL_M = 6.235; LOGT_LN2 = 1.443

   Note: For all experiment plots, we have converted the results to log-scale, for easier identification of patterns.

Description of Design Elements

Step 1 - Creating a Memtable and SSTs

1. KV-store get API

The KV-store get API is implemented in **database.cpp:get()**. This function initially searches for the key in the Memtable using **rbtree.cpp:get()**. If the key is not found in Memtable, the search proceeds over all SSTs, in the order from the youngest to the oldest. In step 1, we implemented a binary search algorithm for searching over the SSTs (**LSMTree.cpp: search_SST_Binary()**).

2. KV-store put API

We implemented the KV-store put API in **database.cpp:put()**. This function accepts a key-value pair. If the Memtable is not full, the pair is directly inserted into the in-memory Memtable by calling **rbtree.cpp:put()**. Otherwise, if the Memtable is full, we flush its contents into an SST (Sorted String Table) by calling **database.cpp:writeToSST()**, then clear the Memtable, and lastly add the key-value pair to the empty Memtable.

3. KV-store scan API

The KV-store scan API is implemented in **database.cpp:scan()**. This function returns a pointer to an array containing KV-pairs that are within the range, from key1 to key2. We achieve this in two steps. First, the Memtable is scanned by calling **rbtree.cpp:scan()**, and the results are stored in an array *sorted_KV* passed by reference. Then, all SSTs from youngest to oldest are scanned by calling **LSMTree.cpp:scan()**.

While scanning each SST, *sorted_KV* maintains two consecutively sorted ranges, separated by *first_end*. The first part contains sorted pairs from previous runs, while the second part contains sorted pairs from the current SST. To merge the results, we call **LSMTree.cpp:merge_scan_results()** function. This merge-sort helper function merges the paris, resulting in a sorted array stored back in *sorted_KV*. We will iteratively perform the scan for each SST until we reach the oldest SST. At that point, *sorted_KV* will contain the final sorted result.

Extra Feature: We've implemented the Bloom filter in Step 3, however, it's less effective during scan operations. To address this, we stored the metadata of the SST in the file name, following the format <LSMTree_level>_<Time at creation><Clock at creation>_<min key in SST>_<max key in SST>_<offset of B-Tree Root>.bytes. Therefore, we can efficiently use the min_key and max_key for scanning. This allows us to quickly determine whether a given SST falls within the range. If it does not, we can simply skip it and proceed to the next SST. We will discuss the file name structure of SSTs in more detail in the section '#10 Static B-tree for SSTs'.
We've also implemented an algorithm to handle sequential flooding during scan, and we will discuss more on this later.

4. In-memory Memtable as balanced binary tree

We implemented the in-memory Memtable as a Red-Black tree, since Red-Black trees provide a more balanced tree with fewer rotations during insertion and deletion, which can make these operations faster in practice. The implementation of the Memtable can be found in **rbtree.h.** When we reach the maximum size of Memtable, we will traverse it based on the order of keys and output all the key-value pairs to a sorted file, then save it in the storage device.

5. SSTs in storage with binary search

The binary search on SST is implemented in **LSMTree.cpp:search_SST_Binary()**. In Step 1, each iteration of the binary search requires issuing one I/O operation to read the page from the SST, with the data being cached in the system's OS cache. By integrating a Bufferpool in Step 2, we can enable direct I/O. For every page we read, it is stored in our bufferpool, thereby avoiding extra I/O if subsequent iterations occur within the same page. We will discuss more on this in the section `#8 Integration buffer pool with queries`.

6. Database open and close API

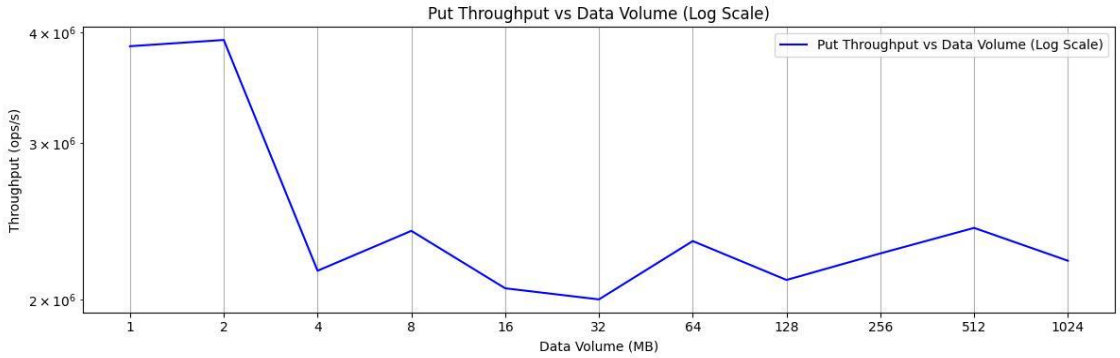
The database open and close APIs are implemented in **database.cpp:openDB()** and **database.cpp:closeDB()** respectively.

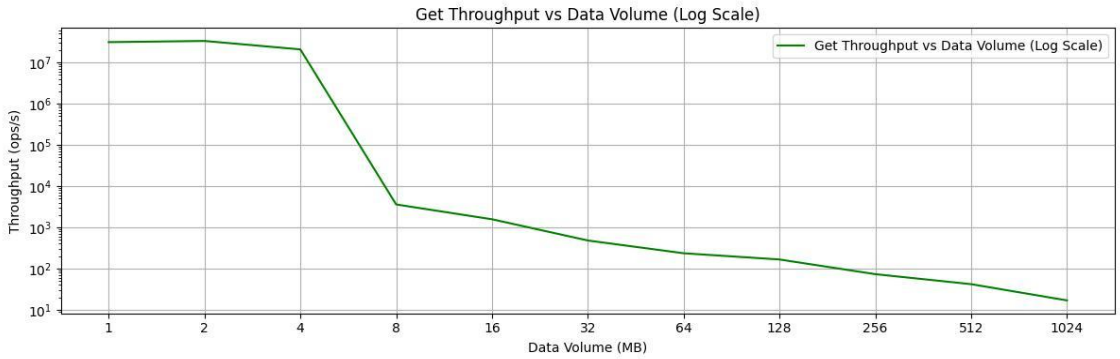
When closing a database, if the Memtable is not empty at this point, we flush it into a new SST file and free-allocated objects to avoid memory leaks. Additionally, given our implementation of Dostoevsky in step 3, and the fact that we include <LSMTree_level> in the SST filename, there's a special consideration: we need to record the size of the largest level by renaming the file of the oldest SST (largest level in Dostoevsky). Having a record of the size is important since it only has one big contiguous SST in Dostoevsky of the largest level, and the size is needed to determine whether it has reached its capacity. We will discuss it further in the Dostoevsky section.

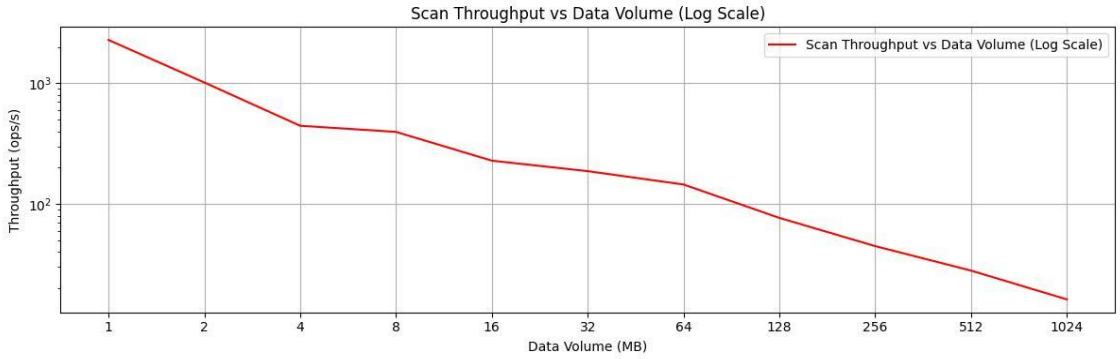
When opening a database, we need to pass in the database's name, *db_name*. We first check if a directory with *db_name* already exists. If it does not, we create a new directory under the name. If it does exist, we need to restore the sorted list of existing SST files, and also record the size of the largest level SST discussed before, lastly revert it to the original filename format.

Experiments - Step 1

In the Step1 performance experiment, we used a 1MB Memtable. For data volumes ranging from 1 MB to 1024 MB, we began by populating the database with the necessary data using `Put` operations and then performed 500 `Get` and `Scan` operations each. We timed these operations and calculated their throughput as operations per second by dividing the total number of operations by the time taken to complete them. The results were plotted on a logarithmic scale to capture the performance across a broad range of data sizes. (Note: as suggested on piazza @180, we only focused on short scans of about one page big)







Put: The put throughputs stay similar at 1MB and 2MB and both of them are high, which can be attributed to the lack of writing into SST at 1 MB data volume and only a single SST file being written at 2 MB, resulting in zero or only a few IOs performed. However, the Put operation shows a significant drop in throughput as data volume expands from 2 MB to 4 MB. This is because, as the data volume grows exponentially, the number of SST files that need to be written increases, leading to more I/O that results in slower throughput rates. Moreover, since our SSTs are uniformly sized at 1MB, it implies that the cost of writing each SST file should be consistent and result in a more stable throughput for Put operations. Then the observed fluctuations in Put throughput between 4 MB to 1 GB, may be attributed to variability or instability in the testing environment.

Get: Initially, the Get operation maintains a relatively steady decreased throughput as the data volume increases from 1 MB to 4 MB. At 8 MB, there is a sharp decline in throughput, which shows a similar pattern as observed in the Put operation. After that, the get throughput gradually decreases as the data volume continues to increase up to 1024 MB. This is because, when the data volume increases exponentially, the Get operation may need to access more SST files to get a certain value. This increased I/O requirement leads to a reduction in throughput rates.

Scan: The throughput for Scan operations decreases almost linearly on the logarithmic scale with increasing data volume. As data volume grows exponentially and results in additional SST files, the Scan operation has to traverse more of these SSTs and increase I/O rapidly. Consequently, it contributes to a decrease in throughput rates.

7. Implementation of Buffer pool as hash table with collision resolution

The buffer pool's implementation is detailed in **bufferpool.h.** Pages are uniquely identified by concatenating the file name with the page's offset within that file. Our bufferpool is structured as a hash table, mapping each page's unique ID to the corresponding 4KB page. By integrating bufferpool, we can enable direct read I/O operation. Before issuing any read to storage, our system will first check if the given page is in the buffer pool. If so, it will skip issuing the I/O.

We used MurmurHash as our hashing algorithm, and the public source code is obtained from: https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp. MurmurHash is preferred over alternatives like xxHash due to its balance of efficiency and uniform distribution of hash values. This balance is critical in reducing collisions and ensuring quick access within our buffer pool.

We employ chaining to resolve hash collisions, which helps in avoiding cache collusion. To ensure accuracy, the page ID is stored alongside the actual 4KB page to make sure to return the correct page in case of hash collisions.

8. Integration buffer pool with queries

We created a wrapper function, **LSMTree.cpp:read()**, to incorporate the buffer pool by first checking if the target page is already in the buffer pool before retrieving it from storage. Once a page is retrieved from storage, it is then stored in the buffer pool. This function has replaced the original pread() in both search and scan operations, as seen in **LSMTree.cpp:search_SST_BTree()** and **LSMTree.cpp:scan_SST()**.

9. Clock eviction policy

The buffer pool implements the Clock eviction policy (see **bufferpool.cpp:evict_clock()**). This choice is slightly better over the traditional LRU eviction policy because Clock eviction approximates LRU's benefits while being more efficient in terms of computation and resource usage.

10. Static B-tree for SSTs

An SST will be created when DB flushes the memtable to storage. At that time, the DB will scan the memtable and store all the elements into a sorted key-value vector (see **database.cpp:writeToSST()**). With this sorted vector, we will treat it as the leaf level of the B-Tree and build up the non-leaf levels based on it.

The B-Tree construction is achieved in **BTree.cpp:convertToBTree()**. First, we pad the sorted vector to become a multiple of pages (for direct I/O). Then, we iterate each element in the vector and find all elements that should appear on the non-leaf levels. This is achieved by counting the index of the element and checking if the index equals the last element in a leaf node.

For each of these non-leaf elements, we send it to a recursion function at **BTree.cpp: insertHelper()**. The purpose of this function is to assign each of these non-leaf elements to their corresponding levels and nodes on the B-Tree. The logic in this function is similar to the one used in node splitting of B-Tree insertion, such that when a node exceeds its capacity, the node will be split into half-and-half by its middle point, whose index can be calculated as 0.5*(KEYS_PER_NODE+1). Then we set a counter for each level of the B-Tree and increment them whenever an element is inserted into a level. Whenever the counter equals the middle point, we know that the next element inserted will be a middle point in a split and we will send it to the next level using recursion. After that, we will allocate a new node on the current level and all the following elements will be inserted into this new node.

There is a corner case in this algorithm, such that we should not escalate the very last element on each level even if its index equals the middle point because the size of the last node is still smaller than a page in this case. To achieve this, we calculate the final accumulated value of the counter at each level beforehand and skip the element if its index equals final_counter - 1. This accumulated value is calculated with: (please check **max_size** variable in **insertHelper()**)

$$maxcount_i = \frac{maxcount_{i+1} - 1}{middlepoint + 1},\ for\ the\ ith\ level$$

After all elements in each non-leaf node are assigned, we iterate these elements one by one to calculate the pointer (offset) values in the node (at **BTree.cpp:insertFixUp()**). Each non-leaf node has a structure of "|...keys...|...offsets....|# of keys|".

Lastly, we write the leaves into the SST file followed by all non-leaf levels (**BTree.cpp: write_non_leaf_nodes_to_storage()**) (this is convenient for LSM-Tree compaction, more on this later). Thus, each SST (with B-Tree) has a structure of "|....sorted_KV (as leaf)....|root|..next level..|...next level...|".

During LSM-Tree compaction, since we use an output buffer to write to the SST page-by-page, we lost the information of having a whole sorted key-value vector (this is also why we store leaves first, and append non-leaf nodes later). As a result, in **LSMTree.cpp: merge_down_helper()**, we allocate a **non_leaf_keys** variable to store all keys that belong to the non-leaf levels. And we iterate each of these keys one by one to feed to the **insertHelper()** to build up the B-Tree (see **BTree.cpp: void convertToBTree()**).

Moreover, when compaction is in progress, all intermediate SSTs would only store the sorted vector without the B-Tree because we know that these SSTs will be compacted further. This mechanism reduces the time spent on B-Tree construction and thus improves performance. This is achieved by setting the **last_compaction** variable in **LSMTree.cpp: merge_down()**.

10.1 B-Tree search in get() and scan() operations

The point query on each SST is implemented in **LSMTree.cpp: search_SST()**. It keeps both the original binary search (**search_SST_Binary()**) and B-Tree search (**search_SST_BTree()**). For the B-Tree search, it first uses **BTree.cpp: search_BTree_non_leaf_nodes()** to search the non-leaf nodes

level-by-level. On each level, it reads a whole node into memory, and performs a binary search within the node to find the file offset to the children. Finally, the offset to the corresponding leaf node is returned, then **search_SST_BTree()** reads in the leaf and performs a binary search to locate the target key-value pair in the node.

For the scan() operation on an SST, similarly, **LSMTree.cpp: scan_helper_BTree()** first uses **BTree.cpp: search_BTree_non_leaf_nodes()** to find the file offset to the node containing the lower bound of the scan range (key1) and locate the minimum qualified key in the node. Then, in **LSMTree.cpp: scan_SST()**, it iterates all the following elements in the following pages until exceeding the upper bound (key2).

BONUS: Handling Sequential Flooding

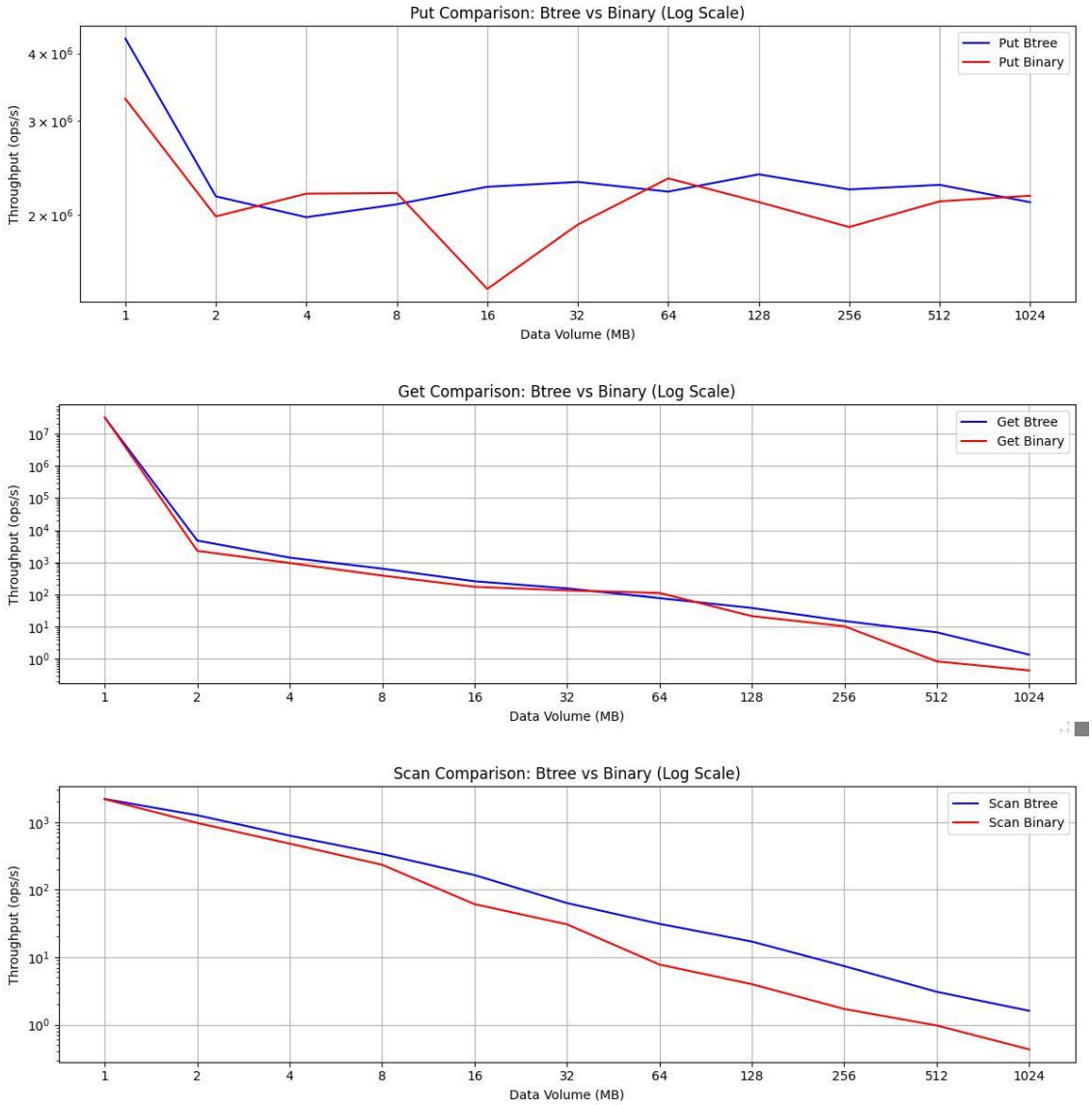
When using LRU or Clock as in this project, a sequential flooding issue happens when a scan operation has a big range, such that most of the buckets in the buffer pool will be filled up by the pages spanned by this scan. However, not all spanned pages are hot, which results in many of the buffer pool buckets storing cold pages. This issue gets worse when a scan range is even larger than the total page number in the buffer pool, and the same scan is performed repeatedly. In this case, pages in the buffer pool are constantly refreshed by newly accessed pages, but there is no buffer hit because the next accessing page will always just be evicted.

To prevent this from happening, we set a counter for each scan operation (**scanPageCount** in **LSMTree.cpp:scan()**). The counter accumulates the number of B-Tree leaf pages the scan has accessed among all SSTs so far, and compares that with a SEQUENTIAL_FLOODING_LIMIT in **LSMTree.cpp: read()**. Once the counter exceeds the limit, all the following page accesses to the leaf nodes will not be saved to the buffer pool anymore (this method is mentioned in Piazza @90). However, we still buffer the accesses of non-leaf nodes to the buffer pool because the B-Tree is very shallow, and so we expect them to be hotter.

We choose the SEQUENTIAL_FLOODING_LIMIT to be 1000 pages, which takes up 40% of the 10MB buffer pool capacity (i.e. once a scan occupies 40% of the buffer pool, we disable the buffer pool for all following reads, so that at least 60% of the buffer pool is still functional).

Experiments - Step 2

In the Step2 performance experiment, we used a 1MB Memtable and 10MB buffer pool. For data volumes ranging from 1 MB to 1024 MB, we began by populating the database with the uniformly randomly distributed data using `Put` operations and then performed 1000 `Get` and `Scan` operations each. We timed these operations and calculated their throughput as operations per second by dividing the total number of operations by the time taken to complete them. The results were plotted on a logarithmic scale to capture the performance across a broad range of data sizes.



The plots indicate that using a B-tree search generally outperforms binary search on Get and Scan. Given that the data is presented on a log scale, the actual performance gap between B-tree and binary search is more substantial than what is visually drawn in the plots.

Put: The performance of Put operations is not influenced by B-tree or binary search, hence comparing them is not applicable. Notably, there is a substantial decrease in Put throughput when the data volume exceeds 2 MB, which corresponds to the point at which data begins to be written to SST files. As the

volume of data increases, more SST files are required to be written and more B-Trees are constructed, amplifying both CPU and I/O and consequently reducing throughput. Particularly, a marked decline in throughput is observed when the data volume reaches 16 MB with the use of binary search. This specific drop could be attributed to performance fluctuations within the teaching lab.

Get: The Get operation performance, when using both B-tree search and binary search, follows a similar pattern, with a marked decline as the involvement of SST files increases. Generally, when dealing with large datasets that require frequent I/O operations, B-Trees significantly perform better than binary search. This is because the B-Trees structure is optimized to minimize disk reads and writes. Unlike binary search, which operates efficiently on in-memory data but becomes less effective as the dataset size exceeds memory capacity, B-Trees are designed to handle large volumes of data residing on disk with fewer I/O operations. Each node in a B-Tree can hold multiple keys, allowing the tree to maintain a shallow depth even for substantial datasets, thereby reducing the number of disk accesses required for any given search. Furthermore, the integration of the buffer pool enhances this efficiency by reducing the need for disk access. This caching mechanism is particularly effective for B-Trees, as their structure often leads to repeated access to certain nodes. In contrast, a binary search on a sorted array, while time-efficient in theory, will suffer from the overhead of loading large data segments from disk into memory for each search operation. Therefore, the B-tree search shows better performance over binary search, with the greatest difference at data volumes of 512MB and 1024MB. Note that, although the Binary-search strangely outperforms B-Tree search at 64MB, it is acceptable, as there might be performance fluctuations with the teaching lab.

Scan: The performance difference in Scan operations between using B-trees and binary search is more noticeable than with Get operations, and this difference increases as more data is populated. As mentioned above, for larger datasets, particularly those requiring disk access, B-Tree is generally faster and more efficient due to its structure that optimizes disk I/O operations. Also, rather than stopping at the first key found, we need to access all of the SSTs from storage and merge their results together to construct the final scan result. This requires even more pages to be read and gives the buffer pool much more pressure for binary search. As a result, B-Tree is more efficient than binary search, since binary search would require repeated searches to access multiple contiguous items, leading to more I/O operations and hence slower throughput.

11. Bloom filter for SST and integration with get

A Bloom Filter is created whenever we build up a B-Tree for an SST. The filter consists of a bitmap, whose size is related to the number of entries on the B-Tree leaves. For each element on the leaves, we set the corresponding bits in the filter from 0 to 1 using several murmur hashes with different seeds (M*ln2 hashes, the choice of M will be discussed later in Monkey).

The implementation of the Bloom Filter class can be found in **bloomFilter.h**, more specifically, a Bloom Filter is created only when 1. The DB flushes the memtable to SST (**database.cpp: writeToSST()**); and 2. After LSM-Tree compaction (**LSMTree.cpp:merge_down_helper()**).

During a get() operation, before LSMTree performs a search in the SST, it first probes the SST's Bloom Filter. If any of the hashes returns a negative, the LSMTree will skip the SST because it is sure that the key is not in the SST. (The probing is implemented in **LSMTree.cpp:check_bloomFilter()** which is called at **LSMTree.cpp:LSMTree::get()** to probe) (The details of implementation will be described in the next sections)

12. Persisting Bloom filters in SSTs

After all elements are set in the filter, the filter will be written to a file in the storage (implemented in **bloomFilter.cpp: writeToStorage()**). The data file is stored under **data/<db_name>/filter/**, and has the same file name as its corresponding SST (this was confirmed in Piazza @177), which makes the db easier to correlate SSTs and Bloom Filters. In the file, the filter begins at offset 0, and there is no other metadata stored in the file because we can infer the number of entries by the level information of the SST (please see **LSMTree.cpp:calculate_sst_size()** for details).

In **LSMTree.cpp:check_bloomFilter()**, after a hash value is calculated, the function infers the corresponding page index of the hash value (the bit) using bit-shifts, and reads the corresponding page from either the buffer pool or storage using **LSMTree.cpp:read()**. If the read results in a buffer miss, then the page will be added to the buffer pool. Please see the next section for how the probing works.

BONUS: Blocked Bloom filters

The core idea is to split the Bloom Filter into multiple cache lines, and the corresponding bits of each key are all in the same cache line, which improves locality and reduces cache misses (so improves performance). When creating the Bloom Filter (**bloomFilter.h: BloomFilter()**), the bitmap is created as an array of bitmaps, such that each bitmap has the size of a cache line (64 Bytes, verified on teaching lab using "getconf -a | grep CACHE"). When setting the bits (in **bloomFilter.cpp: set()**), we added one more hash to decide the index of the cache line bitmap that the key belongs to. Then all the following hashes return an index within the same bitmap and we toggle the corresponding bits to 1.

When probing the filter, in **LSMTree.cpp:check_bloomFilter()**, we use the first hash to calculate the index of the cache line bitmap, from which we can infer its corresponding page index (because all

bitmaps are stored contiguously). Then we read the page and find the corresponding bitmap (at this stage the bitmap should be loaded on a cache line). We use the hashes to probe the key in the bitmap and return false as long as any hash results in a negative in the Bloom Filter. If all hashes hit, then the function returns a true, meaning that the entry may be in the SST.

BONUS: Monkey

The basic idea of Monkey is to find optimal assignments for the False Positive Rate (FPR) on each level of the LSM-Tree that optimizes the memory consumption of the Bloom Filters, given a value of the worst-case expected I/O for an unexisting key look-up.

To achieve this, we used the formula provided in the Monkey paper:

Leveling

$$p_i = \begin{cases} 1, & \text{if } i > L_{filtered} \\ \frac{(R - L_{unfiltered}) \cdot (T-1)}{T^{L_{filtered}+1-i}}, & \text{else} \end{cases}$$

$$\text{for } 0 < R \leq L$$

$$\text{and } 1 \leq i \leq L$$

$$\text{and } L_{filtered} = L - \max(0, \lfloor R - 1 \rfloor) \quad (5)$$

Tiering

$$p_i = \begin{cases} 1, & \text{if } i > L_{filtered} \\ \frac{R - L_{unfiltered} \cdot (T-1)}{T^{L_{filtered}+1-i}}, & \text{else} \end{cases}$$

$$\text{for } 0 < R \leq L \cdot (T-1)$$

$$\text{and } 1 \leq i \leq L$$

$$\text{and } L_{filtered} = L - \max(0, \lfloor \frac{R-1}{T-1} \rfloor) \quad (6)$$

, where p_i is the FPR on level i, R is the worst-case expected I/O, T is the LSM-Tree size ratio, and $L_{filtered} = L$ is the total number of levels that have bloom filters, and $L_{unfiltered}$ is the number of the levels that do not have bloom filters (which is 0 in our project).

We also use Dostoevsky, which uses Leveling on the last level and Tiering on all other levels. So we will use equation (5) on the last level, and (6) on other levels, and the final version of the formula becomes:

- $p_L = \frac{R(T-1)}{T}$ (on the last level ($L \neq i$)), and $p_i = \frac{R}{T^{L+1-i}}$ (on the ith level)

With this said, we can calculate the number of bits per entry on each level:

- On the last level: $2^{-M_L ln(2)} = p_L = \frac{R(T-1)}{T} \Rightarrow M_L = \frac{logT - logR - log(T-1)}{ln2}$
- On the ith level ($i \neq L$): $2^{-M_i ln(2)} = p_i = \frac{R}{T^{L+1-i}} \Rightarrow M_i = \frac{(L+1-i)logT - logR}{ln2}$

Under an assumption of $R = 0.1$ (as mentioned in Piazza @240) and T = 4:

$M_L = \frac{2 - log0.1 - log3}{ln2} \approx 5.3913$

$M_i = \frac{2(L+1-i) - log0.1}{ln2} \approx 2.885(L + 1 - i) + 4.7925$

The implementation can be found at **bloomFilter.h: calculate_num_bits_per_entry()**, which is called in **bloomFilter.h: BloomFilter()** and **LSMTree.cpp: check_bloomFilter()** to get the up-to-date number of bits per entry when creating or probing the filter.

BONUS: Dostoevsky

We have expanded our Basic LSM-tree implementation to incorporate the Dostoevsky compaction policy with a size ratio of 4. So the LSM tree will maintain a single sorted run at the largest level and use tiering at all smaller levels. To achieve this, we added a boolean variable in **LSMTree.h** for the level to indicate if it is the largest level, along with a variable to track the current size for compaction.

BONUS: Min-Heap

With the implementation of the Dostoevsky policy, to make compaction across multiple SSTs more efficient, we used a min-heap from the standard library priority_queue. We have a HeapNode that stores the KV pair and an index identifying which SST contains this pair. This is important for later compaction stages, as it ensures we only merge the pairs from the newest SST file. Additionally, we implemented a custom comparator to create a min-heap, which prioritizes newer files in case of duplicates. This implementation is also in **LSMTree.h.**

13. Compaction/Merge of two SSTs

The compaction algorithm can be found in **add_SST(), merge_down(), largest_level_move_down() and merge_down_helper()** in **LSMTree.cpp.**

When each SST is added to the LSMTree, we call **add_SST().** In this function, the newly added SST is appended to the first level. Under the Dostoevsky compaction policy, if the first level is the largest and the number of SSTs is less than the size ratio of 4, we need to compact it into one large SST by calling **merge_down_helper()**. If the number of SSTs in the first level equals the LSMTree size ratio, we merge the SSTs from the first level down to the next by calling **merge_down().**

The function **merge_down()** merges the SSTs from the current level to the next, which might trigger further compaction at the subsequent level as mentioned above in section `#10. Static B-tree for SSTs`. Hence, we check if the current level compaction is the final compaction in the process, if so, we pass `*last_compaction = true*` when calling **merge_down_helper().** This is to indicate that current compaction is the last compaction, and hence construct the BTree and Bloom filter for it, which avoids unnecessary BTree and filter construction during the entire process.

merge_down_helper() is where the actual compaction from the current level to the next level takes place. We allocate four input buffers for the SSTs being merged and one output buffer for the new SST file. To efficiently handle data from multiple SSTs, a min-heap is used. We initialize the min-heap with the first element from each SST and, while the min-heap is not empty, we pop the smallest element and append it to the output buffer. We then push a new KV pair from the same SST to the min-heap. If the output buffer becomes full, we write its contents to the new SST file and clear the buffer for the next

iteration. After compaction, we determine the min and max keys for the new file and generate its SST filename accordingly.

14. Support update

To update a key with a new value, we simply call **database.cpp:put()** with the key and the new value. If the key already exists in the in-memory Memtable, we update its value. Otherwise, we add the new KV pair to the Memtable. Once the Memtable is flushed to storage, it eventually triggers a compaction process. During compaction, if we encounter entries with the same key across different SSTs being merged, we keep only the more recent version, and supersede the older ones (see **LSMTree.cpp:merge_down_helper()**). To achieve this, we have implemented a custom comparator for creating a min-heap in **LSMTree.h**. This comparator prioritizes the entries from the newer file in cases of duplicated keys. Therefore, during get and scan operations, we can ensure that the most recent version of each key is returned.

15. Support delete

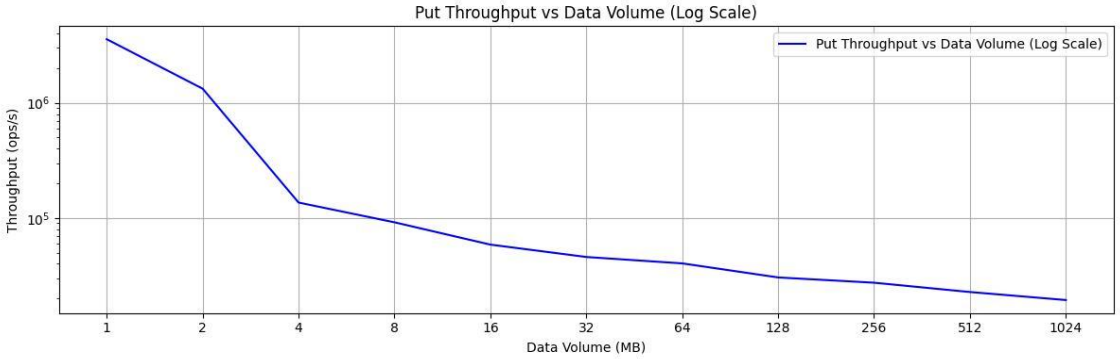
We have also added support for deletes. For the tombstone, we use a minimum 8-byte integer, specifically numeric_limits<int64_t>::min(). To delete a key, we call **database.cpp:del()**, which in turn calls the function **database.cpp:put()** to put the KV-pair with the value as a tombstone. After this KV-pair is flushed to disk, and when it reaches the largest level of the LSM tree, the key marked with the tombstone is discarded during compaction since there are no longer any older versions of the entry in existence. This implementation can be found in **LSMTree.cpp:merge_down_helper()**.

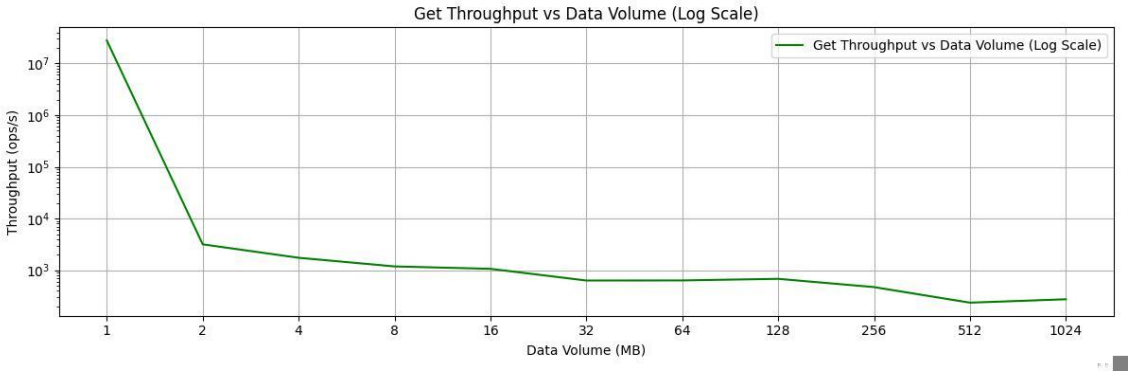
Experiments - Step 3

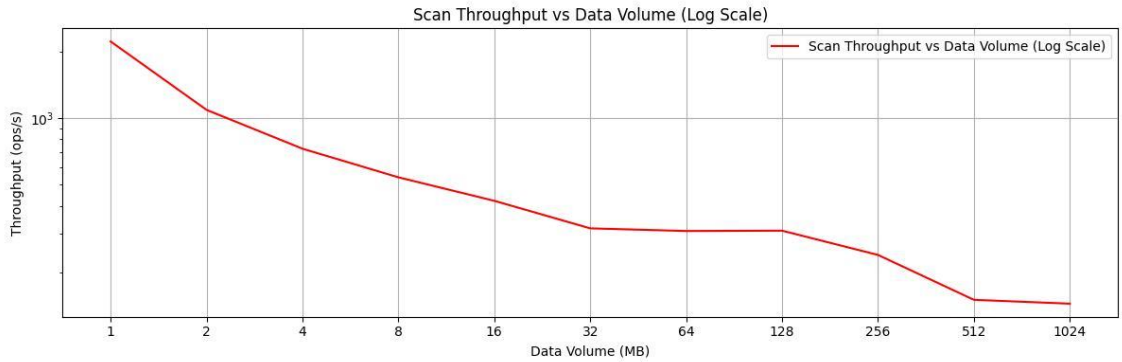
LSM-tree with Monkey (Size Ratio T = 2, Basic LSM-Tree)

In the Step 3 performance experiment, we used a 1MB memtable and 10MB buffer pool. Also, as we implemented Monkey, the Bloom filters use a different number of bits per entry on each level.

For data volumes ranging from 1 MB to 1024 MB, we began by populating the database with the uniformly randomly distributed data using `Put` operations and then performed 1000 `Get` and `Scan` operations each. We timed these operations and calculated their throughput as operations per second by dividing the total number of operations by the time taken to complete them. The results were plotted on a logarithmic scale to capture the performance across a broad range of data sizes.







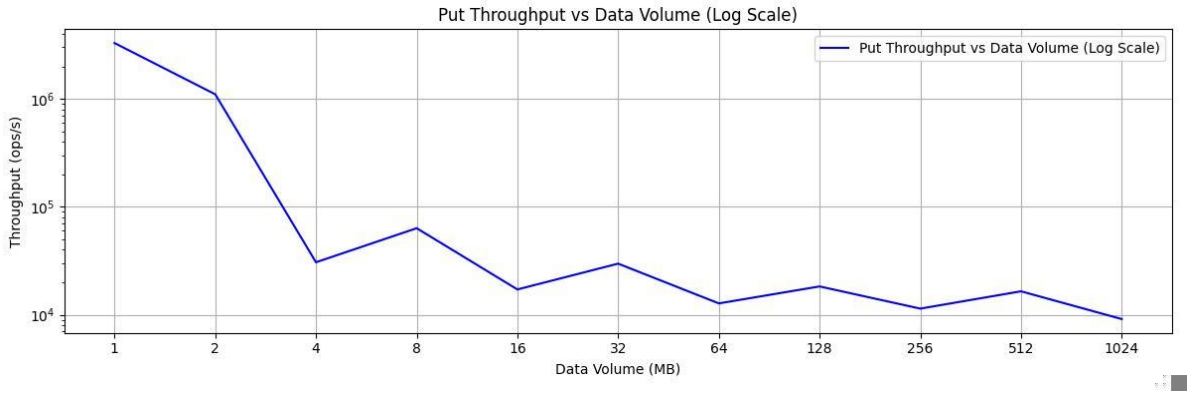
From experiments in step 2, the throughput for both `Get` and `Scan` operations decreases to below 10^0 ops/s when the data volume reaches 1024MB. However, in step 3, the throughput for these operations remains just below 10^3 ops/s, even as the data volume increases to 1024MB. This observation leads us to conclude that the implementation of the LSM-Tree, as well as the extra features, offer a significant performance enhancement for both `Get` and `Scan` operations.

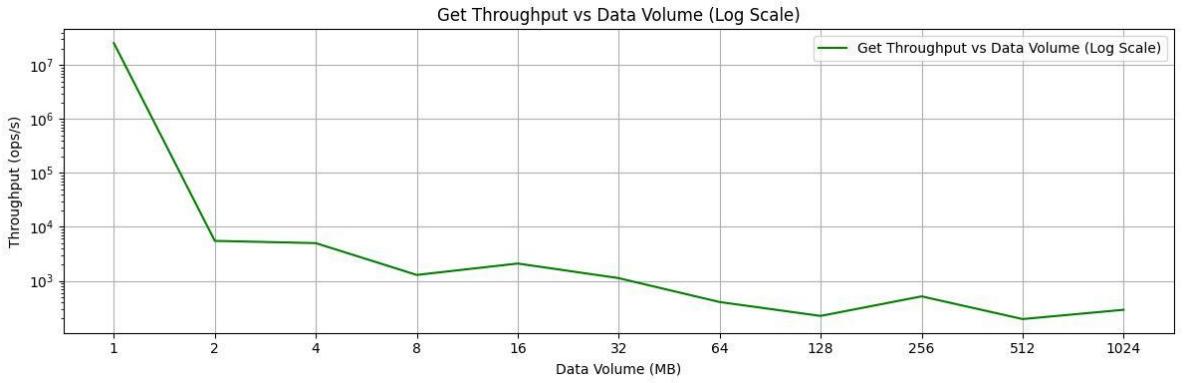
Put: Similar to previous steps, we can see a significant reduction in throughput when the operation starts writing into SST files. However, beyond this point, the put throughputs are no longer steady, while decreasing as data volume increases. This is because as data volume goes up, the LSM tree becomes deeper thus more compactions are executed, resulting in more I/Os performed. Furthermore, the overall throughput in this step is lower than in Step 2, which can be attributed to the extra workload of compaction that arises in an LSM-Tree structure whenever there are two SSTs at the same level.

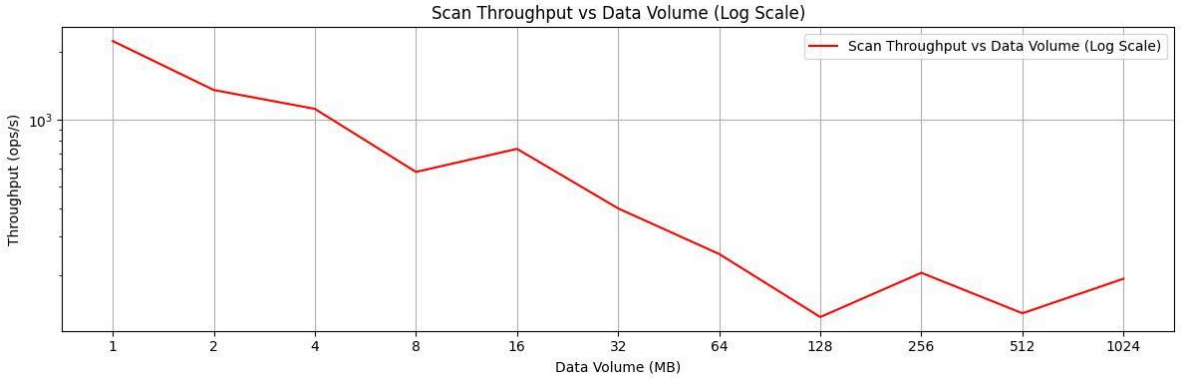
Get: The Get operation performance still follows a similar pattern: there is a marked decline as the involvement of SST files increases. But in comparison to step 2, as the data volume increases to 1024MB, the `Get` operation shows better and more stable performance. One key factor contributing to this enhancement is likely the integration of the Bloom Filter, which efficiently determines whether the target key is in a certain file, thereby reducing unnecessary I/Os. Additionally, the implementation of compaction, which merges SST files, plays a significant role. This compaction process reduces the number of SSTs. Also since the B-Tree has a huge fan-out, it can still keep the number of SST levels low. This prevents the increase in the number of non-leaf nodes that need to be searched. Thereby the system requires much less I/Os. As a result, the overall efficiency and stability of the `Get` operation are enhanced as the data volume grows.

Scan: Like the `Get` operation, `Scan` also exhibits enhanced performance compared to step 2. Even when the data volume increases to 1024MB, the throughput can stay within the range of 10^2 to 10^3 ops/s. This improvement can be attributed to factors similar to those benefiting the `Get` operation: the application of the Bloom Filter and Compaction Policy, both contribute to this increased efficiency. These strategies lead to fewer I/Os required and thus a faster overall performance for the `Scan` operation with large data volumes.

LSM-tree with Dostoevsky and Monkey (Size Ratio T = 4)

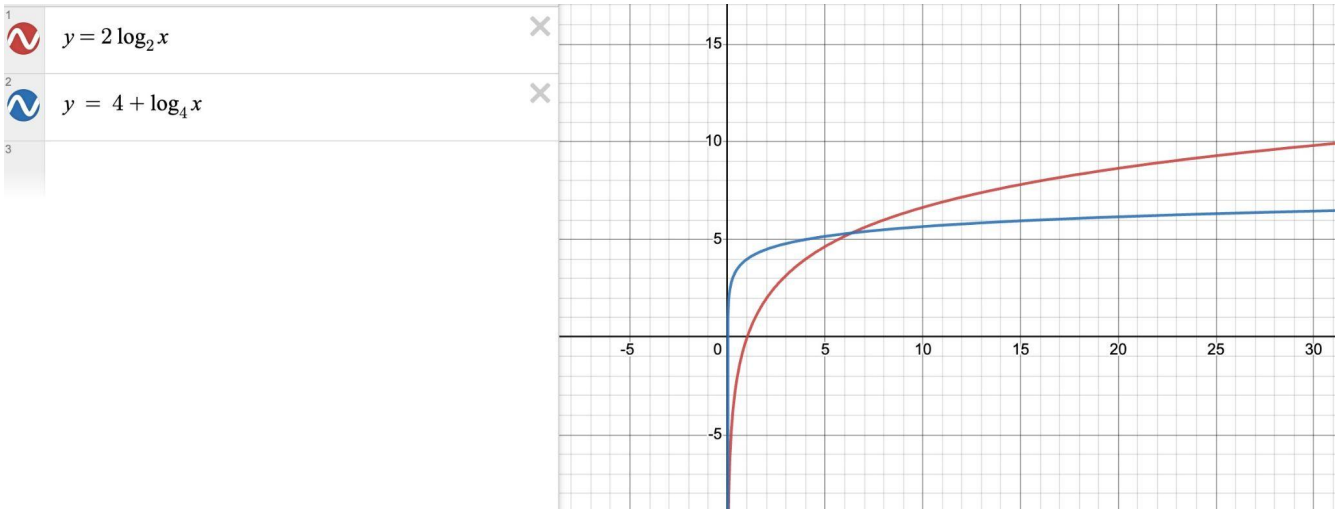






We have expanded our LSM-tree implementation to incorporate the Dostoevsky compaction policy with a size ratio of 4. So we did one more experiment with enabling both Monkey and Dostoevsky strategies. This comparison will be against our previous experiment that only used the Monkey with a size ratio of 2.

Put: The put throughputs under two different ratios stay similar, while the Dostoevsky with a size ratio of 4 performs slightly better than the basic LSM tree when the data volume is large. This is expectable, as Dostoevsky brings the insertion cost from $O(T \cdot L/B)$ down to $O((T + L)/B)$, where $L = log_T(N/P)$. This can be illustrated by the plot below, where the red line represents the theoretical performance curve of the basic LSM tree, and the blue line is the one with Dostoevsky with T=4. As x (i.e. N/P, the data volume) increases, the latter will eventually achieve a lower insertion cost than the basic one. Note that, for the experiment above when T=4, the put throughputs under 4, 16, 64, 256, and 1024 MB seem to be lower than the average trend. This is reasonable as the mutation of the last level on the LSM tree (leveling) has a larger cost than the other levels because it needs more compactions.



Get: The throughputs when T=4 are similar to when T=2. This is reasonable because Monkey maintains the bound steady at $O(2^{-M})$, and the values of M stay similar when T=4 (5.3913) and when T=2 (6.235).

Scan: The scan depends on the number of SSTs presented in the LSM tree since we need to traverse and merge them one by one. Dostoevsky uses tiering at all levels except the last, corresponding to a number of SSTs of $(4 - 1)(log_4(N/P) - 1) + 1$. In comparison, the basic LSM tree has $log_2 N/P$ SSTs. As can be seen from the plots below, when the data volume is low, the Dostoevsky version has fewer SSTs, while as the data volume increases, it eventually results in more SSTs than the basic LSM tree, corresponding to more I/Os and worse performance. This matches our finding for their throughput comparison, where under low data volume, the Dostoevsky version performs better. However, when the dataset is large, the basic one is generally slightly better.