CSC443 - DATABASE SYSTEM TECHNOLOGY

UNIVERSITY OF TORONTO

DEPARTMENT OF COMPUTER SCIENCE

Project Report

Date: April 10, 2023

Contents

1 Design

1.1 Notes About The Report

Before starting our report, we would like to point out some notes and clarifications.

- When we refer to a C++ class, we specify its name by putting its name in a special code format. For example, `className`. In these cases, please find the class implementation by referring to its file `className.cpp`.
- When we refer to a method, we use the name of the class followed by a double colon (::) and the method's name. For example, `Memtable::Put` refers to the method `Put` in the `Memtable` class. So to view a method's implementation, please refer to the class's cpp file, which in this case will be `Memtable.cpp` file.
- The project's code base consists of 5 main directories: `include`, `src`, `lib`, `tests`, and `experiments`. The `include` and `src` directories contain our main implementations, and the `lib` directory contains a third-party implementation for the `xxhash` which we have used as the hash function in step 2's extendible hash table and step 3's bloom filter. The `tests` directory contains our tests for the project, in which we have a set of tests for every main class implementation and the steps of the project. Finally, the `experiments` directory contains the experiments that we have implemented for all 3 steps of the project. The python file `drawGraphs.py` in this directory is used to draw our results.

1.2 Step 1

1.2.1 Database and memtable

In the first step of this project, we start by creating a database (a KV store) with an in-memory memtable, which is implemented using the Red-Black tree data structure. The user can create a database by initializing a `Db` object and providing the parameter `memtableSize`, which is the size of the memtable in terms of the number of key-value pairs. When the memtable fills up, we write all data within the memtable into an SST file in sorted order. We use `uint64_t` as the type of our key and values stored in the database, thus a key-value pair will consist of two `uint64_t` values.

See `Db.cpp`, `Memtable.cpp`, `RedBlackTree.cpp`, `Node.h`

1.2.2 Database's Open API

The user can open the database by calling its `Db::Open` method and providing the directory path in which the DB files will be stored. If the directory already contains SST files, the DB will catalogue these files for future DB operations.

1.2.3 Database's Put API

The Db's Put method `Db::Put` inserts a key-value pair into the database. This operation calls the `Memtable`'s `Put` method, which then calls the `RedBlackTree`'s Put method to insert the pair into the data structure if there is space. If not, the memtable writes all the data within the red-black tree into a new SST file and then resets and inserts the key-value pair into the new empty memtable. When the memtable writes the new SST file, it provides the data in sorted order so that the resultant SST file is sorted. The method `RedBlackTree::InorderTraversal` makes an in-order traversal of the red-black tree and provides the memtable's data in sorted order.

1.2.4 Database's Get API

The Db's Get method `Db::Get`, searches for a given key and returns its value. This operation calls the `Memtable`'s Get method, which then calls the `RedBlackTree`'s Get method to try to find to key within the data structure. If not found, we search through all the SST files in order of creation time by performing a binary search on each of them. The method `SST::PerformBinarySearch` implements this logic where it reads one page of the file and runs `Utils::BinarySearch` on the page. The method `SST::ReadPagesOfFile` reads one page of the file by default and returns a vector of `uint64_t` key-values.

1.2.5 Database's Scan API

The Db's scan method `Db::Scan`, scans and returns a range of keys (key1, key2) in the DB. To do so, it first searches for this range in the memtable and then searches through each SST file of the DB in order, from the youngest one to the oldest one based on their creation time. In scanning the files, which is implemented in `SST::PerformBinaryScan`, we first find the key1's offset by performing a binary search on the files. Then given the key1's offset, we continuously read one page from that offset and insert them into the result list, if the keys are within the range. We stop reading more pages as soon as we reach a key greater than key2 or we have reached the end of the file.

1.2.6 Database's Close API

The method `Db::Close` closes the operation of the database by writing the current contents of the memtable into a new SST file and then resetting its configuration.

1.3 Step 2

In this step, we evolve the KV store to include a buffer pool and improve SST query-efficient by using B-tree techniques. The buffer pool is implemented using an extendible hashing data structure and it maintains $\mathcal{O}(1)$ expected performance for all queries. Two eviction policies (clock and LRU) are also implemented for it. Additionally, the API of the KV store is expanded to allow users to change the maximum size of the buffer pool, and the buffer pool is integrated into the query workflow to enable the database to first attempt to find the target DB page in the buffer pool before retrieving it from the file storage.

1.3.1 Buffer pool and Extendible Hash table

The buffer pool is implemented in the `BufferPool` class and has an extendible hash table member (the `ExtendibleHashtable` class) which is initialized with a specific `EvictionPolicyType`. The buffer pool initializes its extendible hash table by taking the `minSize` and `maxSize` of the directory entries for the hash table which defines the min and max number of directory entries that the hash table can have. The class `ExtendibleHashtable` implements a hash map of directory entries (i.e., bucketId) to their corresponding `Bucket` object. The `Bucket` class represents a directory entry in the extendible hash table, and its `localDepth` member represents the number of bits that it currently uses, which is used to determine whether to split or expand the directory when the bucket fills up. Each bucket holds a linked list of `Page` objects, where each page object represents a database page that holds 4KB of data. A linked list is chosen here over an array or vector due to its property of fast insertion and deletion of objects anywhere in the list. Figure 1 below illustrates the design of the buffer pool.

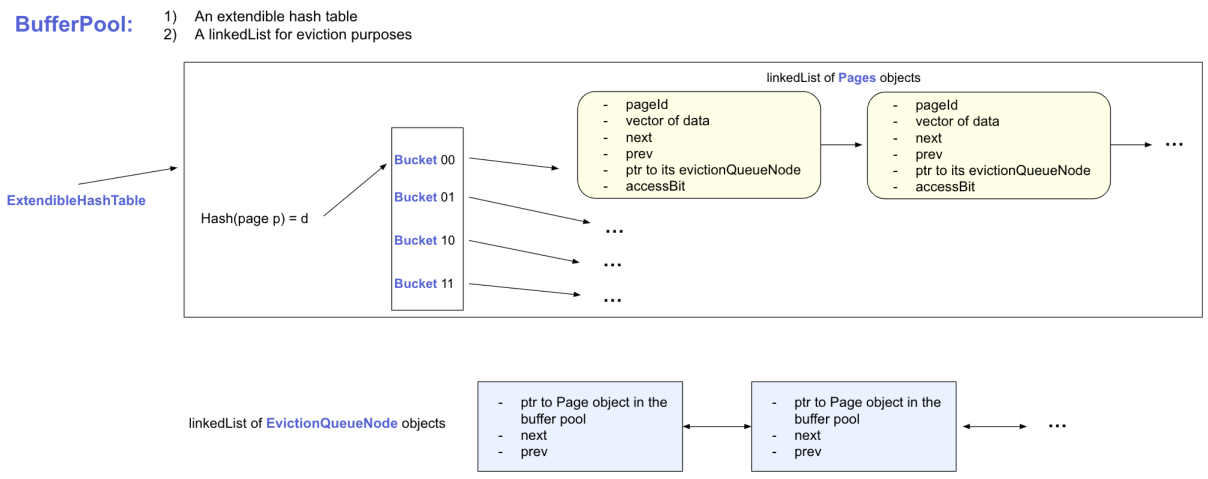
BufferPool:
1) An extendible hash table
2) A linkedList for eviction purposes
linkedList of Pages objects
ExtendibleHashTable
Hash(page p) = d
Bucket 00
Bucket 01
Bucket 10
Bucket 11
- pageId - vector of data - next - prev - ptr to its evictionQueueNode - accessBit
- pageId - vector of data - next - prev - ptr to its evictionQueueNode - accessBit
...
linkedList of EvictionQueueNode objects
- ptr to Page object in the buffer pool - next - prev
- ptr to Page object in the buffer pool - next - prev
...

Figure 1: Buffer pool design

The class `ExtendibleHashtable` keeps track of the current number of bits used to create directory entries in its `globalDepth` member, and the minimum and the maximum number of bits allowed to expand the number of directory entries stored in `minDepth` and `maxDepth`, respectively. One important part of an extendible hash table is the ability to expand the directory automatically when the hash table fills up. Our expansion criteria are as follows:

$$\text{\# of entries in hash table} > 0.8 * \text{\# of directory entries}$$

We decided to use the entries in the hash table rather than the bucket size as we don't want to over-expand the directory and we are making the assumption that we have a good hashing algorithm (xxhash) that will prevent long hash chains in buckets. Once the directory is expanded to the max, the eviction algorithm starts to kick in whenever a new page needs to be inserted into the buffer pool. Finally, to store a page in an extendible hash table, we use a string combination of SST's file name and the page offset as its key in the hash map.

1.3.2 Buffer Pool's Eviction Policies

The code in `LRU.cpp, Clock.cpp, EvictionPolicy.h, EvictionQueueNode.h` handles the eviction policy of the buffer pool. If the buffer pool is initialized with LRU eviction policy, every `Page` in each bucket of the extendible hash table has a `EvictionQueueNode` pointer member that attaches that page to a node in an eviction queue, which is used to keep track of the least recently used pages. In the case of the Clock policy, the page stores a `accessBit` to be used during the eviction process.

1.3.3 Static B-tree SST Files

This step of the project transforms the SST structure from a sorted file into a static B-tree to achieve a worst-case I/O of $\mathcal{O}(\log_B N)$ per query. The expanded database API to supports both types of SST files, in which we defined an enum `SearchType` for this purpose. In the case when the SearchType is `B_TREE_SEARCH`, we first call `SST::SetupBTreeFile` to compute and estimate the number of levels of the B-tree and then write the file using `SST::WriteFile` method. The method `SST::WriteBTreeLevels` (used in the `SST::WriteFile`) writes each level of the B-tree by appending data to each level as required. This method finally calls the method `SST::WriteEndOfBTreeFile` to mark the end of the file by writing one metadata page to the beginning of the SST file.

We used the class `SST` to store various SST file metadata such as the filename, the byte size of data, and the maximum offset to read the leaves of the B-tree. The `SST` class has various private member functions used for writing data to a file, including getting a page ID for use in the buffer pool, adding data to a list of nodes, writing extra data to align it in a file, adding the next internal level fence keys, writing B-tree levels, getting B-tree metadata, writing B-tree metadata, writing and getting a page from the buffer pool. The B-tree SST file consists of metadata of B-Tree including the number of levels and page offset for each level and followed by the actual data in each level of the B-Tree.

Figure 2 shows the design of the B-tree file structure (which includes the bloom filter, introduced in step 3).

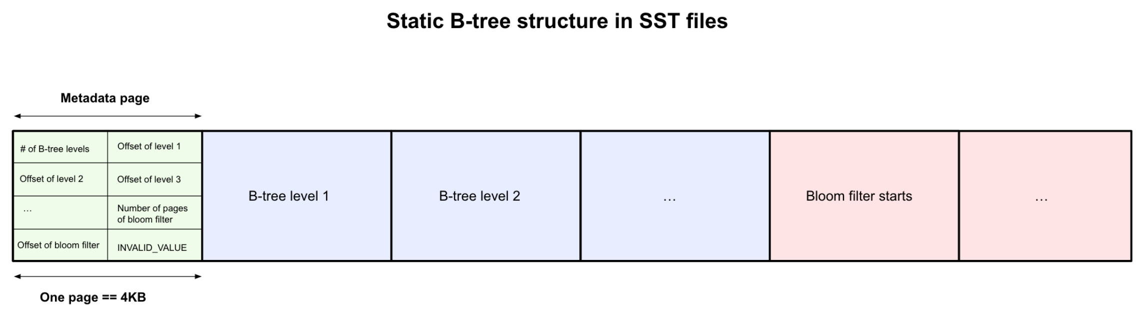

Figure 2: Static B-tree structure design

The method `WriteBTreeMetaData` is responsible for writing the metadata to the starting page of the SST file. The `WriteBTreeLevels` function writes the leaves of the B-Tree to the SST file by adding the fence keys to the next internal level and then writing the leaf level to the SST file. The `WriteBTreeInternalLevels` function writes the internal levels to the SST file. We implement the write to a B-tree file such that we can append to its levels gradually as more data becomes available, which is a feature that is useful for the next phase of the project about the LSM tree. To facilitate that, we have a class `BTreeLevel` which holds the `startingByteOffset` and `nextByteOffsetToWrite` of the B-tree level that it represents. This class also keeps track of its remaining (non-page aligned) data that has not yet been written to the SST file.

In this step of the project, we always write the whole SST file at once by calling `SST::WriteFile` method and proving the data that needs to be written in the leaves of the B-tree. This method then calls `SST::WriteBTreeLevels` to write all of the levels and also write one page of metadata to the first page of the SST file (which is done by calling `SST::WriteEndOfBTreeFile`). The `endOfFile` flag is used to decide whether we want to finish off writing the B-tree file or if this file is still being written in some continuous steps. As said, at this step of the project we always set the `endOfFile` to true since we want to write the file all at once. However, we will use this flag when writing the SST files produced during the compaction process of the LSM tree in step 3. The constants `SST::KEYS_PER_PAGE`, `SST::KV_PAIRS_PER_PAGE`, and `SST::PAGE_SIZE` constants are used throughout our implementation to determine the appropriate offsets and alignment for the data. As we are using `uint64_t` to represent a key or a value in our KV-store, `SST::KEYS_PER_PAGE` has the value of 512 and `SST::KV_PAIRS_PER_PAGE` has the value of 256. Based on these considerations, there are 512 `uint64_t` fence keys in each internal node and 256 key-value pairs in each leaf node of the static B-tree files.

1.4 Step 3

In this step, we transform the KV store implementation into an LSM tree and add a compaction mechanism and a bloom filter for each SST file.

1.4.1 LSM Tree

The LSM tree's implementation follows a basic LSM tree with a fixed size ratio of 2 between any two levels. The LSM tree is implemented in the class `LSMTree` which uses the class `Level` as the implementation for its levels. Anytime the Db's memtable is full, the data of the memtable is written to the LSM tree's first level by calling `LSMTree::WriteMemtableData`. This method first makes sure the LSM tree has a first level and creates it if needed. It then writes the memtable's data to its first level by calling the first levels' `WriteDataToLevel` method. After that, this method calls `LSMTree::MaintainLevelsCapacityAndCompact` method to make sure we maintain the limit of at most one SST file per level. In this recursive method, whenever two SSTs are at the same level, we call that level's `SortMergeAndWriteToNextLevel` method (starting from the first level) to merge-sort those two SST files and write them as a new SST file into the next level. Each level of the LSM tree keeps track of an array of its SST files along with the info about its level number, search type of its SST files, and some parameters about the bloom filters of the SSTs.

We define `inputBufferCapacity` and `outputBufferCapacity` as the sizes of input and output buffers (in number of pages) used during the compaction process. Specifically, during the compaction process, we allocate three buffers - two input buffers for the files being sort-merged and one output buffer for the new file being created. We use two classes `InputReader` and `OutputWriter` to represent the input and output buffers during the compaction process. Each SST file has a member of type `InputReader` that reads `inputBufferCapacity` many pages of that SST file into memory to be sort-merged with the other SST file. The compaction operation also deals with updates and deletes by keeping the most recent version of each key and adding a tombstone to the memtable for deletes. We use `Utils::DELETED_KEY_VALUE` as a tombstone to represent the value of a deleted key. If the compaction operation encounters this value for a key, it considers that key as deleted and does not include it in the resultant SST file. We have two APIs `Db::Update` and `Db::Update` in the Db class that the user can call to update or delete a key. In the context of an LSM tree, updating a key is equivalent to putting a key with a new value and deleting a key is equivalent to putting a key with the value of `Utils::DELETED_KEY_VALUE`.

1.4.2 Bloom Filter

Each SST file in each level has a corresponding bloom filter. The bloom filter is implemented in `BloomFilter` class. This class is initialized by two parameters `bitsPerEntry` and numKeys, where it creates a filter array with the size bitsPerEntry * numKeys and then computes the optimal number of hash functions needed for the optimal performance of the bloom filter by computing std::ceil(log(2) * bitsPerEntry). We represent the filter array as a vector of uint64_t, but we use it as a bit array. That means every time a key is hashed to a bit, we set a single bit of this vector for that. This is handled by shifting the value of 1 an appropriate number of times and using OR for setting and AND for checking if a key is in the bloom filter. We use the same xxhash library used for the buffer pool as our hash function of the bloom filter. To simulate multiple hash functions needed for the bloom filter, we provide different seed values to xxhash thus producing the effect of multiple different hash functions. By experiment, we found that using seeds with increments of numHashFunctions distributes the bits that each key is hashed to well enough, so as to provide reasonable performance.

The bloom filter of each SST file is created at the time when an SST is created. If the SST belongs to the first level, its bloom filter is created all at once by hashing all of the file's keys to the filter array by calling the method BloomFilter::InsertKeys. Otherwise, the SST files that are created during the sort-merge operation will hash their keys to their bloom filter by calling BloomFilter::InsertKey, as we perform the process of compaction. We integrate the Bloom filter into the GET workflow by reading the Bloom filter of the SST and searching that first.

1.4.3 LSM Tree's GET workflow

The method LSMTree::Get(uint64_t key) of the LSM tree is responsible for the Get requests of a key. It starts searching through the SST files of each level (starting from the first level) and calls the SST::PerformBTreeSearch method of each SST file. In this method, first, we fetch the bloom filter pages of that corresponding SST file (either by getting it from the buffer pool or reading it from the storage - lines 450 and 451 of SST.cpp). We then call the method BloomFilter::KeyProbablyExists of the bloom filter. If the bloom filter believes that the key does not exist in this file, we immediately return the value of Utils::INVALID_VALUE which represents that the key does not exist in this file. However, if the bloom filter believes that the key might exist in the file, we call the method SST::FindKeyInBTree which performs a B-tree search on the SST file. Given an input key, the method LSMTree::Get always returns one of these 2 values: the value of the key if the key exists, or the value of Utils::INVALID_VALUE if either the key is deleted (i.e. it encounters a value of Utils::DELETED_KEY_VALUE) or it does not find the key and reaches the end of the file.

1.4.4 LSM Tree's Scan workflow

The method LSMTree::Scan of the LSM tree implements the scan process of an LSM tree on the range (key1, key2). The scan starts by keeping track of the curKeyToLookFor, which is initialized to key1. Each SST file has a `ScanInputReader` member which is a class that is responsible for reading `inputBufferCapacity` number of pages of the SST file during the scan process. The class `ScanInputReader` internally keeps track of the current buffer that it searches through and reads more pages from the file if needed. This class tries to find the requested `curKeyToLookFor` by doing a binary search on its current buffer. The scan operation starts from the first level and continuously queries the `ScanInputReader` member of the SST file that it searches and asks if the `curKeyToLookFor` exists in that file. Note that since we are implementing a basic LSM tree, there will be at most 1 SST file at each level. When a level is searched for the first time, the range of pages in the SST file that contain the range (key1, key2) will be found and set using `GetBTreeScanLeavesRange` method. The scan operation's steps continuously look for the current key within the range that it needs to find until it either reaches the key2 or exhaustively searches all the levels and does not find any more keys within the range. In that case, it stops and returns the vector of results.

1.4.5 Persisting Bloom Filter

As mentioned in step 2, each SST file is a static B-tree file, which consists of a metadata part (written on the first page of the file), and the B-tree levels one after the other. The bloom filter has two pieces of metadata (i.e. the number of pages of the filter array in the file and its starting page), which are added to the metadata page of the file. Also, the filter's array is written after the B-tree's levels.

1.4.6 Bloom Filters and Buffer Pool

The method `PerformBTreeSearch` used in the Get workflow reads the bloom filter by calling the method `SST::GetBloomFilterPages`, which fetches the filter's array either from the buffer pool or file storage. If it is the first time that this bloom filter's array is read, this method reads it from the storage by calling `SST::ReadPagesOfBloomFilter` and then inserts the filter array in the buffer pool (by calling `BufferPool::Insert`). As mentioned in step 2, we use a string combination of SST's file name and the offset of the page as the key in the extendible hash map of the buffer pool. In the case of the bloom filter, we store the whole pages of it in the buffer pool and so we hash it using the offset of the starting page of the bloom filter array in the storage. As we hash a bloom filter using the offset of the first page of its array in the storage, that means we treat the whole bloom filter as if we have hashed one page to the buffer pool. This ensures that we maintain our access time of $\mathcal{O}(1)$ even if we have a lot of bloom filters mapped to the buffer pool. The only factor that storing the bloom filters in the buffer pool will add is the need for more memory, which is expected if we aim to keep the bloom filter of the SST files in the memory.

2 Project Status

We have implemented and finished all 3 steps of the project.

3 Experiments

Having implemented a key-value store database that consists of various components and data structures, we want to find out how each component or data structure affects the performance of the database operations.

We define the throughput of a database operation as the amount of data (measured in MB) that the database operation can process per second. We calculate the throughput as follows:

$$\textbf{Throughput} = \frac{\text{Processed key-value size in MB}}{\text{Elapsed time in seconds}}$$

Similarly, we measure the latency as the amount of time it takes to perform a database operation:

$$\textbf{Latency} = \frac{\text{Elapsed time in seconds}}{\text{Number of data entries to query}}$$

3.1 Extra Experiments

For some extra insight into the performance of the database, we implemented some extra experiments.

1. For step 1, we tested throughput using multiple sizes of memtable to see how changing the size of the memtable affects performance
2. We included latency for all the `GET` query experiments to see how changes in database configurations affect the time it takes for a `GET` query to process.

3.2 Experiments for Step 1

3.2.1 Methods

For Step 1, we want to determine the effects of increasing memtable size and input data size on the performance of various database operations. Thus, our experiment tested the throughput of each of the `PUT`, `GET`, and `SCAN` database operations using a memtable of size 1MB and 4MB, and unique randomized input data of sizes from 1MB up to 1GB.

For each input data size, we run the `PUT` operation first to insert all the data into the database. This ensures there is data in the database when performing `GET` and `SCAN` experiments right after. The GET operation experiment performs searches on a subset of all the data we inserted into the database in randomized order, which helps to help speed up the experiment's run time. The SCAN operation experiment scans for all the data within the database starting from key = 0 (as the scan's lower bound), and the number of entries in the database as the scan's upper bound.

3.2.2 Results of Experiments in Step 1

Graphs shown in Figures 3-6 illustrate our experiments with the PUT, GET, and SCAN operations implemented in step 1 of this project. As an extra experiment to get more insights and show the effect of having different memtable sizes, we show the results using 2 different database instances with the memtable sizes of 1 MB and 4MB.

Figure 3 shows the throughput of the PUT operation while we vary the data size (from 1 MB to 1GB). In each of these two databases with different memtables, we can see that the throughput significantly drops if we insert the data size greater than the size of the memtable. Specifically, in the database with a 1 MB memetable, the throughput drops drastically when we insert 2 or more MB of data. Similarly, in the database with a 4 MB memetable, the throughput drops much faster when we insert 8 or more MB of data. This is expected since in-memory operations are much faster than performing file I/O, which is required when data are larger than the memtable size.

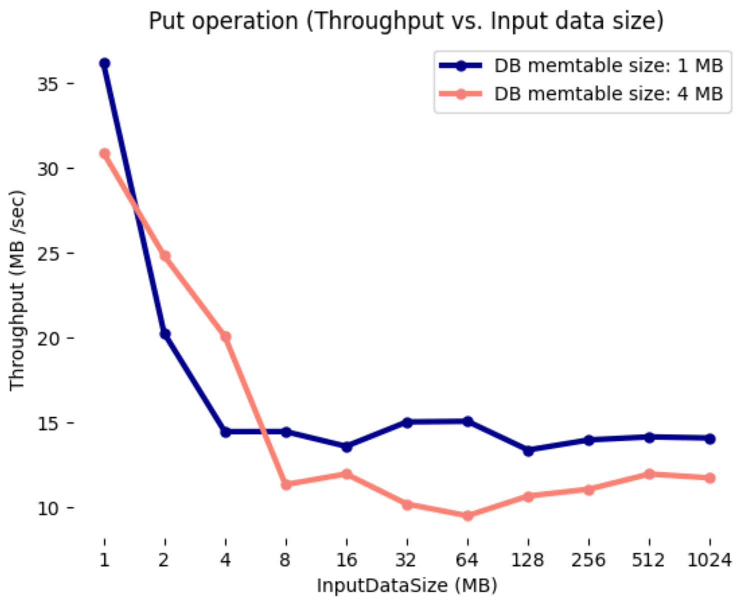

Figure 3: Throughput for PUT operation on various input data size.

The throughput seems to plateau for data size larger than the memtable size which is also expected since the file writing for the sorted file has a minimum overhead, thus the throughput is mostly dominated by the throughput of the disk/SSD. The throughput also appears to be lower for databases with a memtable size of 4MB which is quite unexpected since we would expect smaller and more frequent file writes would generate more overhead. Though it is also possible that this lower throughput could be caused by the CPU overhead of maintaining a larger self-balancing BST for the 4MB memtable.

Figure 4 shows the throughput of the `Get` operation while we vary the data size (from 1 MB to 1GB). We decided to use log-scale on the Y-axis due to the big drop in throughput when the size of the data written to the database is greater than the memtable size, and we don't want to show a flat line for larger data sizes. The graph shows a clear negative linear relationship between the exponentially increasing input data size on the x-axis and log-throughput on the y-axis, which very well outlines the expected $\mathcal{O}(\log N)$ of the binary search algorithm.

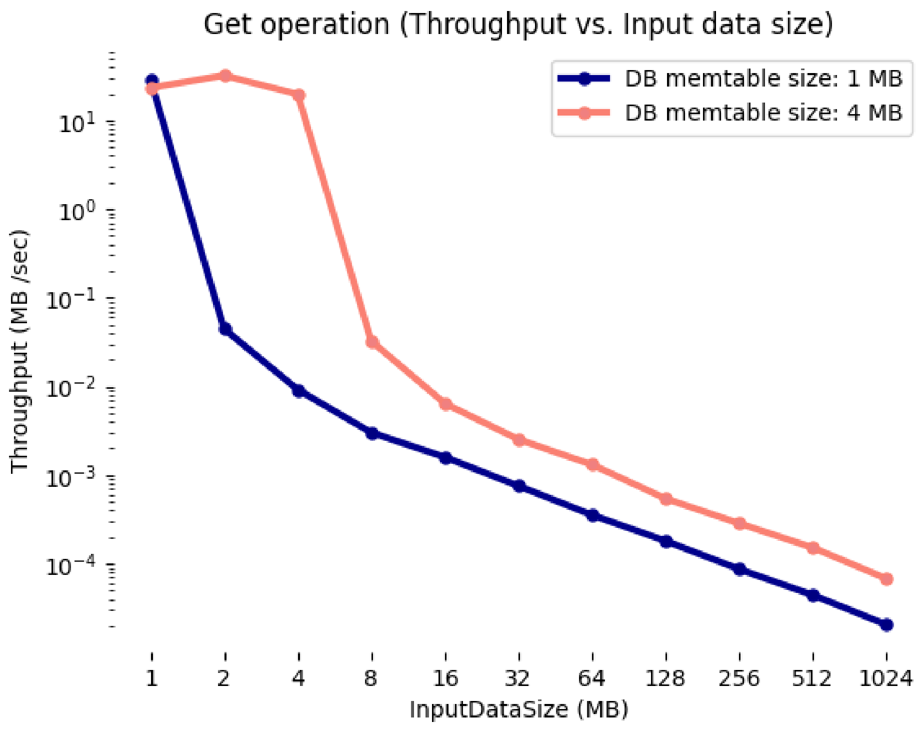

Figure 4: Throughput for GET operation on various input data size.

Similar to `PUT` experiment, we also made the same comparison between the two databases with memtable of sizes 1MB and 4MB. In this case, the 4MB configuration appears to have a slightly higher throughput throughout which is expected since there are more data stored within a single file, and thus more can be done within a single file.

The raw throughput number of `GET` queries is very low, ranging from 0.0012MB/s (1.2KB/s) for an input data size of 2MB down to 0.000081MB/s (0.8KB/s) for an input data size of 1GB. This very well shows the limitations of sorted files when it comes to query operations.

Figure 5 shows the latency of the `GET` queries with increasing input data sizes. This is an extra experiment to get more insights into the amount of time each `GET` query took. As expected, the latency increases as the size of the data stored in the databases grows, and the exponential curves again show off the $\mathcal{O}(\log N)$ runtime of the binary search algorithm.

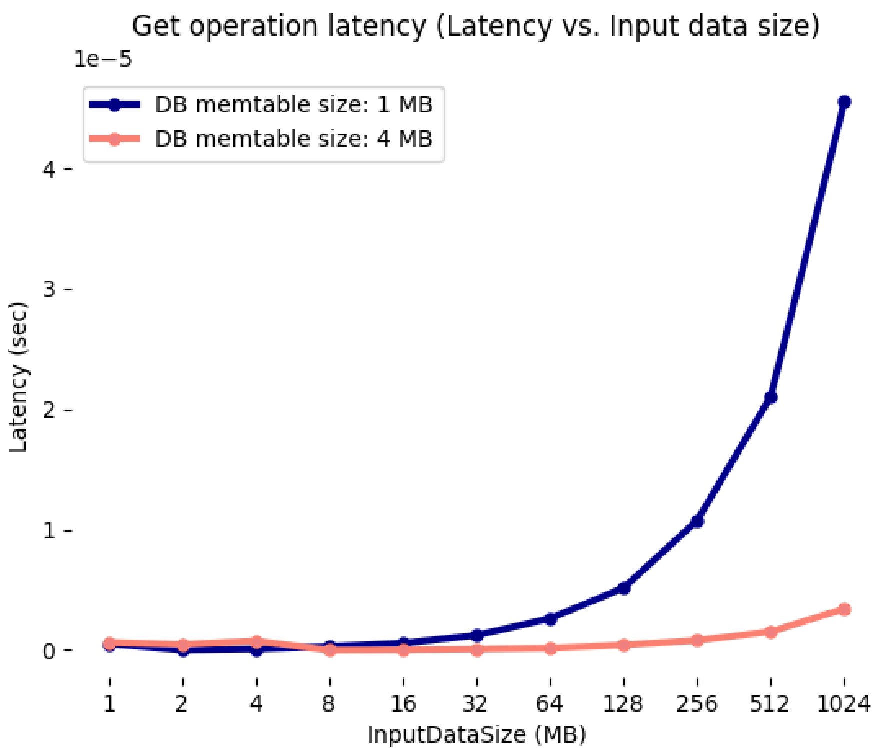

Figure 5: Latency for GET operation on various input data size.

One interesting observation is that the latency of query using the database with a smaller memtable size (i.e. 1MB) appears to be growing much faster than the other databases. This is most likely due to the fact that a smaller memtable produces smaller

SST files, which means there are more files for the same amount of data. This causes a query to look through more files compared to SSTs generated by a 4MB memtable, where the data is stored across fewer files thus reducing the number of files needed to read.

Figure 6 shows the throughput of the SCAN operation while we vary the data size (from 1 MB to 1GB). Similar to GET operation, we decided to use log-scale on the y-axis due to the sharp drop in throughput when the input data size is greater than the memtable size, and we also see somewhat of a negative linear relationship between the x-axis and y-axis which again shows traces of binary search.

The throughput of the SCAN ranges between 12 to 25 MB/sec for the 1MB memtable database and 15 to 35MB/sec for the 4MB memtable database. Comparing this throughput to GET operations which were around 0.00001-0.001 MB/sec, we can see that we are able to scan much more data per second. This is expected as the scan operation can read data contiguously from the same file once the lower bound is found, and it also obtains a much larger amount of data for each read than GET operation.

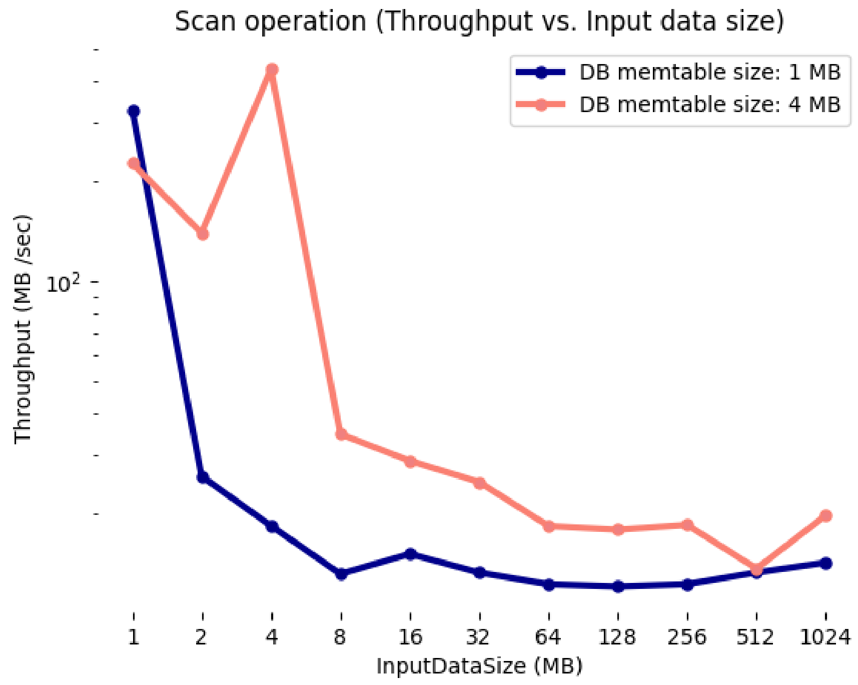

Figure 6: Throughput for SCAN operation on various input data size.

3.3 Experiments for Step 2

3.3.1 Methods

For Step 2, we conducted two experiments:

1. The first experiment will be studying the effects of various buffer pool sizes as well as the eviction policy on the performance of database queries. We start with a database that contains 1GB of unique random data, and then we use a buffer pool of sizes starting from 1MB up to 512MB. With this setup, we recorded the query operation throughput on the database using both LRU and Clock eviction policies, and on two different data patterns.

2. The second experiment will be studying the effects of different SST file structures on the performance of database query and scan operations. Similar to the first experiment, we also start with a database that contains 1GB of unique random data, as well as a buffer pool of size 64Mb (determined from experiment 1 to be the best-performing buffer pool size). With this setup, we perform both GET and SCAN operations on the data sizes, varying from 2MB up to 1GB and recorded the throughput and latency of each of the operations.

3.3.2 Results of Experiments in Step 2

Figure 7 illustrates the first experiment where we measure the throughput of the `GET` operation on random data with a buffer pool of various sizes. The advantage of Clock over LRU is that it has much lower CPU overhead. However, deciding what page to evict is a different matter that can inversely affect the Clock's CPU efficiency. As we can see in this figure, on random queries, the Clock algorithm has a better performance than LRU for smaller data buffers of up to 8 MB, but after that its performance becomes worse than LRU. The reason for this is the randomness of queries. When the queries are random, there is a higher chance that the bits in front of the clock's handle get set, as more and more pages are accessed. This makes the travel time of the handle to be longer as it has to rotate more until it can find a page that it can evict (i.e. has the access bit of 0). As we can see in this figure, the longer travel time of the handle in the clock algorithm shows itself much more for the greater sizes of buffer pool (i.e. greater than 8 MB) and this causes the Clock algorithm to have a worse throughput compared to LRU in these cases.

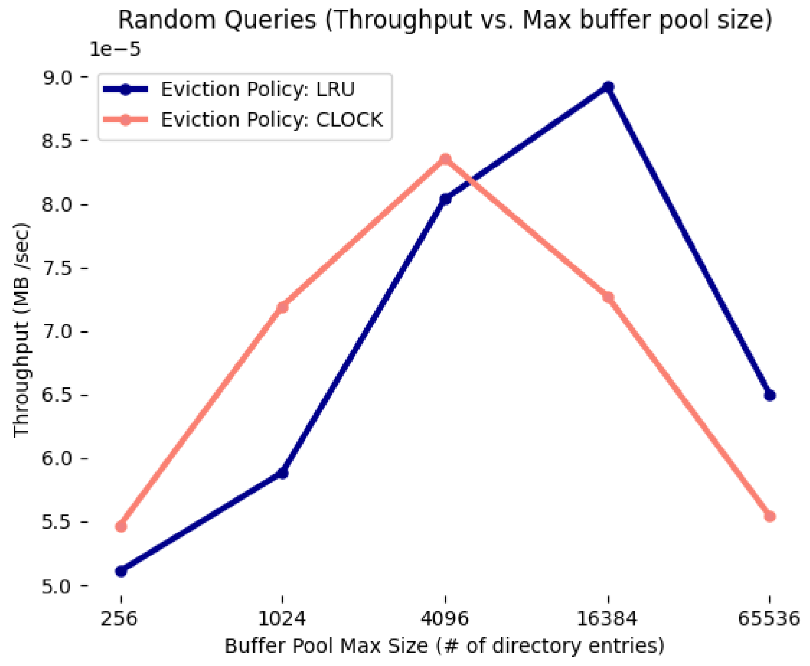

Figure 7: Throughput for Get operation on increasing buffer pool size (Random)

One other observation in this figure is the fact that both LRU and Clock algorithms have a max point and after which their performance drops and obtains a decreasing trend. We think this could be due to the effects of CPU caches. For the LRU algorithm, we use a linked list of `EvictionQueueNode` (an object consisting of only 3 pointers), while in the clock algorithm, we use a list of `Page` objects that have greater sizes. So we believe the larger memory footprint of the list of pages used in the Clock causes it to not fit in the CPU caches and hence increases its access time. As we can see in the graph, LRU's linked list does not fit into CPU caches after the buffer pool size of 64 MB, while this happens for the Clock's list after the buffer pool size of 8 MB.

Figure 8 illustrates the other case of the first experiment, where we measure the throughput of the `GET` operation on data with spatial locality with buffer pools of various sizes. Overall, we can see that the Clock performs better than LRU on data with spatial locality. This is expected as it is more CPU-efficient than LRU. Accessing data with spatial locality enhances this efficiency by maintaining its clock round short.

One thing to note for these two experiments with eviction policies is that we believe they are very sensitive to outside factors. Since we are using randomized data without a seed (we probably should have one for reproducibility), and we are running on a shared server, these could greatly affect the throughput.

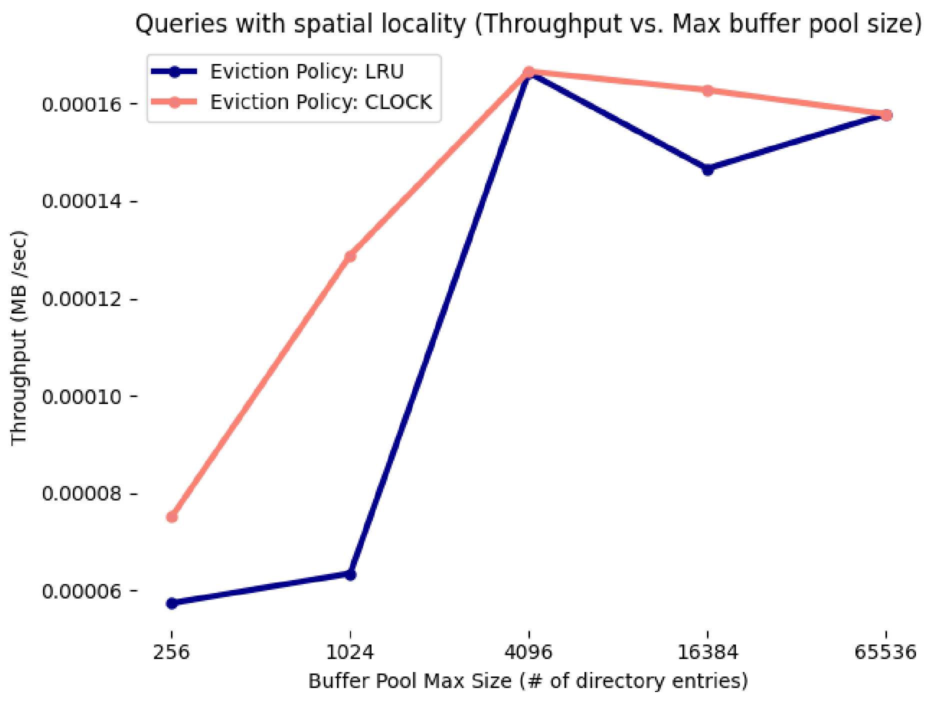

Figure 8: Throughput for GET operation on increasing buffer pool size (Spatial)

Figure 9 illustrates the second experiment, in which we measure the throughput of the `GET` operation and measure the effect of using binary search vs. B-tree search for the SST file structures. Clearly, the B-tree file structure shows a greater throughput compared to the sorted file structure which is expected. Specifically, a B-tree file structure results in a 3.5 greater throughput. One other observation is that, in both of these file structures, the throughput starts to plateau after the query data size grows more than 8 MB, indeed while B-tree still has greater performance. This stable behaviour in throughput is due to the limits of disk/SSD's IO operations, which have a greater overhead as the size of the data grows.

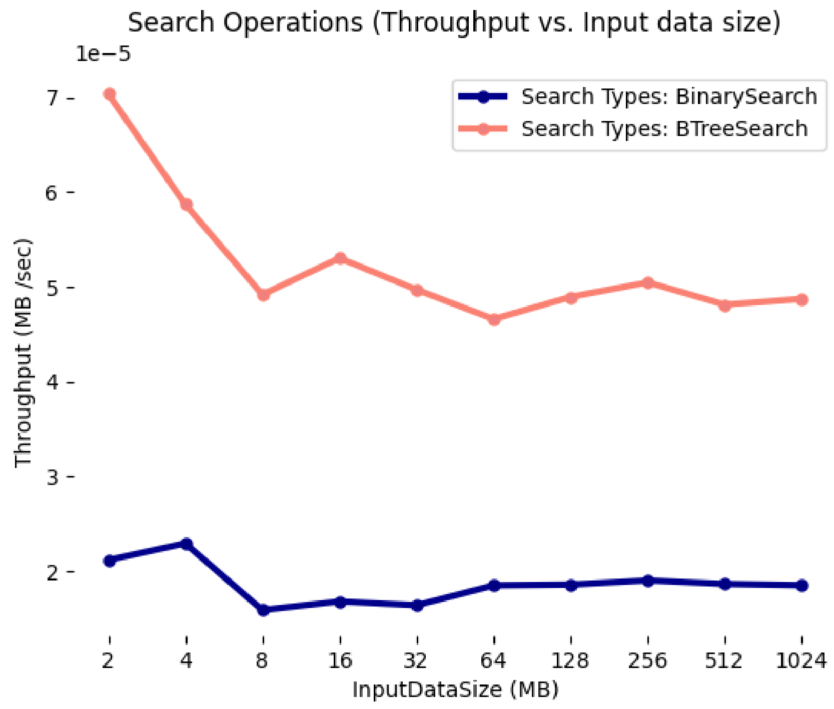

Figure 9: Throughput for GET operation (Binary search vs. B-Tree search)

Figure 10 illustrates the `SCAN` throughput of using binary vs. B-tree search. Again the B-tree file structure shows a greater throughput performance compared to only binary-searchable files. The B-tree search performed better up to the input sizes of 256 MB, whereas the sorted file seem to perform better from this point on. One reason that might explain this is that for bigger inputs, once the lower bound of the scan result is found, in both cases, we have to read sequentially until we reach the upper bound. For large data, the advantage of the fast query provided by the B-tree for finding the lower bound gets dominated by the extra overhead for the sequential read of contiguous data in the B-tree structure compared to a sorted file, thus producing lower throughput.

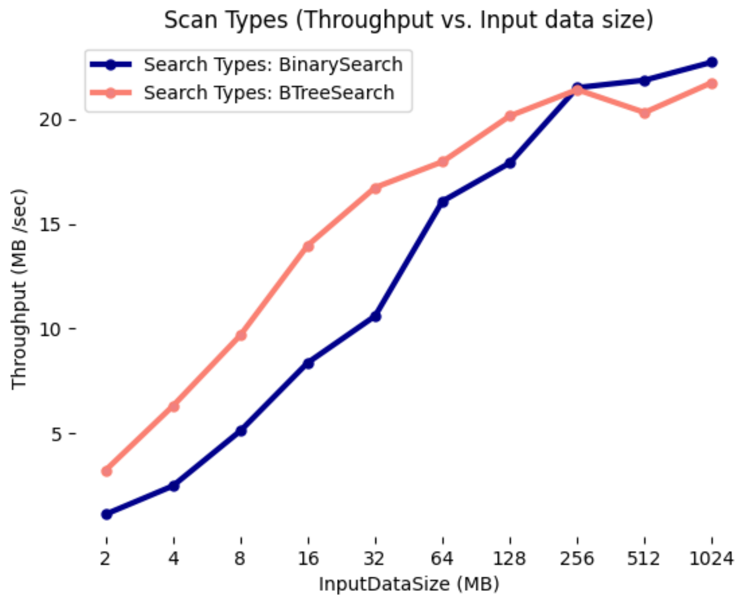

Figure 10: Throughput for SCAN operation (Binary search vs. B-Tree search)

3.4 Experiments for Step 3

3.4.1 Methods

For Step 3, we conducted two experiments:

1. The first experiment will investigate the performance of LSM-Tree data structure under various sizes of data. We tested the throughput of `PUT` (such that data contains some random `UPDATE` and `DELETE` data mixed in), `GET` and `SCAN` operations on input data sizes ranging from 2MB to 1GB using a DB with a memtable size of 1MB, a buffer pool of size 10MB, and LSM-Tree data structure with bloom filter of 5 bits per entry. The 1MB input data is eliminated in this case as we already observed the throughput when all data fits in memory. Similar to the Step 1 experiment, we run the `PUT` experiment first to populate the database with data so that `GET` and `SCAN` can be performed after, and `GET` is also completed on a subset of all data for run time reasons. However, since `SCAN` operation is also slow for LSM-Tree data structure, we also reduced the amount of data to scan to speed up the total run time of the experiment.

2. The second experiment will study the effects of increasing bits per entry for the bloom filter on the throughput of query operations. We start with DB with a memtable of size 1MB, and a buffer pool of size 10MB. We then record the throughput of `GET` operations starting with 2 bits per entry and an input data

size of 2MB all the way up to 1GB, while incrementing the bits per entry every time, we double the input data size.

3.4.2 Results of Experiments in Step 3

Figure 11 shows the PUT throughput on increasing input data size using LSM-Tree data structure. Given the negative log shape of the graph and along with the values on the x-axis increasing exponentially, this perfectly demonstrates the $\mathcal{O}(\log_2 N/B)$ cost of LSM-Tree insertion.

The throughput numbers for PUT using LSM-Tree range around 1.1 to 12.6MB/s. Comparing this to the sorted file in Step 1, it is much lower which is expected as the sort-merge operations performed by LSM-Tree have a lot more CPU overhead.

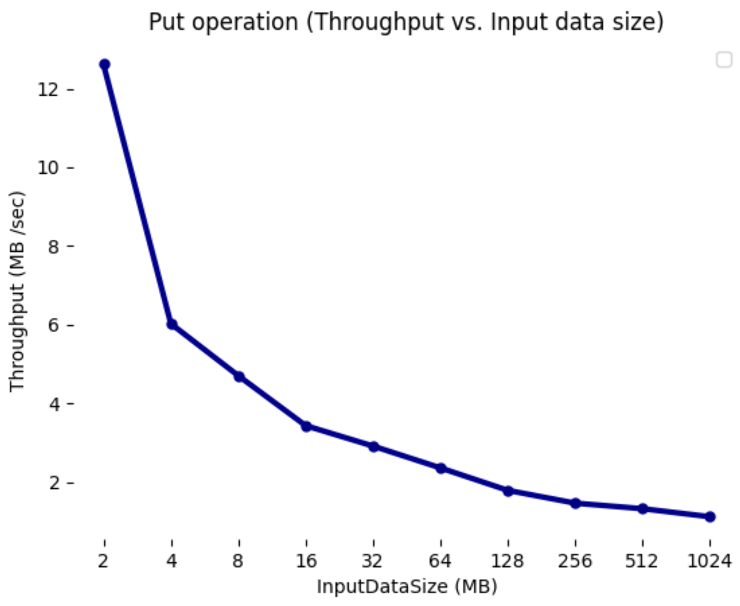

Figure 11: Throughput for PUT operation on various input data size (LSM-Tree)

Figure 12 shows the throughput of GET queries on the LSM tree while we vary the input size. Overall, we expect the throughput to decrease as the size of data written to the database grows, which is shown in the overall trend of the graph. However, as we are using bloom filters, there are chances that the bloom filter can help avoid file I/O in some queries. These instances of early returns can speed up the performance of GET queries even if the size of data in the database is large. This might explain the random spikes of throughput in the graph.

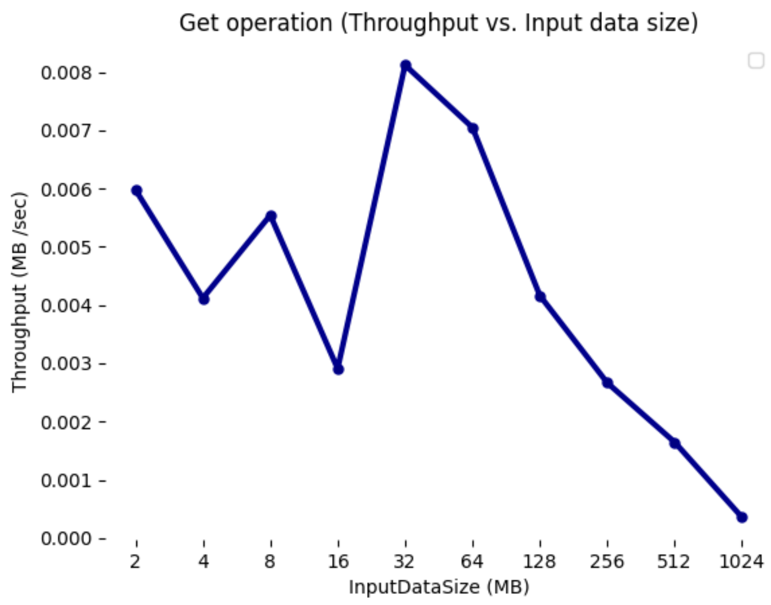

Figure 12: Throughput for GET operation on various input data size (LSM-Tree)

Figure 13 shows the latency of the `GET` queries on the LSM tree. As expected, the latency increases as the size of data in the database grows. This increase is quite steady up to data sizes of around 256 MB, but after that the increase in the latency is exponential. This is an extra experiment.

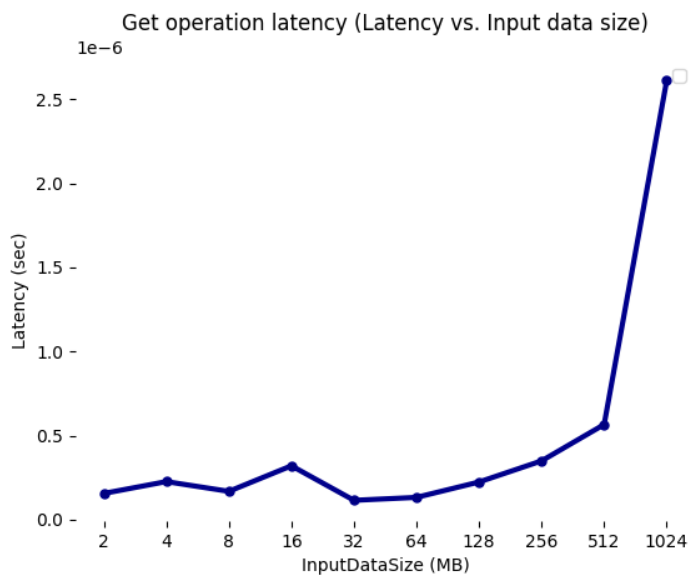

Figure 13: Latency for GET operation on various input data size (LSM-Tree)

Figure 14 shows the last graph for the first experiment of this step, which is the throughput of the Scan operation on the LSM tree. As shown, the scan's throughput decreases exponentially as the data size of the database grows. This is also expected as we know the scan cost of LSM-Tree is $\mathcal{O}(\log_2(N/B) * \log_B(N) + S/B)$, and the log factors can be clearly seen in the shape of the graph.

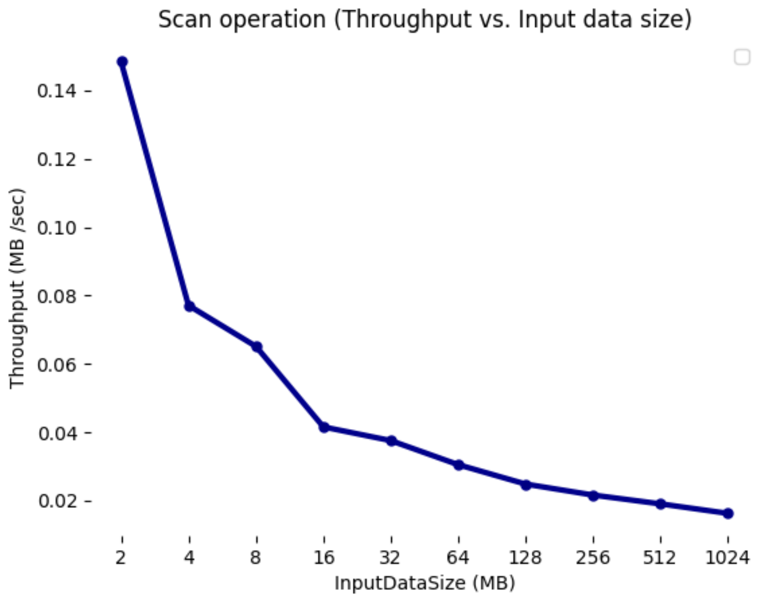

Figure 14: Throughput for SCAN operation on various input data size (LSM-Tree)

Figure 15 shows the throughput of the `GET` queries as we increase the bits per entry for the bloom filters as well as the input data size. In this experiment, as the data grows we increase the number of bits per entry for the bloom filters, which results in a greater bit array for the bloom filters. So we theoretically should expect an increasing trend in the performance of get queries, as as the data grows we allocate more bits per entry, so the false positives rate should decrease. However, a larger bits per entries value means more hashing for every single query. While the former can improve the throughput, the impact of more hashing can affect the throughput negatively. In Figure 12, when the value of bits per entry increases from 2 to 3 or from 5 to 6, we see a great increase in the throughput, while increasing the bits per entry elsewhere lowers the performance. The low performance in higher values of bits per entry demonstrates the effect of a greater number of hashing that is needed in these cases. Also increasing the bits per entry of a bloom filter means increasing its bit array, which in turn can make it outgrow the CPU caches and hence degrade the performance. So overall, we can say although increasing the bits per entry of the bloom filters decreases its false positives rate, the effect of a high number of hashing and memory usage of that can negatively impact that gain. Hence, a lower throughput for larger data sizes is an expected result.

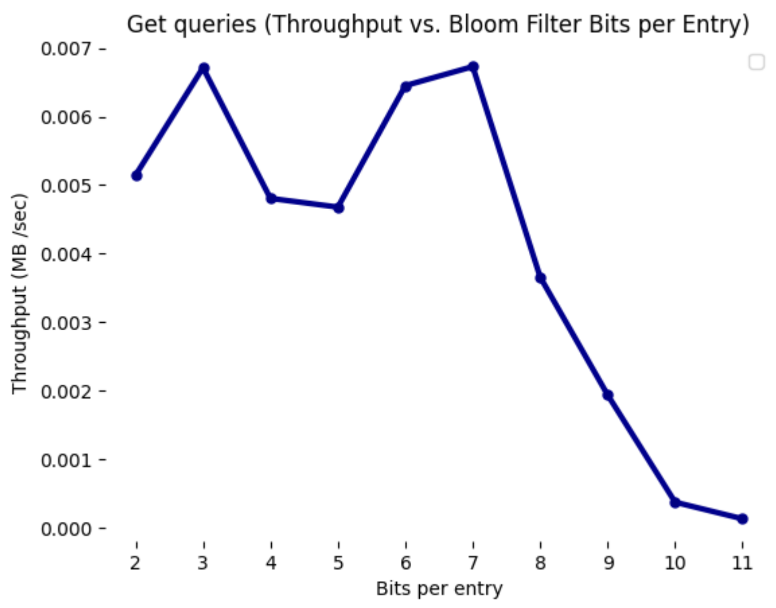

Figure 15: Throughput for GET operation on various bloom filter bits per entry

4 Testing

For testing, we decided to avoid the complication of using a third-party library and thus implemented a very simple test-and-print system where each test will return a boolean value to indicate whether the test passes or not, and we will print out an error message informing any failing tests.

As for the testing methodology, we implemented extensive unit testing for each component of the database such as the memtable, the extendable hash table, and eviction policies to ensure each component functions correctly on its own. We also implemented some integration tests for the entire database by calling the `GET`, `PUT`, and `SCAN` APIs under various configurations to ensure the database functions correctly as a whole.

Here is a whole list of test files in the project directory and the functionality they test:

- `TestBloomFilter.cpp` - unit tests for Bloom filters used in LSM-Tree.
- `TestClock.cpp` - unit tests for the CLOCK eviction policy used in the buffer pool.
- `TestLRU.cpp` - unit tests for the LRU eviction policy used in the buffer pool.

- `TestLSMTree.cpp` - unit tests and integration tests for LSM-Tree implementation.
- `TestMemtable.cpp` - unit tests for memtable implementation which also tests the underlying red-black tree implementation.
- `TestExtendibleHashtable.cpp` - unit tests for the extendible hashtable data structure used in buffer pool.
- `TestUtils.cpp` - unit tests for utility functions used throughout the project, such as binary search and file opening.
- `TestSST.cpp` - unit tests for SST-related operations, such as writing files and performing search and scan on SST files.
- `TestDb.cpp` - integration tests for the entire database under various configurations.

5 Compilation and Running Instruction

To minimize the effort spent on configuration and compilation, we decided to use CMake for our project as it greatly simplifies the process and many IDEs provide robust support for it.

To start, run the following commands:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

To run all the tests in the project from the build directory, run:

```
$ ./tests/test
```

To run all the experiments, execute the shell file in the project directory:

```
$ ./run-experiment.sh
```

This will build the project in release mode (with optimizations), run all of the experiments, and generate graphs using the output CSV files in the project directory.