CSC 443 Team

- ZhiTing Peng (1004956595)
- MingXi Liu (1004758595)
- Junyoung Seok (1004978919)

Description of design elements

Memtable: memtable.h

The memtable is implemented as an AVL tree. The functions get and put are implemented according to the standard algorithms. The function scan is implemented as an inorder traversal through the tree, but only including nodes with keys between the minimum key and maximum key. The minimum 64 byte integer value is reserved for the tombstone.

Buffer Pool and Extendible Hashing - bp_directory.h

The buffer pool is a critical component of a database, and it is comprised of three classes: BPDirectory, BPLinkedlist, and BPPageframe. The BPDirectory class is responsible for managing the buffer pool's data, and it includes a hashtable that maps prefixes to corresponding linked lists. Each linked list contains PageFrame nodes, which hold a single page's content along with metadata like SST name, page number within the SST, and a clock bit, among others.

BPDirectory is where most of the extendible hashing functionality is located, and it accepts parameters like eviction policy, initial prefix bit count for key entries, maximum buffer pool size before eviction, and maximum items allowed in a linked list chain before rehashing everything and doubling the size of the hashtable. Hashing is performed using the XXhash library, and the SST name and page number combination is used to ensure consistency.

To simplify memory allocation and deallocation during expansion, shared pointers were used between prefix entries pointing to the same linked list.

The buffer pool also holds Bloom filters, and as Bloom filters are inserted and removed, it will keep track of how much remaining memory is available and trigger eviction when necessary.

Eviction: BPDirectory::evict_until_under_max_bp_size();

The buffer pool has a maximum size. Once too many pages or Bloom filters are in the buffer pool and the maximum size is exceeded. The eviction policy kicks in to look for a page to evict. Once a victim has been identified, it is simply removed from the buffer pool and attributes of the buffer pool are updated accordingly.

Eviction: LRU Cache - LRUCache.h

If the eviction policy is set to LRU, the buffer pool will maintain an LRU cache of pages that is implemented using a doubly linked list whose nodes store pointers to the PageFrames. There are two cases for when a page is accessed. If the page is already in the buffer pool, it is moved to the front of the LRU cache. Otherwise, the page is inserted to the front of the LRU cache and the node at the end of the linked list is evicted.

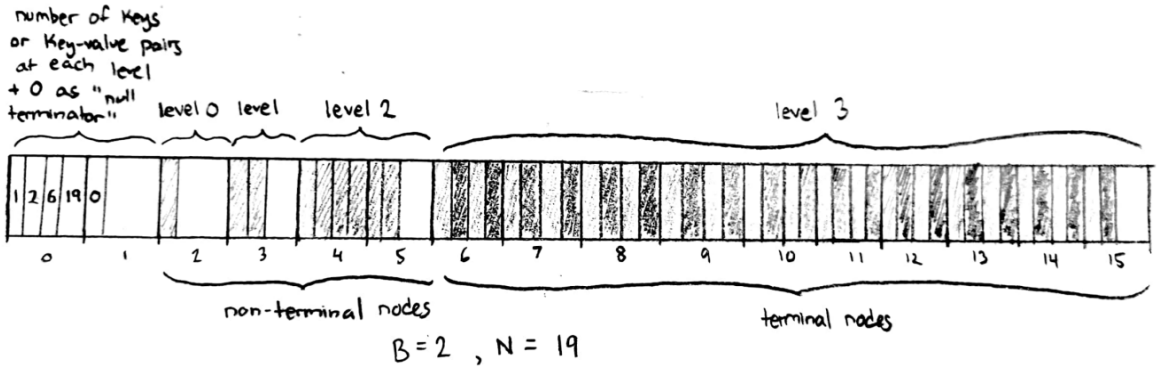
Eviction: Clock Bitmap - ClockBitmap.h, BPDirectory::clock_find_victim()

The clock eviction algorithm is similar to the FIFO (First In First Out) algorithm, but instead of simply removing the oldest page from memory, it checks if the page has been accessed since it was last used. To implement this, a binary clock bit is added to the PageFrame class: 1 meaning it has been recently used before, and 0 meaning it is set up for eviction. When a page is used, its clock bit gets set to 1. For eviction, there is a clock hand that moves around. When eviction is required, the clock hand traverses the buffer pool hashmap in a clockwise fashion until finding a page with a 0 clock hand, and setting that one as the victim, while setting the clock bit of all pages on the way to 0. If the clock hand reaches the end of the hash map, it automatically moves to the beginning and the process repeats.

B-Tree Creation: KeyValueStore::serialize()

B-trees are written to storage when the memtable's remaining capacity is strictly less than the size of one key-value pair. This is done by retrieving all of the key-value pairs in the memtable in sorted order via a scan through the entire AVL tree. These become the terminal (leaf) nodes. Using the number of key-value pairs, the height of the B-tree is calculated. Then, all of the fence keys are iterated over. On each iteration, the appropriate level in the B-tree for the fence key is calculated. These then become the non-terminal nodes. Additionally, various other metadata, namely the number of keys in each level (or key-value pairs for the last level) is also calculated. All of this data is then written sequentially to storage.

The resulting SST structure is as follows. The light bars represent keys and the dark bars represent values.



Data is stored as a clustered B-tree-based index. Because the B-tree is static, all of the nodes are packed into the B-tree without any padding, but with padding between levels so

that levels are page-aligned. Keeping track of the number of keys (or key-value pairs) in each level helps with B-tree traversal.

LSM Tree: KeyValueStore::compact_files()

The LSM tree is implemented as a collection of static B-trees in storage. SST files are named following the format “sst.<level>.<count>.bin”. For example, the first SST will be named “sst.1.1.bin”. The second SST will be named “sst.1.2.bin”. Because the size ratio is two, the compaction policy will merge the two SST files into another one named “sst.2.1.bin”.

Compaction is done in two passes. During the first pass, two input buffers read from the two SST files being compacted, and one output buffer writes the sorted data to a file called “temp”. If duplicate keys are encountered, the most recent key-value pair can be inferred from the name of the SST file. For example, sst.3.2.bin is more recent than sst.3.1.bin. If a tombstone is found, the tombstone will be kept in the final compacted SST file unless that SST file is in the last level.

During the second pass, the non-terminal nodes of the SST file and other metadata are reconstructed from the terminal nodes in “temp”. This data is then written sequentially to a new SST file, and the data in the temp file is appended to this same SST file.

Bloom Filter - BloomFilter.h

Every time a new SST is create/d, a Bloom filter for it is also put inside. Bloom filters are created during the serialize process of the memtable and during merging of SSTs. For the most part, Bloom filters are stored in the buffer pool. As Bloom filters are inserted and removed, the buffer pool will keep track of how much remaining memory is available and trigger eviction when necessary.

** Please note that to find the implementation of each design element in the code, simply search for the bullet point name. **

Project status

What works:

- KV-store get API (1)
- KV-store put API (1)
- KV-store scan API (1)
- In-memory memtable as balanced binary tree (1)
- SSTs in storage with efficient binary search (1)
- Database open and close API (1)
- Extendible hash buffer pool (1)
- Integration buffer with get (2)
- Shrink API (2)
- Clock eviction policy (2)
- LRU eviction policy (2)
- Static B-tree for SSTs (2)
- Compaction/Merge of two trees (3)
- Support update (3)
- Support delete (3)
- Bloom filter for SST and integration with get (3)

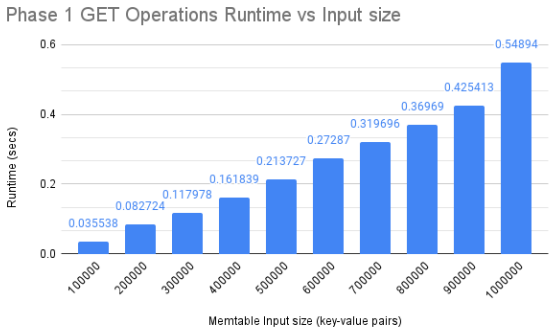
What doesn't work:

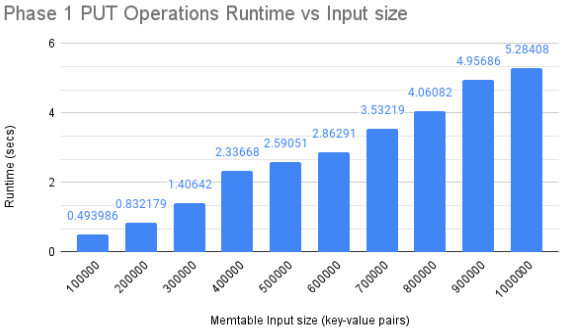
- Shrink API (2) - eviction works fine, but the prefix keys in the buffer pool are not reduced when the buffer pool size decreases
- Very slow when running Phase 3 experiments, we ran out of time so to generate the graphs, we had to reduce the size of the params

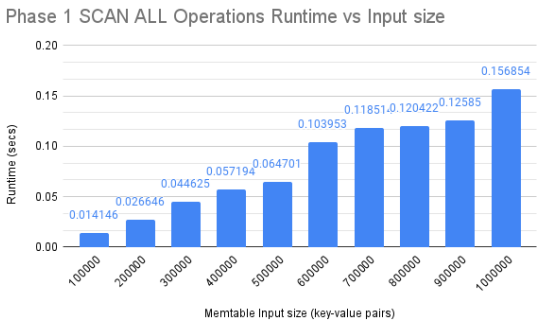
Experiments

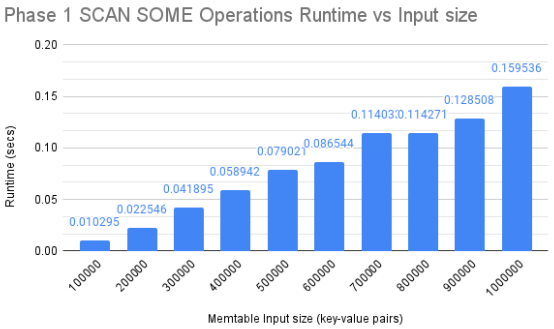
** Please note that the Phase 1 and Phase 2 experiments cannot be executed because the code undergoes changes that cause them to break once Phase 3 is implemented, due to the addition of new features since their execution. **

Phase 1









The experiment's x-axis measures the number of key-value pairs tested, while the y-axis represents the throughput of each operator in seconds.

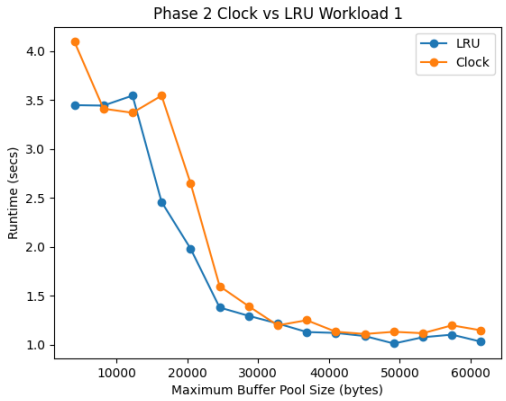
The "put" operation:
Regarding the "put" operation, we notice a gradual increase in throughput as more data is inserted because the system will need to perform more work to insert the data into the memtable. However, as the system approaches its capacity, throughput will decrease since the memtable management becomes more time-consuming, resulting in a slower overall performance.

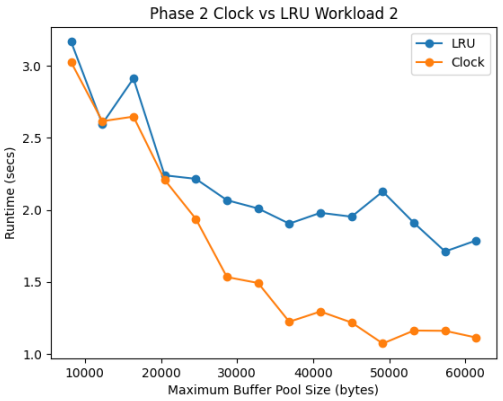
The "get" operation:
As for the "get" operation, we observe a gradual increase in throughput as more data is retrieved. It should be noted that this is a direct result of the increased workload and the need to retrieve more data from the system. Despite the workload expansion, the "get" operation is still quicker than the "put" operation due to the memtable's ability to efficiently retrieve data.

Regarding the "scan" operation, we measured the time taken for the system to retrieve all records ("SCAN ALL") and a certain set interval ("SCAN SOME") from the memtable. As more data is retrieved, we observe a gradual increase in throughput. Scanning a specific

range and the entire AVL tree was found to have similar runtimes since both required traversing to the tree's bottom layer, and sequential scanning was less expensive.

Phase 2 - Experiment 1





Workload 1:

For the first workload, we will randomly retrieve keys in a uniform distribution to test the performance of the clock algorithm. We expect the clock algorithm, also known as the second chance algorithm, to outperform LRU in this scenario.

The clock algorithm maintains a circular buffer of page frames with a clock bit associated with each frame. When a page needs to be evicted, the algorithm examines the page's clock bit and evicts it if the bit is 0. Otherwise, it sets the bit to 0 and moves to the next page frame until it finds a page with a clock bit of 0.

The clock algorithm is expected to perform better than LRU in a uniform distribution workload because it can efficiently identify pages that have not been accessed recently, regardless of their position in the buffer pool. The LRU algorithm, on the other hand, maintains a linked list of page frames and requires more computation to determine which page to evict.

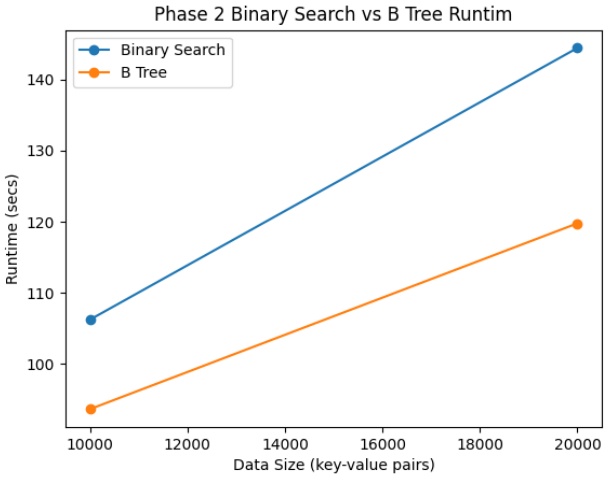
Workload 2:

For the second workload, we anticipate the LRU algorithm to exhibit superior performance compared to clock. To validate this, we will perform a sequence of operations where we will retrieve a single key 100 times, and then fetch distinct entries from the following 20 pages. The workload is specifically designed to favor LRU over clock, as the former algorithm can better identify and evict pages that have not been accessed recently.

In the LRU algorithm, page frames are maintained as a linked list, where the recently accessed page is kept at the top of the list and the least recently accessed page at the bottom. When a page needs to be evicted, the algorithm selects the page at the bottom of the list that has not been accessed in the longest time.

In this workload, LRU is expected to outperform clock because the majority of pages are not frequently accessed. As a result, LRU can efficiently identify and evict such pages, while clock would have to update the clock bit values for all pages repeatedly, leading to inefficiency.

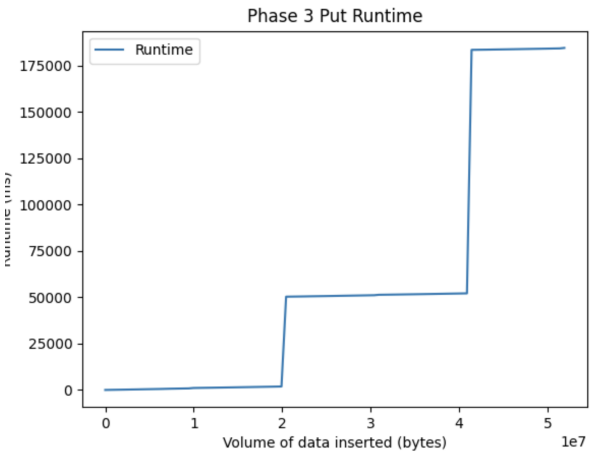
Phase 2 - Experiment 2



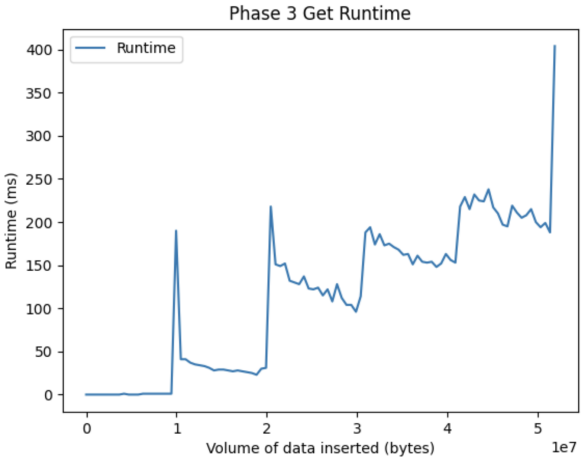
From this, we see that B-tree search is a lot more efficient in terms of runtime of 10000 get operations and 5000 random scan operations because it has fewer layers compared to binary search. B-trees use a hierarchical structure that allows for faster searches, whereas binary search requires more comparisons and takes longer to find the desired value. This means that as the data size increases, B-tree search will continue to perform better than binary search.

Phase 3 - Experiment 1

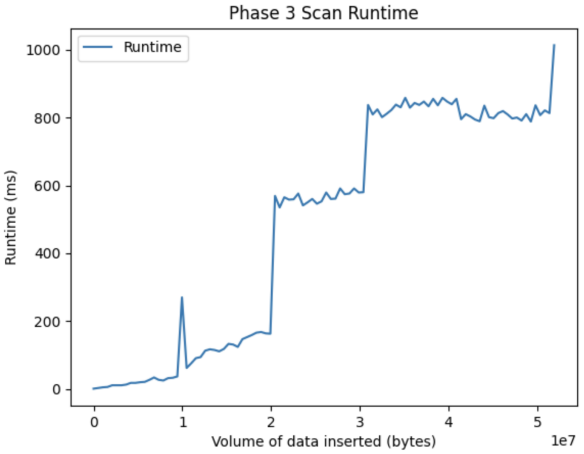
For this experiment, we fixed the buffer pool size to 10 MB, the Bloom filters to use 5 bits per entry, and the memtable to 1 MB. We inserted 5 chunks of data, and at the end of each chunk we recorded the time it took to put the chunk of data, and then we measured the total runtimes of 1000 random gets (first plot) and a scan of the entire database (second plot).



The results here seem to accurately reflect the LSM tree. Since get was run in chunks, runtime for the serialization process was amortized to appear flat. The large jumps that form the staircase can be explained by merging SSTs, which seem to take up very large amounts of time. The plot is increasing as expected because more data needs to be traversed when larger SSTs are merged.



There are a few spikes. When SSTs are first merged/serialized, many of the pages we need to access may not be in the buffer pool. However, once the frequently used pages are loaded into the buffer pool, the average runtimes decrease dramatically until we move onto the next SST. The plot is increasing as expected because more SSTs need to be searched when there is more data.

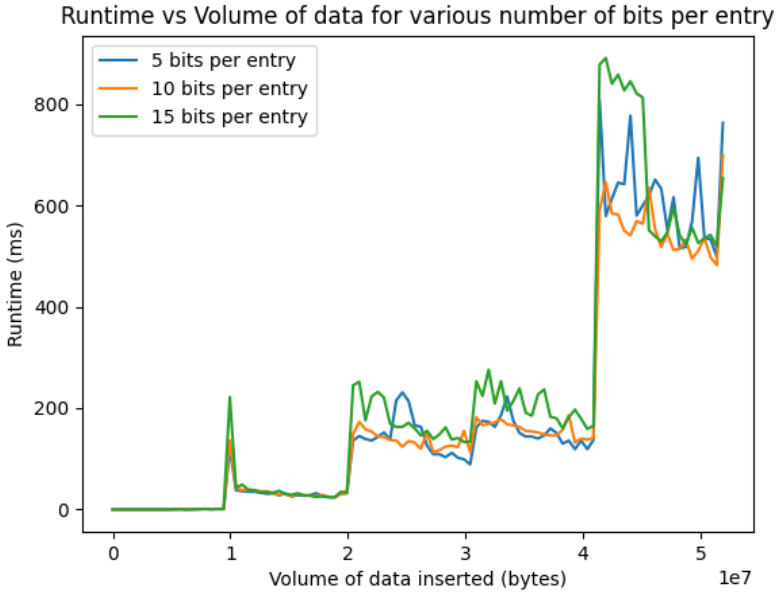


The first spike happened when the volume of data exceeded the memtable size. The second spike happened when the volume of data exceeded the combined size of memtable and buffer pool. The plot is increasing as expected because more SSTs need to be traversed when there is more data.

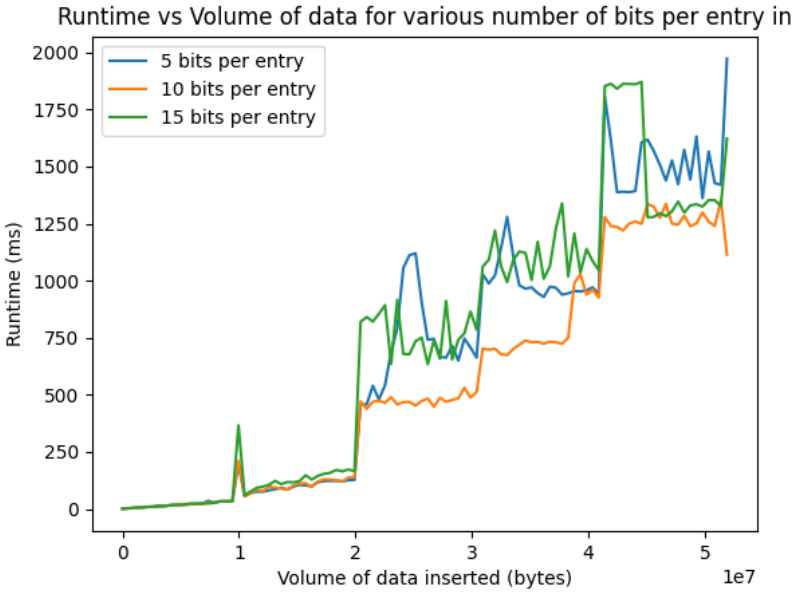
Phase 3 - Experiment 2

For experiment 2, we repeated experiment 1 four times using 2, 5, 10 and 15 bits per entry for Bloom filters. The shapes of the plots for get are understandably similar to experiment 1. However, we found that changing bits per entry had little to no effect on the average runtime.

This is likely because 5 bits can already produce a low positivity rate so any more bits had little effect. When the number of bits per entry is increased, the size of the filter also increases, which means that more memory is required to store the filter. This can lead to slower performance due to increased memory access times.



This next plot shows scan throughput. Interestingly, having 10 bits per entry results in noticeably better performance. We're not exactly sure why this is the case. While increasing bits per entry will result in increased memory usage and yielding less space for the buffer pool to store pages, having false positives can be detrimental to performance since searching a SST is very slow. Maybe 10 bits per entry is a sweet spot.



Testing

Testing is a crucial aspect of software development. In order to ensure the reliability and correctness of our implementation, we designed and executed a comprehensive suite of tests that covered all aspects of the system.

We mostly focused on integration testing, which involved testing the interactions between the different components of the system. We divided the integration testing into several files, each targeting a specific phase of the project. The main goal was to test the basic functionality of the database, including key-value storage and retrieval, scanning, checking the contents of the output SSTs before/after compaction, ensuring that the buffer pool is expanding correctly, etc. We found that as we complete each subsequent phase, the code broke the previous phases' test cases. As we progressed, we made sure that the testing suite was modified so that it passes while preserving the original intention of each test case. Throughout the process, we ensured that our tests were comprehensive, covering a wide range of scenarios and edge cases. Please check out our test cases in the phase*_tests.cpp files.

Compilation & Running Instructions

Please refer to README.md for the make file targets and examples on how to interact with the database.