CSC443 Project Report

Greg Sherman, Murphy Tian, Patrick Zhou

December 9, 2024

Contents

1 Note

Please note that we were granted an extension by the professor, allowing us to submit on December 9th.

2 Instructions

2.1 Setup

Clone the Repo from https://github.com/realtmxi/csc443

2.2 Run & Test

Within the main file there is a sample workflow as well as the experiments from part 1 and 2. Use `make` and `./main` to run the sample.

To run the tests, simply use `make test`, which will compile and run all tests.

3 Project Status

What works:

- The Buffer Pool core function is fully implemented
- The core Memtable is fully implemented as an AVL Tree
- The Database API (Put, Get, Scan, Delete, Open, Close) is fully implemented
- The SST is fully implemented as a BTree. This BTree has a set branching factor.
- The LSM Tree core function is fully implemented. BTrees merge when 2 are in the same level.
- The Bloom Filter core function is fully implemented.
- BTrees support Gets using either BTree search or Binary search on the leaves.

What doesn't work:

- Every bonus is not implemented.
- The SST from Part 1 has been completely overwritten. That is, for each KV-pair, they exist within a BTreePage object, thus not being contiguous in storage. The compromise for Binary searching over the BTree leaves was to deserialize a page per hop, check the min/max keys, then determine where to go from there.
- Although not required, we did not implement removing specific pages from the Buffer Pool during a merge step. We opted to wait for it to evict naturally.

4 Design

4.1 Step 1

4.1.1 KV-Store & Database API

There are a few key functions of the Database API.

Put - Simply calls insert on the AVL Tree Memtable.

Get - Calls get from the Memtable first, then iterates through the SSTs, checking the buffer pool if there

are any relevant pages.

Scan - Keeps a running list of KV-pairs. First scans the memtable, then the SSTs, checking the buffer pool if there are any relevant pages. Once the entire scan range is found, return early.

Delete - Puts a INT_MAX value for the key. This is propogated down the LSM Tree.

Close - Stores the in-memory memtable and its related Bloom Filter to storage.

Open - Takes in a name as a parameter. Checks for the folder in storage and if it exists, load in the SSTs and Bloom Filters.

(private) StoreMemtable - Converts the Memtable to a BTree when full and stores it as the SST. Then calls Compact.

(private) Compact - Checks if there are 2 SSTs on the same level of LSM Tree. If so, perform a BTree merge operation into a new level. Repeat until no 2 levels are the same.

4.1.2 In-Memory Memtable

For our memtable, we used a basic AVL Tree implementation to store the keys and respective values. This tree auto-balances on insert. Each node of the tree is an AVLNode struct containing its key, value, children, and height.

4.1.3 SST Storage

To store the memtable as an SST, we first run a Scan operation from INT_MIN to INT_MAX to get a sorted list of KV pairs. Then we simply loop through each one and write it to a file.

To perform Get operation on the SST, we first get the size of the file to determine the location of the middle KV pair (the middle-most 8 bytes). We then open a file descriptor and read the key, determine the offset of the next midpoint, and continue until we find the correct key (or none at all).

To perform a Scan operation on the SST, we do the same binary search logic as the Get operation to find a the earliest key within the range, then we simply read each KV pair sequentially to the right and construct the sorted list.

4.2 Step 2

4.2.1 Buffer Pool Implementation

The Buffer Pool employs an LRU cache, defaulting to 10MB of memory (roughly 2560 pages). It is a simple implementation:

- The key to access a page is in the form of `filename+page_id`. For example, the key for the 26th page in `sst_0004_12456326.sst` is `sst_0004_12456326.sst26`.
- This key is used in the `page_table_` map. If found, the page is returned to the caller after moving it to the front of the LRU
- If not found, the Buffer Pool accepts an injected `loadPageFromDisk` function to load and return the page from within the buffer pool itself.
- If a loaded page is found, add it to the front of the buffer pool with that key, evicting a page if necessary.

The `GetPageFromId` function is called every single time a page is about to be loaded from storage. This includes every Get and Scan in both BTree Search and Binary Search.

4.2.2 Eviction Policy

Since it is an LRU cache, the eviction policy is just least recently used. When a page is queried through `GetPageFromId`, the page is added to the front of the LRU and a new key is made and added to the map pointing to the page. If the page table is full, we remove the Least Recently Used page, as identified by the last element in `lru_list_` (and also this last element).

4.2.3 Static B-Tree

The design of the BTree in storage is as follows:

Root (4KB)	Page Type (4B)	Page Size (4B)	Key 1 (4B)	Child Page ID (4B)	...	Padding (0-4KB)
Internal 1 (4KB)	Page Type (4B)	Page Size (4B)	Key 1 (4B)	Child Page ID (4B)	...	Padding (0-4KB)
Internal N (4KB)	Page Type (4B)	Page Size (4B)	Key 1 (4B)	Child Page ID (4B)	...	Padding (0-4KB)
Leaf 1 (4KB)	Page Type (4B)	Page Size (4B)	Key 1 (4B)	Value (4B)	...	Padding (0-4KB)
Leaf N (4KB)	Page Type (4B)	Page Size (4B)	Key 1 (4B)	Value (4B)	...	Padding (0-4KB)

The Page Type of the root and internal nodes are 1 whereas the leaf nodes are 2. The Page Size refers to the number of KV Pairs in the node. These are used to properly read the pages from storage. Since these pages are continuous in storage, the first 4KB (Root) has a Page ID of 0, the next (internal node) has Page ID 1, and so on.

To write the BTree upon closing the database or filling up the memtable:

1. Create a BTreePage object for each set of pairs from the memtable that fill a page (in this case, 510 KV pairs fill a 4KB page + metadata).
2. Classify these pages as (sorted) leaves in the BTree object.
3. Using the maximum values from each leaf page, create BTreePage objects classified as (sorted) internal nodes.
4. Repeat the above step over the new internal nodes to create the next layer higher, until only 1 node exists in the layer (the root node)
5. Loop through the internal nodes, pointing the children to the correct page number. Use the number of internal nodes in a given layer to determine what Page ID the next layer starts on.
6. Finally, write the pages to storage top to bottom. The filename is in the form "sst_0000_[timestamp].sst", which is level 0 in an LSM tree (as all memtable writes are).

To perform a Get query on a single BTree SST using BTree Search:

1. Load the first 4KB (root) and do a binary search on the KV pairs to find the closest key greater than or equal to the desired key.
2. Take that Child Page ID and multiply it by 4096 to find the start byte location of the child page in storage.
3. Repeat until the loaded page is a leaf.
4. Perform one last binary search to find the exact key, then return the value or -1.

To perform a Get query on a single BTree SST using Binary Search:

1. Check the size of the file and divide by 4096 to get the number of pages
2. Access the middle page and deserialize it
3. Check the max and min key of this page. If the key to search for is within the range, perform a binary serach wthin the page itself. Otherwise, continue the binary search.

4. return -1 if no key is found within a page or no pages are left.

To perform a Scan query on a single BTree SST:

1. Load the first 4KB (root) and do a binary search on the KV pairs to find the closest key greater than or equal to the start range and less than the end range.
2. Take that Child Page ID and multiply it by 4096 to find the start byte location of the child page in storage.
3. Repeat until the loaded page is a leaf.
4. Perform a binary search to find the smallest valid key. Then read each key after that is less than the end range.
5. If the page is exhausted, simply load the very next page in memory and repeat. Do this until the first key that is out of bounds. Return the list of pairs.

4.3 Step 3

4.3.1 Compaction

To compact two SSTables together, it needs to be triggered at the correct time. The trigger for a compaction happens immediately after saving the SST to storage and appending the filename to our SSTFiles list in our Database object. Since the SST file names are formatted as "sst_[level]_[timestamp].sst" (for example "sst_0002_12345678.sst"), an invariant forms. That is, the SSTFiles are sorted from highest level to lowest level. This is because no compaction can occur for a level without first compacting previous levels. So the compaction process checks the last 2 filenames to see if they are the same level and if so: compacts them, adds the new SST filename to SSTFiles, then repeats until no more compactions occur.

Since our SST is formatted as a BTree, the merging process will merge 2 BTrees together and output a new one. To do this:

1. Read in the first page of each BTree and traverse until we get to the very first Leaf page of each BTree.
2. Create a KV buffer and Max Keys list. Iterate through each page, comparing each key.
3. If one key is less than the other, add the KV to the buffer and increase that page's iterator.
4. If they keys are the same, prioritize first iterator as it is the more recent BTree.
5. If the value is a tombstone AND this merge creates a new level, skip this key entirely.
6. Once a page has been exhausted, immediately break and load in the next page.
7. Once the KV buffer has reached its limit, create a BTreePage object, categorize it as a leaf, and write it to the temporary leaf page file. Add the max key of this page to the Max Keys list.
8. Repeat until there are no more leaf nodes in both BTrees.
9. Create the internal nodes using the same method from 3.2.3, write these internal nodes to a file.
10. Combine the two files and increase the level.

4.3.2 Bloom Filters

Bloom filters are integrated into our system to optimize the performance of the `Get` operation and reduce unnecessary disk I/O by providing a probabilistic check for key existence.

Design and Integration with `Get` The Bloom filter is implemented as a bit array, along with multiple hash functions. During the `StoreMemtable` process:

- A Bloom filter is created for each SSTable with a fixed size and number of hash functions.
- For every key-value pair stored in the SSTable, the key is hashed multiple times, and the resulting indices are set in the bit array.

When a `Get` query is issued:

- The Bloom filter is first consulted to determine if the key is likely to be present in the SSTable.
- If the Bloom filter returns a negative result, the query skips searching that SSTable, thus avoiding unnecessary I/O.
- If the Bloom filter returns a positive result, the query proceeds to perform a binary search within the SSTable.

This integration ensures that the system can efficiently filter out non-existent keys, reducing the number of SSTables accessed during a query.

Persisting Bloom Filters in Storage To maintain the integrity of Bloom filters across system restarts and compacted SSTables:

- Each Bloom filter is serialized to disk alongside its corresponding SSTable.
- The serialized data includes the size of the bit array, the number of hash functions, and the bit array itself.
- Upon loading an SSTable into memory, the Bloom filter is deserialized from the stored metadata file.
- This ensures that the Bloom filter is always synchronized with its SSTable and can be used immediately after database initialization.

Advantages of Bloom Filter Integration

- Significant reduction in disk reads for non-existent keys.
- Low memory overhead due to the compact bit array representation.
- Fast query time for Bloom filter checks, as only hash computations are required.

Overall, Bloom filters are a critical component in optimizing the `Get` operation, improving query throughput while maintaining a simple and effective design for persistence and integration.

5 Testing

The tests are all located in `test/tests.cpp`. There are a variety of unit tests and integration tests. Running `make test` will compile and run the tests, providing a pretty output of the tests that passed and failed.

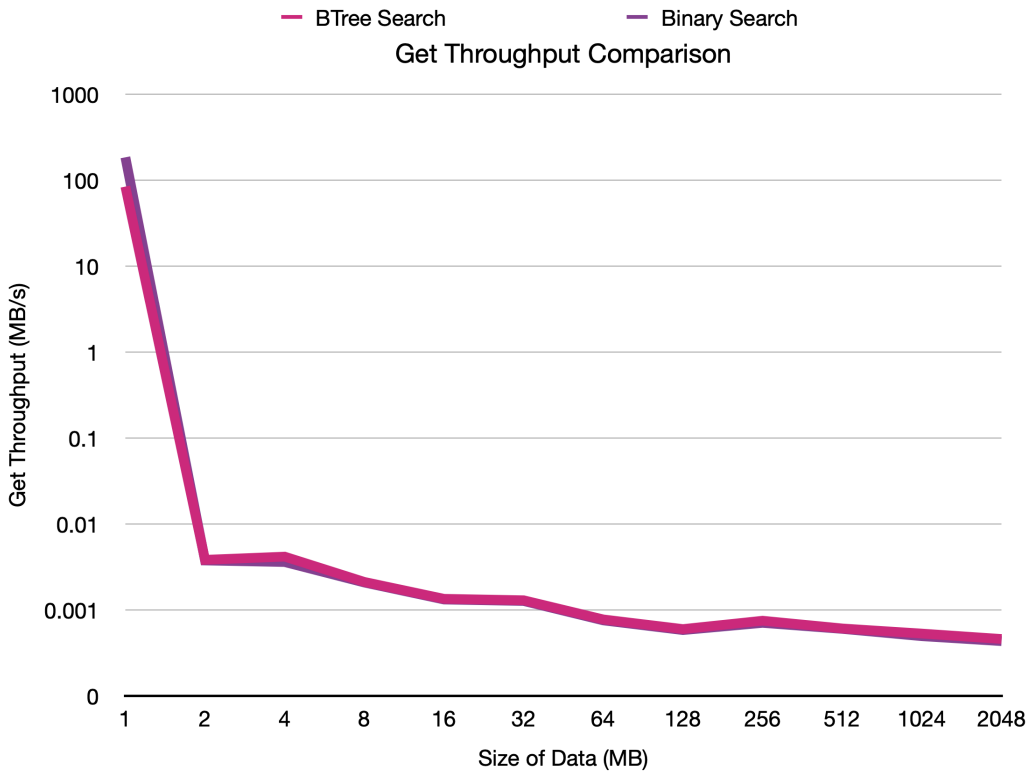
6 Experiments

6.1 Binary Search V. B-Tree

This experiment used 2GB of input data, 10MB Buffer Pool size, 1MB Memtable size, and 8 bits per entry Bloom Filter. We captured data in milestones of powers of 2 MB.

To compare the 2 search types, we created 2 separate databases. One uses BTree Search and the other

uses Binary Search. Every power of 2MB inserted, we took measurements of 100 gets and stored the throughputs.



In our implementation, it appears as though the BTree search is only slightly but consistently faster than the Binary Search (other than the first value). It makes sens that BTree search is faster, but I would have expected a greater disparity.

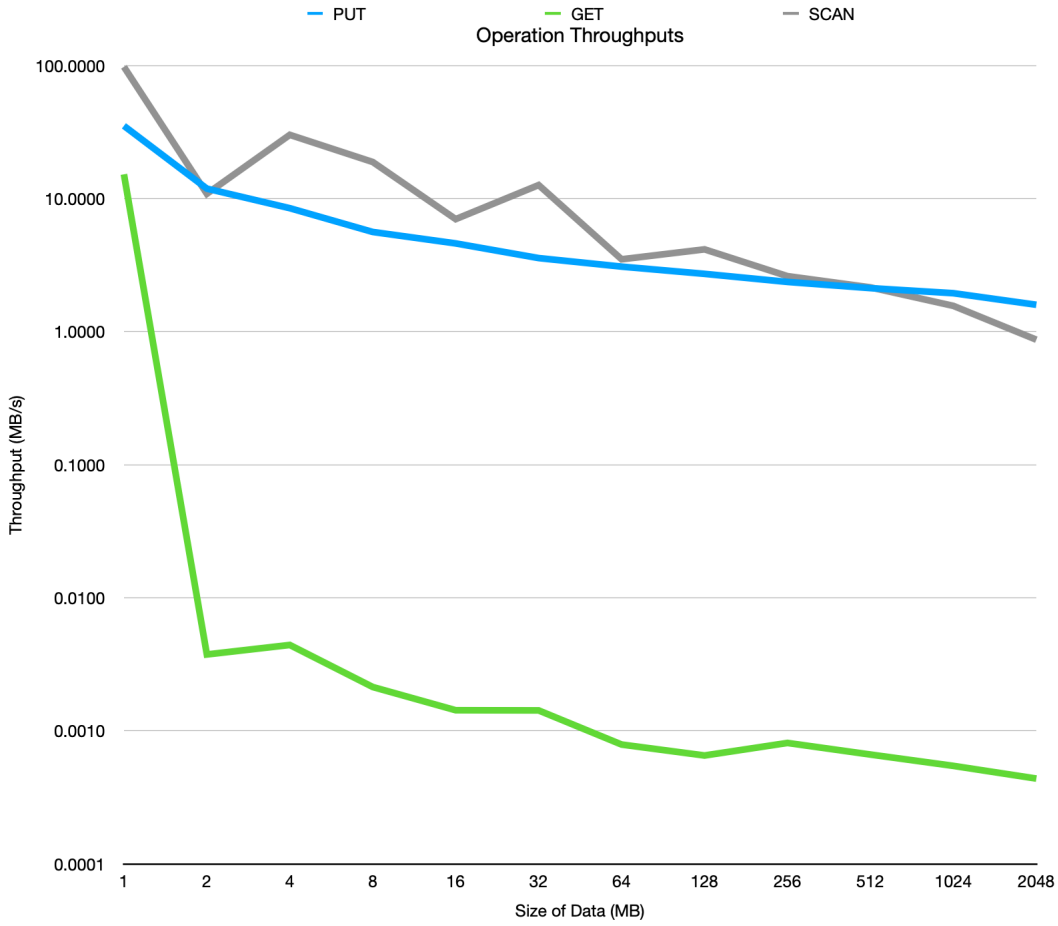
6.2 Put, Get, & Scan Throughput

This experiment used 2GB of input data, 10MB Buffer Pool size, 1MB Memtable size, and 8 bits per entry Bloom Filter. We captured in milestones of powers of 2 MB.

For Puts, we calculated the throughput for 0-1MB, then 1-2, then 2-4 etc.

For Gets, every milestone we performed 100 gets and calculated the throughput using that.

For Scans, every milestone we performed 100 scans for 100 size scan range and calculated the throughput.



Our findings show that the average Put throughput decreases relatively slowly. The reason it remains so fast is because the merge step of LSM compactions take longer as the data size grows. So the individual put operation is not slowing down, but the average.

The Scan throughput also remains relatively high. This is because it performs the same search as the Get, but takes up to 100 more data points in one go, thus having 100x more throughput.

The Get throughput has an obvious drop once SSTs are in play, but the decrease slows down and remains fairly flat.