CSC 2234 Database System Technology
Project Report

Name: Kiiro Huang
Student Number: 1011781957
GitHub Account: FearlessLugia

December 2024

Contents

1 Coding Practices

This project follows the Google C++ style guide.

1.1 Naming Conventions

The naming conventions adhered to in this project are outlined below:

- **File Names:** All lowercase and include underscores: `memtable.cpp`, `buffer_pool.cpp`
- **Type Names:** Use PascalCase (start with a capital letter and capitalize each new word without underscores): `Database`, `BufferPool`
- **Variable Names:** Use snake_case (all lowercase, with underscores between words): `memtable`, `db_name`
- **Class Data Members:** Follow the same format as nonmember variables but include a trailing underscore: `memtable_`, `db_name_`
- **Constant Names:** Begin with a leading `k` followed by PascalCase: `kMemtableSize`, `kPageSize`
- **Function Names:** Use PascalCase: `SSTable::GetFileSize`

1.2 Folder Structure

```
.
├── build
├── docs // this report
├── experiments
│   └── experiment.cpp // experiments main function
├── external // MurmurHash3
├── include // header files
│   ├── b_tree
│   ├── buffer_pool
│   │   └── lru
│   └── lsm_tree
├── src
│   ├── b_tree
│   │   └── b_tree_sstable.cpp
│   ├── buffer_pool
│   │   └── lru
│   │   │   └── lru.cpp
│   │   └── buffer_pool.cpp
│   └── lsm_tree
│       └── lsm_tree.cpp
├── database.cpp
├── memtable.cpp
├── sst_counter.cpp
├── sstable.cpp
├── tests
│   ├── test_b_tree.cpp // unit tests for b-tree
│   ├── test_base.h
│   ├── test_buffer_pool.cpp // unit tests for buffer pool
│   ├── test_db.cpp // integration test
│   ├── test_lsm_tree.cpp // unit tests for lsm-tree
│   └── test_runner.cpp // tests main function
├── utils
│   ├── contstants.h // immutable constants
│   └── log.h // log macro
│
```

```
├── .clang-format
├── .clang-tidy
├── .gitignore
├── CMakeLists.txt
├── experiment_Get.csv // csv output for get throughput
├── experiment_Put.csv // csv output for put throughput
├── experiment_Scan.csv // csv output for scan throughput
├── get_plot.png // get throughput figure
├── plot_generator.py // python script to read csv and plot
├── put_plot.png // put throughput figure
├── readme.md
├── run_experiments.sh // script to run experiments
├── run_tests.sh // script to run tests
└── scan_plot.png // scan throughput figure
```

1.3 Version Control

1.3.1 Git Workflow

As this project is developed by only one member (me!), branches and pull requests are not utilized for code pushes to maintain simplicity.
However, `git rebase` is extensively used to ensure a linear, clean commit history.

1.3.2 Git Commit Messages

The types of Git commit messages and their respective purposes are outlined below:

- **feat:** Introduction of a new feature.
- **fix:** Bug fixes or patches.
- **chore:** Changes not related to the code itself, such as log macro.
- **refactor:** Refactoring production code without adding new functionality.
- **docs:** Documentation.
- **project:** Engineering-related changes, such as updates to Git ignore files or development environment configurations.
- **style:** Code style changes, such as formatting or fixing missing colons, without modifying code logic.
- **test:** Addition of new test cases or refactoring test code without altering production code.

1.4 Logging

A macro named `LOG` is used to handle logging functionality in this project. See `utils/log.h`.

- **Debug mode:** When debugging, log messages are displayed to assist in tracking program execution and identifying issues.
- **Release mode:** During test/experiment execution, log messages are suppressed to maintain clean and focused outputs.

1.5 Test-Driven Design

Test-Driven Design (TDD) is employed in this project to ensure robust and maintainable code. This approach helps to catch bugs early, clarify requirements, and promote a clean and modular design.

2 Compilation & Running Instructions

2.1 Run Experiments

Run the following command to execute the experiments. It may cost some time to run the experiments.

```
$ ./run_experiments.sh
```

The `csv` outputs and the plots will be generated in the `build` folder.

Additionally, a sample result is also put in the project folder, holding the names `experiment_Get.csv`, `experiment_Put.csv`, `experiment_Scan.csv`, and `get_plot.png`, `put_plot.png`, `scan_plot.png`.

2.2 Run Tests

Run the following command to execute the tests.

```
$ ./run_tests.sh
```

3 Description of Design Elements

3.1 Step 1: Creating a Memtable and SSTs

3.1.1 Memtable

In this project, keys and values are both 8-byte integers, which are of type `int64_t` in C++. So, 1 key-value pair occupies 16 bytes.

I create the `Memtable` implemented by std::map, given the dispensations for smaller groups. It takes a parameter called `memtable_size_`. This size is in bytes. When the `Memtable::Size` reaches `memtable_size_`, memtable is full and it flushes from `Memtable::Traverse` to SSTable, and then `Memtable::Clear` the memtable.

See `Memtable.cpp`, `Memtable.h`.

3.1.2 Put API

The `Database::Put` API first puts key-value in the memtable. When memtable reaches its maximum size, it flushes to SST and clears itself.

3.1.3 Get API Workflow

The `Database::Get` API first `Memtable::Get` in the memtable. If not found, it searches in the SSTs from the youngest to the oldest.

The `SSTable::Get` function has a helper function called `SSTable::BinarySearch`. This function first uses `SSTable::GetPage`, where inside it uses `pread`, to read a complete page. Then, it uses a binary search to locate the page that contains the exact key. When the page is found, it is feasible to do a simple linear search to locate the exact key, which are sequential read. However, since the key is already in order, I do another binary search inside the page to search for the key.

3.1.4 Scan API Workflow

Like `Database::Get` API, `SSTable::Get` and its helper function `SSTable::BinarySearch`, `Database::Scan` API also has `SSTable::Scan` and `SSTable::BinarySearchUpperbound` as a helper function. It first scans in the memtable, and then scans in the SSTs from the youngest to the oldest.

This helper function also uses a binary search (but an upper bound version) to locate the page that contains the upper bound of the start key. When the page is found, do another binary search inside the page to get the upper bound of the start key. After finding the upper bound of the start key, do a linear search in this SST to find the lower bound of the end key.

See `Database.cpp`, `Database.h`.

3.1.5 SSTable

As SST is immutable, when flushing memtable to SST, it always writes in a newly created SST.

The SSTs should be assigned unique names. This can be done by using the UUID, ensuring more robustness. But in this project, I only use an increased number for simplicity. During this step the names are like `sst0.bin, sst1.bin`. These namings will later be turned into `btree0_0.bin, btree0_1.bin` for LSM-Tree, to represent a hierarchic structure.

As shown in previous binary search functions, it is important to know whether the key is inside this SST or not. If the maximum key is smaller than the target key, or if the minimum key is larger than the target key, there is no need to scan in this SST. So, I record the `max_key_` and `min_key_` when doing the flush. It helps to pruning when doing the search in multiple SSTs.

See `SSTable.cpp`, `SSTable.h`.

3.1.6 Open API

The `Database::Open` API accepts a string `db_name` as the name and the file path of the database. If the directory already contains SSTs, a singleton `SSTCounter` will read the current number of SSTs and store it. This step will later be changed into LSM-Tree.

See `SSTCounter.cpp`, `SSTCounter.h`

3.1.7 Close API

The `Database::Close` API closes the database. It flushes the current contents of the memtable into a new SST and clears itself. It also completes some deconstruction stuffs.

3.2 Step 2: Buffer Pool & Static B-trees

3.2.1 Buffer Pool Workflow

A buffer pool stands for storing the hot pages. I use the `SSTable::GetPage` function as the main start point for other parts in this project to retrieve a page. Inside this function, it first invokes `BufferPool::Get` to see if the page is in the buffer pool. If it does not hit in the buffer pool, it does an I/O to retrieve the page from storage, and then put this page in the buffer pool by invoking `BufferPool::Put`.

In `BufferPool::Put`, if buffer pool is at the threshold (`kCoeffBufferPool` set to be 0.8 in `utils/constants.h`), it applies the eviction policy.

A singleton class `BufferPoolManager` is used to maintain the only one buffer pool in the database.

See `buffer_pool.cpp`, `buffer_pool.h`, `buffer_pool_manager.h`.

3.2.2 Hash Table Data Structure

The data structure of the buffer pool hash table is of buckets. I use `BucketNode` as the element of buckets in buffer pool. Inside the `BucketNode` there is the `Page`.

The `Page` has a string type `id_`, which is the SST name plus the offset, like `btree0_0_0`. I use `vector<int64_t>` as the type of `data_`.

See `bucket_node.h`, `buffer_pool.h`, `page.h`.

3.2.3 Collision Resolution

I use chaining to handle collision in a hash table. For each `BucketNode`, a pointer called `next_` is set to point to the next bucket.

If the chaining is full, it should evict until the chain has space. So when reaching 80% the threshold, it should start to evict.

3.2.4 LRU Eviction Policy

As for the eviction policy, I apply the LRU policy. I also use a virtual class called `EvictionPolicy`, and the class `LRU` inherits it. This allows more evolvability in the future implementation of other eviction policies.

Diving deep into LRU, I use `QueueNode` as the data structure. Each `QueueNode` has 2 pointers called `prev_` and `next_` to represent the doubly-linked list. For class `LRU`, I maintain `front_` and `rear_` to get the front and the rear of this doubly-linked list.

See `queue_node.h`, `lru.cpp`, `lru.h`, `eviction_policy.h`.

3.2.5 Hash Function

As for the hash function `BufferPool::HashFunction`, I use the murmur hash, which I put in the folder `external`.

See `MurmurHash3.cpp`, `MutmutHash3.h`.

3.2.6 Bonus: Sequential Flooding

I set buffer pool size for 32KB, where each buffer pool contains 8 pages. I set a parameter called `kCoeffSequentialFlooding` as 0.25. When the key range of a `SSTable::Scan` covers over 2 pages, it will apply the sequential flooding. The pages covered during this scan will not be put into the buffer pool.

3.2.7 Structuring an SST as a B-Tree

The `BTreeSSTable` inherits `SSTable`, and it overrides some of the functions to support B-Tree index, including `BTreeSSTable::InitialKeyRange`, `BTreeSSTable::BinarySearch`, `BTreeSSTable::BinarySearchUpperbound`, and `BTreeSSTable::LinearSearchToEndKey`.

The B-Tree SST has 3 layers, the root, the internal nodes, and the leaf nodes. The leaf nodes are exactly the same as the `SSTable`. Performing binary search on static B-Tree implies performing binary search on its leaf nodes using binary search.

In `BTreeSSTable::FlushToStorage`, I first calculate the number of reserved pages for the first 2 layers of the B-Tree, as the root occupies a page, and every internal node in layer 2 occupies a page. Then I write the data to the B-Tree leaf nodes page by page using `BTreeSSTable::WritePage`. After that I write the first 2 layers' nodes, also using `BTreeSSTable::WritePage`.

As for constructing the first 2 layers in `BTreeSSTable::GenerateBTreeLayers`, I set `kFanOut` to be number of the key-value pairs in one page, which is 256. So every root and internal nodes can contain 256 keys in maximum.

During the written of the B-Tree SST, only internal nodes are kept in memory, and leaf nodes are written to disk while being constructed. This method can save memory, avoid random writes, improve query performance, and facilitate subsequent LSM-Tree merging steps.

When reconstructing B-Tree SSTs from storage, it first uses `BTreeSSTable::ReadOffset` to get the first page of the B-Tree SST, and then know the offset where leaf nodes start.

See `b_tree_sstable.cpp`, `b_tree_sstable.h`.

3.3 Step 3: An LSM-Tree

3.3.1 Bonus: Compaction Implementation using Min-Heap

Implementing Dostoevsky requires compacting data from across multiple runs in a given level at the same time. I use `priority_queue` to be the max-heap, and set the third parameter `Comparer` as `greater<>` to make it a min-heap.

The `LsmTree::SortMerge` function accepts multiple BTreeSSTs from a given level and compact it to a new BTreeSST. I use struct `HeapNode` as the element in the min-heap. The function first `BTreeSSTable::ReadOffset` to get the offset of leaf nodes of all SSTs. Then, it continuously `SSTable::GetPage` to retrieve data page by page, and push key-value pairs in the min-heap. Every time the min-heap pop an element, append it to the output buffer, and push the next key-value pairs from the same SST.

See `lsm_tree.cpp`, `lsm_tree.h`.

3.3.2 Bonus: LSM-Tree using Dostoevsky

I set 2 constants for LSM-Tree, `kLsmRatio` for the fixed size ratio between two levels of LSM-Tree, and `kLevelToApplyDostoevsky` for the start level to apply Dostoevsky. For example, I set `kLsmRatio` as 3 and `kLevelToApplyDostoevsky` as 4, meaning that level 0 needs 3 SSTs to merge, level 1 needs 9 SSTs to merge, level 2 needs 27 SSTs to merge, ..., and level 4 needs 2 SSTs to merge.

Every time a new SST is flushed, `LsmTree::OrderLsmTree` should be invoked. For the previous levels, it would invoke `LsmTree::SortMergePreviousLevel`, and `LsmTree::SortMergeLastLevel` at the last level instead. In these functions, they first use LsmTree::SortMerge to do the multiple compaction using min-heap. Then they clear the current level, generate a new SST in storage, and append new SST nodes to the new level or to the current level.

For the previous `SSTCounter` part, now it requires to `LsmTree::ReadSSTsFromStorage` to read SSTs, getting the name of SSTs and sort them per level. Then it invokes `SSTCounter::SetLevelCounters` to set the SST counter.

3.3.3 Bonus: Integration with the Buffer Pool & Immediate Eviction

In `LsmTree::SortMergePreviousLevel` and `LsmTree::SortMergeLastLevel`, after deleting the SSTs that need to be compressed, `BufferPool::RemoveLevel` immediately evicts these pages to clear space in the buffer pool.

3.3.4 Support Deletes for LSM-Tree

The `Database::Delete` API is used to set tombstone in memtable, where `Memtable::Delete` puts `(key, INT64_MIN)` to the memtable.

3.3.5 Current Get Workflow

Currently, the `Database::Get` API first `Memtable::Get` in the memtable. If not found, it searches in the LSM-Tree, where BTreeSSTs are levelled. It searches from the lowest level to the deepest level. In each level, it searches from the youngest SST to the oldest SST.

If anytime it gets value of `INT64_MIN`, that means the key is deleted and the API should return `nullopt`.

3.3.6 Current Scan Workflow

Currently, the `SSTable::Scan` API first `Memtable::Scan` in the memtable. Then, it scans the LSM-Tree, where BTreeSSTs are levelled. It scans from the lowest level to the deepest level. In each level, it scans from the youngest SST to the oldest SST.

When performing `Database::Scan`, if it gets value of `INT64_MIN`, that means the key is deleted and this value should be skipped.

4 Project Status

All required steps for the project have been completed, along with the implementation of the following bonus features:

- **Handling Sequential Flooding**
- **Dostoevsky**
- **Min-Heap**
- **Immediate Eviction after Compaction**

Additionally, following dispensations for smaller groups have been applied:

- `std::map` is used for the memtable, as permitted.
- B-tree search is not performed on the static B-tree structure for SSTs.
- Bloom Filters are not implemented.

There are no known bugs in my code.

5 Experiments

5.1 Notes

5.1.1 Binary search index with query throughput with changing data size

This experiment in step 2 can be performed at the end of step 3. When only performing binary search on B-Tree, Experiment 2 is exactly the same as that of the Get Throughput in Experiment 3.

5.2 Experimental Setup

Experiments were run with the following configurations:

- MacBook Pro 2023
- Apple M2 Pro
- Memory: 16 GB
- macOS: Sequoia 15.1.1

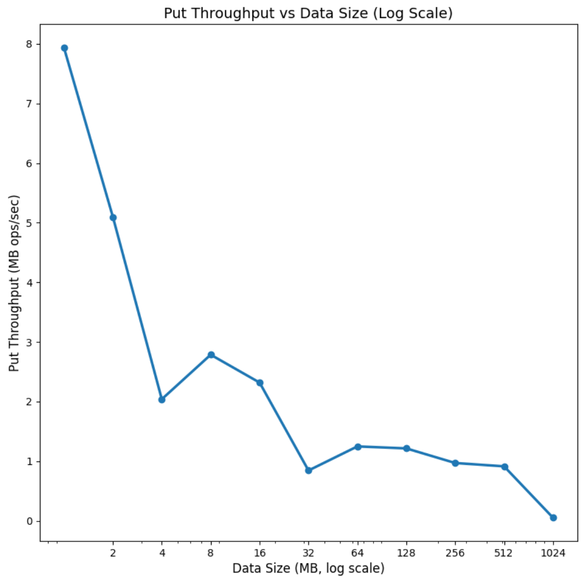

Figure 1: Put Throughput vs Data Size

Experiment configurations:

- Memtable size: 1 MB
- Buffer pool size: 10 MB
- Insert data size: 1 GB
- Query count: 1000 times

Each time the inserted data size reaches a power of 2 (from 1 MB, 2 MB, up to 1024 MB), record the time taken for put, run the queries of get and scan operations, record the time taken for these two operations, and calculate the throughput respectively.

5.2.1 Put throughput with increasing data size

As shown in Figure 1, the x-axis is the data size in MB, shown in log scale, from 1 MB, 2 MB to 1024 MB, while the y-axis is the put throughput in MB operations per second, meaning in one second, it can handle Y MB operations.

The inserted data size is 1 GB. It compacts to generate a level 3 BTree SST `bree3_0.bin`, but not `bree3_1.bin` any more to compact in the level 3 to demonstrate the feature of Dostoevsky. So currently it is a Tiered LSM-Tree, having the insertion cost of

$$O\left(\frac{1}{B} \cdot \log_T\left(\frac{N}{P}\right)\right)$$

where `B` stands for the entries in one page, `T` stands for the size ratio between every two levels in the LSM-Tree, `N` stands for the data size, and `P` stands for the buffer pool size. The figure shows a log shape, which aligns with this formula.

5.2.2 Get throughput with increasing data size

As shown in Figure 2, the x-axis is the data size in MB, shown in log scale, from 2 MB, 4MB to 1024 MB. I omit the 1 MB data point for a better visualization of the trend.

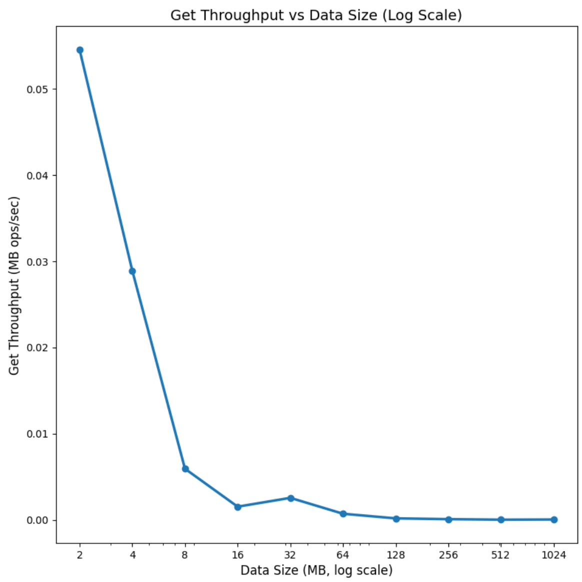

Figure 2: Get Throughput vs Data Size

Performing binary search on LSM-Tree has the query cost of

$$O\left(\log_T\left(\frac{N}{P}\right)\cdot\log_2\left(\frac{N}{B}\right)\right)$$

The figure shows a log shape, which aligns with this formula.

Additionally, if conducting B-Tree search on LSM-Tree, the query cost would be

$$O\left(\log_T\left(\frac{N}{P}\right)\cdot\log_B N\right)$$

5.2.3 Scan throughput with increasing data size

As shown in Figure 3, I also omit the 1 MB data point in scan throughput.

Performing binary search on LSM-Tree has the scan cost of

$$O\left(\log_T\left(\frac{N}{P}\right)\cdot\log_2\left(\frac{N}{B}\right)+\frac{S}{B}\right)$$

where `S` is the number of entries that need to be returned to the users. The figure shows a log shape, which aligns with this formula.

Additionally, if conducting B-Tree search on LSM-Tree, the scan cost would be

$$O\left(\log_T\left(\frac{N}{P}\right)\cdot\log_B N+\frac{S}{B}\right)$$

6 Tests

The tests file are listed below, including both unit tests and integration tests.

- `test_buffer_pool.cpp` - Unit tests for buffer pool
- `test_b_tree.cpp` - Unit tests for B-Tree

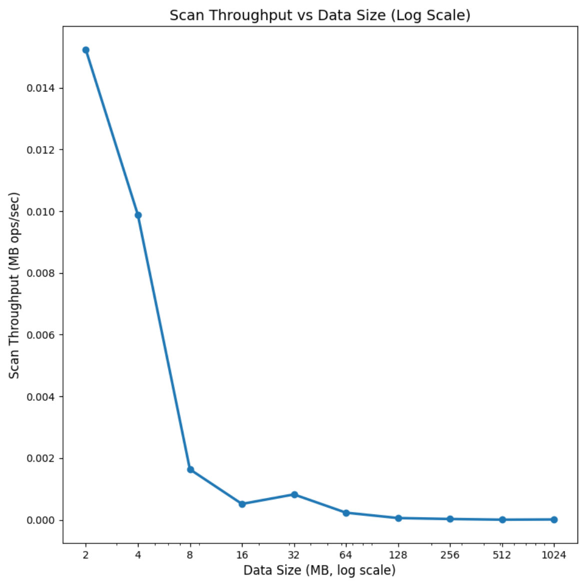

Figure 3: Scan Throughput vs Data Size

- `test_lsm_tree.cpp` - Unit tests for LSM-Tree
- `test_db.cpp` - Integration Tests