# Measurements

Lieyang Chen, Mengwen Li, Shengtan Mao

## Overview:

The benchmark suite aims to measure the main features of ProjectDB: writing, removing, and reading key value pairs. It is designed to be flexible: it supports benchmarking using randomly generated data as well as data from csv files.

The process can be split into two steps. First create a vector of key-value pairs that holds the data that will be used in the benchmark, which may be randomly generated or read from a csv file. Then we perform operations on a ProjectDb database using the vector of key-value pairs and collect the results for the benchmark.

All benchmarks ran under the following system specifications:

CPU: 12 * Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
CPUCache: 12288 KB
Memory: 15.6 GB of RAM
Disk: Samsung 860 EVO 500GB
OS: Docker Ubuntu 20.04, Host 5.11.11-arch1-1 64-bit

## Data for the benchmark:

We first create a vector of key-value pairs that holds the data that will be used in the benchmark. We incorporated the random string generation for the randomly generated values from LevelDB, another key-value storage library. The keys in this situation will be artificial keys of consecutive integers starting from 0. The benchmark suite also supports reading from csv files. The first column will act as the key and the user can select a column to act as the value. We create a vector using these key-value pairs.

There was some debate about whether we should store the data into a vector for then operate on the database or directly operate on the database. The second approach aligns better with common use cases, but we went with the first approach because we want to fairly benchmark csv files. Our key-value database only uses two of the columns out of potentially many more columns in a csv file. If we directly operate on the database, the timing would have to include parsing unused columns, which is problematic. Storing the data into a vector first would allow fair comparison in this case.

### Database configurations:

Several database configurations could affect the performance of the database, specifically:

1. **MEMTABLE_APPROXIMATE_MAX_SIZE_IN_BYTES (defaults to 2 mb)**: This controls how many data we are going to store in MemTable (in memory) before flushing to disk. If the total amount of data that user stores is less than this value, then it's essentially just storing/retrieving value from a `std::map` in memory.
2. **SSTABLE_INDEX_BLOCK_SIZE_IN_BYTES (defaults to 2 kb)**: This represents the size of the block we need to load from disk every time a user looks up a key that's not in MemTable. This also indirectly determines the number of indices we keep for an SSTable.
3. **NUM_SSTABLE_TO_COMPACT (defaults to 5)**: This represents the number of SSTable we will try to compact together. The larger this value, the more efficient we will be on removing duplicate and TOMBSTONE values.
4. **SSTABLE_APPROXIMATE_MAX_SIZE_IN_BYTES (defaults to 4 mb)**: This controls the max size of SSTable we could have on disk. The larger this value is, the fewer SSTables we will have after merges, but this also means that merging could take a longer time.
5. **KEEP_SSTABLE_FILE_OPEN (defaults to false)**: When this is true, all file handles for SSTable will be kept open, this saves some time opening the file, but might lead to too many file handles to be opened, and could result in an exception.

Users are welcome to observe the benchmarks on the expected data and tune the configuration accordingly. Our benchmarks are first done using 250 kb SSTABLE_INDEX_BLOCK_SIZE_IN_BYTES. Then done using the default 2 kb SSTABLE_INDEX_BLOCK_SIZE_IN_BYTES block size configuration.

### Benchmark descriptions:

We examined performance under 7 different tasks under the first database configuration: fill sequential, fill random, overwrite, delete sequential, delete random, read random, read ordered. These tasks are incorporated from LevelDB.

Fill sequential writes all the key-value pairs in the sorted order of the keys to a clean database. Fill random writes all the key-value pairs in the shuffled order to a clean database.

Overwrite writes all the key-value pairs in the sorted order of the keys to a database that was previously filled sequentially.

Delete sequential removes all the key-value pairs in the sorted order of the keys from a database that was previously filled sequentially. Delete random removes all the key-value pairs in the shuffled order from a database that was previously filled sequentially.

Read ordered retrieves all the key-value pairs in the sorted order of the keys from a database that was previously filled sequentially. Read random retrieves all the key-value pairs in the shuffled order from a database that was previously filled sequentially.

### Benchmark for randomly generated values with 250 kb as SSTABLE_INDEX_BLOCK_SIZE_IN_BYTES:

As mentioned before, the keys are consecutive integers starting from 0 and the values are randomly generated strings. In particular, these benchmarks are using randomly generated strings of 100 characters. This table shows the results of our benchmarks. Time units are calculated by time taken by task in microseconds divided by the number of entries (average microseconds/operation).

| entries | fill seq | fill rand | overwrite | del seq | del rand | read seq | read rand |
|---|---|---|---|---|---|---|---|
| 4,000 | 29.91 | 39.60 | 53.84 | 50.60 | 51.37 | 0.474 | 0.554 |
| 8,000 | 53.55 | 78.33 | 102.4 | 99.96 | 103.8 | 0.486 | 0.580 |
| 16,000 | 104.3 | 154.5 | 199.0 | 195.5 | 197.5 | 0.505 | 0.630 |
| 24,000 | 116.0 | 172.1 | 151.9 | 236.3 | 254.4 | 51.45 | 51.51 |
| 32,000 | 123.8 | 183.1 | 148.5 | 258.2 | 336.7 | 2869 | 2883 |
| 48,000 | 131.2 | 198.3 | 156.2 | 248.2 | 481.8 | 5233 | 5230 |

Based on the implementation, for set and remove, the most expensive operations include updating MemTable, which is a O(log(n)) operation size we have to keep the keys sorted; and handling the finished async jobs before actually updating MemTable. From the benchmark for fill, overwrite and del, we can see that the time roughly increases at a speed slightly faster than O(log(n)), but slower than O(n), which corresponds to what we expected.

Comparing sequential operations and random operations, random operations are more expensive. Our understanding is that, although there's no optimization in ProjectDb's

implementation on sequential operations (we do plan to add this optimization later), the operating system and hard drive might also have optimizations for sequential writes.

Comparing the performance for fill versus overwrite and delete; fill is faster than those operations even though the underlying operations are similar. Fill is faster because it starts with a clean table while the other two start with a full table. This means there is less cost for maintenance at the beginning.

Comparing the performance for overwrite and delete, delete seems to take more time than overwrite, but, since the implementation for delete and write is essentially the same, we originally estimate their performance will be similar. The worse performance for delete might be caused by different operations that we do while compacting SSTable compared to write. Another reason might be that with delete, the data we write to disk varies more in size, although very unlikely, it could potentially affect the performance. Also note that the performance for these operations is somewhat unintuitive. Overwrite for 16,000 and delete sequential for 32,000 are the worst out of their respective entries. We are not sure why this happens. We will do some more detailed benchmarks in the future to get a deeper understanding.

The read performance under these settings is really bad. The reason will be explained below when we run it with a different SSTABLE_INDEX_BLOCK_SIZE_IN_BYTES settings. With the current settings, we can see that when the entries increase from 16,000 to 24,000, the time suddenly jumps by around x100 times. This is because with MEMTABLE_APPROXIMATE_MAX_SIZE_IN_BYTES set to 2 mb, and with around 105 bytes per entry, the MemTable will reach its size limit at around 20,000 entries. As a result, some of the key retrievals will start requiring loading blocks from SSTable (on disk) into memory. This is the most expensive operation, and it dominates the read runtime.

### Benchmark for randomly generated values with 2 kb as SSTABLE_INDEX_BLOCK_SIZE_IN_BYTES:

Below is our benchmark when running the same randomly generated key value pairs with only SSTABLE_INDEX_BLOCK_SIZE_IN_BYTES changed:

| entries | fill seq | fill rand | overwrite | del seq | del rand | read seq | read rand |
|---|---|---|---|---|---|---|---|
| 8,000 | 52.94 | 73.84 | 52.67 | 48.84 | 66.94 | 0.106 | 0.136 |
| 16,000 | 102.7 | 153.0 | 102.1 | 93.20 | 143.6 | 0.105 | 0.138 |
| 32,000 | 125.0 | 183.5 | 124.2 | 192.5 | 302.7 | 72.95 | 73.71 |
| 64,000 | 137.0 | 208.9 | 144.5 | 527.9 | 639.7 | 87.24 | 88.82 |
| 128,000 | 141.1 | 226.9 | 166.5 | 1323 | 2297 | 191.4 | 189.2 |

The read tasks have much improved performance for this setting. The other tasks have similar performance. This is because for each read operation, we now only need to load 2 kb of data into memory, instead of 250 kb. The trade-off is that the SSTableIndex in-memory now has more entries. However, since the value for SSTableIndex is just an integer, even if the number of entries increases, it will still not take too much memory.

There are several optimizations we could do in here. One is that our current implementation uses a linear search to locate the block in SSTableIndex, which has O(n) complexity. Since the keys are all sorted, we could switch to a binary search, which will improve the performance to O(log(n)). Another optimization is that, everytime after we load an SSTable block into memory, we could potentially keep it in memory for a longer period of time instead of just discarding it right away. This will tremendously improve sequential read performance.

### Benchmark for "Trending YouTube Video Statistics" csv data with 250 kb as SSTABLE_INDEX_BLOCK_SIZE_IN_BYTES:

We also performed our benchmark using USvideos.csv from "[Trending YouTube Video Statistics](#)" on Kaggle. It contains various information about trending videos on YouTube, but we only used the column for video_id and title. An example entry is as below:

(key: 2kyS6SvSYSE, value: "WE WANT TO TALK ABOUT OUR MARRIAGE").

Here are the results in the same format as the previous table:

| entries | fill seq | fill rand | overwrite | del seq | del rand | read seq | read rand |
|---|---|---|---|---|---|---|---|
| 4,000 | 9.552 | 12.68 | 15.52 | 14.25 | 14.41 | 0.427 | 0.506 |
| 8,000 | 14.89 | 24.09 | 26.07 | 25.04 | 24.97 | 0.426 | 0.520 |
| 16,000 | 24.02 | 47.36 | 44.06 | 43.03 | 42.97 | 0.460 | 0.548 |
| 24,000 | 31.65 | 67.44 | 58.97 | 57.55 | 57.71 | 0.456 | 0.569 |
| 32,000 | 36.68 | 81.54 | 68.80 | 69.95 | 66.90 | 0.488 | 0.586 |
| 40,949 | 43.08 | 93.06 | 77.88 | 74.13 | 74.21 | 0.477 | 0.588 |

Note that the same trend can be observed here as in the previous table. All the operations are much faster in this case even though it is using 250 kb block size. Because the value is generally much smaller than 100 bytes and the data size overall is smaller than 2 mb, all the data can be kept in memory, and the operations are very fast.

**Further benchmarks:**

This benchmark is a basic measurement for the database's performance. We have ideas for a few more benchmarks that would be useful to implement given more time.

We originally planned to compare our benchmark with the LevelDB benchmarks. Much of our benchmark is modified from theirs; we wanted to run our benchmark using their APIs and compare the results.

We have observed the significant impact of configurations on the performance. It would be interesting to further investigate this and see how performance responds to different configurations.

Related:

Benchmark folder: https://github.com/mli9502/ProjectDb/tree/main/benchmark

LevelDB: https://github.com/google/leveldb