KV Store Project Report

Kevin Cai, Junsong Guo

December 6, 2024

1 Introduction

In this report, we present the implementation of a key-value (KV) store as part of the course project for CSC443. Out KV store database can store keys and values as 4 byte integers. The database can be accessed by the user in their terminal with a CLI, which supports insert, update, and delete operations. The database uses a combination of in-memory structures and disk storage to manage large amounts of data in gigabytes. This implementation uses a Memtable, Sorted String Tables (SSTs), a buffer pool, and a LSM Tree with a Bloom filter, and advanced query techniques to optimize for both storage and retrieval performance.

2 Design Details

2.1 CLI

Users can interact with the database locally using our command-line interface (CLI). Below is a list of all the available commands:

```
put <key> <value>           : Inserts a key-value pair into the KV store.
get <key>                   : Retrieves the value associated with a given key.
scan <low> <high>           : Scans and retrieves all KV-pairs in the range
                              [low, high].
open <dbName> <memtableSize> <bufferCapacity> :
                              Opens the KV store with the specified parameters.
close                       : Saves and closes the KV store.
delete                      : Deletes the current opened KV store.
exit                        : Exits the program.
help                        : Displays a help message.
```

`<memtableSize>` is specified in terms of number of key-value pairs and `<bufferCapacity>` is specified in the number of 4 KB pages. These parameters can be changed each time the database is opened.

Related Code: main.cpp

2.2 Memtable

The memtable is implemented as a balanced binary tree (AVL tree). It stores key-value pairs in memory and supports the basic KV-store operations (Put, Get, and Scan). When the memtable reaches its size limit, or when the user closes the database connection in the CLI, key-value pairs stored in the memtable are converted into an SST.

Related code: AVLTree.h & AVLTree.cpp: `scan()`, `insert()`, `getValue()`

2.3 SST

Sorted String Tables (SSTs) are used to store data on disk. The `Get` and `Scan` operations on SSTs use binary search to quickly locate keys, by taking advantage of the sorted nature of the SSTs. Since a buffer pool is used for the database, the SST I/O operations uses direct I/O to avoid double buffering from the OS and our buffer pool. Read and writes to SSTs are done in 4KB sized pages. In our final version of the database, we are not using direct I/O for read write operations, as we discovered that the `O_DIRECT` flag was not supported on Windows Subsystem for Linux (WSL), which is the environment both our team members are working on. We are only able to use direct I/O on the teach.cs machine, however conducting experiments takes a very long time even for a vew MB of data. Hence, we did not use direct I/O and our progress is left in the branch `direct-io`.

Related code: SSTController.h & SSTController.cpp: `save(...)`, `get(...)`, `scan(...)`, `close(...)`; KVStore.h & KVStore.cpp: `put(...)`, `get(...)`, `scan(...)`

2.4 Buffer pool

The buffer pool caches frequently accessed pages of size 4KB in memory to minimize disk I/O. The buffer pool uses a hash table with xxhash function (2016 Stephan Brumme). Hash collisions are resolved by chaining, which is implemented as a linked list for each hash table cell, and the newly hashed page is added to the end of the linked list. The eviction policy we chose is the Clock eviction policy, as it is a fast approximation of the LRU eviction policy. Whenever a page will be read using an I/O, the database first checks the buffer pool for the page and only performs an I/O if the page is not in the buffer pool, then the newly read page will be added to the buffer pool.

Related code: BufferPool.h & BufferPool.cpp: `getPage()`, `putPage()`

2.5 B-Tree

A B-Tree was implemented to compare the performance of querying using B-Tree compared to using binary search on the SSTs. Our implementation only converts SST files into static B-trees, rather than constructing B-trees dynamically. To construct a static B-tree for each SST file, the database first extracts all key-value pairs, then for each `B` (page size / key-value pair size) key-value pairs, they are merged into one B-tree node, with values in the node as the last key in each page. Then this step is continued recursively until only one root node of size `B` is left. Each value also stores the page the value points to beside as an integer value, where interal tree node page numbers are stored as negative numbers and SST page number is stored as positive numbers. Finally all B-tree nodes are stored in a single file.

Related code: BTreeController.h & BTreeController.cpp: `get(...)`, `createBTrees(...)`

2.6 LSM-tree for SSTs

An LSM-tree was implemented as a superior alternative to the plain SST files in previous parts, aimed to optimize for a balance between querying and insertion. It is a basic LSM-tree with a size ratio of 2. While keeping the interface mostly similar to that of plain SSTs, it introduced levels to the individual SST files. Instead of being stored in one single directory, all SSTs are separated into different levels, each represented by a subdirectory in the database. Due to this change, the metadata changed from a single integer representing the number of SSTs to a hashmap specifying the number of SST files in each level. When a new SST is inserted into the DB, it is always added to the first level. Then, the DB recursively checks if a level is full, starting from the first level. When any level has 2 SSTs, they will be compacted into a larger one and inserted into the next level. Also, an external sorting mechanism is implemented to ensure the SSTs with sizes larger than memory can be correctly compacted. For update queries, a new value will be normally put to the DB and the duplication will be resolved in compaction. Similarly, delete queries will normally put a value of `INT32_MIN` for the given key, which represents a tombstone. Once a tombstone reaches the max level, it will be completely removed from the DB. Lastly, when the DB is closed, the metadata will be stored in a file called `metadata`, which will be used to load the metadata when he DB is opened again.

Related code: LSMController.h & LSMController.cpp: `save(...)`, `get(...)`, `scan(...)`, `close(...)`; LSMStore.h & LSMStore.cpp: `put(...)`, `get(...)`, `scan(...)`, `remove(...)`,

2.7 Feature Checklist

Here's a list of features that we implemented:

Note: a friendly reminder that our rubric is a bit different since we are a group of only 2 students.

Step 1 - KV-store get API - yes - KV-store put API - yes - KV-store scan API - yes - In-memory memtable as balanced binary tree - yes - SSTs in storage with binary search - yes - Database open and close API - yes

Step 2 - Implementation of Buffer pool as hash table with collision resolution - yes - Integration buffer pool with queries - yes - Clock or LRU eviction policy - yes

Step 3 - Filter for SST and integration with get - NOT REQUIRED - Persisting filters in SSTs - NOT REQUIRED - Compaction/Merge of two SSTs - yes - Support update - yes - Support delete - yes

Bonus: - Static B-tree for SSTs - yes (extra because smaller group) - Experiments for binary search V.S. static b-tree - yes (extra because smaller group) - Handling sequential flooding - yes

3 Experiments

3.1 SST binary search `Get` query performance

We tested the `Get` query throughput of our database using only the memtable, buffer pool, and SSTs with binary search. For this experiment, the memtable was set to 1 MB, each page in the buffer pool was 10 MB, and up to 1 GB of data was inserted into the database. The performance of the `Get` query was measured at 8 intervals during the insertion process by recording the time to retrieve 1 KB of uniformly random key value data inserted at the time. Figure 1 shows the result of the experiment.

This experiement can be reproduced by executing the binary file `experiment*BinarySearchGet*` in the build directory.

3.2 SST B-Tree `Get` query performance

Using the same environment, memtable size, buffer pool size, and data size, as in experiment Section 3.1, we tested the `Get` query throughput of the static B-Tree. Figure 2 shows the result of the experiment. Get query throughput with our B-tree implementation was slower than our binary search implementation. Theoretically, the B-tree should have a faster running time of $O(\log_B(N))$, whereas binary search has a running time of $O(\log_2(N))$. We suspect that since we only used B-tree on SST of size 1 MB, maybe the size is not large enough to see a speed improvement, as our B-tree only has one level, and data is not pruned significantly compared to binary search.

These experiements can be reproduced by executing the binary file `experimentBTree` in the build directory.

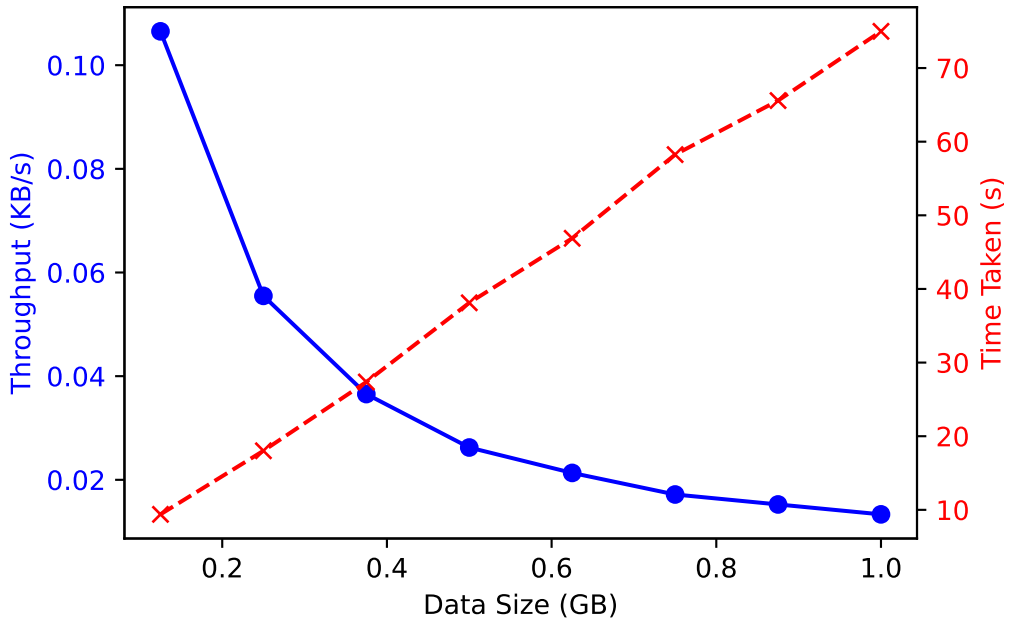

Figure 1: Binary search Get throughput and time taken vs data size

Please note that these experiments take much longer to run. (over 2.5 hours to run all three of them on our machine with 16 GB ram and 12gen intel CPU.)

3.3 LSM `Put/Get/Scan` query performance

Using the same parameters again, we were able to experiment with the throughput of the LSM tree implemented in Step 3. Results are shown in the figures below. It might be surprising to see that the throughput of the LSM-tree is much lower than that of the previous steps because LSM tree is the most comprehensive and universal DB structure that we learned in the lectures. One of the causes could be that the LSM-tree we implemented was a very basic one. It has a size ratio of 2 and did not implement any optimizing strategies such as Monkey or Dostoevsky. Also, since static B-trees were not integrated, we cannot cancel the $O(\log_2(B))$ cost. A final reason could be that since we started the experiment with a fresh DB, it will need to do extra I/Os in order to create new subdirectories when it reaches a new level, this will likely increase the time of the insertion but will diminish as the level grows.

4 Compilation and Running Instructions

To compile the project, use the following command in the project root directory:

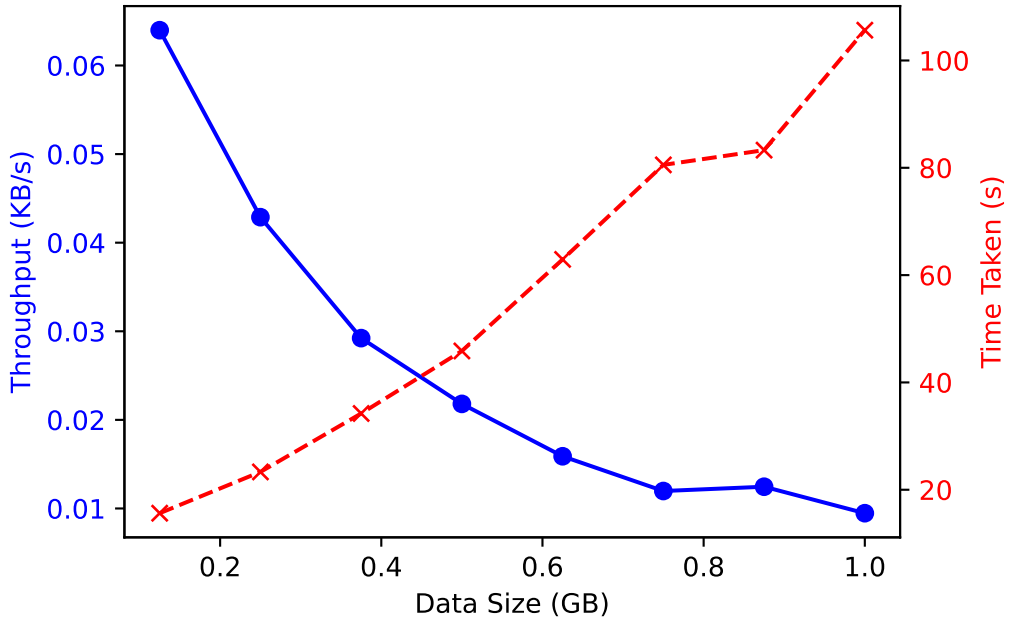

Figure 2: Static B-tree Get throughput and time taken vs data size

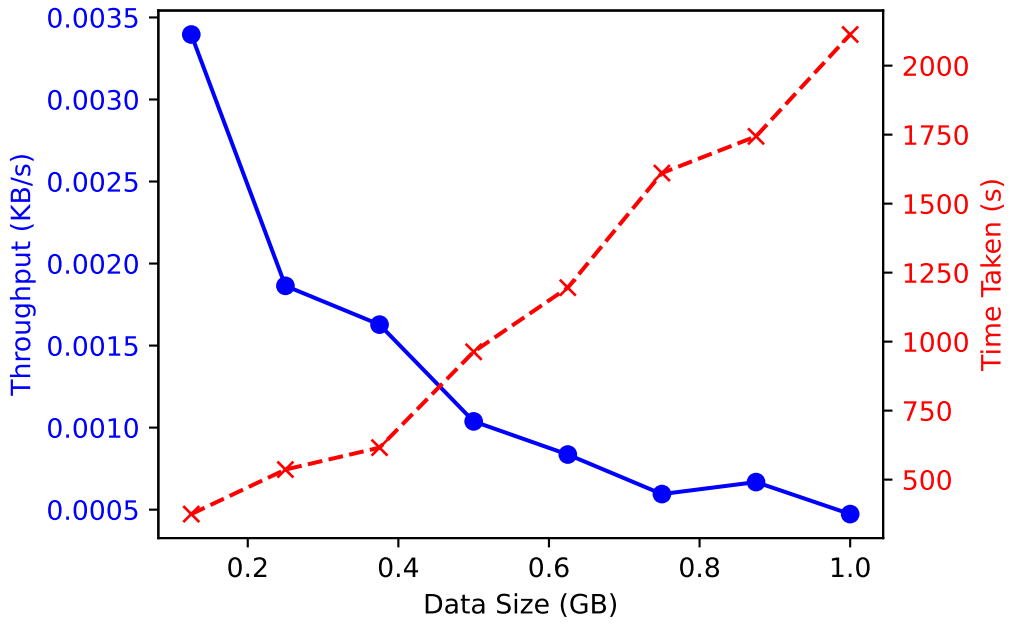

Figure 3: LSM tree Put throughput and time taken vs data size

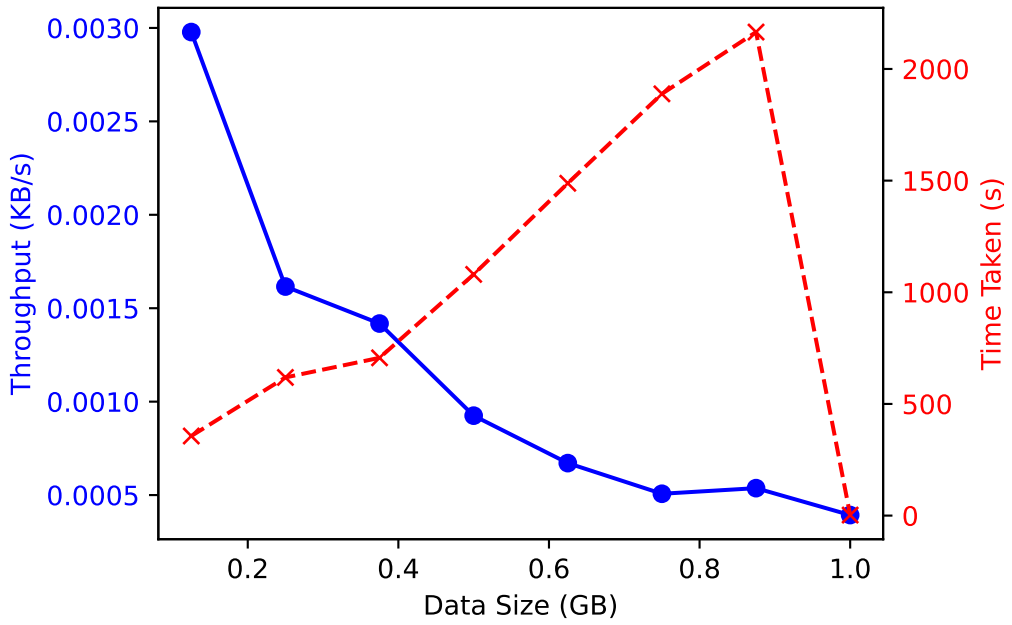

Figure 4: LSM tree Get throughput and time taken vs data size

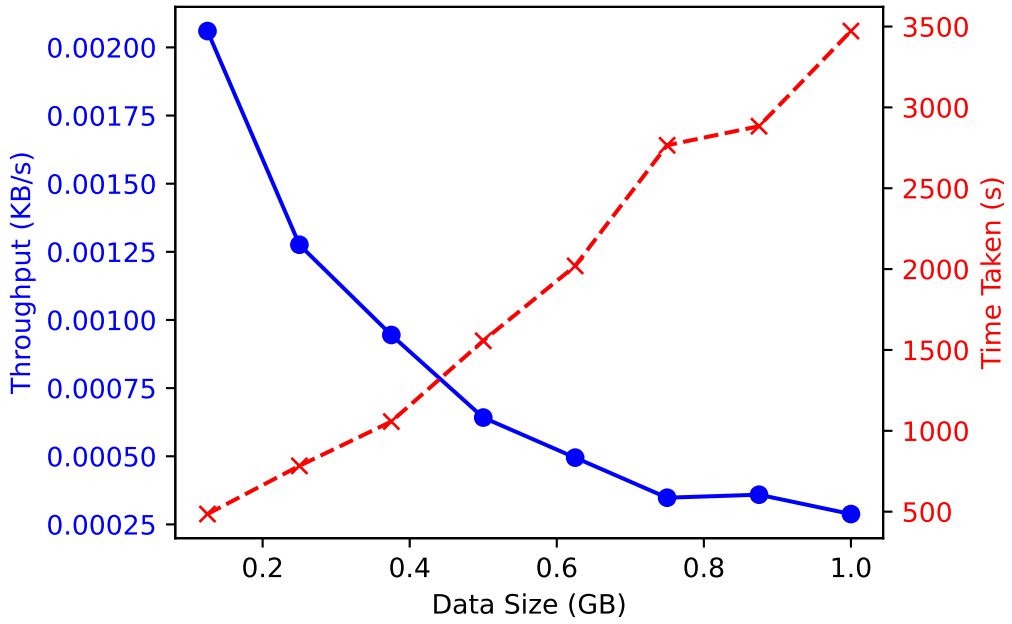

Figure 5: LSM tree scan throughput and time taken vs data size

```
mkdir build-release
cd build-release
cmake ../ -DCMAKE_BUILD_TYPE=Release
make
```

Then to start the CLI, use the following command in the `build-release` directory:

```
./main
```

Note: this executable uses memtable, buffer pool, and LSM-tree.

Please see "Design Details" section above or simply type `help` in the CLI to get command instructions.

5 Testing

To test the functionalities of the databse, use the following command in the `build-release` directory:

```
./tests
```

We implemented unit tests to validate our AVL tree, SST, B-tree, and LSM-tree implementations. Functionalities such as put, scan, remove, update, as well as initializing and re-opening database are all covered in different test cases. The detailed tests we performed are shown in the output when running the tests.

6 Limitations

Although we have tried our best, there are some limitations of the code that could be improved in the future. For example, while we implemented static b-tree and performed corresponding experiments. We did not integrate it with the LSM tree in step 3. If we were able to do it, the performance of the final product could be improved greatly in theory.