KV Store Project Report

Kevin Cai, Junsong Guo

December 5, 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>` and `<bufferCapacity>` are both specified in terms of number of key-value pairs, and these parameters can be changed each time the database is opened.

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.

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.

2.4 Buffer pool

The buffer pool caches frequently accessed pages 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. Dirty pages (pages that are modified) in the buffer pool are written to the corresponding SST when the page is evicted.

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.

2.6 LSM-tree for SSTs

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 `experimentBinarySearchGet` in the build directory.

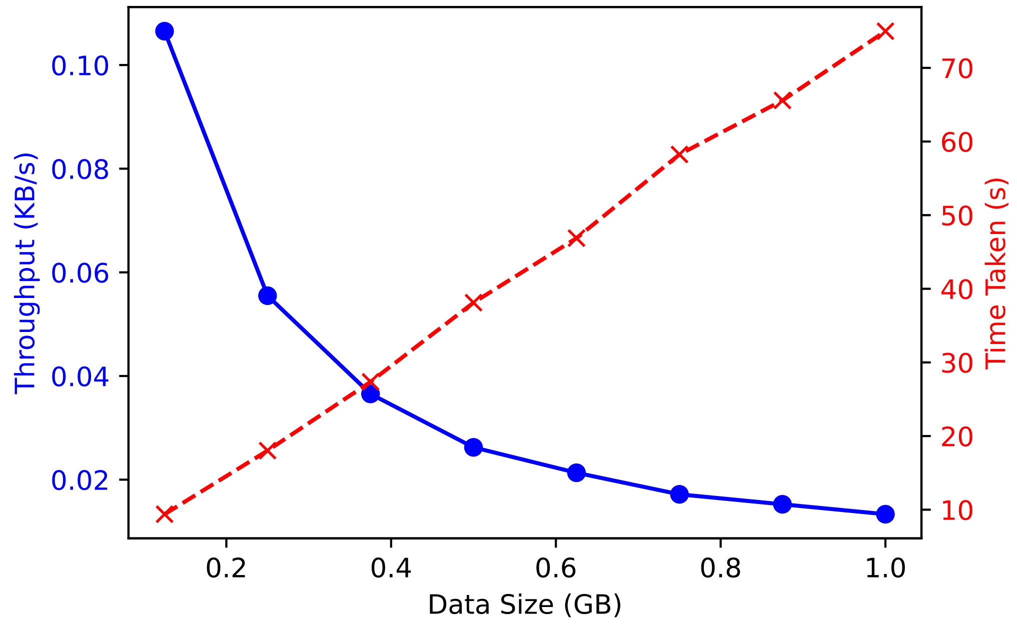

Figure 1: Binary search Get throughput and time taken vs data size

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.

This experiement can be reproduced by executing the binary file `experimentBTreeGet` in the build directory.

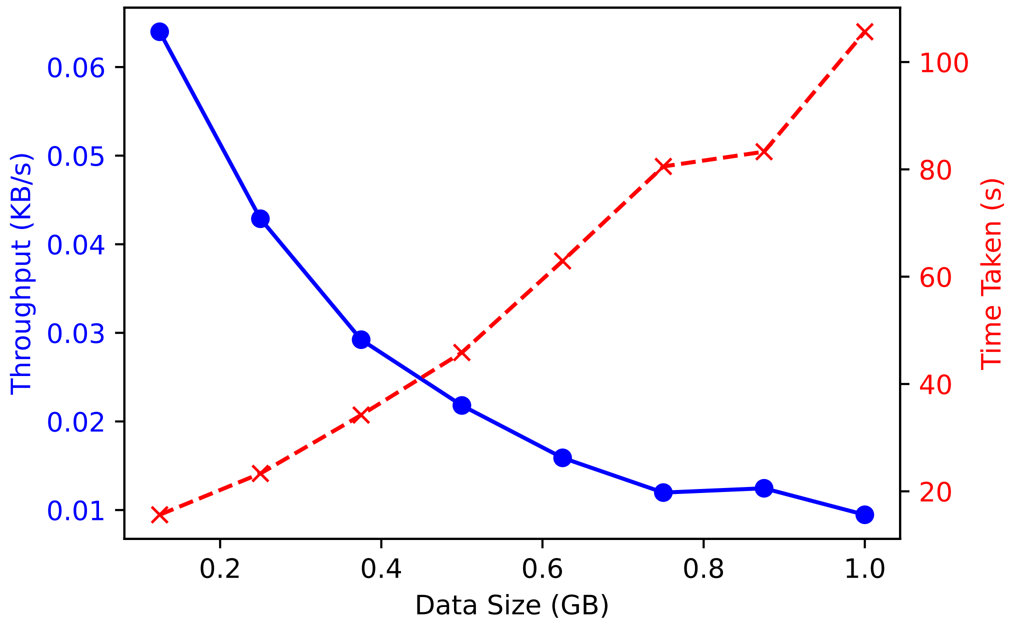

Figure 2: Static B-tree Get throughput and time taken vs data size

4 Compilation and Running Instructions

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

```
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
```

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. The detailed tests we performed are shown in the output when running the tests.