

# KV Store Project Report

Kevin Cai, Junsong Guo

December 4, 2024

## 1 Introduction

In this report, we present the implementation of a key-value (KV) store as part of the course project for CSC443. Our 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 combination of in-memory structures and disk storage, to manage large amount 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.
```

## 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 is 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 Clock eviction, as it's 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 LSM-tree for SSTs

# 3 Experiments

## 3.1 SST binary search 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.

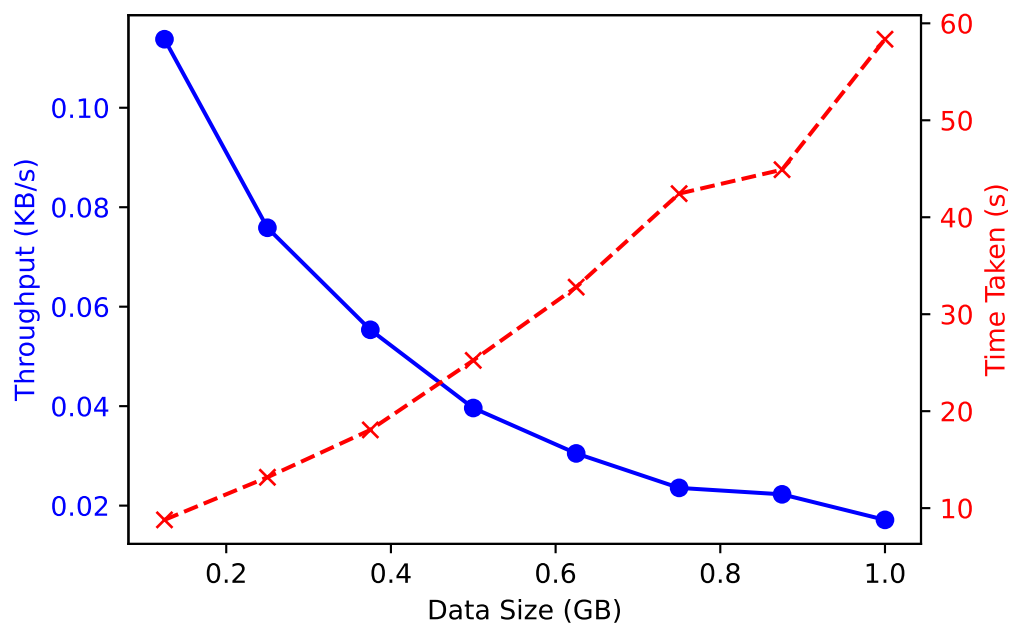


Figure 1: KV Store Throughput and Time Taken vs Data Size

## 4 Compilation and Running Instructions

To compile the project, use the following command:

## 5 Testing