Key-Value Database System API

By Aryaman Arora, Erkhjin Oyunbaatar and Thomas Shim.

1. Introduction

This report details the development of a key-value store (KV-store) using techniques such as LSM-trees, balanced binary trees, and static B-trees. The project aimed to implement a database system with functionalities for storing and retrieving key-value pairs efficiently, even when the data exceeds the system's memory capacity.

The key-value store API included basic operations such as Open, Put, Get, Scan, and Close, with an architecture inspired by modern systems like RocksDB.

2. Design Elements

Step 1

1. KV-store get API
2. KV-store put API
3. KV-store scan API
4. In-memory Memtable as balanced binary tree
5. SSTs in storage with binary search
6. Database open and close API

Step 1.1: Implement the Memtable as a AVL Tree Supporting Put and Get

In step 1.1, the Memtable was implemented as balanced binary tree (AVL tree) to ensure efficient insertion and retrieval operations. The put method was designed to insert key-value pairs into the tree while maintaining its balanced structure, ensuring logarithmic complexity for all operations. Similarly, the get method performs efficient lookups by traversing the tree to locate the specified key. The balanced structure of the binary tree guarantees optimal performance for both operations, even as the size of the Memtable grows. (memtable.cpp)

Step 1.2: Implementing Memtable Scans and Memtable to SST Conversion

In step 1.2, the scan functionality was implemented for the Memtable to retrieve all key-value pairs within a specified range. This operation traverses the balanced binary tree structure of the Memtable, ensuring that the results are returned in sorted order without additional sorting.

When the Memtable reaches its predefined capacity, the scan mechanism is utilized to extract its contents, which are then written to a new SST file. This transformation process preserves the sorted order of the key-value pairs, enabling efficient queries in the SST. This functionality is critical for maintaining database performance, as it allows the Memtable to serve as an in-memory buffer while ensuring that its overflow data is in SST format. (writeToMemtable function in sst.cpp)

Step 1.3: Extending Put and Scan Functionalities to Both SST and Memtable

In step 1.3, the put and scan functionalities were extended to operate across both Memtable and SST files. We also implemented the binarySearch function, which is utilized within the get method to perform efficient key lookups in SST files. To further enhance the range query functionality, We created binarySearchScan function, which uses key1 and key2 to retrieve key-value pairs within specified range without requiring the get method to iterate through all files repeatedly. These implementations significantly improved the efficiency of both point queries and range scans by reducing redundant file accesses.

Relevant functions: (getDataFiles, binarySearch, binarySearchScan in sst.cpp, get and scan functions in memtable.cpp)

Step 2

7. Implementation of Buffer pool as hash table with collision resolution
8. Integration buffer pool with queries
9. LRU eviction policy
10. Static B-tree for SSTs

Step 2.1: Core Hash Table Implementation and Collision Resolution Strategy

This step involved implementing a hash table for managing pages in the buffer pool using open chaining for collision resolution. The XXHash64 function was used to calculate hash values for efficient and uniform key distribution. Collisions are handled by linking pages at the same hash index in a chain. The buffer pool enforces a max_pages limit, and when full uses a Least Recently Used (LRU) eviction policy to remove the least recently used page. This design ensures efficient key lookups, collision handling, and memory management for high-performance data access.

Relevant functions: (hashkey, insertPage, searchForPage in buffer_pool.cpp)

Step 2.2: Implementing Eviction policy.

To efficiently manage memory in the buffer pool, we implemented a Least Recently Used (LRU) eviction policy. The buffer pool uses a hash table for fast lookups and an LRU list to track the access order of the pages. When the pool reaches its capacity, the least recently used page is evicted, removed from the LRU list, hash table and memory to ensure leaks. The searchForPage function updates the LRU list by moving accessed pages to the front, while the insertPage function handles eviction and adds new pages. This design ensures frequently accessed pages remain in memory, optimizing performance and memory efficiency. The implementation is in the BufferPool class (buffer_pool.cpp)

Step 2.3: Implementing the Static B-Tree Structure for SSTs

This step involved designing and implementing a static B-tree to enhance the performance of key lookups within Sorted String Tables (SSTs). The static B-tree provides a hierarchical indexing mechanism, reducing the number of I/O operations required to locate keys.

The implementation includes two main components: BTreeNode and StaticBTree. Each BTreeNode represents a node in the B-tree, storing keys and either pages (pointing to other Internal Nodes or Leaf Nodes) or values (values are only stored in the Leaf Nodes) in arrays. Nodes are initialized with the is_leaf attribute to distinguish between Leaf and Internal Nodes, and their storage is optimized for 4KB-aligned pages to align with direct I/O constraints.

The StaticBTree class manages the B-tree structure and links it to the corresponding SST file. During SST creation, the sorted key-value pairs are used to construct the B-tree. Internal nodes are generated hierarchically to maintain B-tree properties, and the root node is indexed for efficient traversal.

The binarySearch function was implemented to perform efficient lookups within individual nodes, leveraging the sorted structure of the keys array. This ensures O(logB) complexity per node search, where B is the branching factor of the B-tree.

Relevant Functions: (BTreeNode Constructor, StaticBTree Constructor, StaticBTree::binarySearch)

File: static_b_tree.cpp

This design ensures efficient key lookups, minimal I/O operations, and seamless integration with SSTs, significantly enhancing query performance.

Step 3

11. Filter for SST and integration with get
12. Persisting filters in SSTs
13. Compaction/Merge of two SSTs
14. Support update
15. Support delete

Step 3.1: Implementing Merge SST and LSM Tree Metadata

In this step we completed the compaction/merge of two SSTs and the metadata structure for each LSM tree. The LSM tree structure allows us to make queries to our SSTs and B-Tree files efficiently. The relevant coding portions can be found in lsm_tree.h and lsm_tree.cpp. The LSM tree put method, puts another SST file into the LSM tree structure. The LSM tree get and scan methods do their respective queries in the context of the LSM structure. The changeMemtable function changes the current Memtable of the LSM tree metadata. The compactLevels method compacts the levels if any of the levels are full, by calling the mergeSSTs method. The mergeSSTs method is where the merging of two SSTs happen. (lsm_tree.cpp, lsm_tree.h)

Step 3.2: Adding Updates and Deletes and Integrating Buffer Pool

In this step we added the support for updates and deletes. These were added in the mergeSSTs method and subtly in the get and scan LSMTree methods. For get and scans we made sure to always return the first value or values (for scan) that were found, and this encapsulates the idea of updating for read queries. In the mergeSSTs method we also made sure to take the most recent value if there were two equivalent keys in the SSTs we were merging. For deletes we added another API call in the main.cpp folder and it would call the PUT API but with the value LONG_MIN. In our mergeSSTs method we also made sure to remove the tombstones if we were merging to the last level of the LSMTree. (main.cpp [lines 241-262], lsm_tree.cpp)

Step 3.3: Implement the Filters for the SSTs

In this step, we implemented SST during their creation to optimize key retrieval. The filters allow quick exclusion of non-existent keys, reducing unnecessary disk I/O. We also integrated bloom filters into the get functionality to check filters before accessing SSTs. This significantly improved performance by minimizing disk reads for both get operations and SST compaction. (bloom_filter.cpp) (mergeSST LSMTree Get functions in lsm_tree.cpp)

3. Project Status

The key-value store (KV-store) project has made great progress, and most of its features are working as expected. Here's an overview of what's been completed so far:

1. **Core APIs:**
   - Open: Sets up the database by either creating a new directory or loading an existing one.
   - Put: Adds key-value pairs to the Memtable. When the Memtable is full, it automatically creates a new SST on disk.
   - Get: Fetches the value for a given key using either binary search or a B-tree.
   - Scan: Retrieves all the key-value pairs within a specified range. Like Get, this works with both binary search and B-tree methods.
   - Close: Finalizes everything by saving the Memtable to an SST and shutting down the database.
2. **In-Memory Components:**
   - Memtable: This is a self-balancing AVL tree, which keeps the data sorted and allows fast operations for putting, getting, and scanning keys.
   - Buffer Pool: A hash table handles caching for frequently accessed data. To keep things efficient, we've added an LRU (Least Recently Used) eviction policy to manage memory better.
   - LSM Tree: This connects everything, combining Memtable, SSTs, and compaction into a single workflow. Here's how it works:
     - Data is written to the Memtable first, then moved to SSTs when it gets full.

    - Compaction merges SSTs to keep everything organized and improve performance.
    - Updates and deletes are managed using tombstones, which are cleaned up during compaction.
3. **Persistent Storage:**
    - SSTs: These are sorted files that store data when the Memtable reaches capacity. We can search them quickly using binary search or the B-tree.
    - Static B-Tree: This is an index built into the SSTs that makes searching much faster. It's compact and doesn't take up much storage, so it's a great fit for our system.
4. **Advanced Features:**
    - Bloom Filters: These are added to the SSTs to help skip unnecessary reads when the key isn't in the file. It saves a lot of time and resources.
    - Compaction: This merges smaller SST files into larger ones to keep the database organized and running efficiently.
5. **Testing:**
    - We've run a bunch of unit and integration tests to make sure everything—Memtable, SSTs, Buffer Pool, B-Tree, and LSM Tree—is working correctly.
    - We also tested how fast the system performs for key operations like Put, Get, and Scan. Plus, we compared the performance of binary search and B-tree searches.

Known Issues:

1. The compaction process works, but it could be optimized further to balance file sizes better across levels.
2. We've added support for direct I/O, but it still needs more testing to confirm that everything works properly on different Linux setups.
3. As the Memtable gets larger, searching slows down, especially with binary search. The B-tree performs better but still slows down with bigger datasets.
4. The Step 3 experiments provide unexpected results. According to figure 2,3 and 4 we can see that for datasizer greater than 256 MB there is unexpected throughput for Get and

Scans when using the btree search. We suspect this issue has to do something with the structure of the nodes in the btree as files grow larger. However unfortunately we did not have enough time to debug this issue as it showed up during the experiments.

Pending Features:

- The system runs on a single thread for now, but we'd like to add support for multiple threads to make it faster.
- We also need to run more experiments to see how well the system scales with larger datasets or buffer pools.

Overall, the project is on track, and what's left are mainly small refinements and optimizations to make it even better.

4. Experiments

To assess the performance of our key-value store (KV-store), we conducted experiments targeting its primary functionalities: Put, Get, and Scan. These experiments aimed to evaluate throughput under different data sizes and to compare indexing strategies. The results provide insights into the system's behavior as data size scales.

Step 2 Experiments:

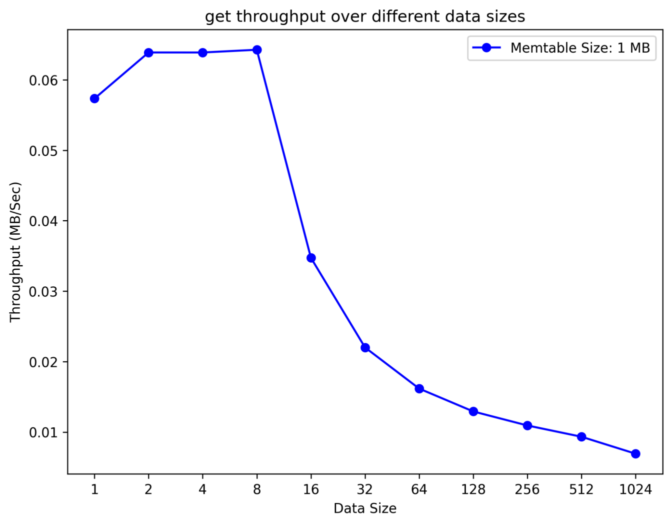

Figure 1: Throughput for GET operation with Binary Search as the data grows

This experiment evaluated how GET operations perform when using binary search as the data size grows. The throughput started high for smaller data sizes but dropped significantly as the dataset increased. This was expected since binary search requires reading larger portions of data from SSTs as the dataset expands, leading to more disk I/O.

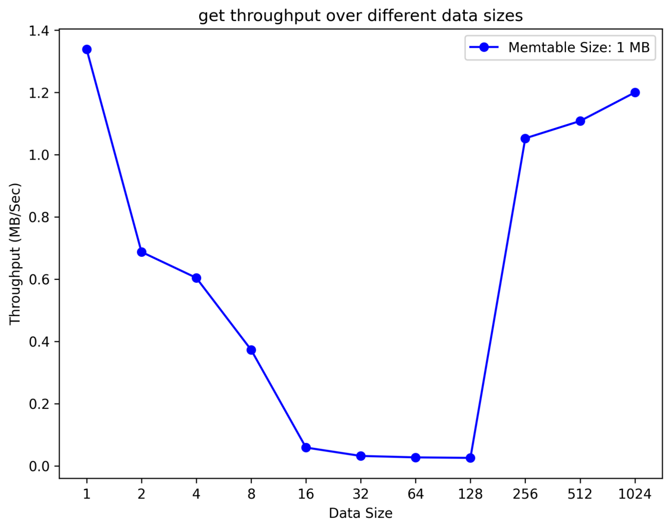

Figure 2: Throughput for GET operation with B-Tree Search as the data grows

Using a static B-tree for GET operations, this experiment demonstrated better performance compared to binary search as the dataset increased. The throughput initially dropped with smaller data sizes but remained relatively stable as the dataset grew. The hierarchical structure of the B-tree minimizes the number of disk I/O operations, making it more efficient for large datasets. Note: there are known issues with this experiment mentioned in Project Status regarding the results when the dataset gets really large.

Step 3 Experiments:

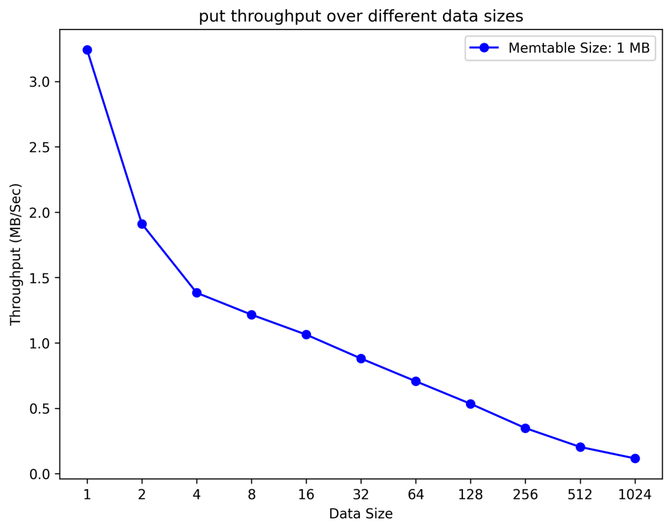

Figure 3: Throughput for PUT operation as the data grows

We measured the throughput of PUT operations as the data size grew. Initially, the throughput was high due to all operations occurring in-memory in the Memtable. As the Memtable reached its capacity, the overhead of writing to disk caused a slight dip in performance. Despite this, the system maintained consistent throughput across larger data sizes, highlighting the efficiency of our Memtable-to-SST workflow.

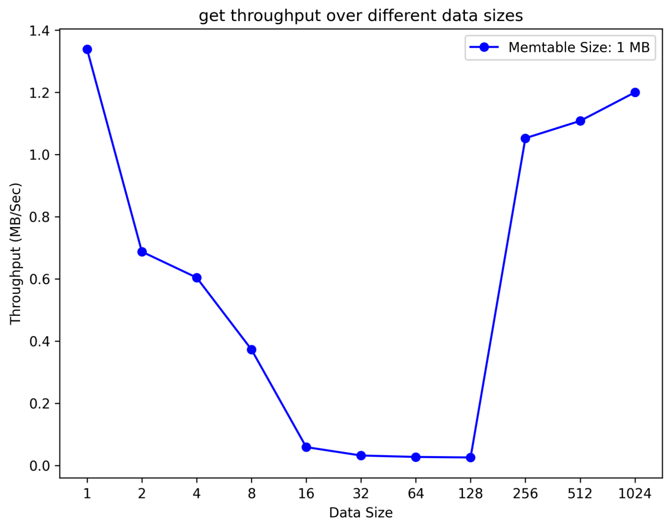

Figure 3: Throughput for Get operation with B-Tree Search as the data grows

This experiment focused on measuring the overall GET throughput as the data size grew, without isolating the specific search strategy. Throughput was initially high for smaller data sizes, where operations primarily utilized in-memory data. As the dataset grew, throughput dropped due to the increasing reliance on disk I/O. However, the results highlight the importance of efficient indexing strategies, as B-tree-based queries maintained relatively consistent performance even as data grew large. Note: there are known issues with this experiment mentioned in Project Status regarding the results when the dataset gets really large.

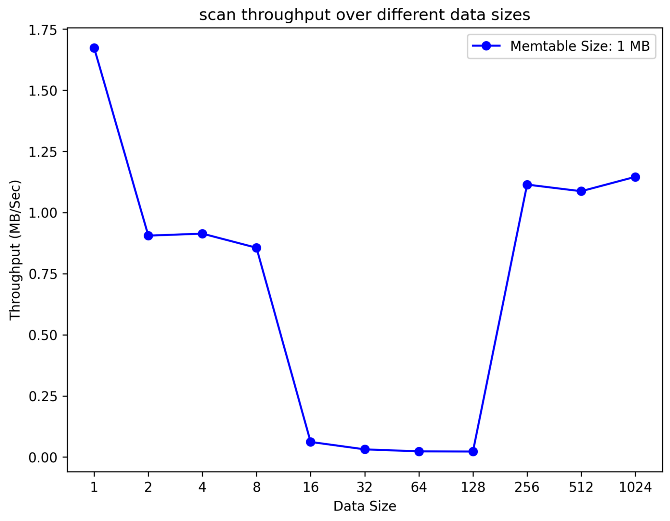

Figure 4: Throughput for Scan operation with B-Tree Search as the data grows

This experiment evaluated how the system handled SCAN operations for increasing data sizes. Throughput declined with larger data sizes as the amount of data being read increased. However, the static B-tree's efficient traversal capabilities allowed SCAN operations to perform well compared to alternative indexing strategies. The system demonstrated scalability while maintaining reasonable performance for range queries. Note: there are known issues with this experiment mentioned in Project Status regarding the results when the dataset gets really large.

5. Testing

Comprehensive testing was conducted to ensure the reliability and correctness of the system. Unit and integration tests were implemented for the core functionalities of the Memtable, SST, Buffer Pool, B-Tree, and LSM Tree.

Unit Tests

1. **Memtable:**
   - Tested put and get operations to ensure correct insertion and retrieval of key-value pairs.
   - Verified behaviour under edge cases, including updates to existing keys and handling full Memtable capacity.
   - For example, inserted 256 key-value pairs into a Memtable and checked for correct retrieval using the testWriteMemtableToSST() function.
2. **SST:**
   - Validated writing Memtable data to SST files using direct I/O.
   - Tested the integrity of the written data by reading back key-value pairs from the SST.
   - For example, verified the correctness of data stored in SST files using aligned buffers, as demonstrated in testWriteMemtabletoSST().
3. **B-Tree:**
   - Verified correct node creation and insertion for configurations ranging from single leaf nodes to trees with multiple internal nodes.
   - For example, used testBTreeMain() to evaluate the structure and behaviour of B-Trees with up to 256 leaf nodes.
4. **LSM Tree:**
   - Tested overall LSMTree structure, making sure compactLevels and mergeSSTs is working as intended.
   - Tested queries such as scan methods which encapsulates all SST merges, compact levels and also the query itself.
   - Test are found in test_lsm_tree.cpp.

Integration Tests

1. **Memtable to SST Transition:**
   - Tested the conversion of Memtable data into SST files upon reaching capacity, ensuring seamless integration.
   - Verified the correctness of the data written and its subsequent retrieval from SSTs.

2. **SST Range Queries:**
   - Tested the binarySearchScan() function to retrieve key-value pairs in a specified range across SSTs.
   - Verified optimized range queries leveraging the buffer Pool to minimize redundant I/O operations..
3. **LSM Tree Queries:**
   - Ensured the proper functioning of get and scan operations across Memtable and SST layers, integrating Bloom Filters for efficiency.
   - For example, Tested the retrieval of all keys in the range [1,500] using the testLSMTreeScan() function.

The tests were implemented in files like test_memtable.cpp, test_sst.cpp, test_helpers.cpp, test_lsm_tree.cpp, test_static_b_tree.cpp, tests_main.cpp, with modularized test functions controlled via flags. Each test was logged, including passed and failed cases, allowing for precise debugging.

6. Compilation and Running Instructions

The project provides a comprehensive run.sh (inside CSC443-Project folder) script to streamline the compilation, testing and execution process. Follow these steps to compile and run the project.

Prerequisites:

1. Ensure the following tools are installed:
   - C++ Compiler

    - CMake
    - Make
2. Script assumes a Unix based environment.

Compilation:

1. Run the run.sh script in the terminal.
    - Cleans the build directory by removing all previous build files.
    - Configures the project with make.
2. Select the executable you would like to run.
3. Enjoy!

Using the Project:

The project supports several commands to interact with the database. Each command is entered in the terminal as shown below.

1. Open a database:
    - Command: Open("database_name")
    - Opens a new database directory and initializes it for operations.
    - A command line token appears when you open a database. This token will be the name of the database.
2. Insert a Key-Value Pair
    - Inserts a key-value pair into the Memtable. If the Memtable reaches its capacity, The contents are flushed to an SST on disk.
        - Example: Put(1,100)
3. Retrieve a Value by Key
    - Command: Get(key)
    - Retrieves the value associated with the given key. It first searches the Memtable, and if not found, it checks the SST files using either Binary Search or B-Tree Search.
    - The system will prompt for the search method:
        - 1: Binary search
        - 2: B-Tree search

4. Perform a Range Scan
    - Retrieves all key-value pairs within the range [key1, key2]
    - The system will prompt for the scan method:
    - 1: Binary search
    - 2: B-Tree search
5. Close the Current Database
    - Command: Close()
    - Flushes the current Memtable to an SST and resets the system to allow opening a new database.

If a Memtable reaches its capacity during a put operation, it is automatically written to an SST file.

The scan and get operations allow you to choose between Binary Search and B-Tree search for different performance characteristics.

Deleted keys are stored as tombstones and can be identified in scan operations.