CSC443 Group Project Report

Jonathan Lam, Ching Hui, Yuchen Zeng

December 2023

1 Compilation & running Instructions

To compile the tests and experiments, run `make` in the root directory of the project. After that, to run either tests or experiments, remain in the root directory of the project and run `./tests/tests` or `./experiments/experiments` respectively.

2 Step 1 - Creating a Memtable and SSTs

2.1 Project status

All required functionality and structures have been implemented for this step.

2.2 Introduction

In the implementation of our database system, the Memtable plays a crucial role as a primary data structure for in-memory storage. The Memtable is designed as a balanced AVL tree, which ensures efficient data management with optimized insertion, search, and range query capabilities. This section of the report delves into the intricacies of the Memtable's implementation, highlighting its structure and functionalities, particularly focusing on the put, get, and scan operations.

2.3 Memtable AVL Tree Structure

The core of the Memtable is structured as a balanced AVL tree, which is a self-balancing binary search tree. Each node in this tree is represented by the Node class that contains an 8-byte key-value pair (16 bytes total for the two together), pointers to the left and right child nodes, and a height attribute that helps maintain tree balance. The root node of the tree acts as the entry point for all operations.

2.4 Memtable Put Functionality

The put function is responsible for inserting new key-value pairs into the Memtable. Insertion into the Memtable starts from the root. The function places the new key-value pair in the appropriate location, following binary search tree rules. If a node with the given key already exists, its value is updated.

In order to keep the AVL structure of the Memtable, a left or right rotation might be necessary after an insertion. This is because post-insertion, the tree might become unbalanced. The AVL tree properties are restored by calculating the balance factor of each node (the height difference between the left and right subtree). Depending on the balance factor, the tree undergoes necessary rotations - left, right, left-right, or right-left - to regain its balanced structure.

If the Memtable has reached its size capacity (memtable_size), the Memtable must flush to SST. The Memtable converts the entire AVL tree into a Sorted String Table (SST) and flushes this data to persistent storage, ensuring efficient memory usage and data durability. When the Memtable is converted into an SST, it is traversed using in-order traversal. This along with the AVL structure of the Memtable ensures that the data being put out to the SST is sorted by key from least to greatest. The Memtable writes data to the SST in 4KB pages. It accumulates up to 4KB of data in a buffer (comprising 256 entries, given that each key-value pair occupies 16 bytes), and upon filling the buffer, writes this data to the SST. In cases where the buffer isn't fully occupied but no more data remains in the Memtable, the buffer is padded with zeros to meet the 4KB page requirement. This methodical approach ensures efficient use of storage space in the SST. Following the successful transfer of data to an SST, the Memtable undergoes a reset, clearing all of its nodes and starting fresh with an empty root_node. This reset is pivotal for the Memtable to continue accepting new data entries without hindering performance or exceeding memory limits.

2.5 Memtable Get Functionality

The get function is responsible for retrieving the value for a given key in the Memtable. The get function uses in-order traversal on the AVL tree. If the key is less than the current node's key, the search moves to

the left subtree; if greater, to the right subtree. Once the node with the matching key is found, its value is returned. If the key does not exist in the tree, the function returns -1, indicating a failed search.

2.6 Memtable Scan Functionality

The scan function is responsible for retrieving and returning all key-value pairs within a specified key range. The range query involves in-order traversal of the tree. The function recursively explores the tree, collecting all key-value pairs that fall within the specified range. The scan functionality supports efficient data aggregation. The collected key-value pairs are aggregated into a vector and returned as the result of the range query. This functionality is particularly useful for operations requiring data analysis over a range of keys. If no keys are found in the Memtable, the scan will return an empty list.

2.7 Database SortedSST Structure and API

The Database is where the I/O operations take place when accessing an SST. The SSTs in this part are ordered in ascending order by the key's value. This allows us to run a binary search over the keys in each SST in order to acquire the value. The sorted structure holds every time we write to a database too which is taken from the in-order traversal of the memtable. The database is initialized by using `Database(memtable_size, bufferpool_capacity)` where `memtable_size` is the number of entries allowed in the memtable and `bufferpool_capacity` is the capacity allowed in the bufferpool. For this part of the project, we do not need to worry about the `bufferpool_capacity` so we can set that value to 0 and also call the function `Database.setBufferpoolEnabled(false)` in order to disable the usage of the bufferpool. The follwing describes our implementation of the database API.

2.8 Databse::Open(const string &db_name, const string &database_type)

The `Database::Open()` function is a critical part of the database system, designed to initialize the database environment, including setting up the directory structure and loading existing Sorted String Tables (SSTs) if available.

2.8.1 Overview

- **Function Purpose**: `Database::Open()` prepares the database for operations by setting up its environment based on the provided database name and type.
- **Parameters**:
  - `db_name`: A string representing the name of the database directory, typically the path to the database directory.
  - `database_type`: A string that specifies the type of database being used, influencing how data is managed and stored. The type used for this phase will be `SORTED_SST`

2.8.2 Functionality

1. **Setting Up Directory Structure**:
   - The function first sets `database_dir` to the provided `db_name`. This variable is used as the root directory for all database-related files.
   - The `memtable` is also informed of the database name through `set_db_name()` call, linking the memtable operations with the database directory.
2. **Directory Validation and Creation**:
   - It checks if the directory specified by `database_dir` exists using the `stat()` function.
   - If the directory does not exist (indicating a new database setup), it creates the directory with read-write-execute permissions for the database.

3. **Loading Existing SSTs**:
    - If the directory already exists, it implies that the database might have been previously used and could contain existing SST files.
    - The function then calls `get_ssts_from_db(db_name)` to load these SST files into the database's current session.
    - This involves scanning the database directory for files with a `.bin` extension (indicative of SST files) and adding their names to the `sst_files` vector.

- The SST file$_n ames are sorted in alphabetical order. This is because the SST s are created with an attaching_\#(number) in the o$

4. **Database Type Configuration**:
    - The `db_type` variable is set based on the `database_type` parameter. This setting tells the database the structure of the SST (sorted or b-tree) to is in use.
    - Similarly, the memtable is informed of the database type through `set_db_type(database_type)`. This allows the memtable to flush its data to the correct SST type and in the correct SST format.

2.8.3 Significance

- **Flexibility and Scalability**: The `Open` function provides a flexible way to either create a new database or seamlessly connect to an existing one, making the system adaptable and scalable.
- **Data Persistence and Recovery**: By loading existing SSTs, the function plays a crucial role in data persistence and recovery, ensuring that data stored in previous sessions is not lost and can be efficiently accessed in the current session.
- **Customization Based on Database Type**: The ability to specify a database type allows for customized behavior of the database system, potentially accommodating various use cases and optimizations.

2.9 Database::Put(const int64_t &key, const int64_t &value)

The `Database::Put()` function adds a new key-value pair to the database, ensuring that they are stored efficiently either in the memtable or in the SSTs.

2.9.1 Overview

- **Function Purpose**: The primary role of `Database::Put()` is to insert or update key-value pairs in the database.
- **Parameters**:
    - `key`: An integer representing the key part of the key-value pair.
    - `value`: An integer representing the value part of the key-value pair.

2.9.2 Functionality

1. **Insertion in Memtable**:
    - The function delegates the task of inserting the key-value pair to the memtable's `put()` function. This is the first attempt to store the data in the database's in-memory structure.
    - The `put()` method of the memtable runs and keeps AVL tree balance as described above.
2. **SST Creation and Loading**:
    - If the insertion in the memtable leads to the creation of an SST (due to memtable reaching its size capacity), the `sstCreated` flag is set to `true`.

- When an SST is created, the `get_ssts_from_db()` method is called to update the database's state with the newly created SST. This involves loading the SST into the database's current session for future data retrieval operations.

2.9.3 Significance

- **Efficient Data Insertion**: The `Put` function ensures that data is inserted efficiently, first attempting to store it in the faster in-memory memtable.
- **Automated Data Persistence**: On reaching the memtable's capacity, a flush to SST ensures data durability and efficient utilization of memory resources.
- **Dynamic Data Management**: The function embodies a dynamic approach to data management, balancing between in-memory storage and disk-based SSTs based on the volume of data being handled.

2.10 Database::Get(const int64_t &key)

The `Database::Get()` function is designed to retrieve the value associated with a specific key. It employs a binary search algorithm when searching in Sorted String Tables (SSTs) for efficient data retrieval.

2.10.1 Overview

- **Function Purpose**: Retrieval of the value for a given key from the database.
- **Parameters**:
  - `key`: An integer key for which the corresponding value is sought.

2.10.2 Functionality

1. **Searching in Memtable**:
   - Initially, the function attempts to find the value in the memtable using the `memtable.get(key)` method.
   - If the value is found in the memtable, it is immediately returned.
2. **Searching in SSTs**:
   - If the key is not in the memtable, the function iterates over the SST files in order from youngest to oldest.
   - For each SST file, the `binarySearch` method is called to find the key within the file.
3. **Binary Search Algorithm**:
   - The `binarySearch` method performs a binary search over the SST file to locate the key.
   - It reads pages (each page being 4KB) from the SST file and searches within these pages for the key.
   - The search continues by adjusting the search range based on the comparison with the last key read from each page.
   - If the key is found, its corresponding value is returned; otherwise, the search continues in other SSTs or returns -1 if the key is not found in any SST.

2.10.3 Significance

- **Efficient Data Retrieval**: The `Get` function provides a fast and efficient means of retrieving data, first checking the in-memory memtable and then leveraging a binary search algorithm for SSTs.
- **Optimized Search**: The binary search algorithm significantly reduces the time complexity, especially in large datasets, by limiting the number of disk reads.

2.11 Database::Scan(const int64_t &key1, const int64_t &key2)

The `Database::Scan()` function facilitates efficient range queries, retrieving all key-value pairs within a specified key range.

2.11.1 Overview

- **Function Purpose**: Retrieve all key-value pairs within a given key range from the database.
- **Parameters**:
  - `key1`: The lower bound of the key range.
  - `key2`: The upper bound of the key range.

2.11.2 Functionality

1. **Initial Checks and Set Creation**:
   - The function starts by ensuring that `key1` is less than `key2` to define a valid range.
   - A set (`valueSet`) is created to store unique key-value pairs.
2. **Scanning Memtable**:
   - The function first retrieves values from the memtable using the `memtable.scan()` method.
   - Retrieved values are inserted into `valueSet`.
3. **Scanning SSTs**:
   - The function iterates over each SST file from newest to oldest and employs `binarySearchScan` to find relevant key-value pairs.
   - Unique pairs found in SSTs are added to `valueSet`.
4. **Binary Search Scan Algorithm**:
   - The `binarySearchScan` method performs a binary search over the SST file to locate the range of keys.
   - It reads pages of data and searches for key-value pairs within the specified range.
   - The search adjusts its range based on the keys found, ensuring that only relevant pages are read.
   - This method enhances the efficiency of range queries by limiting disk reads to only necessary pages.
5. **Result Aggregation**:
   - The function aggregates all key-value pairs from both the memtable and SSTs.
   - A `ScanResponse` object is populated with these aggregated results and returned.

2.11.3 Significance

- **Range Query Efficiency**: `Scan` provides a comprehensive and efficient mechanism for retrieving a range of key-value pairs from both in-memory and disk-based storage.
- **Optimized Disk Access**: The binary search scan approach minimizes disk access, crucial for large datasets, thereby enhancing performance.

2.12 Database::Close()

The `Close()` function in the Database is responsible for finalizing operations on the ‘Memtable‘ before the database is closed. The primary role of this function is to ensure that all the data in the ‘Memtable‘ is appropriately flushed to persistent storage in the form of an SST

2.12.1 Implementation

The ‘Close()‘ method in the ‘Database‘ class calls the ‘close()‘ method of its ‘Memtable‘ instance.

2.12.2 Functionality

When ‘Close()‘ is invoked, the ‘Memtable‘ checks the database type (either BSST or SST). Depending on this type, it calls either ‘convertMemtableToBSST()‘ or ‘convertMemtableToSST()‘. These methods handle the conversion of the in-memory AVL tree structure into a format suitable for storage on disk, ensuring that all data in the ‘Memtable‘ at the time of closing is not lost but rather persisted for future access.

- In the case of BSST, the ‘convertMemtableToBSST()‘ method is called, which converts the AVL tree into a B-tree Sorted String Table, optimized for quick search operations.
- In the case of SST, the ‘convertMemtableToSST()‘ method is called, which converts the AVL tree into a Sorted String Table, ensuring data is stored in a sorted manner.

This approach ensures that the ‘Memtable‘’s state is effectively and efficiently preserved in the database’s storage layer, making the ‘Close()‘ function a vital component of the database’s data integrity and persistence mechanisms.

2.13 Experiments

It is important to note that all three experiments in this section are done on the teach.cs machines which is why the throughput times are slower than one might expect on a device with better specs.

2.13.1 PUT Operation Throughput

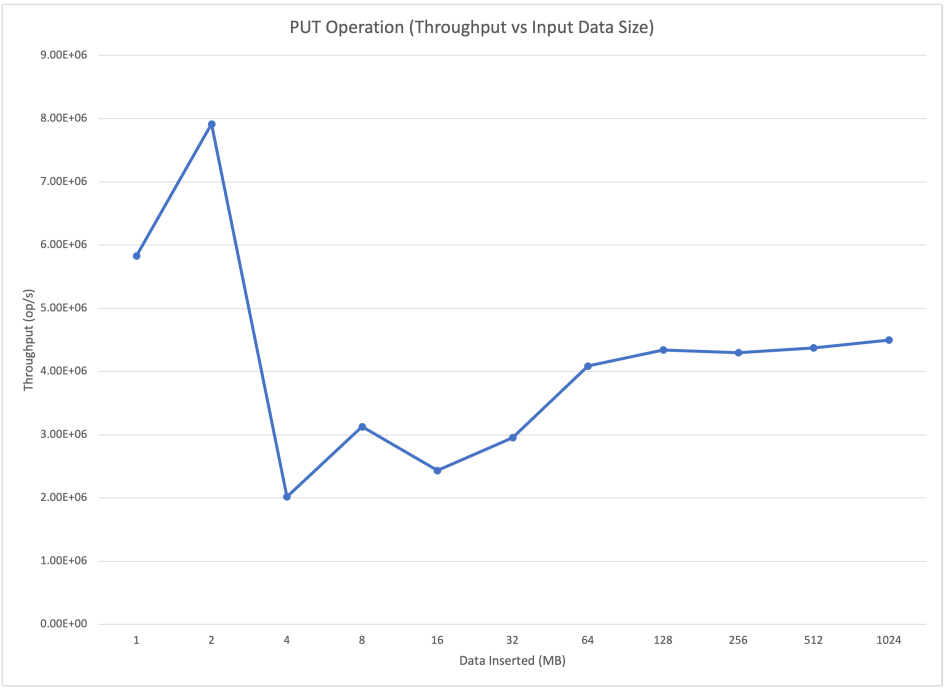

Figure 1: Throughput of PUT operations as a function of data inserted.

The PUT operation graph in Figure 1 shows a peak throughput at a data size of 2 MB with a throughput of 7.29×10^6 operations per second. After that peak, there is a significant drop as the data size increases to 4 MB with a throughput of 2.02×10^6 operations per second. This sudden downturn can be attributed to the memtable's capacity limit. The memtable, an in-memory buffer limited to 4 KB, overflows once the insertion exceeds this threshold, necessitating a time-consuming flush to Sorted String Tables (SSTs) on disk.

After the 4 MB mark, the throughput increases as the volume of data inserted escalates. This trend indicates the system's effective adaptability to handling a larger number of PUT requests. This is possibly due to the more representative sample size of operations. In our experiment, we doubled the data inserted into the database at each benchmark which means that the amount of data grows exponentially after each time we measure the throughput. The larger sample size could be the reason as to why the graph demonstrates a convergence to the throughput of 4.5×10^6 operations per second.

2.14 GET Operation Throughput

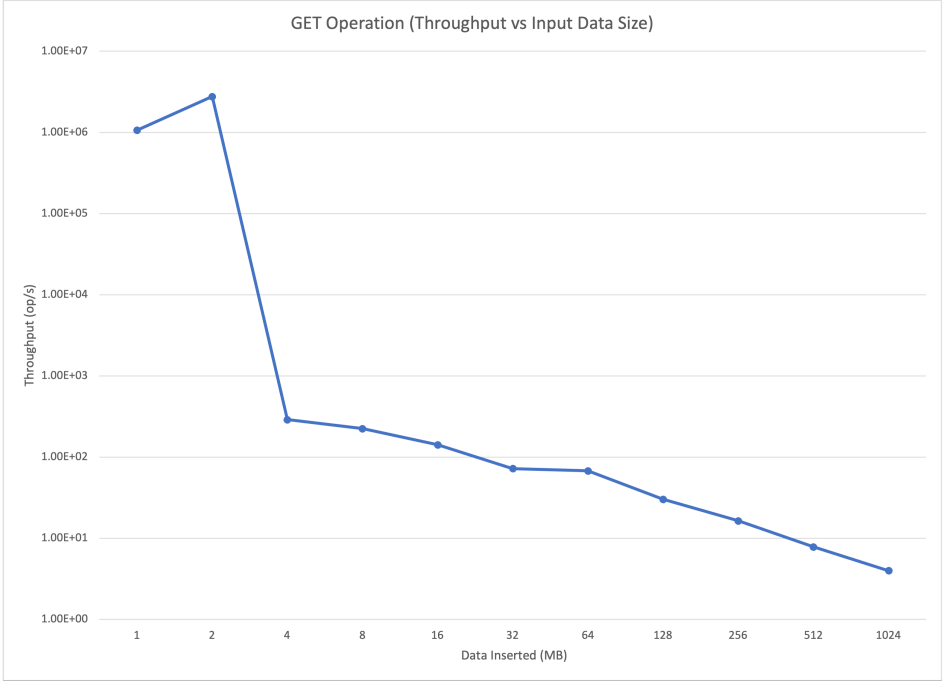

Figure 2: Throughput of GET operations as a function of data inserted.

The GET operation graph shown in Figure 2 demonstrates a pronounced decrease in throughput as the amount of data inserted increases from 1 MB to 1024 MB. The most substantial decline occurs between 2 MB and 4 MB, after which the throughput continues to fall but at a lesser rate. This significant dropoff is again due to the flushing of the memtable at its capacity of 4MB. This is due to memory operations being faster than I/O operations which is why there is a significant drop off. At 4MB we start to search for values that could potentially be in the SSTs which would cause us to use I/O operations which are slower. As the data set grows, the cost of accessing the correct data likely involves more complex lookups and increased access to slower secondary storage. The trend indicates a need for optimized indexing and caching mechanisms to enhance GET operation performance in larger databases.

2.15 SCAN Operation Throughput

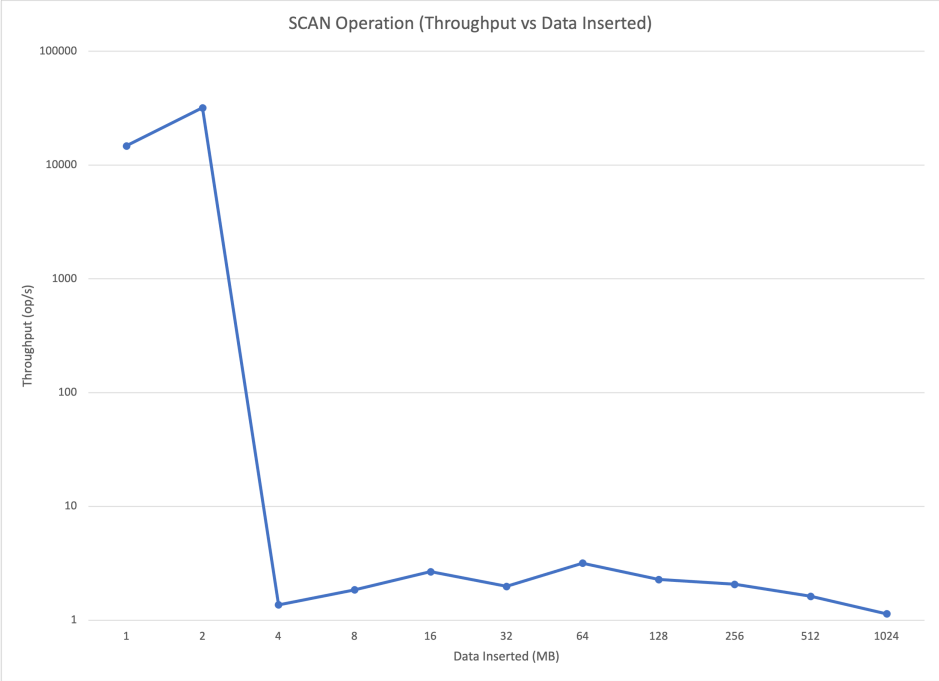

Figure 3: Throughput of SCAN operations as a function of data volume inserted.

In Figure 3, the SCAN operation throughput exhibits a steep drop from 2 MB to 4 MB of inserted data, followed by a more stable but still descending pattern. The initial drop is due to the memtable size of 4MB. Before 4MB of data is inserted, only the memtable is scanned for values which uses memory. When the database hits a size of 4MB, the data is flushed from the memtable and put into SSTs which uses I/O operations. This explains the significant drop-off in performance as I/O operations are more expensive than in memory.

As the data size further increases, the throughput decrease becomes more gradual, which indicates that the system's performance becomes more I/O-bound, with disk seek times dominating the performance characteristics. The graph suggests that scanning operations are significantly affected by the size of the data set, pointing to potential areas for optimization in database scan strategies.

3 Step 2 - Buffer Pool & Static B-trees

3.1 Project status

All required functionality and structures have been implemented for this step.

3.2 Buffer Pool Implementation

The `Bufferpool` class uses a hash table to store pages, where each bucket of the table contains a linked list (`Bucket`) of pages. This design allows for efficient retrieval of pages. The class uses the Least Recently Used (LRU) eviction policy to manage memory by keeping track of the pages accessed and removing the less frequently accessed ones when needed. The use of `MurmurHash3` indicates a focus on efficient and reliable hashing. The class also has dynamic resizing capabilities, which is crucial for adapting to different workload sizes.

3.2.1 Bucket

`Bucket` is a struct representing a frame in the buffer pool. Each `Bucket` contains:

- `page_id`: A unique string ID for the page in the form of `<file name>#<page offset>`, where `#` serves as a divider.
- `data`: A vector of `KeyValuePair`, representing the actual data stored in the page.
- `next`: A unique pointer to the next Bucket in the chain, allowing the implementation of a linked list for each hash table entry.
- `lruNode`: A shared pointer to an `LRUNode`, linking this Bucket to the buffer pool's LRU queue.

3.2.2 Hash Table Structure

The `Bufferpool` keeps track of the following variables:

- `table`: A hash table implemented as a vector of unique pointers to `Bucket` objects.
- `queue`: An instance of `LRUQueue`, used for eviction policy.
- `capacity`: The maximum number of pages that can be stored, each page is 4KB.
- `num_buckets`: The current size of the hash table.
- `num_pages`: The current size of the hash table.

1. **Constructor**
   When constructing a buffer pool, a specific size is provided for its `capacity`. Then, it calculates `num_buckets` as the nearest power of two to the `capacity` and resizes the table accordingly.
2. **Hash Function**
   The Bufferpool utilizes `MurmurHash3` for its hash table implementation. `MurmurHash` is a non-cryptographic hash function developed by Austin Appleby. Distinctively designed for rapid hashing, MurmurHash significantly reduces collision occurrences and shows a performance advantage over CRC, MDx, and SHAx algorithms. Its efficiency makes it an ideal choice for identifying duplicates and is exceptionally well-suited for HashTable index computations. Many contemporary databases employ `MurmurHash`, including Redis, Elasticsearch, and Cassandra. `MurmurHash3`, in particular, has been a fundamental contributor to numerous performance enhancements in database technologies during the current decade. Therefore, it is highly effective for general hash-based lookups, suitable for the buffer pool hash table. The source code of this algorithm is accessible at `MurmurHash` on GitHub.

3. **Collision Resolution**
   We used **chaining** to achieve collision resolution, which deals with the case that multiple pages map to the same bucket in the buffer pool.

   - **Bucket Structure**
     Each `Bucket` in the buffer pool contains a `unique_ptr<Bucket>` next, which points to the next `Bucket` in the chain. This structure allows each bucket to hold a linked list of `Bucket` objects.
   - **Insertion**
     When a new page is inserted by calling `Bufferpool::insert`, the `Bufferpool` computes the hash key given the page ID. If the computed bucket index already contains a `Bucket`, the new `Bucket` is added at the beginning of the linked list for that bucket, effectively pushing the existing buckets down the chain.
   - **Search and Removal**
     During a search using `Bufferpool::search` or eviction calling `Bufferpool::evict`, the `Bufferpool` starts at the first `Bucket` in the relevant bucket's chain and traverses the linked list until it finds the target `Bucket` or reaches the end of the list.

4. **Automatic Memory Management**
   The combination of `vector` and `unique_ptr` in this buffer pool implementation enhances memory safety, provides clear ownership semantics, and ensures efficient memory and resource management, which are crucial for a robust and high-performing buffer pool in a database management system.

3.2.3 Functionality and Cost Analysis

Since the buffer pool does not involve reading from disk, the cost analysis is solely based on CPU cost. N stands for the number of pages in buffer pool(`num_pages`).

1. `void Bufferpool::insert(const std::string &page_id, vector<KeyValuePair> &value)`
   This method adds a new page to the buffer pool.

   (a) It first checks if the buffer pool has available capacity; if the buffer pool is full, it evicts a page. The average cost of this operation is $O(1)$, even when it needs to evict a page. The worst-case cost of this operation is $O(N)$ due to eviction. We discuss this in more detail in the `evict` section.

   (b) Then, it calculates the hash index for the page ID. Using a good hash function like MurmurHash3, the cost of computing the hash key is $O(1)$.

   (c) Then it constructs a new `Bucket` and inserting it at the beginning of the linked list in the identified bucket in `table`. It initializes a new `lruNode` and add it to `queue` to track the usage of this page. This is an O(1) operation since it only involves updating pointers and does not require traversing the chain for both `table` and `queue`.

   Therefore, the average cost of `insert` is $O(1)$, and the worst-case cost is $O(N)$.

2. `vector<KeyValuePair> *Bufferpool::search(const std::string &page_id)`
   This method removes a page from the buffer pool.

   (a) It first computes the hash index for the page ID with $O(1)$ cost.

   (b) Then, it searches the corresponding bucket's chain for the page. The average cost of this operation is $O(1)$, but the worst-case cost is $O(N)$ if many pages hash to the same bucket(long chaining).

   (c) If a page is found, it moves the `lruNode` of the page to the tail of `queue`, indicating recent access, and returns a pointer to the page's data. It returns `nullptr` if the page is not found. The cost of moving `lruNode` is $O(1)$ and is discussed in detail in the **LRU** section.

   Therefore, the average cost of `search` is $O(1)$, and the worst-case cost is $O(N)$.

3. `bool Bufferpool::remove(const std::string &page_id)`
   This method searches for a page in the buffer pool and returns the page if it is found.

(a) It first finds the bucket corresponding to the hashed page ID with $O(1)$ cost.

(b) Then, it traverses the chain in the bucket to find and remove the specified `Bucket` that wraps the page. For `lruNode` and `Bucket`, memory is automatically released when there are no pointer pointing to the object.
The average cost of this operation is $O(1)$, but the worst-case cost is $O(N)$ if many pages hash to the same bucket(long chaining).

(c) Returns true if the page was found and removed, false otherwise with $O(1)$ cost.

Therefore, the average cost of `remove` is $O(1)$, and the worst-case cost is $O(N)$.

4. `Bufferpool::evict()`
This method evicts a page that is the least recently used page from the buffer pool.

(a) It removes the head of the `queue`, which corresponds to the `Bucket` that wraps the least recently used page. The cost is $O(1)$ and is discussed in detail in the **LRU** section. The average cost of this operation is $O(1)$, but the worst-case cost is $O(N)$.

Therefore, the average cost of `evict` is $O(1)$, and the worst-case cost is $O(N)$.

5. `Bufferpool::resize(size_t new_capacity)`
This method adjusts the buffer pool's capacity, and resizes the hash table when the buffer pool is expanded or contracted too much.

(a) It sets `capacity` to `new_capacity` and updates `num_buckets` to the nearest power of two that is greater than or equal to the `new_capacity` with $O(1)$ cost.

(b) If `new_capacity` is smaller than `num_pages`, the buffer pool evicts pages until `num_pages` is no more than `new_capacity`. Each eviction costs $O(1)$ on average but in the worst case, the cost is $O(N)$.

(c) (extra functionality) If the `num_buckets` changes, the buffer pool creates a new hash table with `num_buckets`. The buffer pool then rehashes all the entries from the old table into the new table and then replaces the old table with the new one. This prevents the directory size from becoming a memory bottleneck and prevents long chaining. This operation is $O(N)$ cost.

Mitigating the Worst-Case Scenario by Handling Long Chains:
The primary factor contributing to the $O(N)$ worst-case complexity in the buffer pool is the potential for long chains due to hash collisions. If many entries hash to the same bucket, the chain at that bucket becomes lengthy.
Several measures are implemented to effectively mitigate the occurrence of long chains. These include:

- **Optimal Bucket Sizing:** the `Bufferpool` ensures that there are a sufficient number of buckets. It does this by maintaining `num_buckets` to the nearest power of two greater than or equal to the `capacity` of the `Bufferpool`. This approach aims to limit the occurrence of hash collisions and ensures efficient indexing. By having `num_buckets` that is a power of two, the hash table can distribute entries more evenly across the buckets, reducing the likelihood of long chains and maintaining efficient access times.

- **Efficient Hash Function:** The use of a well-designed hash function like `MurmurHash3` aids in evenly distributing the entries across the available buckets. This uniform distribution is key to preventing the formation of long chains.

- **Dynamic Resizing:** The buffer pool is equipped to dynamically resize based on the `capacity`. This resizing redistributes the entries, maintaining an optimal load factor and preventing any single bucket from becoming overly congested with entries.

3.3 Intergrate Buffer Pool With Queries

3.3.1 Get Query Workflow

We integrated the access to the buffer pool as a part of our `Get` workflow. `Get` retrieves the value for a given key from the database. You can read more about the original `Get` API in **Section 1.9**.

- **A new function is created in the database:**
  `void Database::find_page(const int &fd, vector<KeyValuePair> &buffer, const int64_t &offset, const string &file_name)`
  This function retrieves a specific page from a database file and populates it into a provided buffer. It handles both scenarios where a buffer pool is used and where it is not.
  Parameters:
  - `fd`: File descriptor of the database file.
  - `buffer`: A reference to a vector of `KeyValuePair` that will hold the retrieved page data.
  - `offset`: The offset in the file where the desired page is located.
  - `file_name`: The name of the file from which the page is to be read.

  1. It first constructs a unique identifier in the form of `<file_name>#<offset>`. This ID is used to check if the page is already in the buffer pool.
  2. If the buffer pool is enabled, it searches the buffer pool for the page using the `page_id` by calling the `Bufferpool.search(page_id)` method, which should return a pointer to a `vector<KeyValuePair>` that stores the page if the page is found. If the result is not a `nullptr`, it copies the data into the `buffer` and returns immediately.
  3. If the buffer pool is disabled or the page is not found, it proceeds to read the page into `buffer` directly from the file.
  4. If the buffer pool was enabled, after a successful read from the file, the newly read page data is inserted into the buffer pool by calling `Bufferpool.insert(page_id, buffer)`. This ensures that subsequent requests for the same page can be served directly from the buffer pool.

- **New Design of Get API**:
  1. **Searching in Memtable**:
     - This step remains the same, where the function attempts to find the value in the memtable using the `memtable.get(key)` method.
  2. **Searching in Bufferpool/SST**:
     - If the key is not in the memtable, the function iterates over the SST files by calling `find_page` method. `find_page` prioritizes reading from the buffer pool to minimize disk I/O operations and inserts pages into the buffer pool after reading them from the disk, optimizing future accesses.
  3. **Searching in the Page**:
     - After getting the page, the function performs a search over the page to locate the key. The type of search depends on the type of SST that is in use. It can be Sorted SST or Static B-Tree SST.
     - If the key is found, its corresponding value is returned; otherwise, the search continues in other SSTs or returns -1 if the key is not found in any SST.

- **Significance**
  - When Get is called, if the data is found in the buffer pool, it can be retrieved without accessing the disk, reducing disk I/O operations.
  - This means that repeated reads of the same data are much faster, as they avoid the need for disk access after the initial read.

- The use of LRU ensures that the most frequently or recently accessed pages are kept in memory, while less frequently accessed pages are written back to the disk. This ensures an optimal balance between memory usage and data access speed.

3.3.2 Scan Query Workflow

For scan queries, we have decided to bypass the buffer pool, which presents the following advantages and trade-offs:

- **Advantages:**
  - **Sequential flooding** happens when scan queries load large amounts of data into the buffer pool, which can displace frequently accessed data. By not using a buffer pool for such queries, we prevent this pollution, ensuring that the buffer pool remains efficient for more frequent, random access patterns.
  - The buffer pool is most effective when it contains frequently accessed data. Scan queries typically read data once and may not need it again soon. Avoiding the buffer pool for these queries helps maintain its efficiency for regular queries that benefit more from quick data access.
  - When large scan queries use the buffer pool, they often lead to the eviction of useful data. By bypassing the buffer pool for scans, we avoid the overhead associated with cache eviction and re-population.
- **Trade-offs:**
  - Scan queries will always result in disk I/O, which could slow down query operations.
  - If the same data is scanned multiple times, not using the buffer pool means each scan accesses the disk, leading to lower performance.

3.4 LRU Eviction Policy

LRU Eviction Policy is used in the buffer pool to manage memory by keeping track of page access patterns and finding the least frequently accessed pages when needed.

3.4.1 LRUNode

`LRUNode` Represents a node in the LRU queue. Each `LRUNode` contains:
Attributes:

- `page_id`: A unique string ID for the page in the form of `<file name>#<page offset>`, where `#` serves as a divider.
- `prev, next`: Two `shared_ptr` of `LRUNode`, allowing for the creation of a doubly-linked list.

Constructor: Initializes an `LRUNode` with the provided `page_id` and sets `prev` and `next` to `nullptr`, indicating the node is initially disconnected from any others in the queue.

3.4.2 LRUQueue

Attributes: `LRUQueue` contains `head, tail`, which are two `shared_ptr` of `LRUNode`, allowing for the creation of a doubly-linked list.
Constructor: Initializes an empty LRU queue with `head, tail` set to `nullptr`.
Functions:

- `void put(shared_ptr<LRUNode> node)`
  Adds a new node to the tail of the queue.
  Preconditions: The node must not already exist in the queue.

- `void move_to_tail(shared_ptr<LRUNode> node)`
  Moves an existing node to the tail of the queue, indicating recent use.
  Preconditions: The node must exist in the queue.

- `string remove_head()`
  Removes the head node (the least recently used node) from the queue and returns its key.
  Used for eviction in the buffer pool when it is full.

3.4.3 Memory Management

The use of `shared_ptr` for node management suggests an emphasis on automatic memory management, likely to prevent memory leaks and simplify the handling of dynamic memory.

3.4.4 Doubly-linked List and Pointers to Head and Tail

We have chosen to use a doubly-linked list for our LRU Queue, storing pointers to both the head and tail of the queue.
This approach facilitates easier movement of nodes, as each node is directly linked to its predecessor and successor. Such a design is vital for LRU implementation, which often necessitates moving nodes to the tail of the list in response to access patterns. Additionally, having pointers to both the head and tail simplifies the removal of the head node. Node removal is also more efficient in a doubly-linked list, as it can be achieved in O(1) time without the need for traversal. This efficiency is particularly advantageous in implementing the **Delete** API in **Step 3**.

However, this design incurs extra memory overhead due to the additional pointers, resulting in higher memory consumption compared to a singly-linked list. Implementing operations that involve updating both head and tail pointers can increase the complexity of the code, potentially leading to errors like incorrect pointer updates.

3.4.5 Advantages

- LRU is relatively simple to implement and understand. It provides a good approximation of the hot pages by assuming that past usage is a good indicator of future usage.

- LRU is effective in exploiting temporal locality. If a data item is accessed, it's likely to be accessed again soon. By keeping recently accessed items in the buffer pool, LRU increases cache hit rates.

- LRU automatically adapts to changing access patterns, as it continuously updates the priority of pages based on their access history without the need for manual tuning.

- Since we bypass the buffer pool for scan, we avoid potential disadvantages of LRU policy, such as thrashing, where pages are constantly being loaded and evicted.

3.4.6 Trade-offs

- Implementing LRU requires maintaining a linked-list of pages, which can introduce overhead in terms of time (for reordering the list on each access) and space (for storing the list itself).

- As the size of the buffer pool increases, the cost of maintaining the LRU list can become significant.

3.5 Static B-Trees

3.5.1 Static B-Tree SST

Assume one page is 4KB (4096 bytes). We now outline the design for our static B-Tree SST (BSST) structure and the roles each of its constituents play:

1. The metadata page is at the very front of the SST at offset 0 and contains a collection of 8-byte integers. One of these integers we will refer to as `entries_offset`, and it is the only one important for Step 2. This integer denotes the page offset in the file at which the leaf nodes or entries begin and is an important piece of information that allows the database to traverse the BSST with $log_B(N)$ page-sized reads.

2. The root node is located at offset 4096 after the metadata page. It may be an internal node that stores references to other internal nodes or leaf nodes lower in the B-Tree, or it may itself be a leaf node that stores entries only in the case where a database flushes a tiny amount of entries as a BSST.

   An internal node can store up to 256 (4096 / 16 [8-byte key-value pair]) pairs, and a leaf node can store up to 256 entries for the same reason. This separation of classes can be found in `src/b-tree.hh` Both types of nodes store pairs of 8-byte integers, where leaf nodes store key-value pairs, but internal nodes store a different kind of information:

   For each of its children, an internal node stores a pair of numbers: the maximum key stored by the child and a byte offset into the file where the child page is stored. These pairs are sorted in ascending order by maximum key to enable efficient tree traversal with binary search, as discussed in lecture. Recursively reading and seeking to these offsets, from the root node down to the internal nodes, is how the database traverses the BSST. By having this structure, we ensure that reading each node costs only one storage I/O and so the search I/O cost would be $log_B(N)$, as we saw in lecture.

3. Leaf nodes are placed after all internal nodes in the file, and they are structured like a regular SST from Step 1. Leaf nodes toward the end that are not full of entries are padded with zero to ensure reads are page-aligned. Below is an illustration of the overall BSST structure and where the leaf nodes/entries are placed:

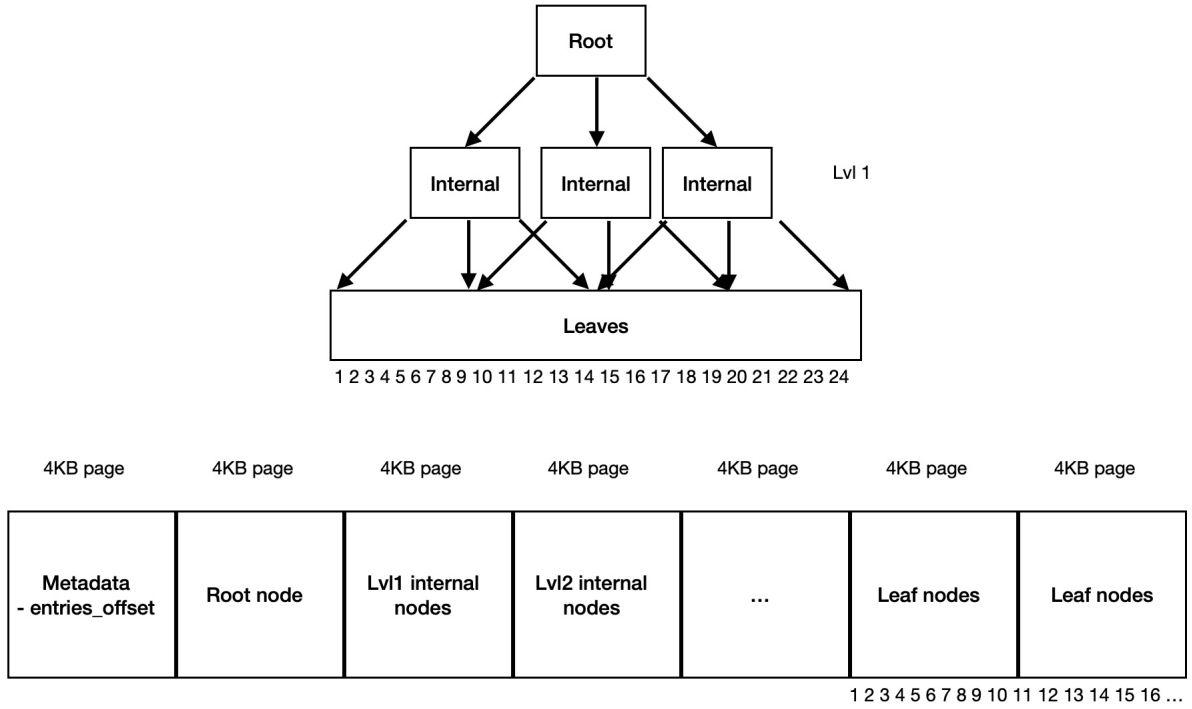

Figure 4: B-tree SST structure

We chose this simple structure to store the leaf nodes so that scans would be simplified. During a scan operation, the database would only need to find the leaf node l_1 that contains the smallest entry in the scan range, then perform consecutive page-sized reads starting at the offset of l_1 to gather entries in the range until the end of the file is reached.

4. The database reads the BSST through the `p_read` function, which accepts a file descriptor and a byte offset. To determine whether it is reading an internal node or a leaf node containing entries, the database will check if the `p_read` offset is $\geq$ `entries_offset`. If it is, then it knows that page is a leaf node, and will perform a binary search to search for the target key. This is the role of `entries_offset` stored in the BSST metadata page, and this logic involving `entries_offset` can be seen in `searchBTree` in `src/database.cc`.

Depending on the database's `db_type`, the database will flush its memtable as a BSST. To facilitate the file structure laid out above, the database first converts the memtable into an in-memory B-Tree. Then, the database performs a Level-Order traversal of the in-memory B-Tree with a Breadth First Search approach to write the nodes into storage. This introduces some overhead when the database flushes its memtable, particularly in terms of computational complexity and I/O operations. However, this method ensures that the data is stored in a sorted and efficient manner, facilitating faster read and search operations later.

The Level-Order traversal ensures that the data is written in a way that preserves the B-Tree's properties, maintaining efficient search capabilities. The resulting file, a BSST, is immutable, allowing the database to take advantage of the read-optimized structure of a B-Tree relative to a regular SST.

The flushing logic can be found in functions `convertMemtableToBSST, constructBTree, and writeToBSST` of `src/memtable.cc` for in-memory B-Tree construction and flushing the B-Tree to storage.

3.6 Step 2 Experiments

The goal of this experiment is to compare our initial binary search to B-tree search. We conduct the experiment by setting memtable size to 8 MB and buffer pool size to 10 MB for both database types, and measure GET throughput at increasing database sizes in powers of 2. That is, at each step the data inserted is doubled. At each data size, we only query a number of queries whose total size is a percentage of the database size at that point to avoid the experiments running for too long. The inserted and queried entries are also consistent between the databases to ensure a valid comparison:

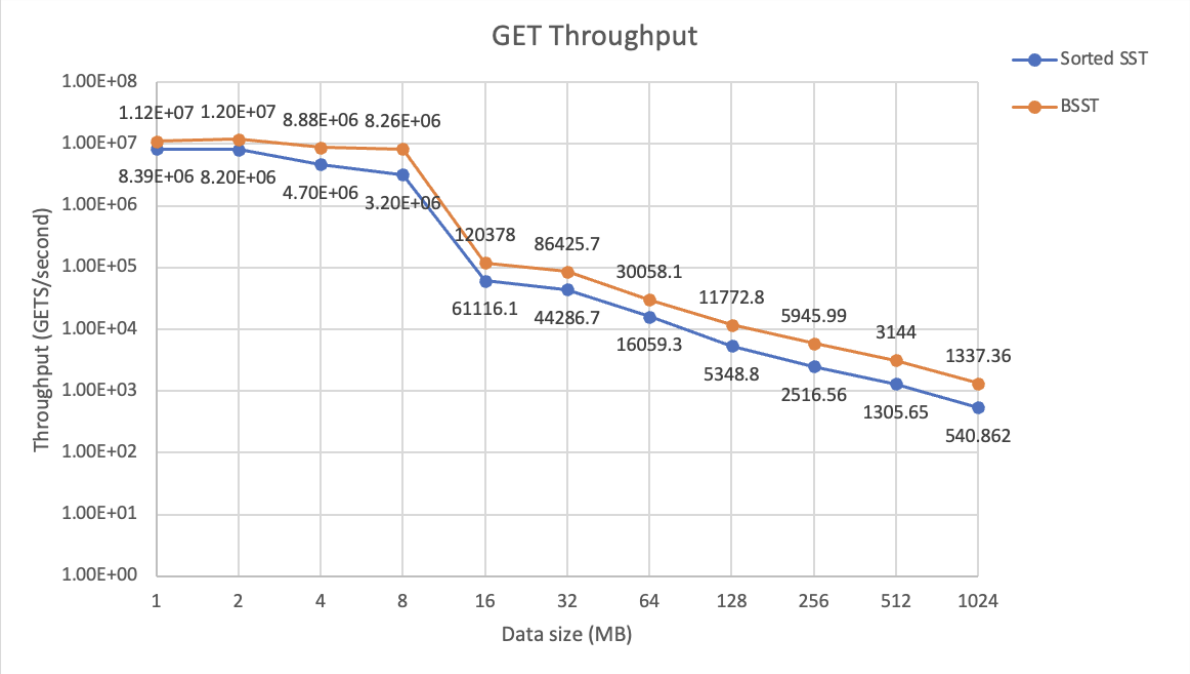

Figure 5: Throughput of GET operations for binary search and B-tree search as a function of data inserted.

As expected, we observe that the B-tree search outperforms the binary search, by around 2 times more GETs per second. This is because as the data size grows, the search cost $log_2(\frac{N}{B})$ for binary search increases much more than the search cost $log_B(N)$ for B-tree search (where $B = 256$), as we saw in lecture. Furthermore, we notice a drop in throughput after 8 MB. This is expected since the memtable was 8 MB, and doubling the database size incurred flushing data to disk. Any queries before 16 MB would have gone through the memtable. This further highlights the performance differences between storage I/O and CPU operations. Furthermore, we couldn't notice the effects of the buffer pool on the GET throughput for both database types. This could be because the point queries we generate were random, thus the buffer pool that's geared towards improving skewed accesses didn't help that much in this experiment.

To more clearly see the performance difference, we plot the B-tree get throughput as a ratio of that for binary search:

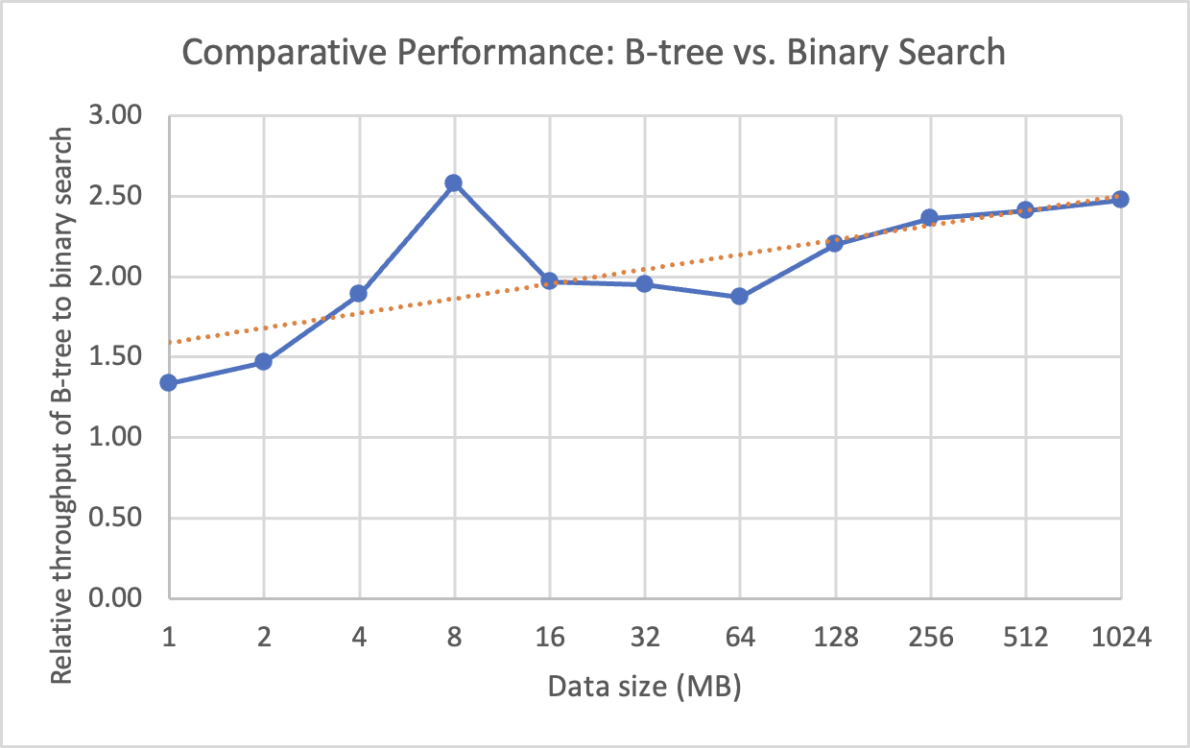

Figure 6: GET throughput of B-tree search divided by throughput of binary search as a function of data inserted.

We observe a steady increase in relative performance for the B-tree, as shown in the trend line. This highlights the increasing performance of B-tree search in comparison to binary search as data size grows. Again, this is because as the data size grows, the search cost $log_2(\frac{N}{B})$ for binary search increases faster than the search cost $log_B(N)$ for B-tree search (where $B = 256$), as we saw in lecture.

4 Step 3 - An LSM-Tree with bloom filters

4.1 Project status

All required functionality and structures have been implemented for this step.

4.2 Bloom filters

Each B-tree SST file in each level has a corresponding bloom filter implemented in the BloomFilter class, which is initialized by two parameters: `num_entries` and `custom_bits_per_entry`. The latter has a default value of 10. The class when initialized creates a filter vector of `num_entries` $*$ `custom_bits_per_entry` bits, and also computes the optimal number of hash functions with `ceil((log(2)) * M)`, where $M =$ `custom_bits_per_entry`. The filter vector houses `int64_t` integers which we use as a bit array. Every time a key is inserted into the Bloom filter and hashed to a bit, we set the bit to 1 if it isn't already 1 by using bitwise OR. When a key is being checked against the Bloom filter, we use a bitwise AND. Like the buffer pool, the Bloom filter uses MurmurHash3 as its hash function. To produce the effect of having $k =$`ceil((log(2)) * M)` different hash functions, we pass in k different seeds to MurmurHash3.

The B-tree SST (BSST) structure thus had to be modified to persist in storage important Bloom filter metadata, the filter itself, and also the seeds:

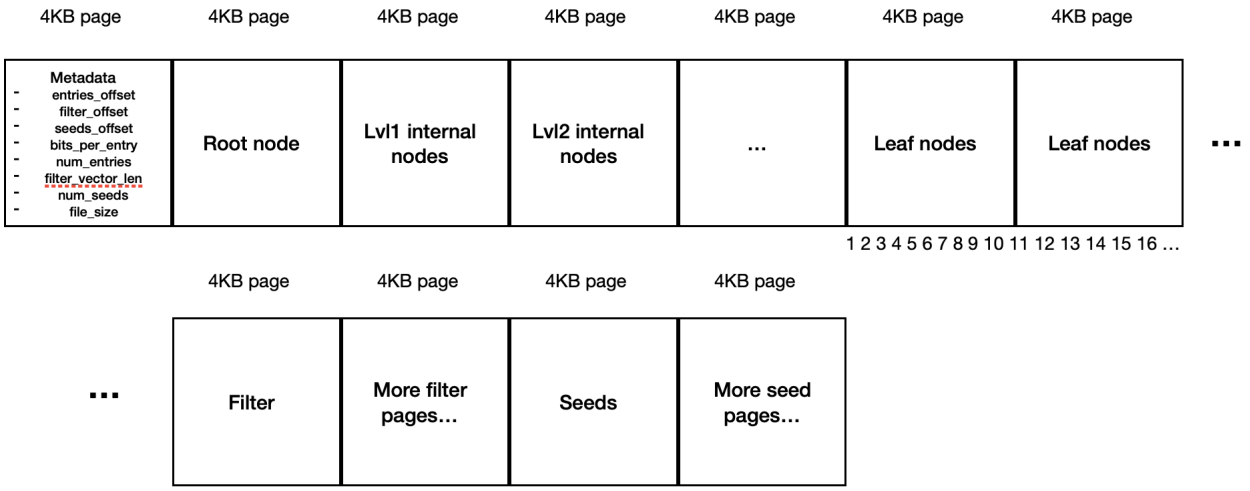

Figure 7: Modified B-tree SST structure to persist Bloom filter

Upon reading a B-tree SST, a BloomFilter object is constructed using the metadata in the metadata page and also the filter and seeds pages at the end of the BSST. This BloomFilter object is then used to check whether a key might be in the BSST. If it returns true, then we run a B-tree search, and if it doesn't then we skip the SST.

4.3 LSM Tree and Compaction

4.3.1 Introduction

This section delves into the implementation of the LSM (Log-Structured Merge) tree within a database system. The focus is on the methods adopted for managing the storage, retrieval, compaction, and state persistence of SST (Sorted String Table) files.

4.3.2 LSM Tree Structure

The LSM tree is represented using a `std::map` in C++, where the key is an integer representing the level in the LSM tree, and the value is a vector of strings. Each string in the vector represents the file_name of an SST file stored in the database. This structure allows efficient organization of SST files across different levels of the LSM tree, facilitating fast access and efficient manipulation during operations like compaction and retrieval.

4.3.3 checkLSMCompaction Method

The `checkLSMCompaction` method facilitates the compaction process within the LSM tree. It iterates over each level of the LSM tree and triggers compaction when it finds two SST files at the same level. It then takes these two SST files and compacts them together into a new SST file which is called SST compaction. The compaction is accomplished by calling the `mergeSortSSTs` method. Post-compaction, the original SST files that were merged are removed from the database and LSM tree, and the new, compacted SST file is escalated to the next higher level in the LSM tree.

4.3.4 mergeSortSSTs Method

The `mergeSortSSTs` method is where the compaction process in the LSM tree takes place. It takes two SST files at the same level and merges them into a single sorted SST file. The method begins by opening the SST files and reading their metadata. Using this metadata, buffers are initialized to store the entries of up to 1 4KB page from these files. The merge sort algorithm is then applied to these entries. In cases where keys are duplicated across files, the method prioritizes the entry from the newer file. After completing the merge sort, the method proceeds to create a new SST file, by calling the `CreateCompactedBSST` method which creates the SST in the Btree format as discussed in part 2. Once the SST is created and flushed to the database, the file descriptors are carefully closed at the end of the operation to prevent resource leaks.

4.3.5 Put Method

The `Put` method manages the insertion of key-value pairs into the database. Initially, these pairs are inserted into a memtable. When the memtable leads to the creation of an SST file, the `Put` method invokes `checkLSMCompaction`. This invocation ensures that the LSM tree remains optimally compacted and efficient in data retrieval.

4.3.6 Close Method

The `Close` method finalizes the database operations before shutdown. If there are entries remaining in the memtable, the method triggers their flushing into an SST file and then checks for any necessary compactions in the LSM tree. It also ensures that the current state of the LSM tree is accurately saved to a file. This allows us to read it back in when opening the database to reconstruct the lsm_tree without any loss in structure.

4.3.7 State Persistence Methods

The `saveLSMTreeState` and `loadLSMTreeState` methods are responsible for persisting and reconstituting the LSM tree's state. The `saveLSMTreeState` method writes the levels of the LSM tree and their associated SST file_names to a file. The `loadLSMTreeState` method reads this file upon database initialization to reconstruct the LSM tree. These methods are essential for ensuring that the LSM tree's state is not lost between database sessions and that the system can resume operations seamlessly.

4.3.8 Conclusion

The LSM tree implementation showcases a structured approach to database management, particularly in handling Btree SST files for efficient data storage and retrieval. The architecture of the LSM tree along with the compaction policy, insertion checks, and state persistence, highlights the robustness of the system. However, areas like memory management in `mergeSortSSTs` and the dependence on file system operations

require careful attention which could lead to performance drops in large databases. This implementation overall achieves the structure and effectiveness of LSM trees in modern database systems, balancing efficiency with complexity.

4.4 Support for Updates

To support updates into the database, we modified the PUT functionality to allow for the value in the key-value pair to be overwritten. The update functionality is identical to the GET Functionality, except it takes a new value and rewrites the value writen in the memtable with that value. We then added a new API to our database: int64_t Database::Update(const int64_t &key, const int64_t &value) which will call the Database::Put function to insert the entry into the Database. Separate APIs for UPDATE functionality clarify the intent of the operations, making the code more readable and understandable, with a small sacrifice of redundancy and overhead.

4.5 Support Delete

In our database system, deletion is managed by designating the value '0' as a tombstone marker. When a key-value pair in the database has a value of '0', it indicates that the pair has been effectively deleted. Consequently, any query for such a key will result in a '-1', signifying the entry's absence due to deletion. Deletion can be achieved through two approaches in our implementation. The primary method involves using the `Database::Delete()` function. When invoked with a specific key, this function internally calls the PUT function, assigning the value '0' to the key in question. This key-value pair, now marked as deleted, is stored in the memtable. Given that the memtable is queried before consulting any SST files, any attempt to retrieve the deleted key will promptly return '-1', reflecting its deletion status. Alternatively, deletion can also be executed directly by calling `PUT(key, 0)`. This approach mirrors the previous method, marking the key as deleted by associating it with the value '0'. Both methods ensure that deleted entries are accurately represented and handled in the database system. Separate APIs for DELETE functionality clarify the intent of the operations, making the code more readable and understandable, with a small sacrifice of redundancy and overhead.

4.6 Step 3 Experiments

In this experiment, we aim to generate an updated measure of throughput for PUT, GET, and SCAN operations. In our changes made for Step 3, the database now stores its flushed data in a Log-Structured Merge-tree (LSM Tree) where the SSTs at each level is a B-tree attached with a Bloom filter to check for a key's membership in an SST. We follow the same experimental setup as the Step 1 Experiment, except we now use an LSM-tree structure with a fixed size ratio of 2 at any level with compaction policy, buffer pool size is fixed at 10 MB, the Bloom filters use 5 bits per entry, and the memtable is fixed at 1 MB. For scans specifically, we limit the operation to scan for 4KB only at each data size. Furthermore, when inserting entries, the value of the entry is either 0 or 1 to simulate updates and deletes, as we reserve the number 0 for entry values to represnt a tombstone. Below are the plotted throughputs for the three operations:

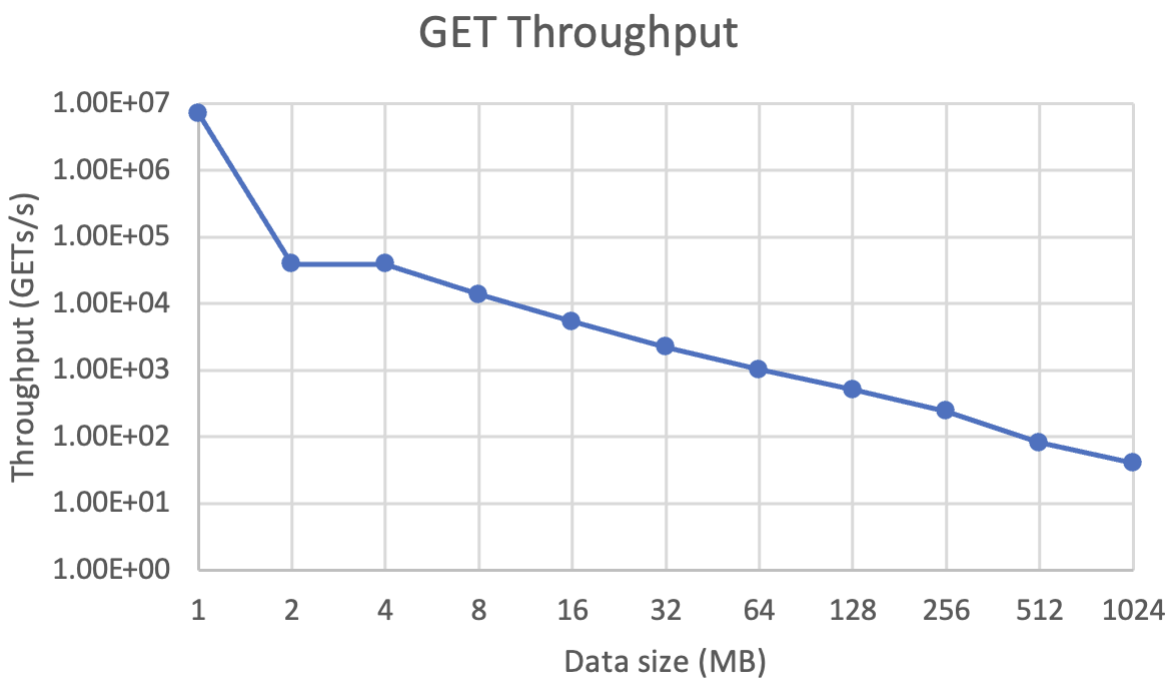

Figure 8: GET throughput of LSM-tree database as a function of database size

Figure 8 shows the LSM-tree's throughput of GET queries as the data size increases, or more precisely, as the data size doubles at each epoch. As expected, the throughput gradually decreased as the data size increased, since proportionally more queries visit the SSTs via disk I/O which is slower than CPU operations. Overall, GET throughput is faster by about two times in this experiment than in Step 1 experiment, where SSTs queries are executed with binary search. This could be attributed to the fact that we now have Bloom filters to avoid unneeded disk I/Os and also the usage of B-tree search within the SSTs. These Bloom filter checks can now speed up GET queries by potentially eliminating the need to traverse an entire SST, especially when the SST is large.

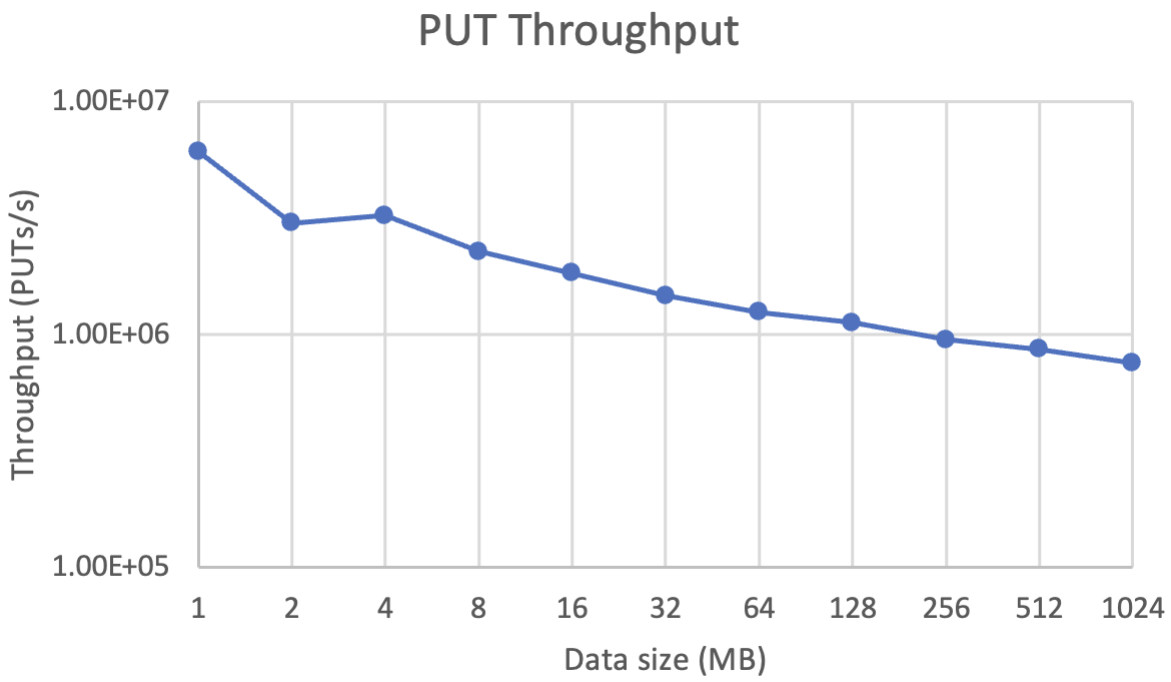

Figure 9: PUT throughput of LSM-tree database as a function of database size

Figure 9 shows the LSM-tree's throughput of entry insertions, where the throughput at each epoch is the time it took for the database to reach the current epoch from the previous epoch. In other words, how long it took the database to double its size to the current size from its previous size. The trend of the data resembles a negative logarithm graph, which resonates with the $\mathcal{O}(\frac{log_2(\frac{N}{P})}{B})$ I/O cost of LSM-tree insertions as the data size increases. As expected, throughput for insertions was much slower than Step 1 experiments, most likely due to the extra computational overheads introduced by the more complicated construction of the B-tree SSTs with metadata and Bloom filters, and also extra disk I/O incurred when the compaction policy is triggered.

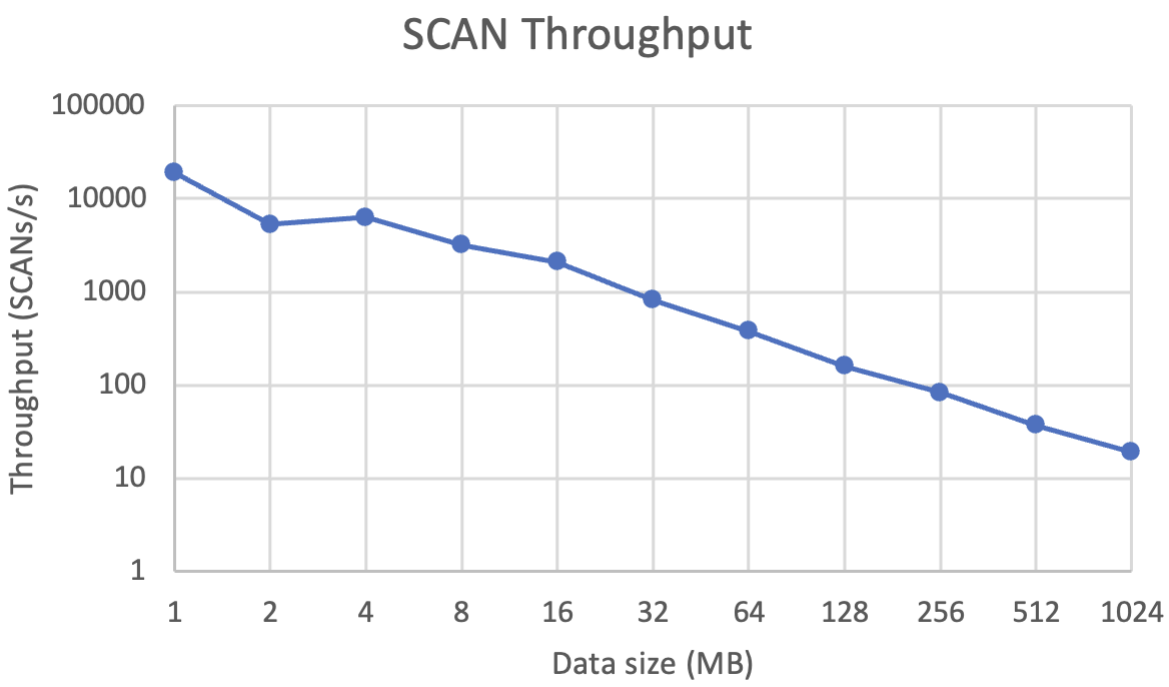

Figure 10: SCAN throughput of LSM-tree database as a function of database size

Figure 10 LSM-tree's throughput of scans. We expected to see a negative log graph since LSM-tree scan costs is $O(log_2(N/P) \cdot log_B(N) + \frac{S}{B})$ I/O, but that may be due to a systematic error in the way the experiment was executed. More scans at each epoch may have produce data that converges into more of a negative log shaped graph. Overall, throughput decreases as the data size increases, as expected, since searching for the right SST may take longer due to the increasing number of levels and also SST (hence B-tree) sizes.

5 Testing

To facilitate unit and regression testing as the database implementation grows more complex, we designed a set of tests for each database component:

5.1 Memtable tests

testPut: Tests the memtable put() functionality is working and the value can be retrieved through get().

testLeftRotation: Fills the memtable with enough values to trigger a left rotation and checks to make sure the AVL structure is still maintained.

testSSTFlush: Tests to see that the memtable is flushed and emptied when the size() hits the capacity.

testValidSST: Tests to make sure that the data flushed from the memtable is correctly represented as an SST.

testPutSameKey: Tests to make sure that the key is updated when the value is changed.

testScan1-3: Tests to make sure that scanning a sequence of numbers in the memtable yields the correct output.

testCloseDatabase: Tests to make sure that the rest of the memtable is flushed to an SST when Close() is called on the database.

5.2 Database tests

testPutAndGetMemtable: Tests to make sure that putting and getting a value in the database still works when a memtable hasn't been flushed yet.

testPutAndGetSST: Tests to make sure that putting and getting a value in the database works when flushed to a sorted SST using binary search over pages.

testGetValue1-5: Tests to make sure that binary search over the sst works. These 5 tests cover the following cases: the first page of the database, the last page of the database, the middle of the database, and everywhere else.

testGetNotFound1-3: Tests to make sure that values outside of the range of keys in the database return -1 (value doesn't exist).

testScanQuery: Tests to make sure the scan query works for values in the SST.

testScanOverPages: Tests to make sure that scanning values over two pages still returns the correct value.

testScanMultipleSSTs: Tests to make sure that correct values are still returned even when there are different values in different SSTs.

testScanMemtableAndSST: Tests to make sure that values retrieved from the memtable and SST are in the ScanResponse query.

testScanOutOfRange1-2: Tests to make sure that when the scan goes out of range of the database values, it does not include it or come out with errors.

5.3 Database update and delete tests

testGetAfterUpdateAndDelete: Test verifies the database's ability to correctly handle Get operations after Put (update) and Delete operations.

testScanAfterUpdateAndDelete: This test evaluates the database's capability to correctly execute Scan operations after a series of Put (update) and Delete operations.

5.4 Buffer pool tests

testInsertAndSearch: This test verifies that the buffer pool correctly inserts and retrieves key-value pairs, ensuring both the accuracy of stored data and the functionality of collision resolution mechanisms.

testEvict: This test checks the buffer pool's eviction policy to ensure that it correctly evicts the least recently used pages when new pages are inserted beyond its capacity.

testEvictWithMultiRead: This test evaluates the buffer pool's eviction strategy in a scenario involving multiple reads and inserts, ensuring that the eviction process works correctly even with frequent access to certain pages.

testExpand: This test assesses the buffer pool's ability to correctly resize (expand) its capacity and maintain the integrity and accessibility of key-value pairs after the expansion.

testShrink: This test checks the buffer pool's functionality when reducing its size, ensuring that it correctly handles the eviction of pages and maintains data integrity for the remaining pages. [testShrink2:] This test examines the buffer pool's shrink operation in a different sequence, where pages are inserted first and then the buffer pool is resized, testing the correctness of data retention and eviction post-resize.

5.5 Buffer pool LRU tests

testPutRemoveHead: This test verifies the functionality of the LRU (Least Recently Used) queue by inserting elements and then removing them from the head, checking if the order of removal is as expected and if the queue is empty after all elements are removed.

testMoveToTail: This test assesses the LRU queue's ability to correctly move a node to the tail of the queue, ensuring that the least recently used element is updated appropriately and removed in the correct order.

testRemoveNode: This test assesses the LRU queue's ability to correctly remove a node from the queue, ensuring that queue is updated appropriately.

5.6 Bloom filter tests

allKeysPresent: Test whether a Bloom filter is correctly constructed and if all the inserted keys return a positive query

duplicateElements: Test whether inserting duplicate keys returns a negative query

numEntriesOne: Test edge case where Bloom filter is constructed to store only one entry

testConstructor: Test constructing an arbitrary Bloom filter, writing it to file, reading the file to reconstruct the Bloom filter with an overloaded constructor and compare the two filters

5.7 B-tree SST database tests

tsetGetAndPut: test if inserting entries then querying those entries returns the appropriate keys

testGetInvalidKeys: test querying keys not in the database, reserved keys, and keys that are out of the restricted range

testGetNewAndOldKeys: test flushing all inserted data to multiple SSTs and querying them all to check if values are correct

testScan: test B-tree scan functionality

5.8 B-tree database update delete tests

testGetAfterUpdateAndDelete: This test checks the database's ability to accurately retrieve data with the Get method after performing a series of Put (update) and Delete operations, ensuring data consistency and correct response to deletions both before and after a flush operation.

testScanAfterUpdateAndDelete: This test assesses the database's capability to perform Scan operations correctly after updating and deleting entries, verifying whether the Scan returns the appropriate set of values and sizes in response to these changes.

5.9 B-tree SST database update/delete tests

tsetGetAndPut: test if inserting entries then querying those entries returns the appropriate keys

testGetInvalidKeys: test querying keys not in the database, reserved keys, and keys that are out of the restricted range

testGetNewAndOldKeys: test flushing all inserted data to multiple SSTs and querying them all to check if values are correct

testScan: test B-tree scan functionality

5.10 In-memory B-tree and B-tree flushing tests

testRootLeaf: test B-tree SST correctness when the root node contains only entries

testHeightOne: test B-tree SST correctness when the data size grows until root contains references to leaf nodes

testThreeInternalNodes: test B-tree SST correctness when data sizes grow until there is the root, three internal nodes, and the rest are leaf nodes

5.11 LSM Tree Tests

testCompactionFileCreated: Test to make sure when two SSTs are at the same level of the LSM tree, it compacts into one SST, flushes it to the next level, and then deletes the original 2 SSTs.

testGetAfterCompaction: Tests to make sure that the GET function still works with the newly compacted SST and that no data is lost.

testScanAfterCompaction: Tests to make sure that the SCAN function still works with the newly compacted SST and that no data is lost.

testGetAfterCompactionLvl3: Tests to make sure that GET still returns the correct values after two level 2 SSTs are compacted into level 3

testScanAfterCompactionLvl3: Tests to make sure that SCAN still returns the correct values after two level 2 SSTs are compacted into level 3

testSSTAfterClose: Tests to make sure that closing the database and reopening it keeps the LSM tree structure and GET and SCAN still returns the correct values.

testScanMultipleSSTs: Tests to make sure that correct values are still returned even when there are different values in different SSTs.

testScanMemtableAndSST: Tests to make sure that values retrieved from the memtable and SST are in the ScanResponse query.

testSSTAfterUpdateDelete: Tests to make sure that when values are updated and deleted in the database, flushed to the SSTS, and merged and compacted together, it keeps the more recent updated and deleted values rather than the older values.