Key-Value Store Database Design

Group Members: Aakash Vaithyanathan, Avnish Pasari

CSC443
Database System Technology

Professor: Niv Dayan

TAs: Navid Eslami, Sajad Fagfour, HongCheng Wei

Table Of Contents

I. Introduction

In this project, we implement a key-value (KV) store database system capable of supporting the following API functions: open, close, get, scan, delete and put. This report is divided into 3 parts, representative of each part of the project. Each part includes sufficient details on our implementation strategies, how we made certain design choices, and the relevant experiment results summarizing the API performances in terms of I/Os. We have also included supporting diagrams to visualize our design choices and project structure. The entire project is done in C++.

Project Status

To our knowledge, everything in our code works as intended. We have written a very comprehensive test suite, and all of our tests pass. We have implemented all parts of the project successfully along with their experiments and described how additional experiments can be run. To our knowledge, there are no bugs in codebase.

Prerequisites

When you proceed to clone our repository, we ask that you **do not** rename the folder “csc443-project” as this is used by one of our functions to dynamically retrieve the parent directory of the project to facilitate reading and writing files in our project. Changing the name of this folder would cause errors in several parts of the project so we would like to ask you to keep the same name.

For this project, we have decided to use C++17 and the GCC compiler. Before interacting with the project, we ask you to please use VSCode and Teach.CS for testing our project as we have created a specific configuration to work best with the above defined version and compiler. The version of Python used in Teach.CS is Python 3.12.3. We ask that you confirm if this version is already installed. If not, any other version would also work.

You should notice on the bottom right of your VSCode a configuration named “CSC443-Config” to be created and selected by default for this project. If not, we ask you to please create one and copy the below configurations as described in the below figure. We would also like to confirm if you’ve installed the C++, MakeFile provided by Microsoft under the extensions tab of the editor.

Note: To make this step easier for you. We decided to push the “.vscode” file generated by our system instead of adding it to .gitignore

```
{
    "configurations": [
        {
            "name": "CSC443-Config",
            "includePath": [
                "${workspaceFolder}/**"
            ],
            "defines": [],
            "compilerPath": "/bin/gcc",
            "cStandard": "c17",
            "cppStandard": "c++17",
            "intelliSenseMode": "linux-gcc-x64"
        }
    ],
    "version": 4
}
```

Figure 1: Configuration settings in VSCode for the project

Compiling The Project

After ensuring you've cloned the project on Teach.CS and are using the "CSC443-Config" configuration setting, you can proceed to navigate to the **src** folder as this contains all the files related to our kv-store implementation. Under this directory, you will find a makefile which you can run by typing **`make`** in the terminal. Doing so will compile the files and create the following executables:

1. **run_app** - Executable to run the code in the main.cpp under the src folder. This file is provided in case you would like to add your own custom code to test any functionality in our project.
2. **run_tests** - Executable to run all the unit tests written in our project
3. **part2_experiments** - Executable for the part2 experiments to be called by *experiments/part2/part2.py* via the subprocess() call.
4. **part3_experiments** - Executable for the part3 experiments to be called by *experiments/part3/part3.py* via the subprocess() call.

If you would like to clean the project after running it, please ensure you're at the **src** directory level in your terminal. Once confirmed, please call **`make clean`** followed by **`./clearDBs.sh`** to clear all databases created from the unit tests or while testing the project.

Running the Application

To run the application, or do custom operations, there is an empty **main.cpp** file under the **src/kv-store** directory. Within the **main()** function, feel free to interact with the API as you see fit.

It is important to note that you must use each part's API together. Please do not use the part 3 version of PUT and then use the GET from part 1.

After compiling, you can call the executable **./run_app** to execute your main script.

Running Unit Tests

After compiling the project, to run the unit tests, you can call the executable **./run_tests** that is generated within the **src** folder of our project.

Please note that the tests can take around 5 minutes to complete running.

Folder Structure

To navigate to our unit tests, please navigate to **/src/test**.

Within **test**, there are many folders that contain the header files and the corresponding .cpp files that contain the actual code to test our implementations. For example, to access the tests for our part 1 GET api, you would navigate to **/part1-api/get** within the **test** folder.

The driver code **main.cpp** is located in the root of the **test** directory, and is responsible for centralizing all the unit tests that we have written, as well as producing the executable that runs all the tests.

Important Note on B-Tree Tests

Since testing the creation of a B-Tree is difficult with a large fanout, we have written unit tests with smaller fanouts and page sizes to make the testing process more manageable. This allows us to more easily verify the accuracy of the functions across a variety of scenarios, ensuring that they work correctly regardless of the fanout size and page sizes. Since these tests assume different common variable values, these unit tests are commented out as to not disrupt the standard behavior of the program with the actual fanout size.

Assuming you start in the **src** directory, to run these tests, do the following:

1. Navigate to **kv-store/util/common.h**
2. Change the value of **B_TREE_FANOUT** to 4, and **KV_PAIRS_PER_PAGE** to 4
3. Navigate to **test/b-tree/bTreeTest.cpp**
4. Uncomment the **testFanout4()** and **testFlushMemtableToBPlusTree()** functions
5. Navigate to **test/main.cpp**
6. Comment out every test function call other than **bTreeTestAll()**
7. Compile and run the tests

Note that the **testFlushMemtableToBPlusTree()** function is a visualization function that we manually verified. The output that is produced can be explained by reading this section on our B-Tree implementation.

Important Note on Buffer Pool Tests

Since collisions are possible when using a buffer pool, in order to test them we needed to design tests that deliberately forced collisions to occur in order to validate the behavior of the buffer pool under such scenarios. Since these tests affect the performance of the APIs that use the buffer pool, these tests are commented out.

Assuming you start in the **src** directory, to run these tests, do the following:

1. Navigate to **kv-store/buffer-pool/buffer-pool.cpp**
2. In the **hash()** function, modify the function to always return 0
3. Navigate to **test/buffer-pool/buffer-pool.cpp**
4. Comment out all the currently uncommented out tests
5. Uncomment out the two collision tests **testBufferPoolCollisions()** and **testBufferPoolEvictionWithCollisions()**
6. Navigate to **test/main.cpp**
7. Comment out every test function call other than **bufferPoolTestAll()**
8. Compile and run the tests

Key Data Structures and Tools

Representing KV Pair Objects

We have decided to implement our KV pair as a class. The key is an unsigned int (uint64_t) that can only take positive values. The value is a signed int (int64_t) that can take both positive and negative values. The class has 2 public methods called getKey() and getValue() to retrieve the KV pair data.

For sorting the KV pair data in our unit tests, the class has overloaded the default operator() method that does the comparison between two KV pairs based on their values.

As mentioned in the constants section of our report, each key and value takes up 8 bytes resulting in a KV pair object size of 16 bytes.

Key Assumptions about KV-Pairs

Since we have represented our key as an unsigned int, a key **cannot** take the value 0 in our database design. A key value of 0 is reserved to denote an uninitialized key. This assumption will be highlighted and explained more in detail in part two under the B-tree section of our report.

Another important assumption for part one of our project is that the key's themselves are unique, that is, no two keys across any SSTs for a given database will have the same value.

Interacting With the Databases and Disk

All APIs and methods used for interacting with the disk in terms of Input/Output (I/O) operations are located in the `disk` folder within the `kv-store` directory under the `src` module. The `disk` folder is further divided into two subfolders:

1. **`binary-file` folder**: Contains implementations of methods used to interact with binary files, which are useful for efficiently storing structured data.
2. **`text-file` folder**: Contains methods for interacting with text files, suitable for simpler, line-based storage operations.

Each subfolder is structured to provide specific I/O functionalities for handling respective file types. This modular organization allows for easy maintenance and extensibility of the storage mechanism. In our implementation, binary files are being used for storing fixed-size records, whereas text files are being used for storing metadata.

The Disk API functions are designed with a modular focus, adhering to single responsibilities such as opening, closing, reading, and writing to files. This approach offers several benefits: Reusable functions, such as `openBinaryFile` and `closeBinaryFile`, can be utilized across different system components. Robust error handling within each function simplifies debugging and ensures reliable file operations. Additionally, extensibility is maintained, enabling new methods to be seamlessly added for either file type without disrupting existing functionality.

Binary File API

The **`binary-file`** folder contains all methods that handle the interaction with binary files. These methods are implemented in `binary-file.cpp` and declared in `binary-file.h`. Below, we detail the key functions:

Opening and Closing Binary Files

1. **`openBinaryFile(const char *path, const char *mode)`**
    a. **Parameters**: `path` (file path), `mode` (file open mode).
    b. **Return**: `FILE *` (file pointer to the opened binary file).
    c. **Description**: Opens a binary file for reading or writing, depending on the specified mode. If an error occurs, it exits with an error message.
2. **`closeBinaryFile(FILE *fp)`**
    a. **Parameters**: `fp` (file pointer).
    b. **Description**: Closes the binary file. If an error occurs during the closing process, it exits with an error message.

Writing to Binary Files

1. **`writeSingleKVPairToBinaryFile(FILE *fp, KVPair &kvPair)`**
    a. **Parameters**: `fp` (file pointer), `kvPair` (key-value pair to write).
    b. **Description**: Writes a single `KVPair` to an open binary file. The file must be open before calling this function, and it does not close the file afterward.
2. **`writeKVPairPageToBinaryFile(FILE *fp, KVPair *arr, int size)`**
    a. **Parameters**: `fp` (file pointer), `arr` (array of `KVPair` objects), `size` (number of elements to write).
    b. **Description**: Writes a page of `KVPair` objects to an open binary file.
3. **`writeCharArrayToBinaryFile(string sstName, char *arr, int size)`**
    a. **Parameters**: `sstName` (string to generate file name), `arr` (character array), `size` (number of characters to write).
    b. **Description**: Writes a character array to a new binary file, generating the file name dynamically based on `sstName`.

Reading from Binary Files

1. **`readPageFromBinaryFile(FILE *fp, int page, KVPair *arr)`**
    a. **Parameters**: `fp` (file pointer), `page` (0-indexed page number), `arr` (array to store the `KVPair` objects).
    b. **Return**: Number of elements read.
    c. **Description**: Reads a page of key-value pairs from the specified page of a binary file. If the page is out of bounds, an empty vector is returned.
2. **`readCharPageFromBinaryFile(FILE *fp, int page, char *arr)`**
    a. **Parameters**: `fp` (file pointer), `page` (0-indexed page number), `arr` (character array to store data).

b. **Return**: Number of characters read.
c. **Description**: Reads a page of characters from the binary file and stores them in `arr`. If the page is out of bounds, it returns 0.

Debugging and Testing

1. **`writeKVPairVectorToBinaryFile(const char *path, vector<KVPair> &kvPairs)`**
    a. **Parameters**: `path` (file path), `kvPairs` (vector of key-value pairs).
    b. **Description**: Writes a vector of `KVPair` objects to a new binary file. This function is used only for debugging and testing purposes and should not be part of the final implementation.

Text File API

The **`text-file`** folder contains all methods used to handle text files. These methods are implemented in `text-file.cpp` and declared in `text-file.h`. Below, we provide an overview of these functions:

Opening and Closing Text Files

- **`openTextFile(const char *path, const char *mode)`**
    - **Parameters**: `path` (file path), `mode` (file open mode).
    - **Return**: `FILE *` (file pointer to the opened text file).
    - **Description**: Opens a text file for reading or writing, depending on the specified mode. If an error occurs, it exits with an error message.
- **`closeTextFile(FILE *fp)`**
    - **Parameters**: `fp` (file pointer).
    - **Description**: Closes the text file. If an error occurs during the closing process, it exits with an error message.

Reading and Writing to Text Files

- **`readTextFile(const char *path, vector<string> &lines)`**
    - **Parameters**: `path` (file path), `lines` (vector to store lines read from the file).
    - **Description**: Reads all lines from a text file and stores them in the provided vector. Each line is stripped of its newline character before being added.
- **`appendNameToTextFile(const char *path, const char *name)`**
    - **Parameters**: `path` (file path), `name` (name to append).
    - **Description**: Appends a new name to the text file on a separate line. If an error occurs, it exits with an error message.
- **`isNameInTextFile(const char *path, const char *name)`**
    - **Parameters**: `path` (file path), `name` (name to search for).
    - **Return**: Boolean indicating whether the name is in the text file.

- **Description**: Checks if a specified name exists in the text file and returns `true` if found.

II. Part 1: Creating a Memtable & SSTs

All code related to part 1 can be found under **`src/kv-store/part1-api`**

Keeping track of created databases and SSTs

All databases created by the user using the **open()** API command are created under the **databases** folder with the input name. When any database is created, we also create a text file under that database folder with the name **file_names.txt**. This text file keeps track of all existing SSTs for this specific database.

The **database** folder also has a text file called **db_names.txt.** This text file keeps track of all the databases that were created by the user.

Design choice - Why we have db_names.txt and file_names.txt

The need for a db_names.txt file stems from the need to check for existing databases with the given name before creating a new one from open() API command. Issuing multiple I/Os to the database and reading each database folder name would be an expensive operation. To tackle this problem, we have decided to keep track of the created database names in a db_names.txt file. When we call the open() API command with the given database name, we first read the db_names.txt file to check if any database with the input name already exists. This results in us making 1 I/O call to check for an existing database with the name.

A similar argument can be made for why each database has a file_names.txt. This is because the get() API command requires us to search the SSTs in order from youngest to oldest. By keeping track of a file_names.txt file which keeps track of the names of all SSTs for the current database, we know that the SST name at the very end of the file is the youngest SST that needs to be searched first. We can issue 1 I/O to store in memory the name of all the SSTs for a given database and read the SSTs in reverse order. Since a given database will only contain a finite number of SSTs, we can be sure that storing the names of all the SSTs for the current database would not require a lot of memory.

Important Constants Defined For Part I

For part one of our project, we have the following constants defined:

- PAGES_PER_SST = 256
- PAGE SIZE = 4KB
- KV_PAIR_SIZE = 16 bytes[1]

[1] 8 bytes for keys, 8 bytes for values

- KV_PAIRS_PER_PAGE = 256[2]
- MEMTABLE_SIZE = PAGES_PER_SST * KV_PAIRS_PER_PAGE = 4096 KV pairs

Memtable: Represented as an AVL Tree

All code related to avl tree can be found under **`src/kv-store/avl`**

In this project, we have decided to denote our memtable as an AVL tree. The balanced nature of the AVL tree guarantees that the height difference between left and right subtrees is always kept to a minimum, guaranteeing efficient $O(nlogn)$ insertions and lookups, making it an ideal choice for implementing a memtable to buffer insertions in our KV-store.

Each node in the tree is denoted by a class called **AVLNode** that takes in a KVPair object and initializes the left and right pointer of the node to null in its constructor.

The memtable itself is represented as an **AVLTree**, which uses instances of the AVLNode to maintain the structure of the tree**.** Some of the key methods from this class are:

- **insert(KVPair kv)** - insert a KV pair object into the AVL tree. If a duplicate key is inserted, the existing value is overridden with the new value.
- **get(uint64_t key)** - returns a pointer to the KV pair object with a given key in the memtable. If no such key is found, return nullptr.
- **scan(uint64_t start, uint64_t end)** - performs a range query of objects in the memtable between start and end keys, inclusive.
- **clearTree()** - performs an inorder traversal of the memtable and clears all initialized nodes emptying the memtable without deleting the pointer to the memtable object.

The AVLTree class also contains helper functions to help us perform these externally facing functions, most notably:

- **getBalanceFactor(AVLNode *node)** - calculates the difference in height between the left and right subtrees of a node. A balance factor greater than 1 or less than -1 indicates that the tree needs rebalancing.
- **rotateLeft(AVLNode *node), rotateRight(AVLNode *node)** - performs a left or right rotation on a given node to correct an imbalance

In addition to this, the class also has the following methods that are mainly used for debugging purposes. Note that these methods return vectors, but it has no impact on performance because these functions are only used for debugging purposes.

- **inorderTraversal()** - performs an inorder traversal of the memtable.
- **preorderTraversal()** - performs a preorder traversal of the memtable.
- **postorderTraversal()** - performs a postorder traversal of the memtable.

[2] 256 x 16B KV-Pairs = 4KB

Part 1 Design Choices

Use of Global Variables

For part one of our project, we have decided to use three key global variables that are defined in the **global.h** file found under the **utils** folder. These variables are as follows:

- AVLTree* memtable - pointer to the memtable that is used
- vector<string> sstNames - vector of SST names for the currently open database.
- string currentOpenDB - Name of the current open database.

By defining the global variables, we can reference these using **extern** in the relevant header files and avoid having to pass the memtable, database names, or sstNames to all the functions that need it. The reason this choice of defining constants works is because at a time, we know that only one database will be open and any changes made to these variables will be for the active database the user is interacting with.

Flushing Memtable to Disk

There are two scenarios where we empty the contents of our memtable. The first is when we reach the memtable size and must flush its contents to a Sorted String Table (SST). The other is when we call the close API method and must flush the existing data in the memtable into an SST and close the currently open database.

In both these cases, a potential solution would be to empty and clear the memtable, deleting the root node and reinitializing a fresh memtable to start with. However, we have decided to not delete the pointer to the memtable but instead only ensure we clear all children nodes of the root thus having a fresh memtable to begin with for both the above-listed cases. This design also ensures that we don't accidentally create any dangling pointers due to repeated deletes and reassignment of the newly created memtable.

KV Store API Implementation

Introduction

For each part of this project, we have implemented a separate '**db'** folder which has its own open() and close() API functions. A quick overview as to why this is necessary is due to the fact that for part 1, SSTs only have the KV pair data. However, for part 2, we now structure this SST as a B-tree which also includes internal nodes in this SST. Consequently, the naming convention of the SSTs created in part 2 changes compared to part 1. The same is true for part 3 as now we will have an LSM tree with a different naming convention and global variable that would keep track of the relevant metadata for us.

For these purposes, open() and close() API functions need to be unique to each part of the project since they would handle different metadata and flush different types of SSTs.

We will provide a brief overview once again in parts 2 and 3 of this project with more relevant details.

Open API

The Open() API command creates a database directory with the given name and prepares it for use. Before we create a database with the given name, we first check if a database with that name already exists in our project or not. To do so, we issue a read to the **db_names.txt** file and check if the input database name exists in it or not. If a database with that name exists in our database, we load the names of all the SSTs currently part of this database into the relevant global variable. The function returns an integer value described below:

- Return value of 1 - Database directory with the name successfully created
- Return value of 2 - Database directory with the name already exists and we load in memory the existing SST names of this database in a vector.
- Return value of 3 - Error creating database with the given name.

For a successfully created database name, the API appends the name of the created database into the db_names.txt file.

The function signature is as follows: **int openDB(const char *name);**
The file path for this is: **`src/kv-store/part1-api/db`**

Close API

The Close() API command closes the current open database and flushes the contents of the memtable into an SST. It is possible that the memtable may not be completely full when the close command is called. This function also empties the memtable and resets the state of the other relevant global variables like sstNames and currentOpenDB.

The function signature is as follows: **void closeDB();**
The file path for this is: **`src/kv-store/part1-api/db`**

Get API

Searching for a key in memtable

The Get() API command retrieves the value of a given key from the database if such a key exists. The function first searches for the key in the memtable utilizing the **memtable->get()** function. If such a key exists, we return the value associated with this key.

Searching for a key in SST from youngest to oldest

If the key does not exist in the memtable, the function proceeds to search the SSTs for this database in the youngest to oldest order. This is where we make use of our previously mentioned design choice of **file_names.txt**. Each database will have a file_names.txt that lists the names of the SSTs for this database. We proceed to read the SSTs in reverse order of these names and perform a **binary search** on each SST. To determine the number of pages in an SST, we implemented a function called getFileSize() to return the total size in bytes for the file. By dividing the file size by size of a page in our database, we can determine the number of pages for the SST we are currently performing binary search on to find the probable page. We start reading the SST from the middle page using the **readPageFromBinaryFile**() function mentioned in the Disk API section of our report. This function reads 4KB worth of KV pair data. Since we know that a page can have at most 256 KV pairs in it as

mentioned in the constants section of our report, we store these KV pairs read from the binary file temporarily into a static array of size 256.

Determining if the current SST is worth reading over or not

Upon reading the middle page of the SST, we then check the min-max range of the page. If the input key lies in the min-max range of the page KV pair data, we know this is a **probable** page where we may find our input key. In this case, we proceed to do a binary search on this read page to find the key's value. If the key exists, we return its value using the KV.getValue() method.

If the key does not exist in the min-max range of the read page from the SST, we update the middle page number based on the binary search algorithm. We continue to do this until we satisfy the binary search condition (leftPageNumber <= rightPageNumber). Once we do not satisfy this condition, we know that the key does not exist in this SST and we proceed to read the next youngest SST and repeat the algorithm mentioned above. If after reading all the SSTs from this database we were unable to find the key, we return **INT_MIN** to the user. This constant denotes the case that the key was not found.

The function signature is as follows: **int64_t get(uint64_t key)**;
The file path for this is: **`src/kv-store/part1-api/get`**

Put API

The Put() API command writes the input KV pair object to the memtable for the current database. The function calls the **memtable->insert()** to write this KV pair into our AVL memtable. Before we insert the data, we check if our memtable is at capacity.

Flushing the SST to disk

If our memtable is at capacity, we first flush the contents into a new SST and write the contents to a binary file using the function **flush_to_disk()**. Since the SSTs are numbered as SST_1, SST_2, etc, we know that the name of the new SST to be created is going to be the adding a prefix string of "SST_" to the current size of the sstNames global variable. Once we have the new SST name, we open the binary file and call **writeMemtableToBinaryFile()** which performs an inorder traversal of the memtable.

Optimizing our write I/O operations by buffering

In the above-mentioned writeMemtableToBinaryFile(), we keep track of an output buffer that has a maximum size of 4KB. Instead of writing the KV pair data every iteration from the in-order traversal, we add these to the output buffer and wait until our buffer fills upto a page worth of data. Once, at capacity, we call **writeKVPairPageToBinaryFile()** which writes the data from the output buffer at once. Internally, the function calls fwrite() which issues a single I/O to write the data as opposed to individually writing every KV pair. This ensures that any write operation we do to a file happens at the granularity of a **page**. This optimization helps reduce our total write I/O cost when flushing the memtable to disk from **O(N) → O(N/B).**

If the memtable was not at capacity, we simply proceed to insert the input data into it by calling **memtable->insert()**.

The function signature is as follows: **void put(uint64_t key, int64_t value);**
The file path for this is: **`src/kv-store/part1-api/put`**

Scan API

Performing a scan over the memtable

The Scan() API command retrieves all the values between the input start and end key from the database, inclusive. The function first searches for the range of KV pairs between the scan range in the memtable utilizing the **memtable->scan()** function. Since for the scan API, we do not know how many values will fall in the range, we utilize a vector<KVPair> to keep track of the complete scan result. It is possible that the end key may not exist in the scan result.

Performing a scan over the SST from youngest to oldest

After performing a scan of the results from the memtable, we proceed to read all the SSTs in the oldest to youngest order. It is important to note that unlike the get API that needs to search the SSTs from youngest to oldest, for the scan API, we need to search all the SSTs hence we read all the SSTs in the order they were created. By reading the names of the SSTs using the file_names.txt, we proceed to perform a **binary search** on each SST. We start reading the SST from the middle page using the **readPageFromBinaryFile**() function mentioned in the Disk API section of our report. This function reads 4KB worth of KV pair data. Since we know that a page can have at most 256 KV pairs in it as mentioned in the constants section of our report, we store these KV pairs read from the binary file temporarily into a static array of size 256.

Determining if the current SST is worth reading over or not

Upon reading the middle page of the SST, we then check the min-max range of the page. If the values between the scan range lies in the min-max range of the page KV pair data, we know this is a **probable** page where we may find our start key. In this case, we proceed to read the KV pairs in order for this page of the SST and continue to add the KV pairs that lie in our scan range into the **results** variable. Since we know that an SST is sorted, all subsequent pages for this SST will have key values greater than the start key of the scan API. We will continue to read the remaining pages of the SST in order and keep adding valid entries that lie within the scan range, that is, the key values are less than or equal to the end key range. Once we have read all the pages of this SST, we proceed to read the next SST, performing a binary search from the middle page, and repeating the same process as listed above.

If the key does not exist in the min-max range of the read page from the SST, we update the middle page number based on the binary search algorithm. We continue to do this until we satisfy the binary search condition (leftPageNumber <= rightPageNumber). Once we do not satisfy this condition, we proceed to read the next SST and repeat the algorithm mentioned above. After we have read all the SSTs, we finally return the **results** variable with the scan API results. The results of the scan function are sorted before returning to the user. As mentioned in the KV pair class definition section of our report, we have overridden the default comparison operator method that compares the keys in our results variable. By calling the standard library's sort() method, we can automatically sort our vector<KVPair> objects using our defined operator().

Finally, we sort the results of our scan and return it to the user.

The function signature is as follows: **vector<KVPair> scan(uint64_t start, uint64_t end)**;
The file path for this is: **`src/kv-store/part1-api/scan`**

Unit Tests & Integration Tests

As we develop our solution for part one of the project, we have extensively tested our code base writing unit tests for each aspect of our project. In total, we have 33 unit tests written for part one of our project. A breakdown of these is as follows:

- Memtable (AVL Tree) [`/test/avl`] - 14 tests
- Disk API and Open API [`/test/disk`] - 6 tests
- Get API [`/test/part1-api/get`] - 4 tests
- Put API [`/test/part1-api/put`] - 2 tests
- Scan API [`/test/part1-api/scan`] - 5 tests
- Close API [`/test/part1-api/close`] - 2 tests

In addition to the unit tests, we have also written an end-to-end test utilizing various API methods simulating a realistic database interaction by a user.

III. Part 2: Extendible Buffer Pools & Static B-Tree

Background & Key Assumptions

Important Constants for Part 2

- BUFFER_POOL_SIZE = 2560[3]
- B_TREE_INTERNAL_NODE_SIZE = (KV_PAIRS_PER_PAGE - 1) = 255
- B_TREE_FANOUT = 256

[3] Buffer Pool is fixed to be 10MB. 10 MB = 2560 x 4KB Pages.

Note that the B-Tree constants will be explained in more detail in Step 1: Creating B-Tree Internal Nodes.

Naming of Database Pages

Unlike the sequential page accesses of each SST in part 1, in part 2 it is necessary to effectively manage random accesses to pages across many different SSTs. As such, it is essential to establish a naming convention to uniquely identify each database page. In our approach, we aimed to find a balance in keeping the identifier lightweight while ensuring that each page remains distinct and easily accessible.

The page IDs in our databases are structured as follows through a concatenation of different attributes of the page:

1. SST Number
2. Number of Internal Nodes in the SST
3. Page Number within the SST

For example, if I was looking for page 10 on SST4, and SST 4 had 7 internal nodes, the pageId would be defined as the following: **SST4-7-PG10.**

Buffer Pool

All code for buffer pool can be found under: **`src/kv-store/buffer-pool`**
In our KV Store, the buffer pool is designed to efficiently cache frequently accessed database pages in memory, minimizing costly storage I/Os. In our implementation, we use a hash table combined with the LRU (Least Recently Used) eviction policy.

Hashing

For our buffer pool, we have decided to use the MurmurHash3 function, created by Austin Appleby. Since the MurmurHash3 function produces a 128-bit value, and our hash table can only support log_2(BUFFER_POOL_SIZE) bits, we modulo the produced hash value with our buffer pool size to get a hash value that fits our buffer pool. This is appropriate because we can map the well-distributed set of hash values from MurmurHash3 to the limited size of our buffer pool, thus preserving hash uniformity.

As mentioned before, hash collisions are managed using chaining, which ensures efficient management of hash collisions without degrading lookup performance.

By using a well-defined hash function for page lookup as well as chaining for resolving hash collisions, we can maintain O(1) expected performance for lookups and insertions into our buffer pool.

Hash Table Implementation - The Frame Class

We represent a unit of storage within our buffer pool with the **Frame** class. Every frame will store the **pageId** of the page that currently occupies the frame, as well as a pointer to the next **Frame** object in a particular bucket via a linked list, because we resolve collisions through chaining.

Observe that a **Frame** can store either **KVPairs** (if we are bringing either a B-Tree internal node, or SST data into the buffer pool), or it can store **bits** (if we are bringing a page of a particular bloom filter into the buffer pool).

As such, a Frame will store a single page of data. This page of data can consist of either **KVPairs**, or it can consist of **char** values to represent the bits of a Bloom Filter. Alongside this, it will also store the number of values in the page, in the respective variables **numKVPairs** or **numBitsInBloomFilter**.

Note that a Frame cannot store both **KVPair** and **char** values. We have different constructors and retrieval methods specific to whether the Frame object is storing **KVPair** values or **char** values. Because the Page IDs will contain whether the page is or is not a Bloom Filter page (this will be covered in IV. Part 3: LSM Tree With Bloom Filters of the report - basically, if a pageID starts with 'BF' we know it is a bloom-filter page), we can ensure that no undefined behavior occurs and ensure that the two functionalities remain independent within the **Frame** class.

Hash Table Implementation - The Buffer Pool Class

Our hash table is represented by the **BufferPool** class, which manages a collection of Frame objects. We have fixed our buffer pool size to be 10MB, defined by a constant BUFFER_POOL_SIZE.

In addition to this fixed-size **hashTable**, the BufferPool class also manages attributes that track its current **size**, as well as a deque named **lruQueue** representing the state of the LRU eviction, which will be discussed in a subsequent section.

Some of the key methods from the BufferPool class are the following:

- **hash(string pageId)** - computes a hash value for the given page ID.
- **evictPage()** - evicts the LRU page by removing it from the buffer pool.
- **updateLRU(string pageId)** - moves the page with the given pageId to the most recently used position of the LRU deque.
- **put(string pageId, KVPair *page, int numKVPairs)** - inserts the given page into the buffer pool. Evicts a page if the buffer pool is full.
- **get(string pageId, KVPair *outputPage, int *numKVPairs)** - retrieves the page with the given pageId from the buffer pool. Returns 1 if the page is found, and returns 0 if the page is not found. If the page is found, copy the page into **outputPage** and the number of KV Pairs into **numKVPairs.**
- **putBfPage(string pageId, char *bitmap, int numBitsInBloomFilter)** - inserts the given page of bits from the Bloom Filter into the buffer pool. Evicts a page if the buffer pool is full.
- **getBfPage(string pageId, char* bitmap, int *numBitsInBloomFilter)** - the number of bits in this particular page of the Bloom Filter

put() Implementation in the Buffer Pool

First, recall that Frame objects can take in either **KVPair** or **char** values based on whether or not we are bringing a Bloom Filter page into the buffer pool.

As such, the implementation of **put()** is the same for both **put()** and **putBfPage()**, which are used to put a page of **KVPair** values and **char** values into the buffer pool, respectively.

The pseudocode for the **put()** function is as follows. We will denote the data actually stored within a page as **pageData.**

```
put(pageId, pageData, sizeOfPageData):
	hash = hash(pageId)

	create a new Frame, calling the appropriate constructor based on if the data being
	inserted is a bloom filter or not

	if the buffer pool is full, evict the LRU page

	add the page to the buffer pool based on its hash value, resolving collisions with
	chaining if needed

	update the size of the buffer pool, and update the LRU queue
```

get() Implementation in the Buffer Pool

Again, recall that Frame objects can take in either **KVPair** or **char** values based on whether or not we are bringing a Bloom Filter page into the buffer pool.

As such, the implementation of **get()** is the same for both **get()** and **getBfPage()**, which are used to retrieve a page of **KVPair** values and **char** values from the buffer pool, respectively.

The pseudocode for the **get()** function is as follows. We will denote the output buffer for the data as **outputBuffer**.

```
get(pageId, outputBuffer, outputSizeOfBuffer):
	hash = hash(pageId)

	retrieve the frame currently in the buffer pool with the corresponding hash value

	traverse through the linked list (due to potential chained collisions) until we find
	the target page

	if the page is found:
		update the LRU Queue (as we have just accessed the page)
		copy the frame’s page data into outputBuffer
```

```
        copy the frame's page data size into outputSizeOfBuffer
        return 1 (success)

    else the page was not found:
        return 0 (failure)
```

Eviction Policy - Building the LRU Deque

We decided to use the LRU eviction policy for this project due to its high eviction effectiveness. To implement this eviction policy, we developed a double-ended linked list that serves as a queue to track the usage of the pages cached in the buffer pool. Using a double-ended queue allows us to efficiently manage insertions, removals, and updates to both ends of the queue, and thus keep track of the eviction order.

In our implementation of **DoubleEndedQueue,** it is built using a doubly linked list of **DoubleEndedQueueNode** objects.

The **DoubleEndedQueueNodes** contain the following attributes (along with their getters and setters):

- **string pageId** - representing the pageId that is in the queue. Note that we only need to track this value because we can determine its location in the buffer pool by hashing it, and then evicting it after locating it.
- **DoubleEndedQueueNode *prev** - a pointer to the previous node in the linked list, characteristic of a doubly linked list
- **DoubleEndedQueueNode *next** - a pointer to the next node in the linked list, characteristic of a doubly linked list

To implement the actual **DoubleEndedQueue**, we need to ensure that we can access the queue from both the head and the tail. As such, our **DoubleEndedQueue** class contains the following attributes:

- **DoubleEndedQueueNode *head** - a pointer to the node at the front of the queue. In our implementation, this represents the least recently used page.
- **DoubleEndedQueueNode *tail** - a pointer to the node at the end of the queue. In our implementation, this represents the most recently used page.
- **int size** - keeps track of the size of the queue, so we know when the queue is filled up and an eviction is necessary

Some of the key methods from the **DoubleEndedQueue** class are the following:

- **front()** and **back()** - returns the pageId at the front and back of the LRUQueue
- **pop_front()** - removes the head of the queue from the LRUQueue, evicting the least recently used page
- **push_back(string pageId)** - adds a new page to the end of the queue, marking it as the most recently used
- **erase(string pageId)** - removes the page corresponding to pageId from the queue, allowing us to update the page's position when it is accessed again
- **move_to_back(string pageId)** - moves a page to the end of the queue, marking it as most recently used. Used when a page is accessed that is already in the buffer pool.

DoubleEndedQueue Implementation Details

As mentioned previously, the implementation of the **DoubleEndedQueue**, but we will quickly discuss two functions that we have written for our specific use case.

The **erase(string pageId)** function is used to locate any page within the **DoubleEndedQueue** and remove the page.

The **move_to_back(string pageId)** function calls **erase()** and then **push_to_back()** on a specific page, effectively moving it to the most recently used position of the queue. This function is used when we need to update the LRU Queue within the buffer pool when we access a page that's already in the buffer pool.

Structuring SSTs as B-Trees

When modifying our SST structure to be stored as B-Trees, we must modify the way in which we flush the memtable to disk (as a new SST) when the memtable gets filled up.

In this section, we will focus solely on the process of converting the in-memory data from the memtable into a static B-tree structure, and storing it into disk as a new SST file. The specifics about how these B-tree structured files are subsequently used within our system will be covered in the sections detailing the API implementation for part 2.

Implementation Details

In our implementation, we have narrowed this transformation into 2 main steps:

1. Creation of the internal nodes of the B-tree
2. Linking of internal nodes to their child nodes in the B-tree

This transformation is represented by the **BTreeUtil** utility class. This class contains 2 key static methods:

- **deque<vector<BTreeNode *> > createBPlusTree(uint64_t *sortedMaxKeys, int numPages)**
- **KVPair *flushMemtableToBTree(uint64_t sortedMaxKeys, int numPages, int& numInternalNodes)**

The function **createBPlusTree()** is responsible for creating the B-Tree. The implementation of this function will be covered within *Step 1: Creating B-Tree Internal Nodes*.

The function **flushMemtableToBTree()** is an all-in-one function that the API uses to generate a list of key-pointer pairs for all the internal nodes, that it can then use to subsequently store into disk as a new SST file. This function will:

1. Call createBPlusTree() to create the B-Plus tree
2. Set numInternalNodes to the number of internal nodes
3. Handle the logic surrounding linking internal nodes to child nodes, of which is covered in *Step 2: Linking Internal Nodes to Child Nodes.*

The implementation of this function is covered within Implementation: flushMemtableToBPlusTree().

Representation of B-Trees in Files

Important Remark: Pointers

Note: What will henceforth be known as a “pointer” is actually represented by the page offset of the child node. For example, if an internal node on page 5 points to another internal node on page 10 of an SST, the “pointer” value would be 10. We let these pointers be represented as an 8-byte integer, making key-pointer pairs equivalent to a KV-Pair in size.

We will denote these key pointer pairs with angle brackets, i.e. <key, pointer>.

Padding Pages

Recall that a key cannot take the value 0, as it is reserved for the system.

Observe that internal nodes in the B-tree do not necessarily have to be completely filled (i.e. 4KB page aligned). However, we need to ensure that each node is 4KB page aligned so that our I/O cost isn’t unnecessarily increased to read a given page/node.

As a solution, we use the reserved key 0, and we pad the rest of the internal nodes with <0, 0> to ensure that our pages are 4KB page aligned. Observe that this is actually an optimization that we came up with, as this ensures that there is no space wastage using this method.

For example, 4KB pages can store 256 key-pointer pairs. If there are only 56 key-pointers, for instance in the root node, we pad the rest of the 4KB page with 200 <0,0> key-pointer pairs to ensure that our root node is still 4KB aligned.

Important Remark: The n-th Child Node

Recall that when building the internal node with the maximum value of the n-1 child nodes, the n-th node is accessed when the desired key is greater than the maximum value in the internal node. This intuition is synonymous with the teachings from lectures.

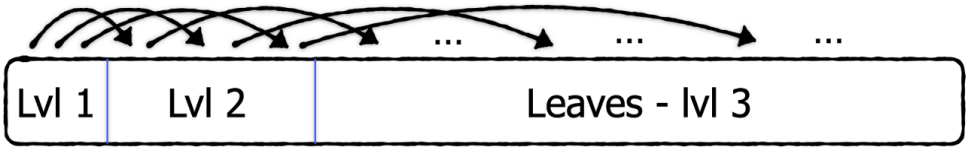
However, since the internal node contains no key that points to the n-th child, we use the key 0 as a key-pointer pair to the n-th child.

Thus, any internal node will be a 4KB page structured as the following:

- <key1, pointer1>
- …
- <0, last child pointer>
- <0,0> until page is 4KB aligned

Structure

Below is a visualization of the structure of our B-Tree, taken directly from the assignment handout.



SST static B-tree structure

The internal nodes are structured as padded pages of key-pointer pairs. The leaf nodes are structured the same way as an append only file, and are structured as non-padded contiguously allocated 4kb pages.

Step 1: Creating B-Tree Internal Nodes

In this section, we will cover how we construct the internal nodes of the B-tree given the sorted max keys from the memtable.

For sanity sake, we have created a class called **BTreeNode**, which contains a singular attribute that is a vector of keys that are a part of the node.

Background & Setup

Assumptions:

- An inorder traversal of the memtable has already been completed. Consequently, we have sorted data in the form of contiguously allocated 4kb pages with no padding that will make up the leaf nodes of our B-tree.

Constants:

- `B_TREE_INTERNAL_NODE_SIZE = KV_PAIRS_PER_PAGE - 1 = 255`[4]
- `B_TREE_FANOUT = B_TREE_INTERNAL_NODE_SIZE + 1 = 256`

Parameters:

- **uint64_t *sortedMaxKeys** - A sorted array of the maximum key in each page (leaf node). These keys will make up the base layer of internal nodes.
- **uint64_t numPages** - The number of pages (leaf nodes)

Returns:

- **deque<vector<BTreeNode *> >** - An ordered queue of vectors representing the internal nodes on each level of the constructed B-Tree. This queue does not contain the leaf nodes. Each vector in the deque represents a level in the B-Tree from top to bottom. For example, the root node would be the first vector, and the level above the leaf nodes would be the last vector.

Design Choice: Using a Deque of Vectors

Because we construct the internal nodes from the base level all the way up to the root node, using a deque provides efficient O(1) access to insert higher levels at the front of the deque, providing a

[4] Key/Pointer Pairs are 16 bytes. Internal nodes of size 4KB can store 255 of these plus one key-pointer pair for the case that the child node's keys are all larger than the maximum key in the internal node.

natural representation of the bottom-up nature of the algorithm. This helps to make the code more understandable. Additionally, using a deque and vectors allows for dynamic resizing, which is essential for handling varying levels of the tree as it grows.

Note that the use of these built-in C++ data structures also does not compromise our experience in low-level programming and database development, as they are tools that just allow us to focus on implementing the core logic and algorithm design with regards to constructing the B-Tree.
The tradeoff with using a deque is the memory overhead due to having to manage multiple memory blocks, but we decided that the tradeoff is worthwhile due to the ease of insertion and code clarity.

Algorithm: createBPlusTree()

Below is the pseudocode for the algorithm of transforming the sorted array of maximum keys into the internal nodes of the B-Tree. Our algorithm constructs the B-Tree from bottom-up, continuing iteratively until there is only 1 node in a level, indicating that we have reached the root node.

```
createBPlusTree(sortedMaxKeys):
	partition sortedMaxKeys into groups of size B_TREE_FANOUT

	if last group has size B_TREE_FANOUT:
		split the group into 2 groups of equal size
	else if last group has size <= ceil(B_TREE_FANOUT / 2):
		combine the last 2 groups and split the combined array into 2 equal-sized
		groups
	NOTE: at the conclusion of this step, every partition will be more than half full

	baseLevel = vector of BTreeNodes for the base level
	pop the last key from the last partition (otherwise it can bubble up to root)

	internalNodesDeque = []
	internalNodesDeque.push_front(baseLevel)
	while most recently inserted level isn't the root node (i.e. level.size() == 1):
		pop the max keys from each node in the most recently inserted level (except
			for the very last node)
		partition those max keys into groups of size B_TREE_FANOUT
		split arrays if needed to ensure every partition at least half full (same
			logic as in the first pass)
		level = vector of BTreeNodes for this level
		do not pop last key from last node this time
		internalNodesDeque.push_front(level)

	return internalNodesDeque
```

Upon execution of this function, we will have an ordered deque of the levels of internal nodes for the B-tree representation of the memtable to be flushed.

Note that the steps within the while loop are extracted into a helper function called **createBPlusTreeInternalNodesHelper.**

Example: createBPlusTree()

Here is a slightly more concrete example to help visualize what is going on in the **createBPlusTree()** function. In this example, we will work with a **fanout = 4.**

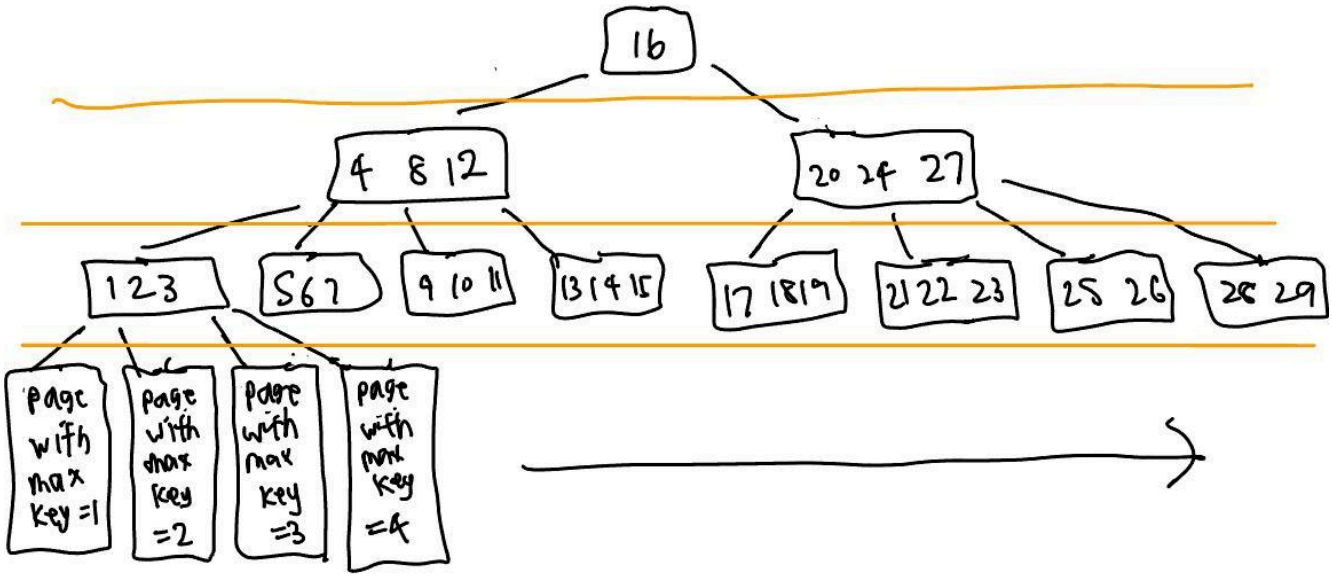
Parameters:

- **uint64_t *sortedMaxKeys =** [1, 2, 3, 4, …, 30] , **int numPages =** 30
    - We assume that there are 30 pages in the SST, and that within each SST, the max value in each of those pages is [1, 2, …, 30]

Steps:

1. We first start by partitioning sortedMaxKeys into partitions of size 4
   ```
   Partitions = [1, 2, 3, 4] [5, 6, 7, 8] … [25, 26, 27, 28] [29, 30]
   ```
2. Since the last partition [29, 30] is not more than half full, we combine and split the last 2 partitions
   ```
   Partitions = [1, 2, 3, 4] … [25, 26, 27] [28, 29, 30]
   ```
3. We pop the last key from the last partition, since this can bubble up to the root
   ```
   Partitions = [1, 2, 3, 4] … [25, 26, 27] [28, 29]
   ```
4. We turn these vectors into **BTreeNodes**, and create a vector to hold them. This is our bottom level of internal nodes now.
   ```
   baseLevel = [BTreeNode(1, 2, 3, 4), BTreeNode(5, 6, 7, 8), … BTreeNode(28, 29)]
   ```
5. We insert this level into our deque of all the nodes
   ```
   internalNodesDeque = [baseLevel]
   ```
6. Since baseLevel.size() > 1, we need to continue building up. This is the beginning of the while loop, and starting from step 7, we step into our helper function.
7. We pop the max keys from each node in the most recent level (`baseLevel`), except for the very last node, and then partition them into groups of size 4
   ```
   maxKeys = [4, 8, 12, 16] [20, 24, 27]
   baseLevel = [BTreeNode(1, 2, 3), BTreeNode(5, 6, 7), …, BTreeNode(25, 26),
                BTreeNode(28, 29)]
   ```
8. We don't need to split the array. We create the **BTreeNodes** for this level and add it to **internalNodesDeque**
   ```
   level = [BTreeNode(4, 8, 12, 16), BTreeNode(20, 24, 27)]
   internalNodesDeque = [level, baseLevel]
   ```
9. Continue through the while loop, since level.size() > 1
10. Pop the max keys from each node in the most recent level, except for the very last node, and partition them into groups of size 4
    ```
    maxKeys = [16]
    prevLevel = [BTreeNode(4, 8, 12]), BTreeNode(20, 24, 27)]
    ```
11. Create the **BTreeNodes** for this level and add to **internalNodesDeque**
    ```
    level = [BTreeNode(16)]
    internalNodesDeque = [level, prevLevel, baseLevel]
    ```
12. Since level.size() == 1, we do not need to continue. We have completed the construction of the internal nodes of the B-Tree.

At the conclusion of the execution, we will have an ordered deque of the levels of internal nodes for the B-tree representation of the memtable to be flushed. Below is a visualization of the B-Tree.



Step 2: Linking Internal Nodes to Child Nodes

In this section, we will cover the logic surrounding linking internal nodes to child nodes. To do so, we will iteratively traverse the B-tree, level by level, and left to right.

For each key in each node, we can determine its corresponding child's page number with the following equation.

$$\begin{aligned}\text{childPageNumber} = \text{currentPageNumber} + \\ (\text{numNodesInLevel - currNodeNumberInLevel (1-indexed)}) + \\ \text{numKeysInLevelBeforeCurrentNode} + \\ \text{keyNumberInLevel (1-indexed)}\end{aligned}$$

By using this equation on each key within the internal nodes of the B-tree, and padding the pages to make them 4KB aligned, we return an array of key-pointer pairs for all the internal nodes represented as 4KB pages, to be used by the API when flushing the memtable to disk.

This logic is covered within the **flushMemtableToBPlusTree()** function. The function was introduced briefly earlier, but we will go into more detail about how this function works in the following section. This is the only function that the API interacts with to create B-Trees. As such, it is vital to understand how it works.

Implementation: flushMemtableToBPlusTree()

Parameters:

- **uint64_t *sortedMaxKeys** - A sorted array of the maximum key in each page (leaf node). These keys will make up the base layer of internal nodes.
- **uint64_t numPages** - The number of pages (leaf nodes)
- **int &numInternalNodes** - The number of internal nodes in the constructed B-Tree. This value is set after the construction of the B-Tree.

Returns:
Recall the definition of a pointer in the context of our KV-Store. Recall from Important Remark: The n-th Child Node that any internal node will be a 4KB page structured as the following:

```
[<key1, pointer1>, ..., <0, last child pointer>] + <0, 0> until page is 4KB aligned
```

Then, the **flushMemtableToBPlusTree** returns an array of these <key, pointer> pairs representing all the internal nodes of the B-Tree**.** Additionally, since the value of **numInternalNodes** is also provided, we know that the return value is a sequence of pages with Key-Pointer pairs, and we know how many Key-Pointer pairs there are, since each page is 4KB aligned.

Algorithm: flushMemtableToBPlusTree()

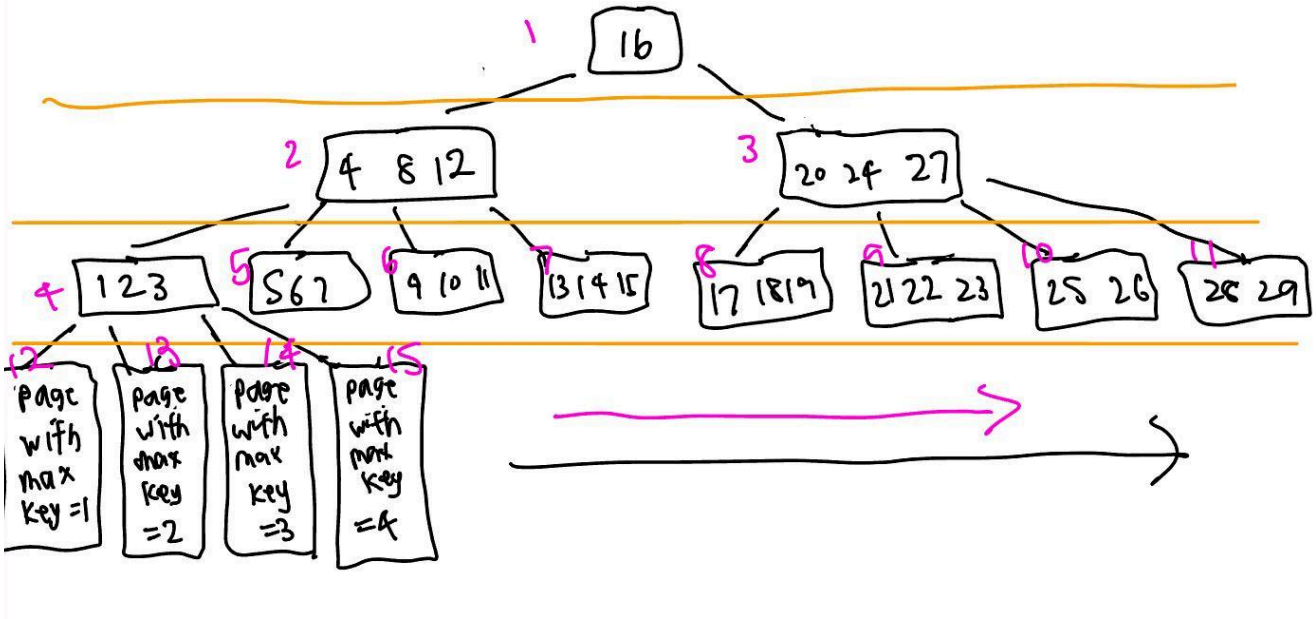
Below is the pseudocode for the algorithm of flushing the memtable to a B+ Tree.

```
flushMemtableToBPlusTree(sortedMaxKeys, numPages, &numInternalNodes):
       bPlusTree = createBPlusTree(sortedMaxKeys, numPages)

       numInternalNodes = count internal nodes in bPlusTree

       internalNodes = KVPair[numInternalNodes * KV_PAIRS_PER_PAGE]
       for each level in bPlusTree:
              for each node in the level:
                     for each key in the node:
                            calculate child page node using equation from above
                            if we are on the last child:
                                   add <0, childPageNumber> to internalNodes
                            else:
                                   add <key, childPageNumber> to internalNodes

                     zero pad the rest of the page, if not 4KB aligned

       return internalNodes
```

Example: flushMemtableToBPlusTree()

As an example, we will continue from the example given for **createBPlusTree()**. Observe in the following visualization that all the pink numbers are the pointer values. This is continued from the previous example.

Recall that our fanout = 4, and that pages can store 4 KV-Pairs.



Using the equation and running the algorithm, we would return the following:

```
internalNodes = [<16, 2>, <0, 3>, <0, 0>, <0, 0>,
                 <4, 4>, <8, 5>, <12, 6>, <0, 7>,
                 <20, 8>, <24, 9>, <27, 10>, <0, 11>
                 <1, 12>, <2, 13>, <3, 14>, <0, 15>,
                    ...
                ]
```

API Implementation

Open API

The open() API has the same implementation as part 1. The key difference to note as mentioned above is that now the vector<string> sstNames global variables contains the following naming convention for SSTs: **SST2-1** where the 2 followed by the SST denotes the number of the SST created and then number followed by the delimiter **'-'** denotes the number of internal nodes for this SST.

The function signature is as follows: **int openDBPart2(const char *name);**
The file path for this is: **`src/kv-store/part2-api/db`**

Close API

The close() API has the same implementation as part 1. The key difference to note is that since for part 2, our SST is structured as a B-Tree SST, we now call the flushToDiskPart2() function which handles flushing any existing values into the memtable into an SST before closing the database and clearing the contents of relevant global variables thus ensuring all data has been safely written to the database folder.

The function signature is as follows: **void closeDBPart2();**
The file path for this is: **`src/kv-store/part2-api/db`**

Put API

For part 2 of our implementation, the function implementation for put() remains the same. The following section discusses the change from part 1 to flush the contents of the memtable into a B-tree SST.

Flushing the memtable as a B-Tree SST

Before we flush the memtable to disk, we first check if the memtable is empty. If it is, the function returns immediately as we do not want to create an empty SST. This is evident in the case when we flush fully-filled memtables into B-Tree SSTs and the user issues a call to closeDB(). In this case, we would ensure that we check the contents of the memtable before creating any SSTs.

Since in part 2, we are structuring our SST as a B-Tree SST, we need to ensure that all the internal nodes are first written to the SST. Unlike in part 1, **memtableTraversal()** function now instead stores the **KVPairs** data into an array of at most memtable size and the **keys** into an array of at most memtable size. These two variables are later used to write the contents of the memtable into B-Tree SST.

Writing the B-Tree internal nodes to SST

We first make sure to write the internal nodes into our SST. Since each internal node stores the maximum key from each page of the SST, we loop over these retrieved keys array and construct a smaller **maxKeys** array from it which holds the **max keys** for each page in the SST. These maxKeys are then passed to our static function **BTreeUtil::flushMemtableToBPlusTree()** to retrieve an array of **<key, pagePointer>** pair of internal nodes for the B-Tree. It is important to note that here, pagePointer refers to the pageNumber to the data where this key can be found in our SST. Since a pointer and int page number take the same size of 8 bytes, we decided to proceed with using an integer to represent this information. Upon successfully constructing our B-Tree of internal nodes, we write these to the SST. To ensure we optimize our write operations, similar to part 1, we make use of an output buffer to write our internal nodes to the SST.

Writing the KV pairs to SST

Upon writing the internal nodes of the B-Tree into our SST, we then proceed to write our memtable KV pair data to the SST. To ensure we optimize our write operations, similar to part 1, we make use of an output buffer to write our KV pairs to the SST.

The function signature is as follows: **void putPart2(uint64_t key, int64_t value);**
The file path for this is: **`src/kv-store/part2-api/put`**

Get API

Binary Search

The get() API for part 2 is similar to our implementation using binary search in part 1 get() API function where we first search for the key in memtable and then proceed to search for the SSTs from youngest to oldest order. We will discuss below the changes for part 2 implementation.

Accounting for the number of internal nodes in our B-Tree SST

For binary search, since we are searching for the key in our SST, we need to search for the middle page for the KV pair data of the SST. However, we need to make sure we offset the left index for the binary search by the number of internal nodes in our SST. Since our SSTs are named as 'SST#-#InternalNodes', we can identify the number of internal nodes by using the delimiter '-' and set the left index for the start of the binary search from that value.

Searching for the page in the buffer pool

When performing the binary search, before reading the KV Pairs from the page, we first check if that page exists in our buffer pool or not. If the page exists, we read it from the buffer pool taking constant time (O(1)). If the page doesn't exist, we read the page into the buffer pool and load it into the array of KV Pairs that stores at most a page worth of data. It is important to note that for every page we read during the binary search, we add the page to our buffer pool regardless of whether the page contains the data or not.

The rest of the structure of the algorithm remains the same as our part 1 get() API function.

The function signature is as follows: **int64_t get_binarySearch(uint64_t key)**;
The file path for this is: **`src/kv-store/part2-api/get`**

B-Tree Search

The get() API for B-Tree search begins by first checking if the key is 0 or not. In our implementation, we assume that keys cannot be equal to zero. We then proceed to first searching for the key in the memtable. If the key exists, we return the value associated with this. If not, we proceed to search for the key in the SST.

Searching for the page in the buffer pool

For B-Tree search, we begin our search by setting our page_number to 0 and traversing the SSTs from youngest to oldest order. We search until the number of internal nodes in our B-tree. It is important to note that in our implementation, our pages are 0-indexed. For the current page number, we search if the page exists in the buffer pool or not. It is important to note that for every page we read during the b-tree search, we add the page to our buffer pool regardless of whether the page contains the data or not.

The page ID that we are looking for has the following structure: "SST3-1-PG0". If the page exists in the buffer pool, we retrieve the value in O(1) constant time. If not, we read the page into the KV pair array which has the size of a page. Since we are traversing the internal nodes of the B-Tree, it is important to note that the KV pair here represents <key, page number> pair since the page number and value are both 8-byte integers.

Identifying the required data page-number to read from b-tree internal nodes

In our implementation, once we read the page worth of internal node data, we traverse this page data in reverse since we are interested in finding the first entry that is <0, page_number>. Once we find this, we know we have found the end-index for the keys in this node of the b-tree.

We now have the start index of 0 and end index found which can be used to perform binary search to find where the key exists in the node. The mid-index we find from the binary search is the index whose value is the **page number** we are interested in reading as this is the page that contains the KV pair data that we want to return to the user.

On the identified page number, we know this is the probable page that contains the key that we are interested in. We first check if this **data-page** exists in the buffer pool or not. If it does, we read the page content in O(1) time else we load it into the buffer pool. On this read page, we perform the binary search and determine if we have found the value or whether we need to search another SST and repeat the above-described algorithm.

The function signature is as follows: **int64_t get_bTree(uint64_t key)**;
The file path for this is: **`src/kv-store/part2-api/get`**

Scan API

Binary Search

The scan() API for part 2 is similar to our implementation using binary search in part 1 scan() API function where we first search for the results in memtable and then proceed to search for the SSTs from youngest to oldest order. We will discuss below the changes for part 2 implementation.

Accounting for the number of internal nodes in our B-Tree SST

For binary search, since we are searching for the start key for scan range in our SST, we need to search for the middle page for the KV pair data of the SST. However, we need to make sure we offset the left index for the binary search by the number of internal nodes in our SST. Since our SSTs are named as ‘SST#-#InternalNodes’, we can identify the number of internal nodes by using the delimiter ‘-’ and set the left index for the start of the binary search from that value.

Searching for the page in the buffer pool

When performing the binary search, before reading the KV Pairs from the page, we first check if that page exists in our buffer pool or not. If the page exists, we read it from the buffer pool taking constant time (O(1)). If the page doesn’t exist, we read the page into the buffer pool and load it into the array of KV Pairs that stores at most a page worth of data. It is important to note that for every page we read during the binary search, we add the page to our buffer pool regardless of whether the page contains the data or not.

Walking the valid pages of the SST

Once we find a probable page number from the binary search, we set the variable that determines if we need to continue performing binary search to false as now we need to walk the valid pages of this SST. We need to read the identified probable page number into the KV Pair array variable and once again check if the page ID we are interested in exists in our buffer pool or not.

Upon checking and reading the data and loading it into the buffer pool if required, we read the kv pairs in this SST’s page and add all the valid entries in the scan range. Once we find a key that doesn’t lie in the scan range, we know that any future pages of this SST aren’t to be searched since an SST is sorted. Hence we can break out of reading this SST and proceed to read the next youngest SST.

The results of the scan function are sorted before returning to the user. As mentioned in the KV pair class definition section of our report, we have overridden the default comparison operator method that compares the keys in our results variable. By calling the standard library's sort() method, we can automatically sort our vector<KVPair> objects using our defined operator().

The function signature is as follows: **int64_t scan_binarySearch(uint64_t key1, uint64_t key2)**;
The file path for this is: **`src/kv-store/part2-api/scan`**

B-Tree Search

The scan() API for B-Tree search begins by first checking if either of the input keys are 0 or not. In our implementation, we assume that keys cannot be equal to zero. We then proceed to the scan results in the memtable.

Searching for the page in the buffer pool

For B-Tree search, we begin our search by setting our page_number to 0 and traversing the SSTs from youngest to oldest order. We search until the number of internal nodes in our B-tree. It is important to note that in our implementation, our pages are 0-indexed. For the current page number, we search if the page exists in the buffer pool or not. It is important to note that for every page we read during the b-tree search, we add the page to our buffer pool regardless of whether the page contains the data or not.

The page ID that we are looking for has the following structure: "SST3-1-PG0". If the page exists in the buffer pool, we retrieve the value in O(1) constant time. If not, we read the page into the KV pair array which has the size of a page. Since we are traversing the internal nodes of the B-Tree, it is important to note that KVPair here represents <key, page number> pair since the page number and value are both 8-byte integers.

Identifying the required data page-number to read from b-tree internal nodes

In our implementation, once we read the page worth of internal node data, we traverse this page data in reverse since we are interested in finding the first entry that is <0, page_number>. Once we find this, we know we have found the end-index for the keys in this node of the b-tree.

We now have the start index of 0 and end index found which can be used to perform binary search to find where the key exists in the node. The mid-index we find from the binary search is the index whose value is the **page number** we are interested in reading as this is the page that contains the KV pair data that we want to return to the user.

Walking the valid pages of the SSTs

On the identified page number, we know this is the probable page that contains the start key that we are interested in. At this point, we begin to walk the pages of the SST. We first check if the **data-page** exists in the buffer pool or not. If it does, we read the page content in O(1) time else we load it into the buffer pool. On this read page, we keep adding the valid KV pairs into the results vector. Once we find a key that lies outside the scan range, we know that any remaining pages of this SST need not be

searched since our SST is sorted. We then break out and proceed to search the next youngest SST and repeat the algorithm.

The results of the scan function are sorted before returning to the user. As mentioned in the KV pair class definition section of our report, we have overridden the default comparison operator method that compares the keys in our results variable. By calling the standard library's sort() method, we can automatically sort our vector<KVPair> objects using our defined operator().

The function signature is as follows: **int64_t scan_bTree(uint64_t key1, uint64_t key2)**;
The file path for this is: **`src/kv-store/part2-api/scan`**

Unit Tests & Integration Tests

As we develop our solution for part two of the project, we have extensively tested our code base writing unit tests for each aspect of our project. In total, we have 21 unit tests written for part two of our project. A breakdown of these is as follows:

- Buffer Pool [`/test/buffer-pool`] - 6 tests
- B-Tree [`/test/b-tree`] - 4 tests
- Get API [`/test/part2-api/get`] - 4 tests
- Put API [`/test/part2-api/put`] - 1 tests
- Scan API [`/test/part2-api/scan`] - 6 tests

Note: Since openDB() for part 2 is analogous to part 1 and the same applies for closeDB() for part 2, we haven't implemented any unit tests for these. The close API calls flushToDiskPart2() which has been tested internally in the Put API test. Furthermore, we have included an end-to-end test for part 2 which incorporates all the functions for part 2.

IV. Part 3: LSM Tree With Bloom Filters

All code related to part 3 can be found under: **`src/kv-store/part3-api/`**

For part 3, we would be structuring our SSTs into an LSM Tree with a size ratio of 2. We will do so by implementing a compaction mechanism, adding a filter for each SST, and loading filters into the buffer pool when accessed.

Bloom Filters

The file path for this is: **`src/kv-store/bloom-filters`**

In our KV-Store, we implement Bloom Filters alongside our LSM-Tree as a probabilistic data structure allowing us to efficiently test whether a given key is in a level of the LSM-Tree. This will help us reduce the amount of unnecessary disk I/Os during membership testing.

Choosing Bits Per Entry and Number of Hash Functions

As described in the project handout for the experiments, we fix our bits per entry to be M = 8.

As the false positive rate is equal to $2^{-M*ln(2)}$, this allows to ensure a false positive rate of 2.14%, assuming the optimal number of hash functions.

Consequently, as given by the formula $ln(2) * M$, the optimal number of hash functions is 6. In our project, this value is calculated at compile time, which allows us to ensure no matter how many bits per entry we decide to have, that we have the optimal number of hash functions.

Optimization: Generating Different Hash Functions

Assume that the number of hash functions is optimal, which in our case, is 6. But since our implementation allows for a variable number of hash functions based on the required false positive rate, we will denote the optimal number of hash functions as H.

Because H can be variable based on what the bits per entry is, we need to have a way of generating H hash functions. If we computed a new hash function for all $i < H$, for example setting the seed of a hash function and computing it H times, this can be very computationally expensive.

To address this, we use the Kirsch-Mitzenmacher technique, described in [this research paper](). This technique aims to reduce the number of hash function computations required when using Bloom Filters by deriving additional hash values from base hash values, eliminating the need to compute a new hash function H times. The technique works using double hashing, by using 2 base hash functions `h1(x)` and `h2(x)` and simulating additional hash functions by computing

```
h(x) = h1(x) + (i * h2(x)).
```

This allows us to greatly speed up the process of constructing the bloom filter, because for each key we only need to compute the hash function twice, as opposed to computing the hash function H times. Because MurmurHash already produces 2 64-bit hash values, we use these hash values as `h1(x)` and `h2(x)`. By nature of MurmurHash, these 2 halves are sufficiently independent such that we can use them as separate hash functions.

Furthermore, the paper details that using this technique does not actually cause any loss in asymptotic false positive probability. This implies that the derived hash values from this technique are effectively independent for practical purposes like our Bloom Filter, ensuring that we maintain our expected performance characteristics like false positive rate.

Implementation Details

We represent the Bloom Filter with the **BloomFilter** class, which manages a bitmap representing the Bloom Filter, which is sized dynamically according to the corresponding level of the LSM-Tree it is being constructed for.

Some of the key methods from the **BloomFilter** class are the following:

- **calculateBitmapSize(int level)** - given the level in the LSM-Tree, calculate the size needed for the bitmap. This function is called in the constructor to initialize the Bloom Filters' bitmap.
- **insertKey(uint64_t key)** - inserts a key into the Bloom Filter, using the Kirsch-Mitzenmacher technique to efficiently compute the hash functions and update the bitmap.
- **getKeyBitLocationForHashFunction(uint64_t key, int level, int hashFunctionNumber, int& bfPageNumber, int&bfPageOffset)** - given a key and the hash function number, returns the page number and page offset of the target bit. Used by the GET API to locate a specific bit in the Bloom Filter bitmap associated with a specific key.

Algorithm: insertKey()

Below is the pseudocode for inserting a key into the Bloom Filter. The important thing to note is that for each hash function, we use the **Kirsch-Mitzenmacher** technique to compute the hash value.

```
insertKey(key):
	hash1, hash2 = hash(key)
	for each hash function i:
		use Kirsch-Mitzenmacher technique to compute hash value
		bitmap[hashValue] = 1
```

Explanation: getKeyBitForHashFunction()

Because the Bloom Filters are stored in storage, accessing its bits requires translating the hashed key values into specific storage locations. By nature of the bloom filters, the GET API needs to check the bit value for a given key, for each hash function.

But since the API needs to know exactly the page and offset of the target bit is in storage, it needs an efficient mannerism to locate a specific bit given a key. This function **getKeyBitForHashFunction()** is used for this purpose.

The parameters and their purposes for the function are as follows:

- **uint64_t key -** the key we are trying to determine exists
- **int level -** the level of the LSM tree we are currently on, in order to determine the bitmap size
- **int hashFunctionNumber -** the hash function number, so we can use the appropriate hash function to determine the location of the target bit
- **int &bfPageNumber -** once calculated, assigns this parameter the value of the page number so it can be accessed accordingly
- **int &bfPageOffset -** once calculated, assigns this parameter the index of the bit within the page, so it can be accessed accordingly.

This function plays a crucial role in bridging the gap between the GET API and storage, with regards to using the Bloom Filters to determine whether a specific bit is set to 0 or 1. The GET API will use the values of **bfPageNumber** and **bfPageOffset** in order to check the value of a given bit.

Changes made in relation to part 2

Introduction

In this updated version, we have introduced a new variable, lsmNames, which is a vector of strings similar to the previously used sstNames. This variable keeps track of metadata for the LSM tree, facilitating operations like get(), put(), and scan(). The lsmNames vector serves the same purpose as sstNames—to store the names of SST files in the database. However, there are some important differences in how it is structured and used.

lsmNames Variable

The `lsmNames` vector is zero-indexed, with each index corresponding to a specific level of the LSM tree. For instance, `lsmNames[0]` is always an empty string because it corresponds to level 0 of the LSM tree, which is the buffer handling all insertions (the memtable). Using a vector instead of an array for `lsmNames` is necessary because we cannot predict how large the LSM tree will grow, and consequently, how many levels (and SST files) will be created. This dynamic nature is the same reason we originally chose a vector for `sstNames`.

File Structure Changes

Previously, we stored B-Tree nodes and their corresponding SST data in a single file. Now, we have separated them into two distinct files for greater flexibility during compaction. When merging levels in the LSM tree, we cannot load entire SSTs into memory. Instead, we use two input buffers and one output buffer to merge pages from two SSTs into a new SST. The output buffer flushes data to disk whenever it fills.

During this flush process, we are still constructing the B-Tree in memory. Storing only the maximum key of each page allows the in-memory B-Tree to be significantly smaller—by a factor of 256, given that only a single maximum key per page is kept. However, since we cannot prepend data to the start

of the SST file (where the B-Tree would logically reside), placing the B-Tree in the same file as the SST data would be inefficient. Prepending would require rewriting the entire SST.

To avoid unnecessary rewriting, we now store the B-Tree in a separate file. The same logic applies to Bloom filters. By giving them their own files, we avoid costly operations and maintain more efficient read/write processes.

File Types and Naming Conventions

Now, our system consists of three types of files:

1. **SST Files (Data)**: Containing all the key-value pairs, with filenames prefixed by `SST` (e.g., `SST1`, `SST2`, `SST3`, ...).
2. **B-Tree Files**: Containing B-Tree nodes, with filenames prefixed by `BT` (e.g., `BT1`, `BT2`, `BT3`, ...). The file `BT1` corresponds to `SST1`, `BT2` to `SST2`, and so forth.
3. **Bloom Filter Files**: Containing Bloom filter pages, with file names prefixed by `BF` (e.g., `BF1`, `BF2`, `BF3`, ...). Similar to the B-Tree files, `BF1` corresponds to `SST1`, `BF2` to `SST2`, and so on.

All these files are binary files. Unlike in the previous version, we no longer use filenames like `SST1-7` to indicate the number of internal B-Tree nodes in the SST file. Since the B-Tree is now separate, the number of internal nodes can be determined by dividing the B-Tree file size by the page size and rounding up to the nearest integer.

Relationship Between lsmNames and Files

The `lsmNames` vector only contains SST file names for the levels of the LSM tree. If an SST file is listed in `lsmNames`, its corresponding B-Tree (`BT`) and Bloom filter (`BF`) files also exist. For example, if `lsmNames[4]` is `SST32`, it implies that `SST32`, `BT32`, and `BF32` all exist as part of the LSM tree's file structure.

API Implementation

Put API

Introduction

Our implementation of the `put()` API has evolved significantly from Part 2, mainly due to the introduction of a fully structured LSM tree architecture. This new design brings changes to how we handle insertions, compactions, updates, and deletions. In addition, it affects how we create and manage SST, BT, and BF files.

LSM Tree Overview

The `put()` function now integrates all user insertions into an LSM tree. Each level of the LSM tree corresponds to a single SST stored on disk. Along with each SST, we maintain corresponding B-Tree (BT) and Bloom Filter (BF) files. When `put()` is invoked, it ensures that new data eventually makes its way into this hierarchical storage, undergoing any necessary updates or deletions along the way.

Key Assumptions and Insights

1. **Size Ratio of LSM Tree Levels:**
   We use an LSM tree with a fixed size ratio of 2 between consecutive levels. Whenever there are two SSTs at the same level, they are merged (compacted) into a single SST and pushed down to the next level.
2. **Levels Are Either Empty or Half-Filled:**
   The `lsmNames` variable stores the names of the SSTs at each level. If a level is empty, `lsmNames` stores an empty string. When a level is half-filled, it holds one SST name. As soon as it becomes full (i.e., it would need a second SST), compaction is triggered, pushing the data down and leaving the current level either empty or half-filled again. This equilibrium ensures that levels are never fully occupied for long, maintaining a balanced structure.
3. **Compactions and In-Memory Buffers:**
   During compaction, we do not load entire SSTs into memory at once. Instead, we use three buffers (each buffer is 4KB): two input buffers for reading pages from the two SSTs being merged, and one output buffer for the newly created SST. We repeatedly append the smallest key from the inputs to the output buffer. When the output buffer is full, it is flushed to disk. This incremental approach minimizes memory usage and keeps compactions efficient.

Put Method

1. **Memtable Insertion:**
   Incoming key-value pairs are first inserted into an in-memory memtable, implemented as an AVL tree. If a key already exists, it is updated with the new value. The memtable acts as a buffer, allowing fast writes and simple updates. Deletes are also handled at this stage by inserting a tombstone value (INT64_MIN).
2. **Flushing to Disk and Creating an SST:**
   Once the memtable reaches its capacity, we invoke `flushToDiskPart3()`. This function:

    - Converts the memtable into a sorted SST.
    - Inserts the new SST into the LSM tree at the appropriate level.
    - Handles all necessary compactions triggered by this insertion.
3. After flushing, the memtable is empty, ready to accept new entries.

Updates and Deletions

- **Updates:** If a key is re-inserted with a new value, the old value is replaced in the memtable. When data moves through compactions, if the same key appears multiple times, only the most recent version (from the newest SST) is retained.
- **Deletions:** Deletion is represented by inserting a tombstone instead of a normal value. Tombstones remain with the key through compactions until they reach the last level of the LSM tree. At this final level, tombstones indicate that no older versions exist, and thus they can be safely removed.

Compaction Mechanics

Compaction is driven by the `lsmNames` variable. Here is how the process works:

1. **Finding the Target Level:**
   When inserting a new SST (after flushing the memtable), we first try to insert it at `lsmNames[0]`. If that slot is occupied, we move to the next level, merging and pushing data down until we find an empty slot. If no empty slot exists, we add a new level at the end of the `lsmNames` vector.
   This process identifies an `index` representing the level where the new SST will ultimately reside.
2. **Compaction Strategy:**
    - **From Level 0 to `index-2`:**
      We use `compactionWithoutDelete()` to merge SSTs without generating B-Trees or Bloom Filters, and without removing tombstones. Since these intermediate levels will be compacted again, there is no need for the extra overhead of building BT and BF files at these stages. After each compaction, the old SST files (along with their BT and BF files) are removed.
    - **At Level `index-1`:**
      Here, we perform the final compaction before placing the SST at the target level `index`. At this point, we know that the SST resulting from this compaction will not be merged again, so we create the BT and BF files for it.
3. **Handling Tombstones at the Final Level:**
   If the `index` corresponds to the last level of the LSM tree (i.e., `index == lsmNames.size()` before insertion), we call `compactionBTandBFWithDelete()`. This function removes tombstones (since this is the last level), creates the BT and BF files for the final SST, and discards old SSTs and their associated BT/BF files.
   If the `index` is not the last level, we call `compactionBTandBFWithoutDelete()`. This function leaves tombstones intact but still creates the BT and BF files for the SST. It also discards the old SSTs and their associated BT/BF files.

Creating and Managing Files

- **SST Files:** Contain key-value data.
- **BT Files:** Contain the B-Tree structure. Created only for the final SST in each compaction chain.
- **BF Files:** Contain the Bloom Filter pages, also created at the final compaction level or whenever the final SST in a chain is formed.

After compaction completes at the final level (`index`), the `lsmNames` entry at that level is updated with the new SST name. The levels above it (0 to `index-1`) are emptied out in `lsmNames` because their SSTs have been either merged into this new SST or pushed down further.

The `put()` operation not only inserts and updates key-value pairs in an LSM tree but also efficiently manages compactions, file creation, and tombstones. By using the `lsmNames` variable, a well-defined compaction flow, and separate files for SST, BT, and BF data, we achieve a scalable and flexible storage layer that can handle unlimited insertions, updates, and deletions over time.

Get API

Binary Search

The get() API for part 3 is similar to our implementation using binary search and buffer pool in part 2 get() API function where we first search for the key in memtable and then proceed to search for the key in our LSM tree going from top to bottom level. We will discuss below the changes for part 3 implementation.

Searching for the bloom filter in the buffer pool

Depending on the number of hash functions defined, we will loop over each hash function number and call the **BloomFilters::getKeyBitLocationForHashFunction()** which takes in the current level of the LSM tree, the number for the hash function to be computed and sets by reference the bloom filter page number to search and the bit location for which we need to check if the value is 0 or 1. If the checked value is 0 for any of the computed hash functions, we do not need to compute the rest of the hash functions since we are sure the key is not present.

Once we get the page number to search, we will construct our pageID which has the following format as the example: “BF1-PG5”. We will check for this pageiD in our buffer pool and determine if the bloom filter page exists in our buffer pool or not. If it does, we retrieve the data in O(1) time, else we load it into the buffer pool.

Upon performing the buffer pool we check, we inspect the bit value at the location provided to us by the above function. If the value at that index is 1, we can know that the key exists due to a true positive or a false positive. If the value at that index is 0, we can be sure that the key will not exist at this level of the LSM tree since a bit cannot be set from 1 to 0 and a bloom filter cannot have **false negatives**.

The rest of the structure of the algorithm remains the same as our part 2 get_binarySearch() API function. For binary search, since we are searching for the key in our SST which as discussed in the introduction section of part 3 is in it's own binary file, unlike part 2, we **do not** need to account for any internal nodes as the B-Tree nodes are in a separate binary file. Hence the left-index for the binary search would begin from 0.

The function signature is as follows: **int64_t get_binarySearch_part3(uint64_t key)**;
The file path for this is: **`src/kv-store/part3-api/get`**

B-Tree Search

The get() API for part 3 is similar to our implementation using b-tree search and buffer pool in part 2 get() API function where we first search for the key in memtable and then proceed to search for the key in our LSM tree going from top to bottom level. We will discuss below the changes for part 3 implementation.

Searching for the bloom filter in the buffer pool

Depending on the number of hash functions defined, we will loop over each hash function number and call the **BloomFilters::getKeyBitLocationForHashFunction()** which takes in the current level of the LSM tree, the number for the hash function to be computed and sets by reference the bloom filter page number to search and the bit location for which we need to check if the value is 0 or 1. If the checked value is 0 for any of the computed hash functions, we do not need to compute the rest of the hash functions since we are sure the key is not present.

Once we get the page number to search, we will construct our pageID which has the following format as the example: "BF1-PG5". We will check for this pageiD in our buffer pool and determine if the bloom filter page exists in our buffer pool or not. If it does, we retrieve the data in O(1) time, else we load it into the buffer pool.

Upon performing the buffer pool we check, we inspect the bit value at the location provided to us by the above function. If the value at that index is 1, we can know that the key exists due to a true positive or a false positive. If the value at that index is 0, we can be sure that the key will not exist at this level of the LSM tree since a bit cannot be set from 1 to 0 and a bloom filter cannot have **false negatives**.

Identifying the true page-number identified from B-Tree search

Since now, our b-tree internal nodes and SST data are separated into their own respective binary files, we know that unlike in part 2, the index identified doesn't point to a page number within the same SST.
We need to keep in mind that the page number it points to is with respect to the current page we were at in the b-tree. Therefore, the new formula to retrieve the data page to search when we read the SST binary file is given as follows:

```
page_number = page_number - num_internal_nodes
```

Taking this change into consideration, the rest of the structure of the algorithm remains the same as our part 2 get_bTree() API function.

The function signature is as follows: **int64_t get_bTree_part3(uint64_t key)**;
The file path for this is: **`src/kv-store/part3-api/get`**

Scan API

Use of struct to store metadata regarding the LSM scan result

To perform a scan over the LSM data, we know that we would require having an output buffer for each level of the LSM tree. For our implementation, we have the output buffer of size **1 page** for each level. To better structure our code for readability, we have decided to use a struct to store all the metadata we would need when loading data into the output buffer for the core scan algorithm. The below is the definition of the struct:

```
struct scanBuffer {
      KVPair page[KV_PAIRS_PER_PAGE];
      int numKVPairsInpage = -1;
      int currentPageNumber = -1;
      int maxpages = -1;
}
```

Design choice: What purpose does the struct serve

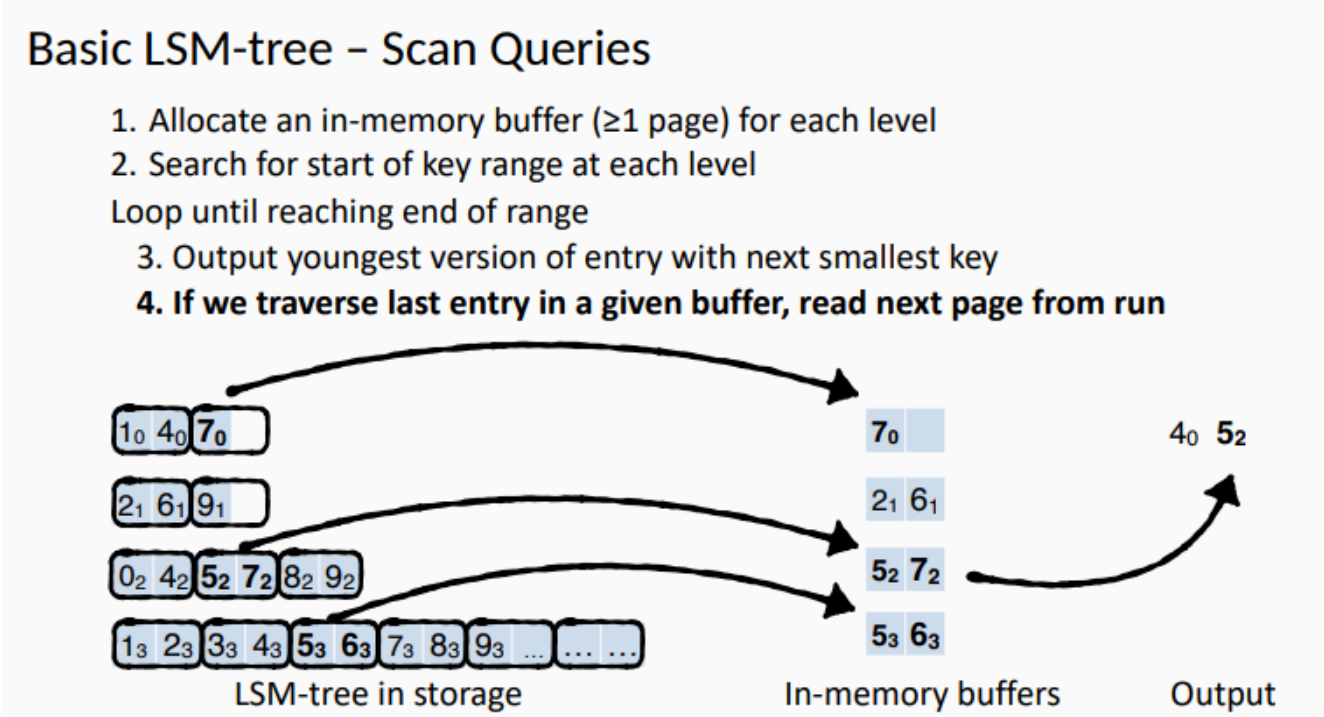
Since the scan algorithm for an LSM tree requires loading a page worth of data for each level simultaneously, we decided to use a struct to better organize the information that we read. Since each page from each level can have varying sizes at any given point, a struct allows us to group relevant variables that we require in a succinct manner while ensuring we avoid accidental updates to any other buffer at another level. The sturct has a **KVPair page** array of maximum size KV_PAIRS_PER_PAGE as defined from the constants in important constants section of our report. The struct also has **numKVPairsInpage**, **currentPageNumber** and **maxpages** to keep track of page related metadata for the currently read page at the level. As we read relevant pages at a level into our buffer, we update the struct variables accordingly. As mentioned above, this happens for every buffer at every level.

Binary Search

The scan() for part 3 of our project starts off by checking if either of the keys is 0 or not. This is because in our implementation, we assume keys cannot take the value 0. Following this, we proceed to get the scan result from the memtable. It is important to note that since bloom filters do not support scan, we do not use a bloom filter for the scan API implementation.

Loading pages from buffer pool for each level buffer

Using the above struct, we begin by populating the buffer for each level. We begin by performing a binary search on the probable page and find if the page exists in our buffer pool or not. This process is done for each level in our LSM tree that isn't empty.



We implement our scan function following the high level psuedocode described from the lectures. We first allocate an in-memory buffer of size one page for each level of the LSM tree using our struct above. We then find the start of our scan key range at each of the levels and continue to loop until we reach the end of the key range. As we find valid entries, we push them to the results vector. Since we know that the more recent version of KV pairs lie in the higher level of the tree, using the **findKeyInVector()** function, we can determine if a version of the key in scan range was inserted into the results vector or not. Once we find that we have read the last KV pair in the currently read page for a level, before the next iteration, we read the next page for this level and check if this page already exists in the buffer pool or not. If it does, we proceed to access it in O(1) time and otherwise add this page to the buffer pool and update the struct scanBuffer for this level.

This algorithm continues to run until we search for all valid pages for every level of the LSM tree. Finally, before returning to the user, we scan the results vector and remove any tombstone entries since these indicate that the KV pair has been deleted. In our implementation, tombstones take **INT64_MIN** value. After removing any tombstones in our results vector, we finally return the output to the user.

The function signature is as follows:

int64_t scan_binarySearch_part3(uint64_t key1, uint64_t key2);

The file path for this is: **`src/kv-store/part3-api/scan`**

B-Tree Search

The scan() for part 3 of our project starts off by checking if either of the key is 0 or not. This is because in our implementation, we assume keys cannot take the value 0. Following this, we proceed to get the scan result from the memtable. It is important to note that since bloom filters do not support scan, we do not use a bloom filter for the scan API implementation.

Identifying the true page-number identified from B-Tree search

Since now, our b-tree internal nodes and SST data are separated into their own respective binary files, we know that unlike in part 2, the index identified doesn't point to a page number within the same SST.
We need to keep in mind that the page number it points to is with respect to the current page we were at in the b-tree. Therefore, the new formula to retrieve the data page to search when we read the SST binary file is given as follows:

```
page_number = page_number - num_internal_nodes
```

Using the above struct and the identified page_number, we begin to populate the scan buffer for each level. We begin by performing a b-tree search on the probable page and find if the page exists in our buffer pool or not. This process is done for each level in our LSM tree that isn't empty.

Implementing the Scan API

We implement our scan function following the high level psuedocode described from the lectures. The rest of the implementation remains exactly the same as the implementation of scan using binary search in this section.

The function signature is as follows:

int64_t scan_bTree_part3(uint64_t key1, uint64_t key2);

The file path for this is: **`src/kv-store/part3-api/scan`**

Delete API

The delete API implementation is used to indicate that an input KV pair is said to be deleted from our KV store database. To achieve this, we call the put API from part 3 and pass the input key and a value of INT64_MIN to mark that the latest version of this KV pair denotes that the entry is to be deleted.

The function signature is as follows:

void deleteKey(uint64_t key);

The file path for this is: **`src/kv-store/part3-api/delete`**

Unit Tests & Integration Tests

As we develop our solution for part 3 of the project, we have extensively tested our code base writing unit tests for each aspect of our project. In total, we have 50 unit tests written for part 3 of our project. A breakdown of these is as follows:

- Bloom Filters [`/test/bloom-filters`] - 5 tests
- DB Tests [`/test/part3-api/db`] - 3 tests
- Delete Tests [`/test/part3-api/delete`] - 3 tests
- GET (Binary Search) Tests [`/test/part3-api/get/get-binarySearch`] - 6 tests
- GET (B-Tree) Tests [`/test/part3-api/get/get-bTree`] - 6 tests
- PUT Tests [`/test/part3-api/put`] - 7 tests
- SCAN (Binary Search) Tests [`/test/part3-api/scan/scan-binarySearch`] - 10 tests
- SCAN (B-Tree) Tests [`/test/part3-api/scan/scan-bTree`] - 10 tests

In addition to the unit tests, we have also written an end-to-end test utilizing various API methods simulating a realistic database interaction by a user. This end-to-end test can be found under **/test/end-to-end-test/part3**.

V. Experiments

Introduction

Background

Within the directory titled ***experiments*****,** there are 2 folders for running the experiments for each part. The folder named ‘part2’ includes experiments for part 2 of the project and the folder named ‘part3’ includes experiments for part 3 of the project. Each folder will have the files labeled **main.cpp** and **partX.py**, where X is either 2 or 3 depending on which experiment you want to run.

The file **main.cpp** is the code for the experiment, it performs the experiment and measurements.

The file **partX.py** is the driver code for running the experiment. It will call the executable created by **main.cpp** and process the results, creating corresponding CSV files per the project requirements while also generating graph figures to visualize the results using the Python library matplotlib.

IMPORTANT: Note that you should NOT run both the experiments for part 2 and part 3 simultaneously since they would interfere with each other and terminate unexpectedly giving you a bunch of errors. This is because SSTs created by one experiment could be modified by another experiment if they are run together. To avoid this, only run one experiment at a time and begin the next one only after the current one has finished running.

Measurements

These experiments are designed to measure the throughput (operations per second) as the volume of data stored in the databases scale exponentially from 1MB up to 1GB.

Fixed Constants

For all of the experiments, they are run with the following constants:

- Memtable Size: 1MB
- Page Size: 4KB
- Buffer Pool Size: 10MB
- Bloom Filter Bits Per Entry: 8

Prerequisites

Note that to run the experiments, it is assumed that you have Python3 and the **matplotlib** and **pandas** libraries installed. If you are running the experiments on teach.cs, these prerequisites are already taken care of.

Running the Experiments

As such, to run the experiment, do the following steps:

1. Compile the entire project
2. Navigate to the directory associated with the experiment that you want to run
3. Run the command `python3 partX.py`

Disclaimer: The experiments can take a while to run since we are inserting close to 1 GB worth of data and then issuing various queries over them. On average, it should take approximately 13 mins for part 2 experiments to finish running and 18 mins for part 3 experiments to finish running.

How the Experiments Were Run

To reproduce similar results, **teach.cs** was the environment that we were using to run these experiments.

Disclaimer: Note that the performance of the system, specifically when running these experiments, could have been influenced by various factors including traffic on **teach.cs.**

Part 2 Experiments

How It Works

We fix the number of keys in the experiment to be searched for as 500.

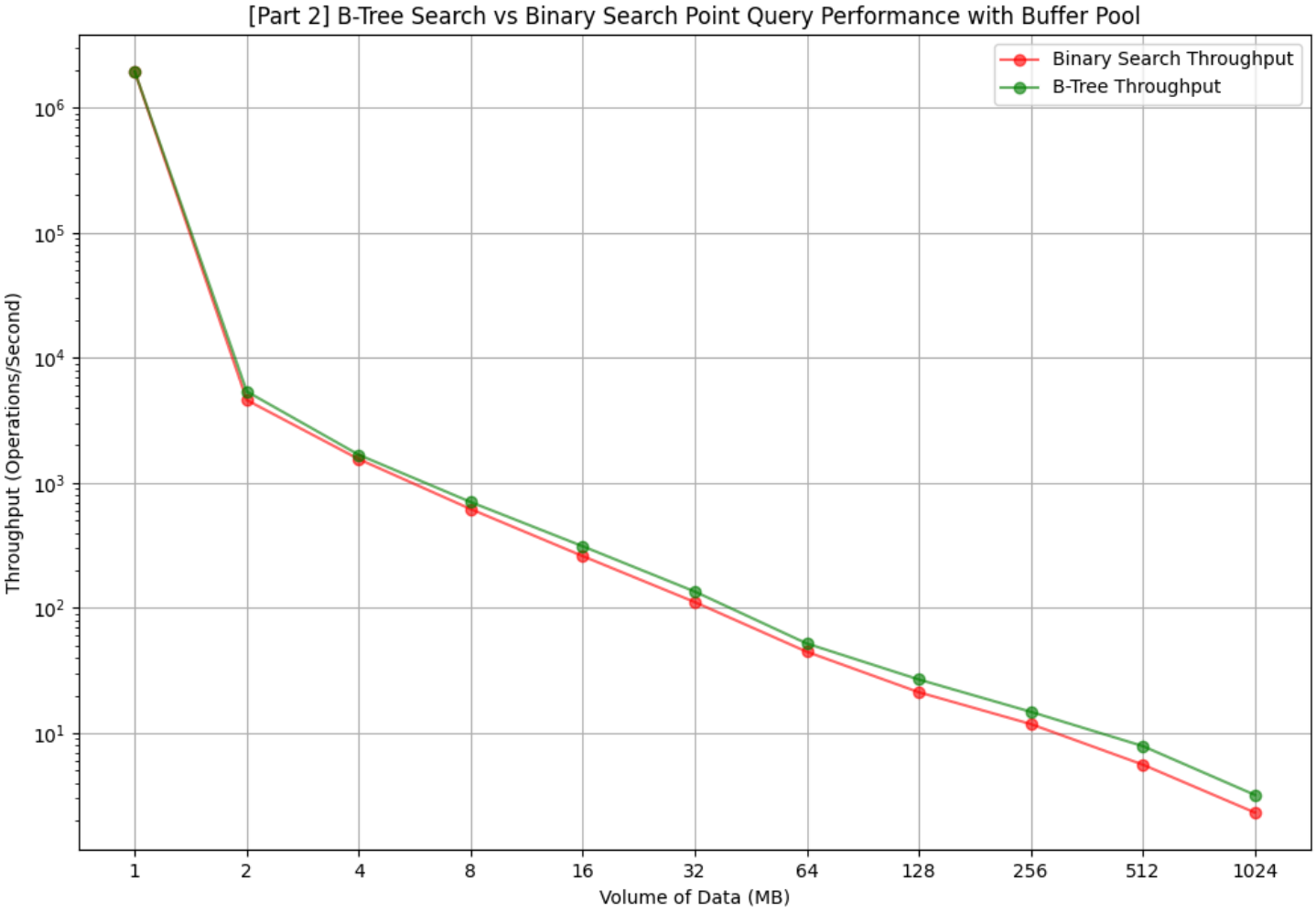
From there, we iterate through each value 1, 2, 4, …, 1024 (exponentially) representing the number of MB of data that we are inserting into our databases.

First, we insert the specified volume of data into the database. These insertions are uniformly randomly distributed as well. Since we are only measuring point queries, this operation is not timed.

Then, for each iteration, we generate a uniformly random array of size 500 keys to search across our entire keyspace, which is [1, UINT64_MAX]

Then, with the buffer pool enabled, we issue 500 uniformly randomly distributed point queries for each implementation of GET - binary search and BTree implementations. For each implementation, we measure the time it takes and calculate the throughput accordingly.

Results: B-Tree Search vs Binary Search Point Query Performance with Buffer Pool



Analysis

This graph illustrates the throughput (operations/second) of binary search and B-tree search for point query performance under varying data volumes, with the buffer pool enabled. The x-axis represents the volume of data in megabytes (MB), while the y-axis, plotted on a logarithmic scale, shows throughput in operations per second. The buffer pool plays a key role in reducing storage I/O operations by caching frequently accessed pages, and the effects of this optimization are evident.

Initially, at small data volumes (1–4 MB), the throughput for both binary search and B-tree search is very high, as most or all operations can be executed within the in-memory buffer pool, avoiding costly storage I/O. As the data size increases, throughput decreases consistently for both methods,

reflecting the growing impact of storage I/O and the computational cost of searching larger datasets. The B-tree search consistently outperforms binary search across all data volumes, with the gap widening at larger sizes (e.g., 512–1024 MB). This performance advantage is due to the hierarchical nature of B-trees, which reduces the likelihood of cache misses, particularly for internal nodes frequently pinned in memory by the buffer pool. In contrast, binary search requires accessing many scattered pages, leading to higher chance of buffer pool misses, and consequently, lower throughput.

Part 3 Experiments

How It Works

We fix the number of keys to search for (i.e. for our GET operations) to be 7000.
We fix the scan range (i.e. for our SCAN operations) to be 7000 as well.

We iterate through each value 1, 2, 4, …, 1024 representing the number of MB of data that we are inserting into our databases.

For each iteration:

We generate a uniformly random array of size 7000 keys to search for our GET operations.

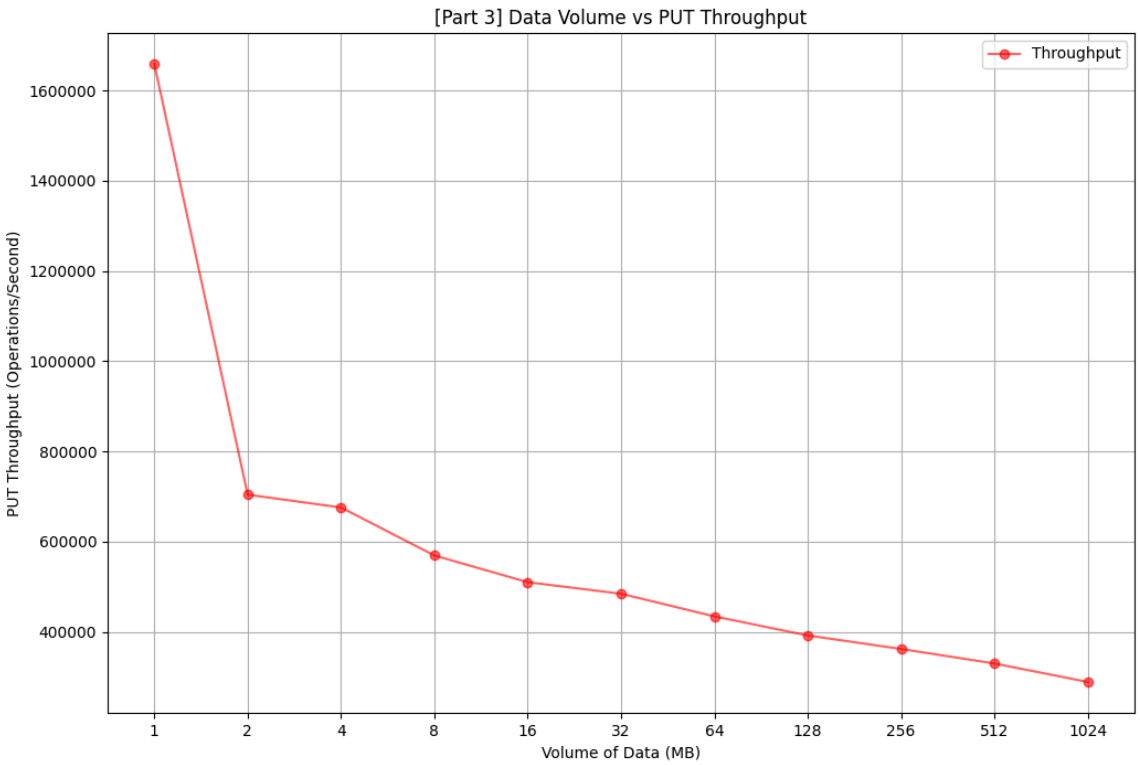
We then uniformly randomly generate the left bounds and right bounds for our SCAN operations such that it gives us a uniformly random range of 7000.

Now, we measure the time it takes to perform X MB of insertions, and then determine the throughput based on how long it takes.

Then, we measure the time it takes to perform 7000 point queries (both using the binary search and the B-Tree implementation), and then determine the throughput based on how long it takes. Note we use both the buffer pool and the bloom filters for GET operations.

Finally, we measure the time it takes to scan a range of 7000 (using both the binary search and B-tree implementation), and then determine the throughput based on how long it takes. Note that the buffer pool is enabled for SCAN operations. However, we do not use bloom filters for scans since bloom filters do not support scans.

Results: Data Volume vs PUT API Throughput



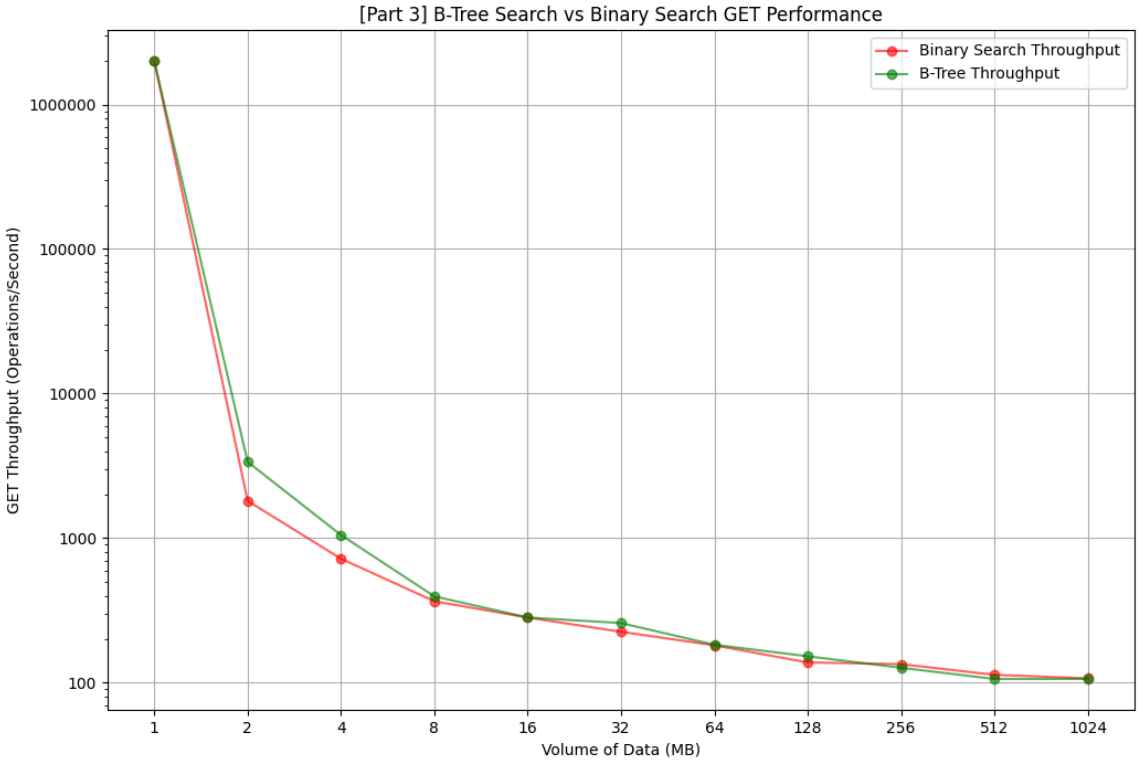
Analysis

We can see that when the data size is equivalent to the memtable size, we see a large jump in throughput, because we do not need to flush to disk, and all of our data fits in memory. As a corollary, we also do not need to incur the costs of creating the Bloom Filters, or B-Tree structure of the SSTs after we flush them to memory.

As we insert more data, we can see that the throughput decreases relatively consistently. Because the LSM Tree size ratio is 2, at each x-axis data point (2, 4, 8, …), our LSM tree actually gains a level. Each new level in the LSM tree adds additional write amplification due to the need for sort-merging during the compaction process. While this helps us in throughput for reads, we incur higher costs when inserting data due to the compaction process. This explains the consistent decrease in throughput as the data size increases and the LSM tree grows deeper in level.

Additionally, with each compaction, we need to construct new Bloom Filters and B-Tree internal nodes for the newly created level in order to keep them up to date. This is also another contributing factor to the observed decrease in throughput as the data size continues to decrease.

Results: Data Volume vs GET API Throughput



Analysis

First, as always, observe that when the volume of data is equal to the memtable size, we see a relatively consistently high throughput due to the lack of need for costly storage I/Os. Also, similarly to the part 2 results, observe that the B-Tree search is usually at least as efficient as binary search - the intuition is the same as from part 2 results.

Instead, here we will compare and contrast the performance in the part 3 API versus the part 2 API.

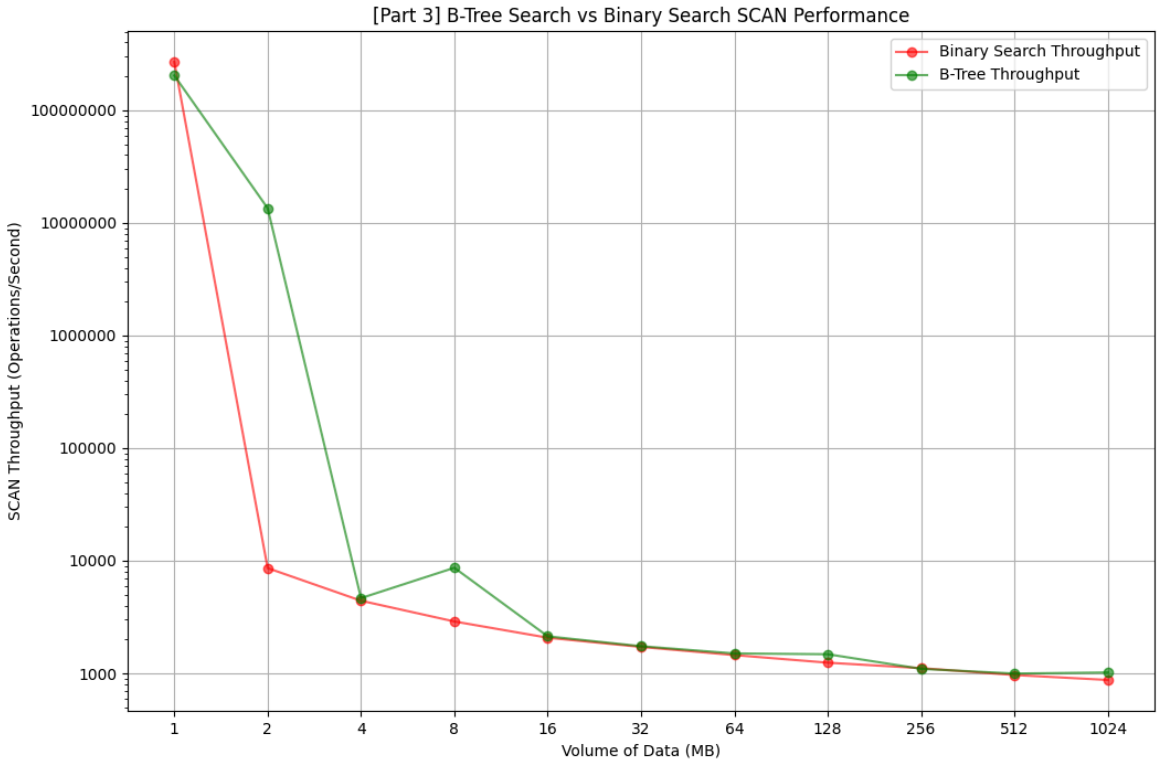
Observe that at low volumes of data, the part 2 API actually outperforms the part 3 API. This behavior is not unexpected - the overhead of creating the Bloom Filters and B-Trees does not counteract the potential savings in lookup time at these smaller data sizes. At low data volumes, the number of levels in our LSM Tree is small, meaning that the system won't need to scan through a significant amount of SSTs to find the desired key. However, with larger amounts of data, the part 3 API begins to perform increasingly better than the part 2 API for point queries.

This increasingly large discrepancy between the part 2 API and the part 3 API as data volumes get increasingly larger can be largely attributed to the use of Bloom filters combined with our LSM Tree. Bloom Filters play a very key role in quickly determining if a key exists in a particular level of the LSM Tree, and if it isn't we can skip querying that entire level all together. This optimization becomes increasingly effective as the dataset grows, the number of levels grows and the number of keys in

deeper levels grows as well, and the bloom filters allow the system to avoid a potentially significant number of unnecessary disk reads.

In contrast, the part 2 API lacks this capability, and must scan all SSTs regardless of whether the key exists, which creates substantially larger storage I/O cost and overhead.

Results: Data Volume vs SCAN API Throughput



Analysis

Initially, at small data volumes (1–2 MB), both binary search and B-tree search exhibit exceptionally high throughput, with binary search slightly outperforming B-tree search at 1 MB. This is likely due to minimal overhead at these smaller data volumes, where most operations can be handled efficiently in memory. However, as the data volume increases beyond the in-memory threshold, both methods show a sharp decline in throughput. B-tree search exhibits more consistent performance at intermediate data volumes (e.g., 4–16 MB), possibly due to its hierarchical structure and lower likelihood of cache misses for internal nodes. At larger data volumes (64 MB and beyond), the throughput of both methods converges, indicating that the I/O and sequential scanning costs dominate regardless of the search method. The overall decline in throughput with increasing data size reflects the growing I/O and computational costs of scanning larger datasets.

VI. Bonus

Flags for Buffer Pool and Bloom Filters

In our implementation, we introduced two flags to control the use of bloom filters and the buffer pool: `BUFFER_POOL_ENABLED_FLAG` and `BLOOM_FILTER_ENABLED_FLAG`. These flags are defined in the `global.h` file within the `utils` folder.

By default, both flags are set to `true`, enabling bloom filters and the buffer pool. However, you are free to adjust these flags as needed. For instance, you can set either flag to `false` in the `main.cpp` file for various parts of the project (e.g., Part 2 or Part 3 experiments) to evaluate the system's performance under different conditions. This flexibility allows you to compare and contrast the performance implications of using or not using the buffer pool and bloom filters in your experiments or tests.

No Metadata used in the implementation

Currently, the only metadata we store is a list of database names and SST file names in two files: `db_names.txt` and `file_names.txt`. However, it is possible to eliminate the need for these separate metadata files entirely. One approach is to dynamically obtain the names of all files in the database directory by invoking a command like `system("ls")` within our code. This would allow the system to self-discover all SST files without relying on explicitly maintained metadata lists.

Additionally, we could adopt a more descriptive and timestamp-based naming convention for SST files to further reduce the need for external metadata. For example, instead of naming a file simply `SST1`, we could incorporate date and time information directly into its filename, such as:

- `SST-21052023-143527-123.sst`
  Here, `21052023` might represent `ddmmyyyy` (21st May 2023), `143527` the time (14:35:27), and `123` a millisecond count or a unique sequence number.

If we need to reflect the LSM tree level as well, we could name files like:

- `SST-21052023-143527-level2.sst`
  Here, `level2` indicates that the SST belongs to level 2 of the LSM tree.

By incorporating timestamps, levels, or other pertinent information directly into the filename, we remove the dependency on separate metadata files. Each file becomes self-describing, making it easy to identify and manage without external references.

Benefits of Eliminating Separate Metadata Files
By relying solely on intrinsic file naming and directory listings, we simplify the system and reduce overhead. This approach makes the database more robust and easier to maintain since no additional files are required to keep track of database or SST states. In the long run, it ensures a more portable and resilient setup, as the essential “metadata” is always co-located with, and inseparable from, the data itself.

BTree and Bloom Filters

Because our group size was reduced from three members to two, the B-Tree and Bloom Filter implementations—along with their integration into our API, the related tests, and the corresponding experimental results—are bonus features that we implemented.

THANK YOU!!