Group Member:
Lone Ji (jilong1, 1005275827, long.ji@mail.utoronto.ca)
Jiaqing Ren (renjiaq1, 1004892351, jiaqing.ren@mail.utoronto.ca)
Boyuan Cui (cuiboyu1, 1003900433, boyuanbryan.cui@mail.utoronto.ca)

Project Description

List of major classes in our project:
- `Database` in `include/database.h`
    - Contains get/put/scan APIs exposed to users
    - Consists of two major attributes: memtable of class `Memtable` and storage of class `StorageType`
- `Memtable` in `include/memtable.h`
    - Contains get/put/scan APIs for in-memory buffer
- `StorageType` in `include/storageType.h`
    - An abstract class that contains get/scan APIs for storage
    - Contains method Flush that takes the data in the memory buffer and writes them to file.
    - This class manages bufferpool and bloom filters.
    - By using this abstract class, it is easier to switch between the naive storage method in Step 1 and the LSM-Tree implementation for experiments
- `NaiveStorage` in `include/naiveStorage.h`
    - Extend class `StorageType`
    - Implement the naive SST storage method in Step 1.
    - Contains the bufferpool (class `ExtendibleHashTable`), the eviction algorithm (class `EvictionAlgo`), and a list of SST files represented as class `sstable`
- `LSMTree` in `include/lsm_tree.h`
    - Extend class `NaiveStorage` since Get/Scan API are the same with `NaiveStorage`
    - Override the Put API to support LSM tree compaction
    - Implements the LSM tree storage in Step 3
- `sstable` in `include/sstable.h`
    - Represent a single SST file in storage. Contains get/put/scan APIs for one file
    - Supports both B-Tree indexing and binary search
    - Stores the KV pair data, bloom filter, and b-tree internal nodes
- `ExtendibleHashTable` in `include/extendibleHash.h`
    - Implement the bufferpool extendible hash table in Step 2
- `EvictionAlgo` in `include/evictionAlgo.h`
    - An abstract class that contains APIs for bufferpool eviction algorithm
    - This class makes it easier to switch between Clock and LRU algorithm for experiments
    - Clock and LRU algorithm are implemented as class `ClockEviction` and `SimpleLRU` in file `include/clockEviction.h` and `include/simpleLRU.h`, respectively

- `BPTree` in `include/BPTree.h`
    - Support B-tree creation, B-tree search and read/write B-tree binary file.
    - Store B-tree structure inside class.
- `BloomFilter` in `include/bloom_filter.h`
    - Implement functions for bloom filter query and creation

SST file naming convention in database:
- SSTs are named with integers representing their recency.
- If `StorageType` is `NaiveStorage`, higher number indicates more recent; if `StorageType` is `LSMTree`, lower number indicates more recent, as well as its level in LSM-Tree.

Design Elements

KV-store get API (1) - 1 points
- Implemented as function`Database::Get()` in file `src/database.cpp`
- First call Get API of memtable; if key not found, call Get API of storage

KV-store put API (1) - 1 point
- Implemented as function `Database::Put()` in file `src/database.cpp`
- First call Put API of memtable. If memtable is full, call Flush API of storage to store the data and then clear the memtable

KV-store scan API (1) - 2 point
- Implemented as function `Database::Scan()` in file `src/database.cpp`
- Call Scan API of both memtable and storage, then combine their results

In-memory memtable as balanced binary tree (1) - 4 points
- Implemented as class `Memtable` in file `src/memtable.cpp`
- Maintains an AVL-tree structure.
- Supports `Put`, `Get`, `Scan` and `Clear` method
    - Put: take a key-value pair and insert them into memtable.
    - Get: get the corresponding value for the given key
    - Scan: given a key range, traverse the AVL-tree inside memtable and return all the key-value pairs where key is between the given key range.

SSTs in storage with efficient binary search (1) - 3 points
- Implemented as private function `binary_search_page_of_key` in file `src/sstable.cpp`
- The function returns the page number of the storage page that may contain the query key. The storage then puts this page in the bufferpool and looks for the KV-pair in this page.

Database open and close API (1) - 2 points
- Implemented as function `Database::Open()`and `Database::Close()` in file `src/database.cpp`
- Open API will configure the database with the given configs (e.g. bloom filter bits, B-tree or binary search, etc.), which makes the experiment more extendible.
- Close API will flush all data in the memory buffer to the file system.

Extendible hash buffer pool (2) - 6 points

- Implemented as class `ExtendibleHashTable` in file `src/extendibleHash.cpp`
- Uses chaining for each bucket in the directory. The bufferpool expands and shrinks automatically such that each bucket in directory contains 1 page on average
- For each page, in addition to the data, we also store the type of the data and the file where it comes from.
    - The type of data is one of KV pair, B-tree, and Bloom filter. It is used to identify different data when querying the bufferpool
    - The source file where the data comes from is used when the original file is modified (due to compaction), we can identify these dirty pages and clear them at once.

Integration buffer with get (2) - 2 points

- Implemented in function `NaiveStorage::Get()` in file `src/naiveStorage.cpp`
- For each get, we will first look for the keys in bufferpool; if found, return it; if not, we will call Get for each SST file and put the page in bufferpool if found.
    - The same procedure is done for bloom filter and b-tree. The only difference is that for bloom filter and B-tree, we will put them in bufferpool regardless of whether the key exists in that SST, but for KV-pair, we will only store the page in bufferpool if the key actually exists.

shrink API (2) - 2 point

- Implemented as private function `ExtendibleHashTable::ShrinkDir()` in file `src/extendibleHash.cpp`
- Shrinking of directory is done automatically depending on the number of pages in the bufferpool

Clock eviction policy (2) - 4 points

- Implemented as function `ClockEvction::accessAndUpdate()` in file `src/clockEviction.cpp`
    - Given a newly access page number, return evicted page number, if no eviction, return -1
- Implemented unit test for the algorithm in `clock_test.cpp`

LRU eviction policy (2) - 4 points

- Implemented as function `SimpleLRU::accessAndUpdate()` in file `src/SimpleLRU.cpp`
    - Given a newly access page number, return evicted page number, if no eviction, return -1
- Implemented unit test for the algorithm in `lru_test.cpp`

Static B-tree for SSTs (2) - 4 points

- Implemented as class `BPTree` in file `src/BPTree.cpp`
- Has a supporting child class `file_node` working as the node of the B+tree structure.
- Has 2 public methods:
    - writeBtreeToFile: takes an array of keys, creates a B+tree based on the keys and writes them to Binary files. In the first section, the function computes the levels of the B+tree and number of nodes for each level. Then the function creates a twoD array to store those nodes. In the second half, the function loop through each

level of the twoD array, and links each child node to its corresponding parent and updates the node attributes.
Finally, nodes in the twoD array are written to a binary file.

- searchBTree: takes a key and searches for the value in the file. The function reads the data from the file and stores them to an array. Then the function loop through the array and do B+tree search. The function takes a `isScan` keyword. If isScan is true, the function returns the matching index of the data for that key, or (index + 1) if the key is not found in the keys of the leaf nodes.

Bloom filter for SST and integration with get (3) - 5 points
- Implemented in function `NaiveStorage::Get()` in file `src/naiveStorage.cpp`
- Function related to bloom filter is implemented in file `include/bloom_filter.h`

Compaction/Merge of two trees (3) - 5 points
- Implemented as private function `LSMTree::Compaction()` in file `src/lsm_tree.cpp`
- To compact two SSTs, we will use only their KV-pair data. After a new SST is created from compaction, we will re-construct the bloom filter and B-tree with the new data.

Support update (3) - 3 points
- Update can be done by just calling `Database::Put(<key>, <new_value>)`

Support delete (3) - 4 points
- Implemented as function `Database::Delete()` in `src/database.cpp`
- Equivalent to calling `Database::Put(<key>, TOMBSTONE)`

Evict dirty pages from bufferpool (bonus implementation)
- Whenever a compaction occurs, we will evict all the pages in the bufferpool of the SST that is modified to clear up space since those pages are no longer valid.
- This is implemented by recording the source file of each bufferpool page, so that we can identify the dirty pages during compaction.
- We also need to add an `evict()` function to `EvictionAlgo` class to explicitly tell the algorithm to evict a specific page in order to make the algorithm consistent with the content in bufferpool.
- This is an improvement of the design since the dirty pages are no longer valid but take up space. Even though we will ultimately clear them from the bufferpool, we could evict them immediately such that other valid pages will not be kicked out.

Project Status
- The project is completed as we have implemented every required element, though there are some optimizations we could implement if we have more time.
- One optimization is related to the bufferpool implementation. Currently the bufferpool capacity is limited based on the number of pages since in the beginning we did not consider the case of supporting pages of variable sizes, so a potential optimization we could have made is to extend the current implementation to support limiting bufferpool capacity based on bytes. Because of this, in experiment 3, in order to create a bufferpool

of size 10MB, we assume that on average the size of each page is 4KB, and we limit the number of pages in bufferpool based on this assumption.

- Another optimization is related to bloom filter memory consumption. Currently, after each compaction in LSM-Tree we will re-construct the bloom filter in memory first and then flush it to storage. However, since the size of bloom filter is proportional to the number of entries in SST, we could run out of memory as the data volume increases. This problem occurs when we try to run experiments on teach.cs; the program crashes as we try to insert 1GB of data. Therefore, in the following experiments, we reduce the data volume correspondingly to circumvent this memory issue. We could try to find ways to optimize the memory consumption of bloom filter.
- One known bug of our project is that when running our unittest (see Tests section below for detail), when the bufferpool is enabled, there is a chance for the test to fail, but most of the time it will pass.

Experiments

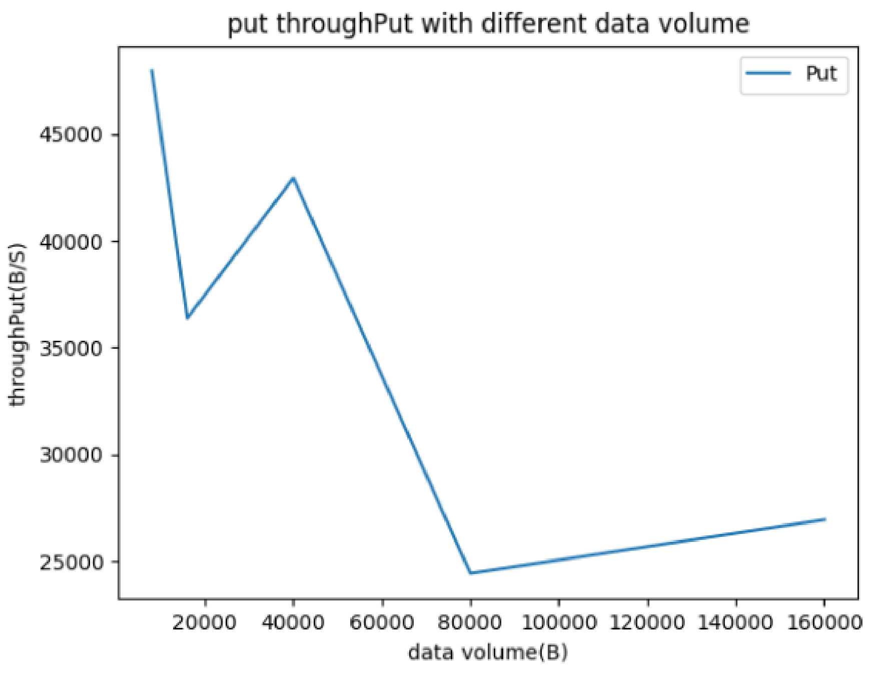
Step 1 Experiments

Experiment Setup:

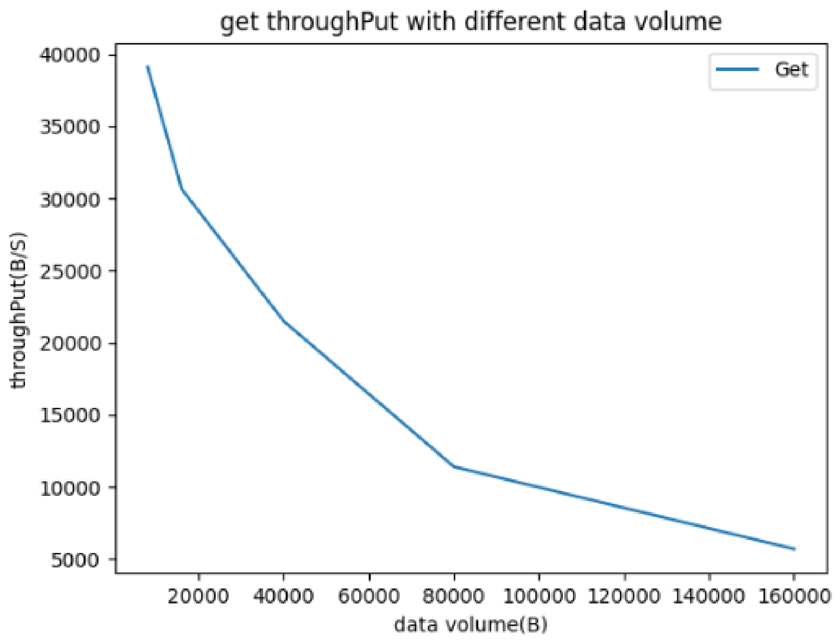
For this experiment, we first insert the corresponding volume and then measure the throughput by performing 100 put/get/scan operations on the database with these volumes. For each volume we repeat the experiments five times and take the average

Figures:

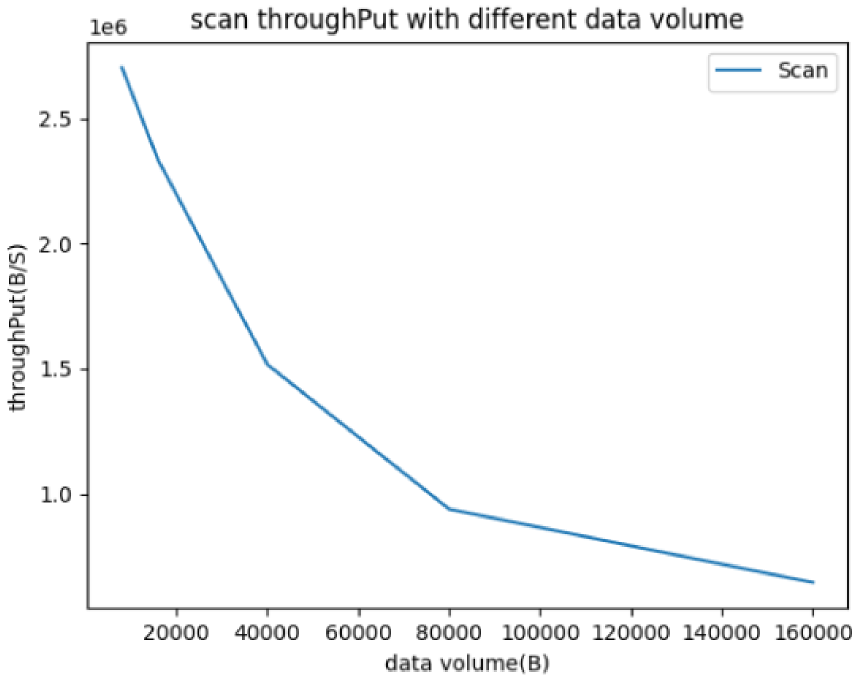
- Data volume VS put performance:



- Data volume VS get performance:



- Data Volume VS scan performance:



<u>Findings:</u>

Scan and get throughput decreases as data volume increases because we need to iterate through more SSTs as we perform get and scan operation.

Put throughput also decreases as data volume increases. But there is an increase in throughput when data size is between 20000-40000.

Step 2 Experiments: Binary Search VS B-Tree Indexing

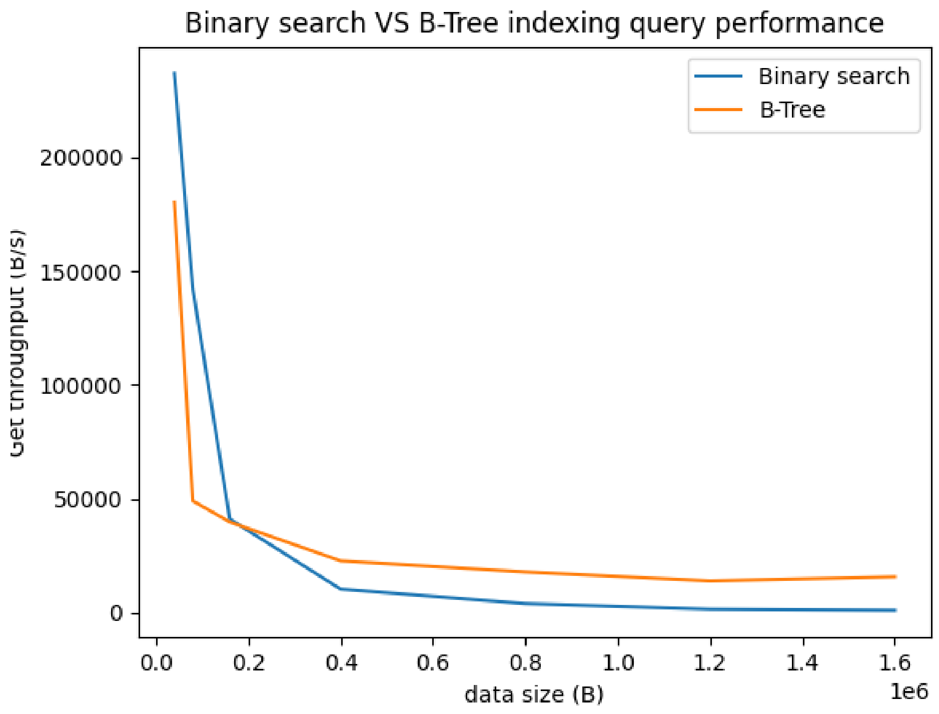
<u>Experiment Setup:</u>

We tested the performance of SST queries on 7 different total numbers of entries: 5000,10000,20000,50000,100000,150000,200000. The memtable capacity is 5000 entries; bufferpool is enabled with 100 pages capacity and Clock eviction algorithm.

To test the Get performance of B-Tree and Binary Search, we first uniformly generate 1000 random keys that already exist in the database such that the random generation time does not interfere with the throughput measurement. Then, we initialize two different databases and measure the time to get all the keys.

<u>Figures:</u>

- Binary search vs B-tree index with query throughput with changing data size



<u>Findings:</u>

The throughput for both B-Tree and Binary search decreases as data size increases since they need to iterate through more SSTs. The performance of B-trees generally surpasses that of binary search trees. This can be attributed to the B-tree structure being stored in the buffer pool, which results in constant I/O time complexity. Additionally, the node size in a B-tree is roughly equal to the size of a page, leading to a higher branching factor. B-trees also maintain a balanced structure, ensuring that search queries exhibit approximately constant CPU time complexity.

Step 2 Experiments: LRU VS Clock

<u>Experiment Setup:</u>

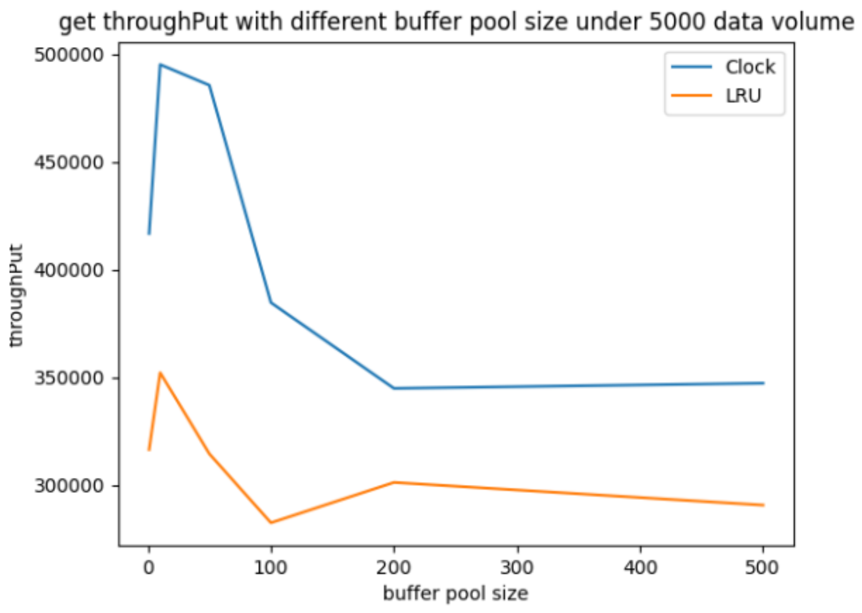
For this experiment, we insert a constant volume of data into two database, one with Clock eviction algorithm and one with LRU eviction algorithm. We measure the throughput by performing get operations 5000 times on the database. We repeat the experiment five times and take the average.

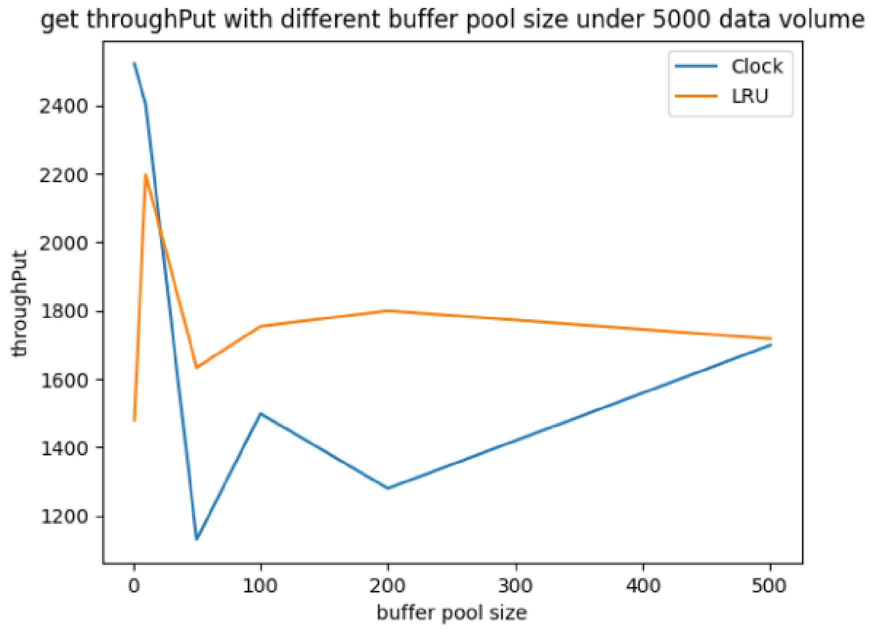
In the first figure, we create a naive workload where the database constantly accesses the same key, by doing this the page hit rates can be maximized for both Clock and LRU algorithm, and Clock will perform better than LRU due to CPU and memory overhead in LRU.

In the second figure, we create a skewed workload where LRU generally performs better. We first generate a set of random keys that are all powers of 2 by left-shifting 1 with a uniformly sampled number of bits. As an example, 2^4, 2^5, 2^7, and 2^8 all have the same probability of being queried. However, for small powers of 2, such as 1, 2, 4, their differences are small, so they are most likely to appear on the same page, while for large powers of 2 like 1024, 2048, 4096, they are likely stored on different pages. Therefore, we will have a higher probability of hitting a page that stores small keys than those storing large keys, hence creating a skewed workload.

<u>Figure:</u>

- LRU vs Clock eviction policy with query throughput with different buffer pool size





Findings:

The data structure used in LRU is much more complicated than Clock (queue and map vs bitmap), so Clock eviction takes less memory and CPU than LRU eviction. In the first figure above, we tried to maximize the bufferpool hit rate by repeatedly accessing the same key, so the Clock eviction always performs better than the LRU eviction. In the second figure, we tried skewed workload so that the overhead of CPU/memory cost in LRU can be neglected, so we can see that as the bufferpool size increases the LRU eviction performs better than Clock eviction.

Step 3 Experiments: LSM-Tree Throughput

Experiment Setup:

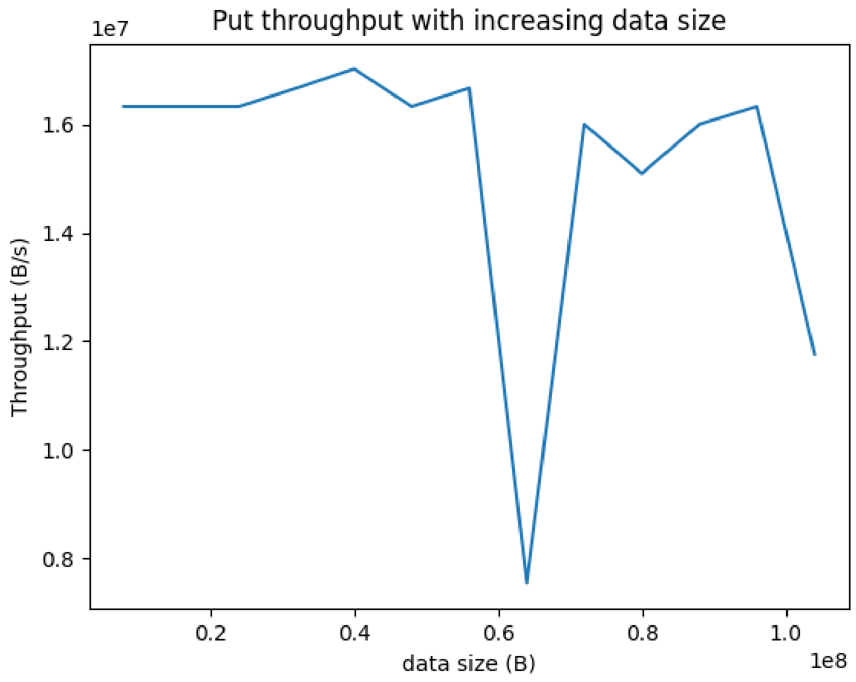
We set up the experiment as required in the course project handout. The bufferpool eviction algorithm is Clock and we use binary search for queries in SST. We measure the put/get/scan throughput each time after we inserted 1 million entries. For each throughput measurement, we put/get/scan 100 entries and calculate the time.

For Put, we insert the same keys each time such that the logical space of the database does not increase. For Get and Scan, we query for random keys and range since we want to ensure that the operation does not always just query the memtable.

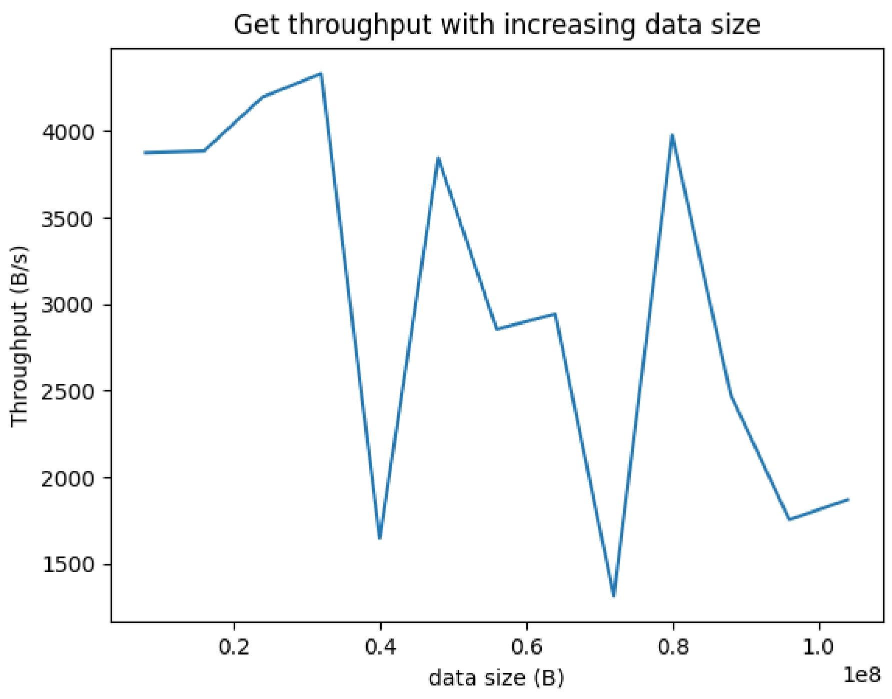
NOTE: Due to the memory and storage limitation on teach.cs, we use 100MB data instead of 1GB, but we believe that the findings should be similar.

Figures:

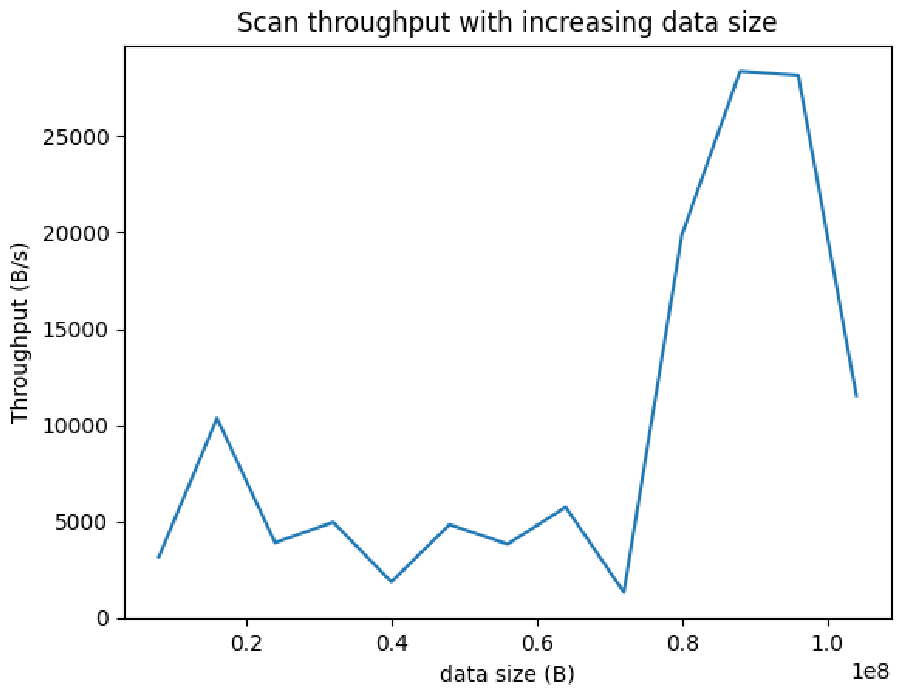
- Put throughput with increasing data size



- Get throughput with increasing data size



- Scan throughput with increasing data size



Findings:

We can see that the Put throughput is significantly higher than that of Get and Scan, which is expected since LSM-Tree is a write-optimized data structure. Scan has higher overall throughput than Get, which is expected since large range read is generally better than random read.

The Put throughput is roughly constant over time except for a few point. This is expected since we are always putting entries in memtable. The only major factor of Put performance is the cost of compaction, which only occurs occasionally. The few points where the performance declines is probably due to compaction of SSTs.

The Get throughput gradually decreases as the data size increases. This is expected since as the data size increases, the LSM-Tree becomes deeper, and the deeper the level is, the more entries there are. Since we are using binary search for this experiment, the time complexity for Get is O(<LSM level> * <binary search cost of each element>). Therefore, the performance declines as the data size increases.

The Scan throughput is roughly constant and there is a sudden increase in performance at the end. This is not quite what we expected but it is also reasonable. The Scan time complexity is O(<LSM level> * <binary search cost of start&end key> + <cost of range read>). Here, the cost is similar to but smaller than Get, since we are only searching for 2 elements and the range read cost is smaller than random read, so it is expected that Scan throughput does not decrease as much as Get as we increase the data size. As for the performance improvement at the end, one potential explanation is that at that point, there is only one SST file in LSM-Tree at the deepest level, so Scan cost is basically the cost of finding two elements and the cost of range read, which is significantly small.

Step 3 Experiment: Bloom Filter

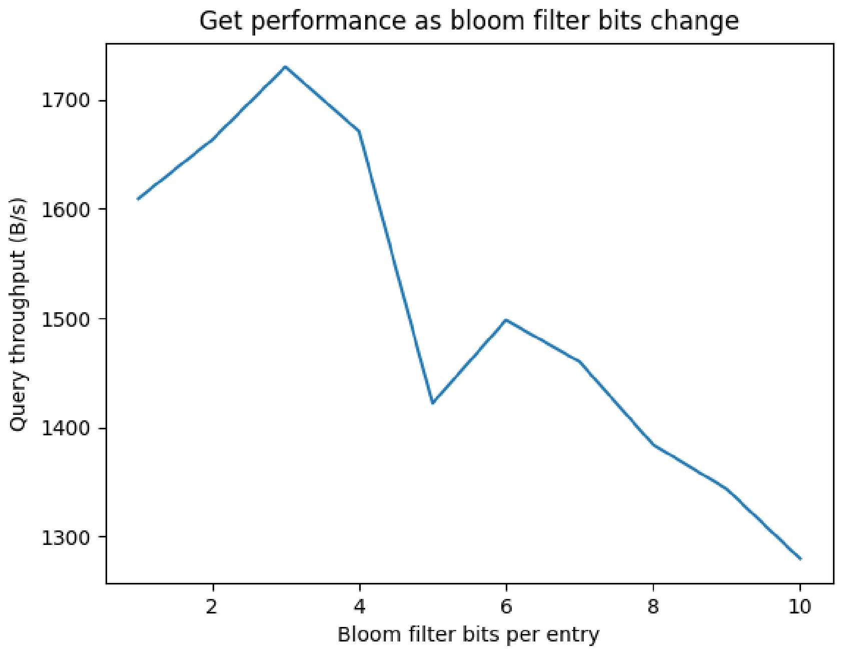
Experiment Setup:

We used the same setup as the experiment above, but tested on different bloom filter bits from 1 to 10. For each bloom filter bit, we first inserted all the data, and then we generate 100 random keys existing in the database. Lastly, we perform Get operation on these keys and measure the time.

NOTE: Due to memory and storage limits on teach.cs, for each bloom filter bit, we only inserted 50MB data instead of 1GB.

Figures:

- Get performance for changing bloom filter bits with growing data size



Findings:

As we can see, the query throughput first increases slightly as we have more bits, but then it starts to decline as the number of bits increases. This is not quite what we expected since theoretically, the false positive rate (FPR) decreases with more bits per entry, so we should have higher query throughput. However, as we have more bits, we need to have more hash functions (since optimally K=ln2*M), so we will have more memory access per query, which increases the CPU cost. Therefore, a reasonable explanation for the plot above is that the CPU costs overwhelm the performance improvement we obtained from low FPR as the bloom filter bits increases.

Tests

- Implemented in `test.cpp`
- We first insert continuous KV pair data where key and value are the same from 0-4095 into the database. Then, we randomly select a set of keys to query; we also randomly select a subset from the query keys to delete from the database. Afterwards, we call Get on the selected query keys and check whether the deleted keys have value of TOMBSTONE. We also select a random range to test Scan.
- To run unittest of the database, run `make unittest`, then `python3 test.py`. All other class-specific unittest are in `test/` folder.

Running Instruction:

All experiments are tested on teach.cs

1. Run `make` to generate all executable files
2. Run `python3 step_1_experiment.py`, `python3 step_2_experiment.py` and `python3 step_3_experiment.py`to run experiments and generates graphs

a. `python3 step_1_experiment.py` generates 3 graph: put.png, get.png, and scan.png

b. `python3 step_2_experiment.py` generates 3 graph: `lru_vs_clock_workload_0.png` where clock performs better, `lru_vs_clock_workload_1.png` where lru performs better, and `binary_search_vs_b_tree.png`

c. `python3 step_3_experiment.py` generates 4 graph: put_experiment_step_3.png, get_experiment_step_3.png, scan_experiment_step_3.png, and bloom_filter_expieriment_step_3.png

Notice: If running on teach.cs, sometimes there will be resource busy error because we are running on a large dataset