

CSC443 Mini Database Report

By Jessica Boritz, Yichen Cai, Rebecca Nhan

Step 1 - Creating an AVL Tree Memtable with SSTs

Design Choices for Step 1

For our memtable, we used an AVL Tree.

INT_MIN was chosen as a reserved value to represent "key not found".

Because `scan()` needs to return all the key-value pairs in range, `scan()` returns a pointer to a `kv_pairs` struct, which contains a 2d array 'kv' containing all the matching key-value pairs and an int `kv_size` which represents the number of pairs in `kv`. When a user no longer needs the results of their scan, they can free this `kv_pairs` struct by calling `freeKVpairs`.

SST Implementation

For our SSTs, we used `.bin` files with unique names, each containing a sorted array of key-value pairs (sorted by key). Unique names are `sst<num>.bin`, with `sst_<sst_counter>.bin` being the newest SST, and `sst0.bin` being the oldest SST.

Any time that the memtable is filled up, or the database is closed, the memtable is transformed into an SST with the `transform_to_sst` method by scanning the memtable and then putting the resulting sorted array into the SST file.

Step 2 - Implementing a Buffer and converting SSTs to B-Trees

Design Choices for Step 2

For hashing pages in our buffer, we gave each page a numeric `pageID`.

Pages in our buffer are identified by SST number and the offset into the SST by the formula

$$\text{PageID} = (\text{SST_Number} * \text{memtable_size}) + (\text{offset} / \text{Page Size})$$

This formula allocates pages proportional to the memtable size to each SST, which is a high upper bound on the number of pages each SST will actually use.

A tighter bound could be determined,

The eviction policy we chose to use was LRU.

Bonus: Expandable Buffer Pool

An extendible directory is implemented for the buffer pool to resize the buffer as needed. The expandable directory is defined in `initialDepth` and `maxDepth`, which specifies the initial and max number of bits in the directory. The user can also resize the buffer at any time by calling `setMaxDirectoryDepth`, which sets the max number of bits in the directory. We use the first `Depth` bits of the hashed `pageID` (defined above) to decide which bucket this page should go in. Similarly, when reading we calculate the `pageID` based on the offset to try to retrieve the page. Each bucket can be chained with 5 pages. When expanding the buffer pool, the directory is doubled by adding one more bit (`Depth+1`). And then all the existing pages are rehashed into the new bucket taking one more bit from their hashed result. The expansion happens when the buffer pool has reached its capacity (and chaining limit), i.e. the new page cannot be placed into the buffer without being placed into a bucket that has the max amount of chaining. When the buffer is already at the `maxDepth`, it will evict pages following the eviction policy. Similarly, when the user manually decreases the `maxDepth`, the eviction policy will kick in and evict pages until the size of the directory can be contained within the new `maxDepth`.

SST B-Tree Implementation

In converting our SST to a BTree, we restructure to SST files to contain the internal nodes and leaves of the BTree.

Each page contains one internal node of the BTree, which contains the max key and links to the page number of at most `B` children. After the internal nodes, we have pages for the leaves of the BTree (the key-value pairs in our database). Each leaf page contains at most `B` key-value pairs. This is done in our `transform_to_sst` function, when we transform our memtable to an SST whenever it fills up, or the database closes. In this, calculations are done in order to find the number of layers we need, the number of internal nodes we need, and the number of leaf pages we need in order to construct the BTree and write it to the SST page by page. This calculation is also repeated when retrieving the data in a `get` or `scan`.

`B` was chosen to be 512, as with a page size of 4KB and an int size of 4B, we can fit exactly 512 key-value pairs in one page.

As the SST sizes are variant, we have an additional file, `sizes.bin`, where the `i`th file represents the size of the `i`th SST.

Bonus: Sequential Flooding

Sequential flooding occurs when the same sequence of queries of multiple pages longer than the buffer is brought to the buffer pool multiple times in a row. To prevent this but also not adding additional overhead to the queries, we added a field in the database called `buffer_args`. The purpose of this field is to save the previous parameters for retrieval from the database.

Sequential flooding is most likely to happen when the same data is retrieved over and over again for a specific purpose. This sequence of queries will likely be issued under the same parameters, say `scan(1, 10000000)`. Every time a get/scan query is issued to the database, we check to see if the last query is exactly the same as the current one. If it is, we do not write to the buffer pool regardless of whether we have read new pages from the disk. This way, for queries smaller than the buffer pool size the buffer pool will function as usual, as the first read will bring the page to the buffer (without flooding), and later repeated ones can read all the pages from the buffer. And for large queries (larger than buffer pool size), it will buffer the pages of these queries for the first time, and after that repeated queries will only read from the buffer but never write new pages to it, preventing sequential flooding.

Step 3 - Converting DB to a LSM Tree and adding Bloom Filters

Design Choices for Step 3

In adding support for deletes, we needed a way to denote tombstones. Thus, we chose INT_MAX as being the reserved value that indicated a tombstone. Users are blocked from entering INT_MIN and INT_MAX as keys or values into the DB, and will be presented with an error message if they attempt to do so.

Because delete is a reserved keyword in C++, the delete function is instead named `remove`.

The `update` function is simply a wrapper that calls `put`. In this way, updates can also be performed without the use of the `update` function. In supporting updates and deletes, the DB now requires each key to be unique, with duplicate entries for one key being removed upon compaction and filtered out during scans. The entry returned will always be the most recently inserted version.

LSM Tree Implementation

We use a basic LSM Tree with a fixed size ratio of 2. In converting our SSTs into an LSM tree, we retain our BTree structure for the SSTs, but change the way insertion happens - whenever there are two SSTs on the same level, we merge them.

Now, `sst0.bin` is the first layer of the LSM tree, and also the youngest SST.

`sst_<sst_counter>.bin` is the last layer of the LSM tree, and also the oldest SST.

Now, when we call `transform_to_sst` to insert new data into our LSM Tree, we first insert to the youngest layer, `sst0.bin`. If there is already an SST on that layer, we sort-merge it with the new data to `sst1.bin`. If there is already an SST on layer 1, we sort merge it with the sort-merged data from `sst0` to `sst2.bin`. We continue doing this until we sort-merge to a layer that does not already have an SST there.

Our compaction in `transform_to_sst` when sort-merging two SSTs utilizes three buffers - two input buffers and an output buffer as directed in the handout. A temp file (`temp1.bin`) is used to store the sort-merged SST if the next layer already has an SST, before merging it with the next layer.

To support our new LSM structure, the formula for allocating pages to each SST has now changed. Instead of giving each SST pageIDs proportional to the memtable size, we now give each SST pageIDs proportional to $2^{\text{sst_number}} * \text{memtable size}$.

Bonus: Bloom Filter Implementation with Monkey

The bloom filter was designed with Monkey in mind. We have implemented a function that calculates the number of bits required for each level following the Monkey model. Following the bits per entry set by the user, we calculate the bits needed for any given level in the LSM tree.

The number of entries of the LSM tree at each level can be calculated by `num_entry=pow(2, level) * (mem_table_size)`, then the last level will have `num_entry * (bits_per_entry - 1)`, and other levels will have `num_entry * (bits_per_entry + (lsm_max_level - 1 - level))`. With the size of the bloom filter and the number of bits per entry at each level, we could easily calculate the optimal number of hash functions at each level. With that we could create an array of bits representing the bloom filters. Each level, there's a bloom filter file corresponding to the main data file. The bloom filter file is stored in a binary format with compacted data (the bits are grouped into bytes and written to the file). The reading of the file also requires decoding of the compacted file. This is to ensure space efficiency of the filters.

Project Status

The database supports all the required functionalities with bonus parts as described above. It is able to put, get and scan data, open and close the database.

To build and run our set of test cases included in `test.cpp`, build via `make tests` and run via `./tests`. To include our DB in another program, you must include `main.cpp` and `buffer.cpp`. The test suite includes 15 individual tests for different functions of the database.

1. Test simple put and get of the DB, making sure what you put is what you get.
2. Test the scan function that returns the whole DB.
3. Test scans that only return part of the DB.
4. Test the transformation to SSTs when memtable is full. Retrieve the value back from SSTs. SSTs are organized using binary trees.
5. Test the transformation to SSTs with a B-Tree setup in the SSTs.
6. Test get function with invalid keys.
7. Test large scans that include memtable and SSTs, where SSTs are organized using binary trees.
8. Test large scans but the SSTs are using B-Trees.
9. Test the opening and closing of the DB, to see if the data persist and the memtable are correctly stored to storage. SSTs are implemented with binary trees.
10. Test the same opening and close of the DB, but with B-Tree SSTs.
11. Test updating the value in the LSM-Tree.
12. Test removing a kv pair and using get to check if it is properly removed.
13. Test removing a kv pair and using scan to check if it is properly removed.
14. Test bloom filters, and check its FP rate.
15. Test the general case of creating, populating, getting and scanning a large database where the SSTs have undergone compaction

The buffer pool implementation is separately tested in `test-buffer.cpp`, which can be built by calling `make test-buffer`. It tests the insertion, reading and eviction of the buffer. Due to the nature of the problem, the buffer pool is examined through dumping the buffer pool content with the `printBuffer` function.

Execution Guide

The database can be run with a set of APIs.

To create a Database:

```
DB* Open(std::string name, int max_size, int initial_directory_size,
int max_directory_size, int search_alg, int Bloomfilter_num_bits);
```

It takes the name of the database, the max memtable size (in terms of the number of KV pairs in the memtable), the initial and max directory size for the buffer pool, search algorithm of binary search or B-Tree (0 or 1 respectively) and the number of bits in the bloom filter. If the name corresponds to an existing database, it will be loaded.

Once the pointer to the DB is obtained, its various DB class methods (operations) can be called:

```
int put(int key, int value);
int get(int key);
int update(int key, int value);
int remove(int key);
struct kv_pairs* scan(int key1, int key2);
```

For `put`, `update` and `remove`, the return value is 0 on success. For `get`, the return value is the corresponding value of the key. If the key does not exist, `INT_MIN` will be returned. The `scan` function searches for a range of keys and allocates memory on the heap for the return. The return value is a pointer to a `kv_pair` struct defined as:

```
struct kv_pairs {
    int** kv;          // Array of [key, value] pairs
    int kv_size;       // Size of kv
};
```

The `kv` is a pointer to an array of arrays, with each subarray containing `[key, value]`. And `kv_size` is the size of the array above, in other words, the length of the returned `kv_pairs`.

Since the `kv_pairs` struct is allocated on the heap, we also included a DB method to easily free all the memory allocated for a `kv_pairs` struct. Simply call:

```
void freeKVpairs(kv_pairs* p);
```

At any point, users can specify the number of bits they wish to use in the Buffer Pool directory. Expanding the upper bound or shrinking the directory by calling:

```
void set_max_dir_size(int size);
```

Lastly, to close the DB and move any temporary in-memory data to storage:

```
int Close();
```

Note that the `Close` function is a method of the DB class. While `Open` is a function that can be called on its own. An example use case of all the above can be found in the README.md file in the repository.

Experimental Results

Experimental Setup

To perform experiments, three experiment programs were created, one for each step in the project. To perform experiments for a step, please first select the branch corresponding to said step in Github, then follow the instructions below. All experiments were run remotely on the CS teaching labs via SSH, which could have an impact on the performance as pointed out by many teams.

For step 1, the experiments are in `exp1.cpp`, which can be built via `make exp1` and run via `./exp1`. This program contains three tests: put, get, and scan.

For step 2, the experiments are in `exp2.cpp`, which can be built via `make exp2` and run via `./exp2`. This program contains five tests: put, get_binary, get_BTree, scan_binary, and scan_BTree.

For step 3, the experiments are in `exp3.cpp`, which can be built via `make exp3` and run via `./exp3`. This program contains five tests: put, get_binary, get_BTree, scan_binary, and scan_BTree.

In each experiment program, the user can select to perform either one of the available tests or to run all of the tests in-order. Results for each test will be output to a CSV labelled by the step it corresponds to and the test run.

In each test, the performance of the specified function is recorded as the data size in the DB grows, starting with 1MB and doubling each time until a final test with 1GB of data. To derive the number of entries, note that each entry in the DB takes 8 bytes of space (4B for key, 4B for value).

For the put test, we record the time taken to insert data, one key-value pair at a time, into a newly-created DB, until the DB contains data elements equivalent to the specified data size. The throughput can then be calculated as the size of the data inserted divided by the time taken. This will provide an amortized cost, accounting for the infrequent-but-large cost of flushing the memtable to disk, and in step 3, also accounting for the costs of performing compaction.

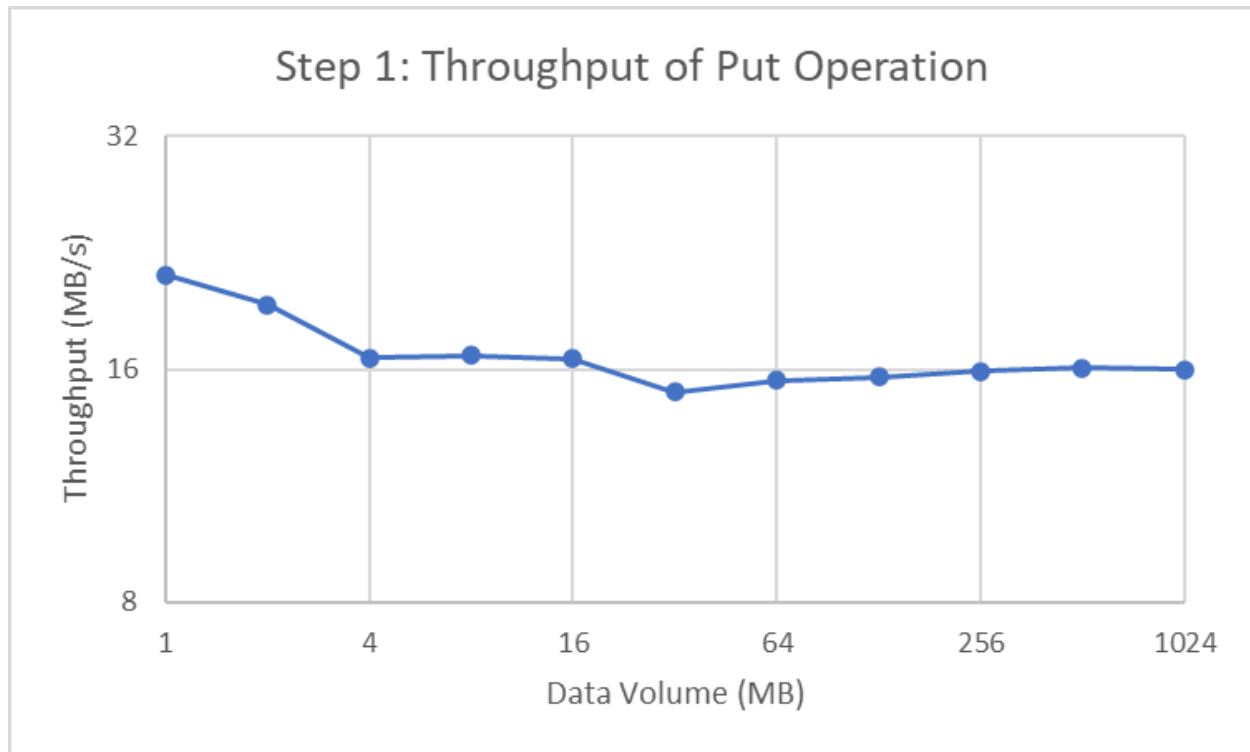
For the get and scan tests, we record the time taken to perform a fixed number of operations, regardless of the data size: 4096 gets, and 128 scans. Each scan contains 16 elements in-range. The throughput can then be calculated by the size of the data gotten / scanned divided by the time taken. Note that the get and scans are tested with uniformly spaced out valid keys. That limits the performance boost from buffer pool and bloom filters. The performance of the two are tested separately to control the variables.

Buffer pool tests are performed on the same database size (128MB). We tested repeated (sequence of) gets from the database using. In particular, 128 groups of gets are issued, within each group, 10 gets are issued within the same page. The first get will bring the page into the buffer and subsequent gets will only need to read the buffer pool. Then this entire 128 groups are read again.

Bloom filters are tested by getting non-existing keys from the database (64MB). The test is performed on getting existing keys at the deepest level, and getting non-existing keys. The idea is that the non-existing keys should be quickly filtered out by the bloom filter and thus speeding up the process.

Other than the data size and search algorithm, all parameters are kept constant between tests. Memtable size is fixed to 1MB, buffer size is fixed to 10MB, B is fixed to 512, and bloom filter bits are fixed to 5 per entry.

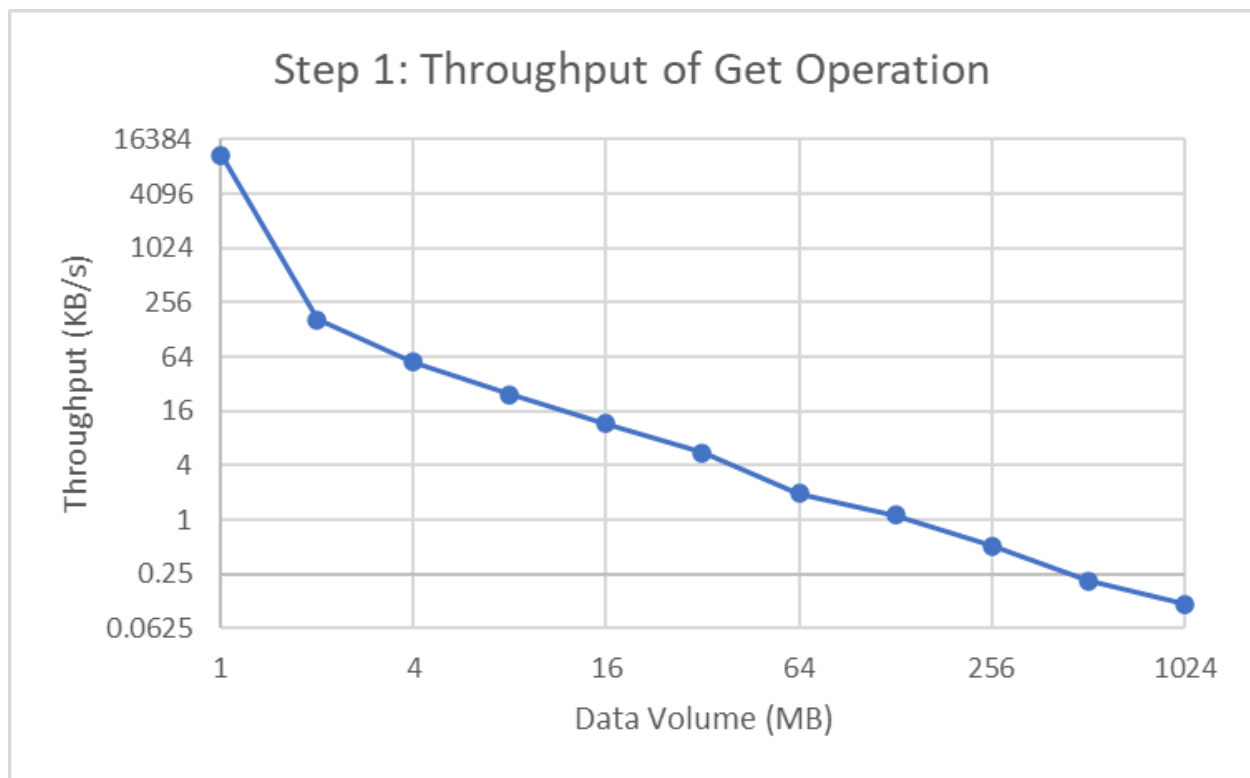
Figure 1: Throughput of Put Operation in Step 1



In step 1, the throughput of the put operation starts at ~20MB/s for a 1MB DB (when the entire DB fits into the memtable). As we insert more data and must begin flushing our memtable to the disk, the throughput quickly drops down to around ~16 MB/s, where it stays roughly constant regardless of the DB size.

This indicates that when amortized, the cost of flushing the memtable to disk is minimal.

Figure 2: Throughput of Get Operation in Step 1

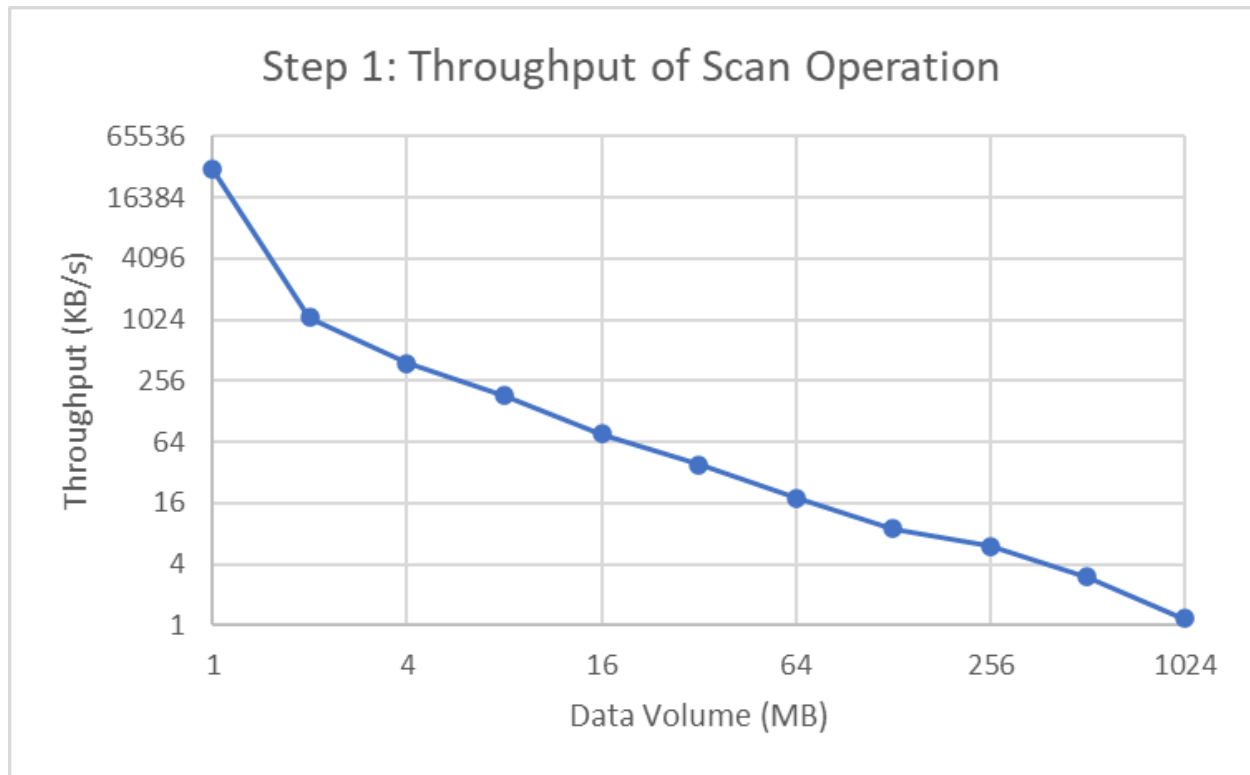


Note that unlike in experiment 1, experiment 2 has throughput measured in kilobytes per second, not megabytes per second.

In step 1, the throughput of the Get operation scales quite poorly with DB size, greatly decreasing as we go from 1MB to 2MB of data and then decreasing linearly as the data volume grows.

Intuitively, this result makes sense: our memtable is of size 1MB, and thus our SSTs are also of size 1MB. When there is only 1MB of data, we are able to search exclusively within the memtable, and have high performance. As the DB size grows, a higher proportion of the data is now stored in the SSTs, and the number of SSTs grows. Although binary search has logarithmic runtime, we must binary search each SST in-order, and the number of SSTs in step 1 grows linearly with data size.

Figure 3: Throughput of Scan Operation in Step 1

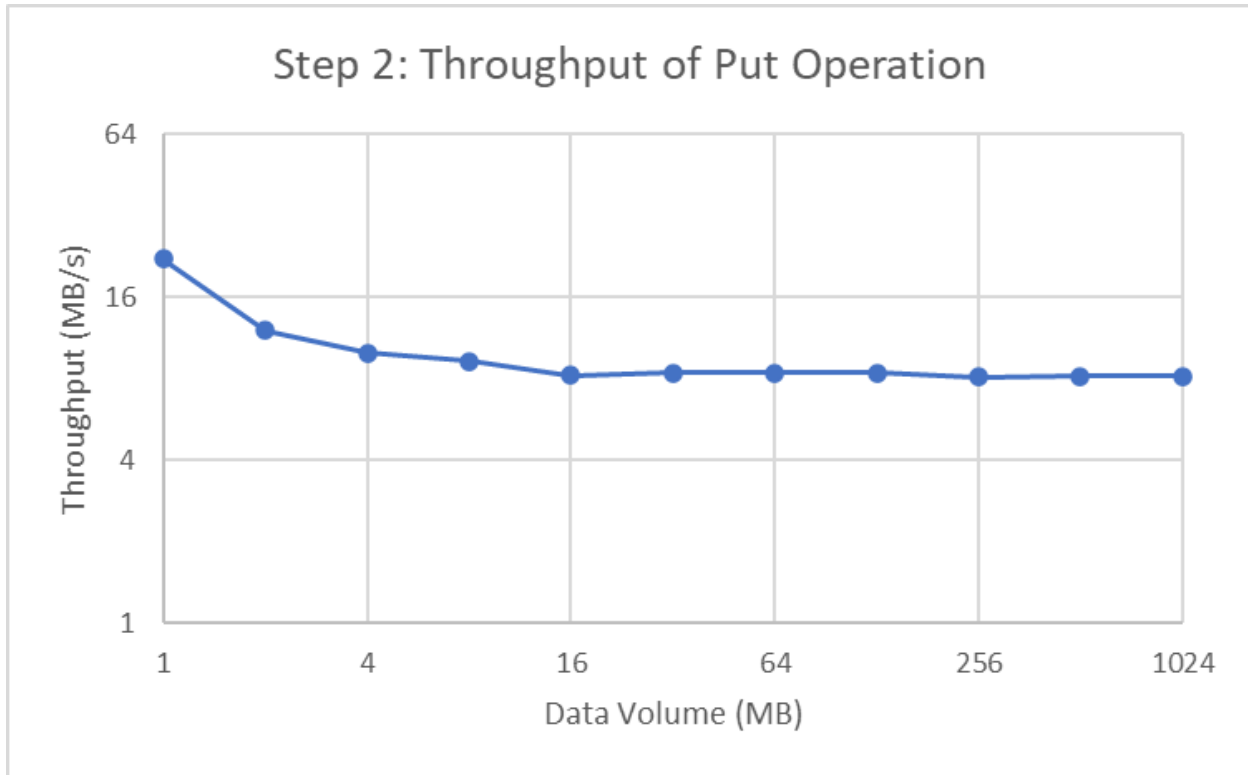


Like experiment 2, experiment 3 also has throughput measured in KB/s.

Similarly to in experiment 2, the throughput of the Scan operation scales poorly with DB size, also greatly decreasing as we go from 1MB to 2MB of data and then decreasing linearly as the data volume grows.

The intuition here is the same: although our scan operation has logarithmic runtime in relation to individual SST size, we must perform scans on each SST in the database, and the number of SSTs grows linearly.

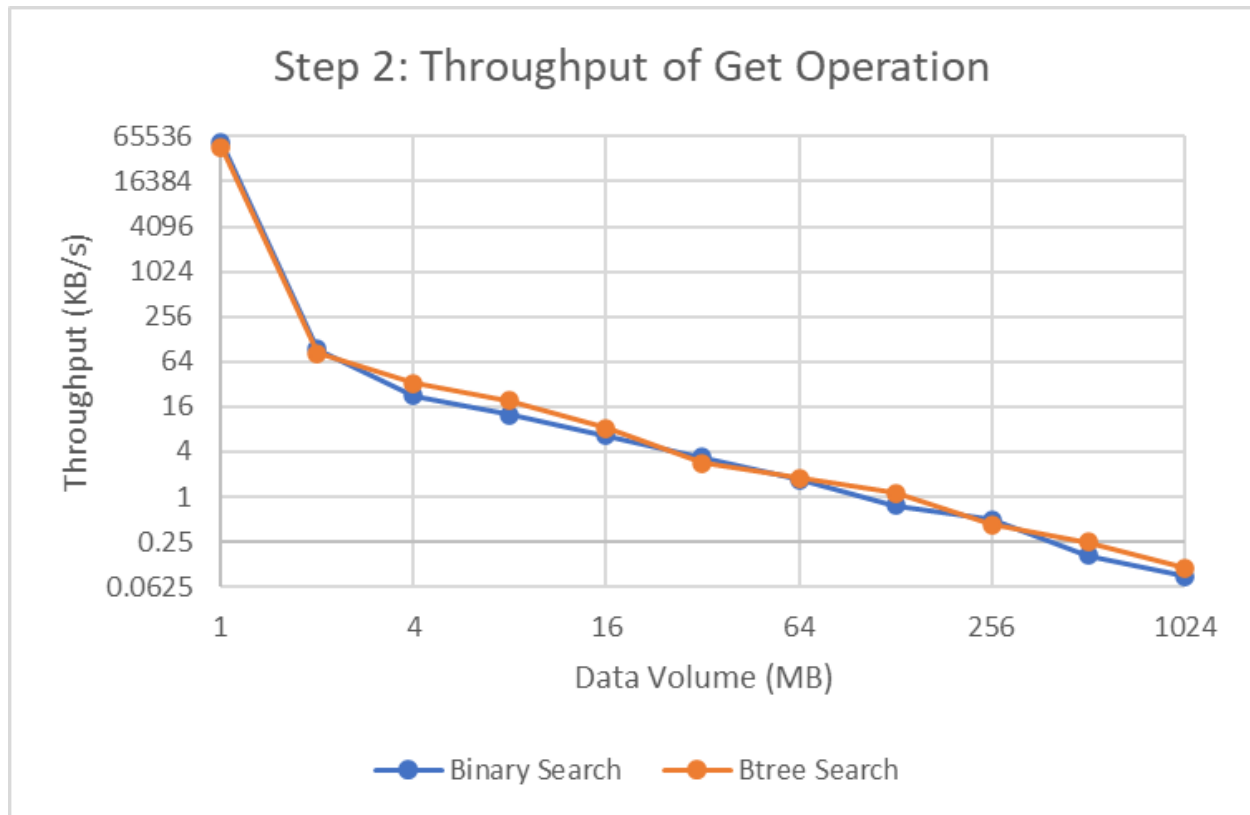
Figure 4: Throughput of Put Operation in Step 2



In step 2, the throughput for Puts with a DB size of 1MB starts at ~20MB/s similar to in step 1, but as we increase the DB size beyond the memtable size the throughput quickly drops down to around ~8MB/s and stays consistently at this level regardless of the data size.

This is a similar result as was obtained in step 1, but with a larger dropoff as we cease to be able to store the entire DB in the memtable. This indicates that our algorithm for writing B-Tree SSTs to disk in step 2 is slower than our algorithm for writing sorted array SSTs used in step 1.

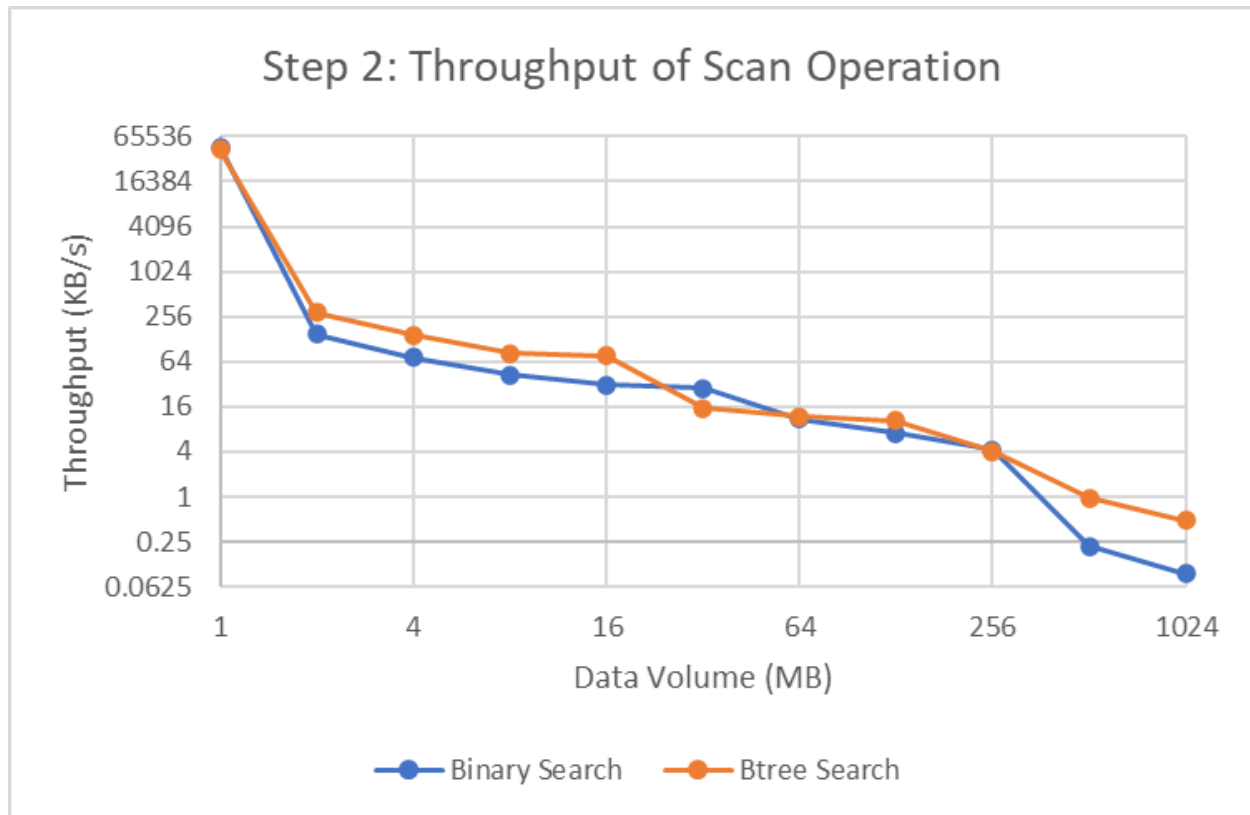
Figure 5: Throughput of Binary vs BTree Get Operation in Step 2



The throughput for gets in step 2 is similar to the throughput for gets in step 1, regardless of if we use Binary Search or B-Tree Search (however, B-Tree search performs slightly better on average): a sharp dropoff in runtime as we go from our DB being entirely in-memory to being partially on disk, and then a linear decrease as the DB size continues to grow.

The cause of this pattern is the same as in step 1: we must check each SST until we find the key we are looking for, and the number of SSTs grows linearly.

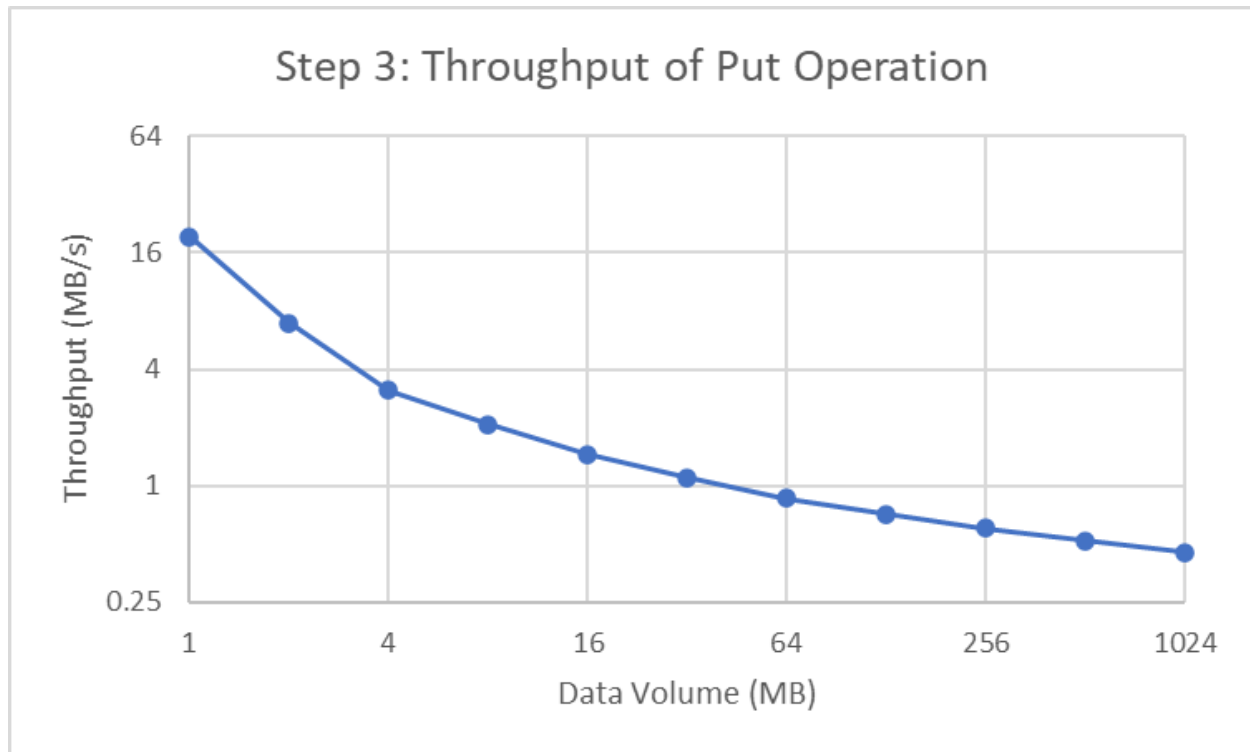
Figure 6: Throughput of Binary vs BTree Scan Operations in Step 2



The throughput for scans in step 2 is also similar to the throughput in step 1: a sharp dropoff in runtime as we go from our DB being entirely in-memory to being partially on disk, and then a linear decrease as the DB size continues to grow.

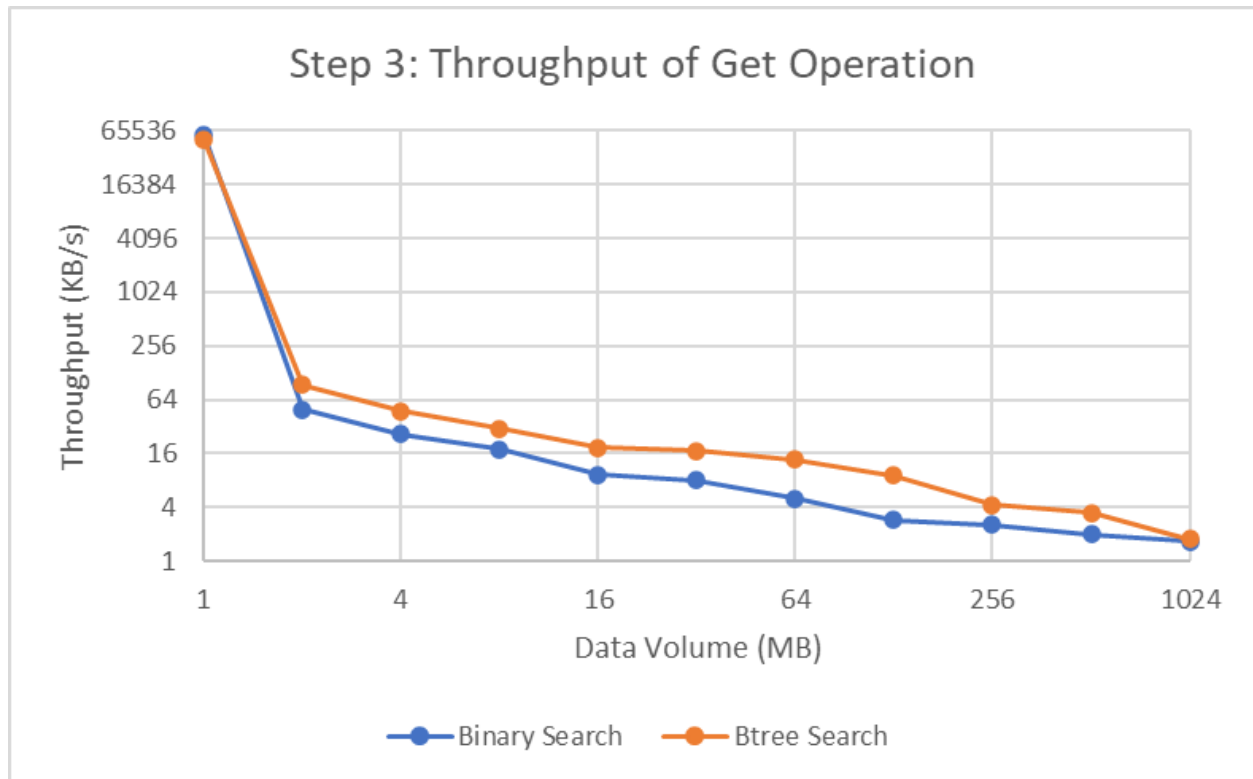
Similar to the results in the prior experiment, B-Tree search performs on-average better than binary search.

Figure 7: Throughput of Put Operation in Step 3



In step 3, the throughput for puts is worse than the throughput experienced in step 2, continuously decreasing as the data volume grows instead of quickly reaching a performance floor. This is because unlike in step 1 or step 2, we must additionally perform compaction on the flushed SSTs whenever an SST already exists at the same level as one we have created, and the size of the SSTs we must compact grows as the size of the DB grows.

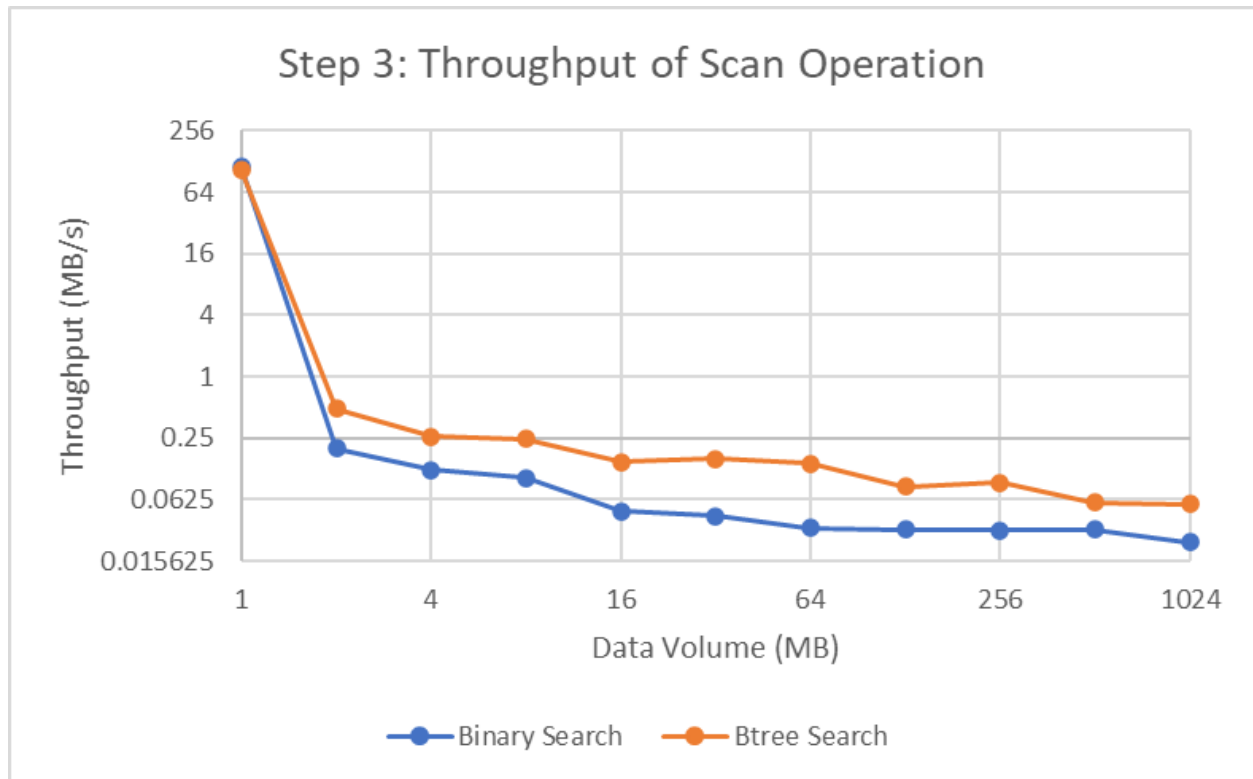
Figure 8: Throughput of Binary vs BTree Get Operation in Step 3



In step 3, the throughput for the get operation, both with Binary and B-Tree search, outperforms the throughput in the prior two steps, as it no longer decreases linearly as the data volume grows. This can be credited to the fact that we now have far fewer SSTs to traverse in step 3, thanks to the LSM structure.

B-Tree search performs on-average better than Binary search.

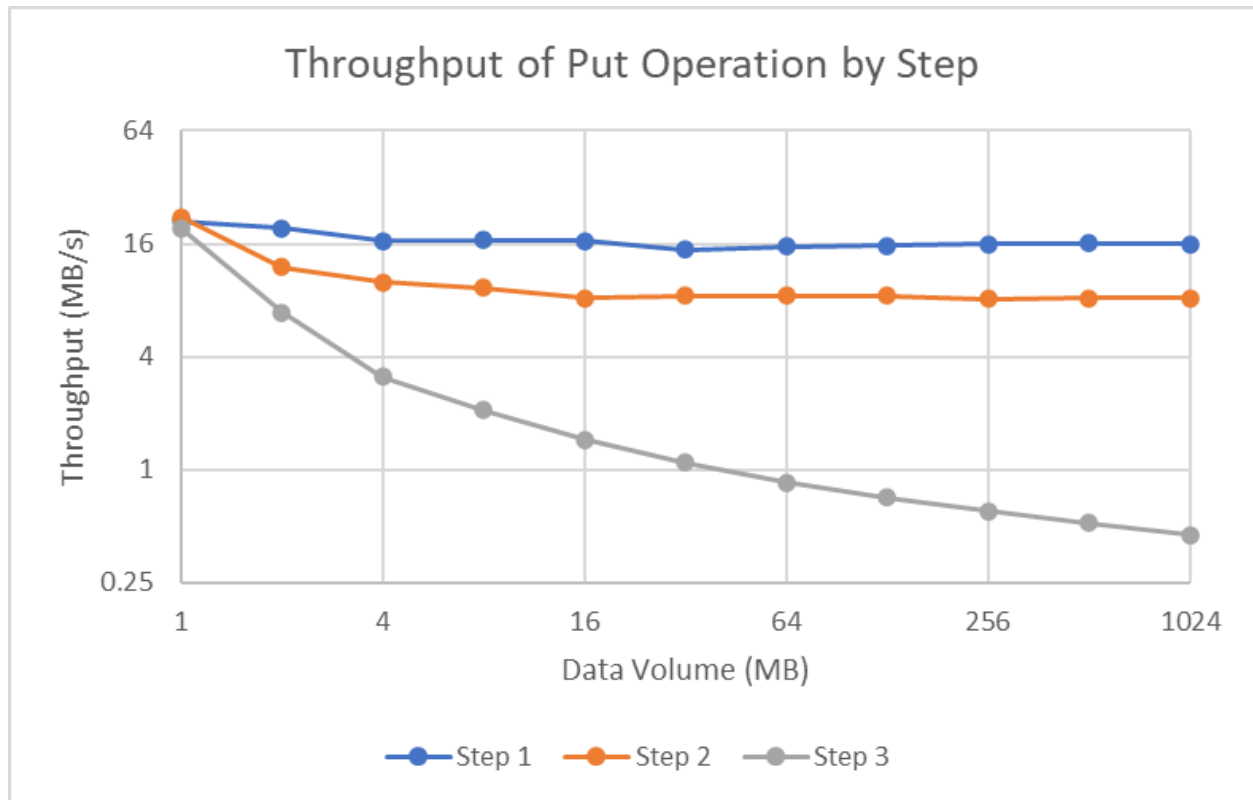
Figure 9: Throughput of Binary vs BTree Scan Operations in Step 3



In step 3, the throughput for the Scan operation also outperforms the throughput in steps 1 and 2, both with Binary Search and B-Tree Search. This can be credited to the fact that we now have far fewer SSTs to traverse in step 3, thanks to the LSM structure.

B-Tree search outperforms Binary search.

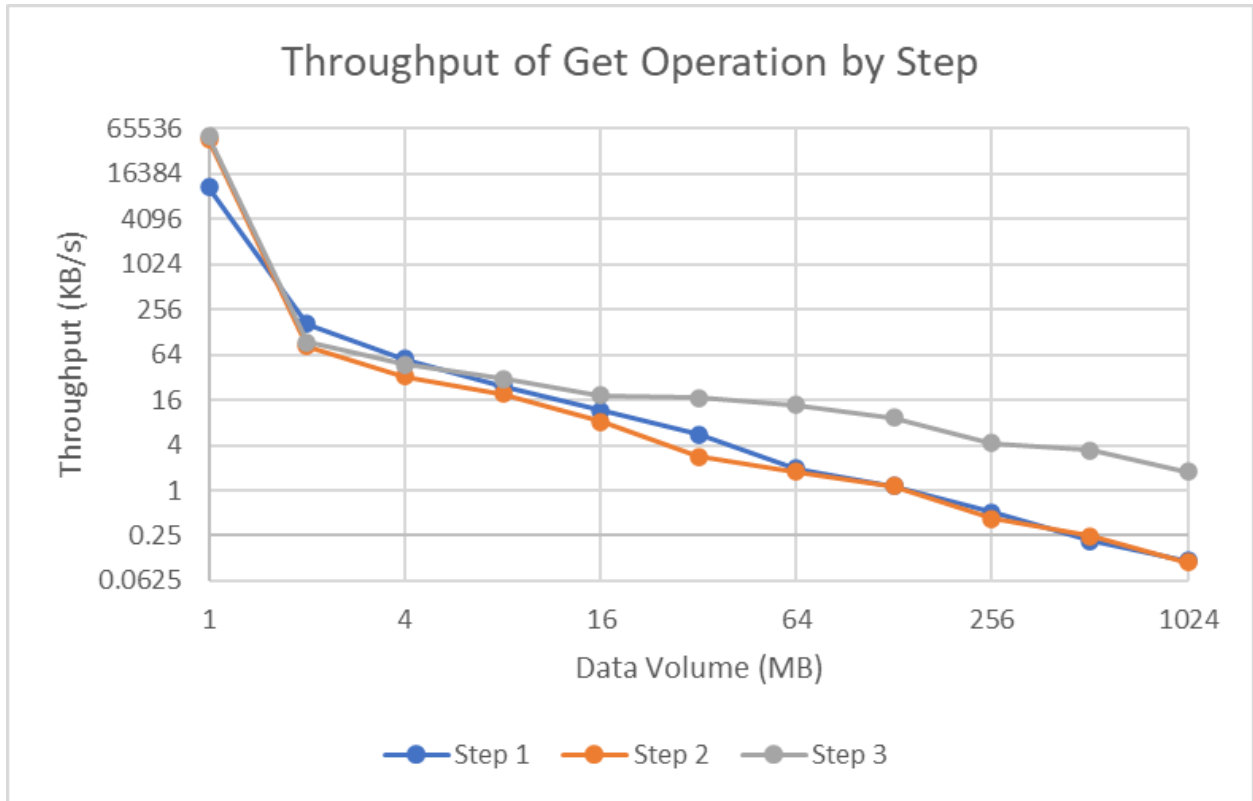
Figure 10: Comparison of Put Operation, Step 1 vs Step 2 vs Step 3



Comparing the results across the three steps, we can note that the performance of put as the DB size grows decreases with each step. In steps 1 and 2, as the volume of data grows the throughput settles around ~16MB/s and 8MB/s respectively, whereas in step 3 the throughput continues to decrease with data volume, decreasing below 0.5MB/s once we have inserted 1GB of data.

Although this distinction may not be large in a small DB, with 1GB of data step 2 performs ~16x faster than step 3. This is the difference between taking ~2 minutes to populate the DB with 1GB of data in step 2, and taking ~32 minutes to populate the DB in step 3.

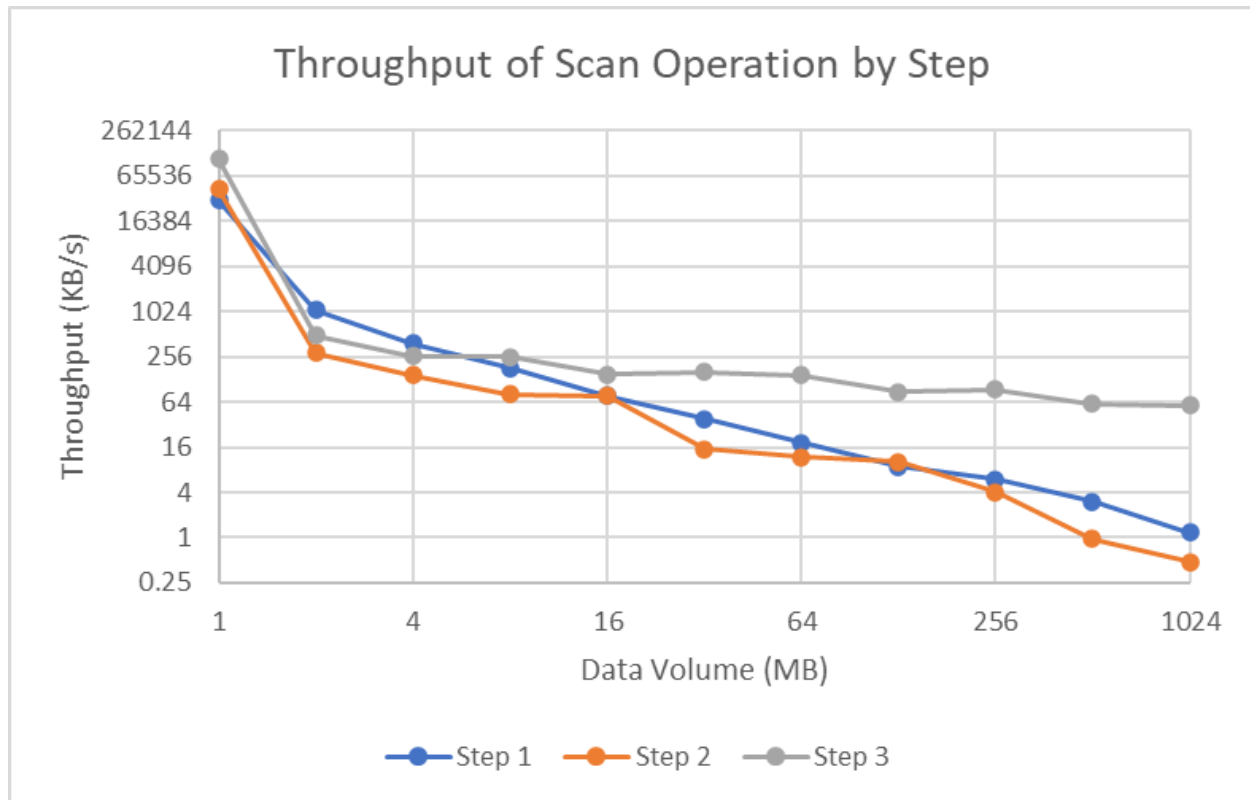
Figure 11: Comparison of Get Operation, Step 1 vs Step 2 vs Step 3



Note that in comparing the performance of the Get operation across the three steps, B-Tree search was chosen for steps 2 and 3, as it performed on-average better than binary search.

The performance of Get is similar in steps 1 and 2, but greatly improves in step 3 as the overall data size grows. This can be credited to the vast decrease in the number of SSTs in step 3 compared to the prior two steps: with 1GB of data, we have 1023 1MB SSTs in steps 1 and 2, and 10 SSTs in step 3, ranging from 1MB to 512MB. Although these 10 SSTs contain the same volume of data, we traverse each SST in logarithmic time, and thus greatly decrease our overall time taken.

Figure 12: Comparison of Scan Operation, Step 1 vs Step 2 vs Step 3



Note that in comparing the performance of the Scan operation across the three steps, B-Tree search was chosen for steps 2 and 3, as it performed on-average better than binary search.

The performance of scan across the three steps is similar to the performance of get: steps 1 and 2 perform similarly (however, step 2 is now slower than step 1), and step 3 vastly outperforms both of them.

Because our example scans are performed over a small range, the majority of the time in performing scans will be spent looking for values in-range and not iterating over in-range data, so similar reasoning applies for scan performance as applied for get performance.

Buffer Pool Experiment

The buffer pool experiments are conducted on the database with and without the buffer. The time taken for the two groups of 128 gets with 10 gets in each group is:

	No Buffer	With Buffer
Total time for 2560 gets	44.6704 seconds	1.39613 seconds
Throughput	0.000437521 MB/second	0.0139895 MB/second

The effect of the buffer pool is obvious and very helpful. In real life scenarios, the hit rate will be much lower than this, but the experiment clearly demonstrated the effectiveness of the buffer pool.

Bloom Filter Experiment

With the bits per entry set to 5 (and using the Monkey algorithm), we issued 100 gets and recorded the time it takes for the “gets” when it is getting existing keys from the lowest level (true positives). And we compared that to the time for 100 “gets” of non-existing elements (true negatives and false positives).

	Existing elements	Non-existing elements
Total time for 100 gets	0.363418 seconds	0.116401 seconds

With only 5 bits per entry, we can already see a significant decrease (68%) in terms of the search time with the help of the bloom filters.