Design Elements

The implementation begins with our main "Client" structure that contains the database name, a Memtable structure with a Memtable size, SST count, storage type, and clean up all as attributes. It then comes with 6 functions: open, put, get, scan, close, and flush. Each of these functions were made as per the handout descriptions with flush being an extra one. Flush is a helper that is used to get the current contents of the Memtable into a new SST.

The Memtable was implemented as an AVL tree. This was done through the AVLTree struct with its AVLTreeNode structs. The AVLTree contains two attributes, a root, and a size (current size). The root is an AVLTreeNode struct (the root of the AVL tree). The AVLTree struct has 6 functions: new, put, get, scan, scan_all, and size.

- The new function simply creates an AVLTree structure that is initialized as empty.
- The put function takes a key and a value to add into a new AVLTreeNode to add to the AVL tree. **Note:** that the put function is equipped with all the balancing and rotation helper functions that a regular AVL tree needs. It also replaces the value if the key already exists within the Memtable.
- The get function takes a key and returns a value if the key is found in the AVL Tree.
- The scan function takes a start and end key range with a KV hashmap where all the results are inserted in order to eliminate any duplicates.
- The scan_all function behaves like the regular scan but does not take a range or hashmap and simply returns the entire Memtable as a vector of KV pairs.
- Finally the size function returns the current size of the Memtable.

Going back to the Client structure here is how the functions work now that we have an understanding of the Memtable.

For Part 1 of the handout (with AppendOnlyLog system):

They can all be found around: src/lib.rs and src/serde.rs

To save a value: client.put(key, value)

All key value pairs are first saved to Memtable. `self.memtable.put(key, value)`
The call returns a bool which, if true, indicates the Memtable's capacity is reached, and our codes call flush to save the current Memtable to a SST file.

The flush function uses memtable's scan_all to generate a sorted list of (key, value). Then pass it in `self.storage.flush`, which calls the serializer function in serde.rs to save to file.

Note we have a Trait DiskStorage that implements get,scan and flush. We have different structs (object types): AppendOnlyLog, BTree, LSMTree that all implement this trait. This facilitates polymorphism in our codes to support users selecting different SST types.

To retrieve a value: client.get(key)
Our database stores i64 key and i64 value.
We first search for the key in memtable, if nothing is found, we then search through the SSTs.

Note SST files are generated using the same format of name: output_X.bin, X is the index under the directory of the database's name. E.x. databasename/output_0.bin, databasename/output_1.bin, etc.

To scan for values: client.scan(start_key, end_key) *end_key is included

After careful consideration, we decided to pass around a hashmap to record all the kv pairs encountered in Memtable and SSTs. We are indeed aware of the overhead of using such a hash, but not using it would require a get query for each element in range, which would cost more runtime and memory access (repeated visits of the same SST).
Our Memtable scan function receives the hashmap and does binary search in the AVL tree. In Part1 for scan, we iterate through SSTs, from greatest to least index in name (= read from youngest to oldest). We binary search in each SST, starting with the middle page: deserializing the page into a kv array + checking the first and last elements of the array to see if the start_key is in between. Once we find the page index and the array index to start, we continuously read on until hitting a key that is greater than end_key.

For the Get and Scan implementations with SST:
We first deserialize the first and last page to get the smallest and largest key of the file to check if start_key is inside it at all.

Then we apply binary search algorithm across pages. For each iteration, we deserialize one page into an array. We check if start_key is in range compared to the min & max key of this page. If in range, we further binary search on the kv array to find the target kv pair.

To open our database: client.open(db_name, config)

The open function takes in a database name and a KVConfig object. KVConfig is another struct we define with fields of user specification, such as memtable_size, bufferpool_size, storage_type and clean_up. The storage_type specifies the format of the SST file, which decides the corresponding algorithm to use for reading and scanning SSTs. The clean_up specifies whether SST files should be deleted after closing the database. When opening the database we first check if a database folder with the name already exists, and if it does we look to see its contents and fill in the Client object accordingly. This is to reopen the database.

To close our database: client.drop()

This one simple call will flush current data in Memtable to a SST. If clean_up is true, all the generated SST files under the database name's directory and the directory itself will be removed.

For Part 2 of the handout (with the static BTree implementation):

One of the major differences was that we had a buffer pool implementation through a BufferPool struct. This can be found in the src/buffer/ folder. The implementation of the buffer pool was made so that the buffer pool had two structures: A hashmap, and LRU.
The self made hashmap was implemented as a vector of a specific size (passed through KVConfig) where the size was equal to the maximum number of pages it could hold. The vector would hold references to BufferNode structs that would be implemented with chaining for collision. This means that not only did the BufferNode structures have a key and page content attributes but also a link reference to the next and previous nodes in their own respective chains.
On the other hand the LRU was implemented with LRUNodes which behaved similarly to the chaining explained earlier. Each LRUNodes would have a next and previous pointer to another LRUNode or None. On top of that the main LRU body (LRUMain) would have a pointer to the last and first LRUNodes in the structure as to be able to add and evict nodes efficiently and without pointer chasing.
Note that these two structures were linked by having two weak pointers. One from the BufferNode to the LRUNode, and vice versa.

The outside code would interact with the buffer pool through a single API call, "find_page". This function takes an sst_name and page_offset (in bytes) and outputs the required page back. The way it does this is by first trying to find it in the buffer pool by creating a BufferKey structure which is a mix of the sst_name and page_offset's, hashing it, finding the bucket and going down the chain (if required). If it find the page in the buffer then it will update the LRU position by putting it at the back of the eviction queue and returning the page. If it does not find it then it finds it in the SSTs (in storage), inserts that new page in the buffer, and if the buffer is already full then it runs the eviction policy (which is evicting the first page in the LRU queue).
Having the outside code only worry about a single API call makes it very easy in the buffer implementation in the query calls as it handles everything needed.

Another major change was the use of static B-trees as opposed to append only logs. The implementation difference can be found in the src/storage/btree.rs file.
Since for this assignment we are only constructing a static B tree without the need to implement insertion, update and deletes, the process is relatively intuitive and simple. We reverse engineer the Btree in bottom-up fashion using a sorted list.

The BTree SST file's format is as follows:
Internal node (a full page: max 256 reference==255 keys in one node):

(key0, page_idx that points to the next internal node when target_key < key0),
(key0, page_idx that points to the next internal node when target_key >= key0),
(key1, page_idx that points to the next internal node when target_key >= key1),
(key2, page_idx that points to the next internal node when target_key >= key2),
(key1, page_idx that points to the next internal node when target_key >= key3),
…

Leaf node (a full page: 256 kvpair):
(key0,val0),
(key1,val1),
(key2,val2),
(key3, val3),
…

Given the sorted list, we calculate the amount of pages it takes up, considering each page as a leaf node. Then we take the first key from every leaf node, considering them as candidates for the first internal layer's nodes' key. We emit the first candidate since in our format key0 appears twice in front, meaning the next candidate can point to the first candidate's leaf page as prev. Then we go thru a recursive algorithm to group candidates together into an internal node + promote some of them as keys for next internal layer's nodes. The algo we invented is as follows:

1. Calculate amount of nodes needed for this layer:
   Number of pointers = len(lower layer);
   Amount of nodes = ceil(Number of pointers / 256)
2. Calculate keys per node:
   (Number of pointers - 2 * amount of nodes) / amount of nodes
3. Some nodes get 1 key more
   ```
   // internal node with idx < excess_keys get an extra key
   let excess_keys = (num_ptrs - (2 * curr_level_num_nodes)) % curr_level_num_nodes;
   ```
4. Traverse through the candidates list, group them into one internal node (using the calculated amount of keys each get)
   *The node after a grouping is promoted to be candidates key for next layer

This algorithm guarantees a balanced BTree where all leaf nodes are on the same layer/level.

Part 3 of the Handout

The Bloom filter is contained in kv/src/filter.rs, and is used with the LSM get function in kv/src/storage/lsm.rs.

The code for serializing and deserializing bloom filters is in kv/src/filter.rs.

The code for compacting and merging SSTs is in kv/src/storage/lsm.rs, in the merge_ssts function.

We support both updates and deletes, which is contained in the kv/src/lib.rs code. We use the minimum i64 to represent a tombstone, which is used for deletions.

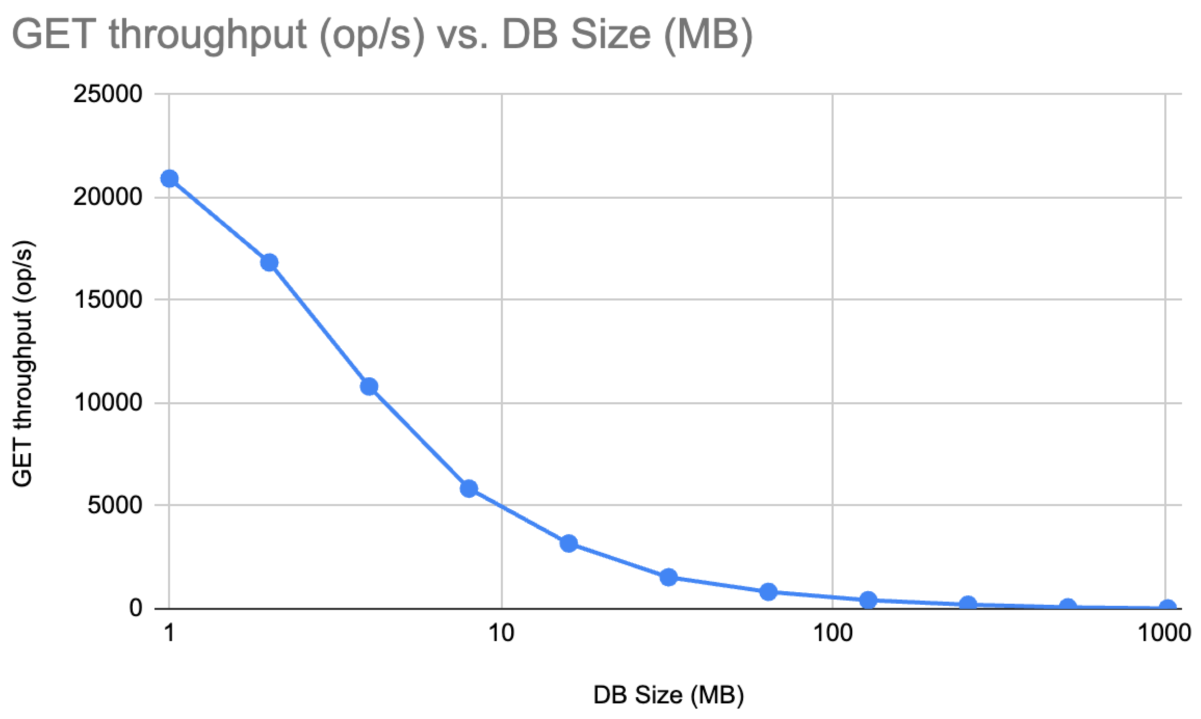
Project Status

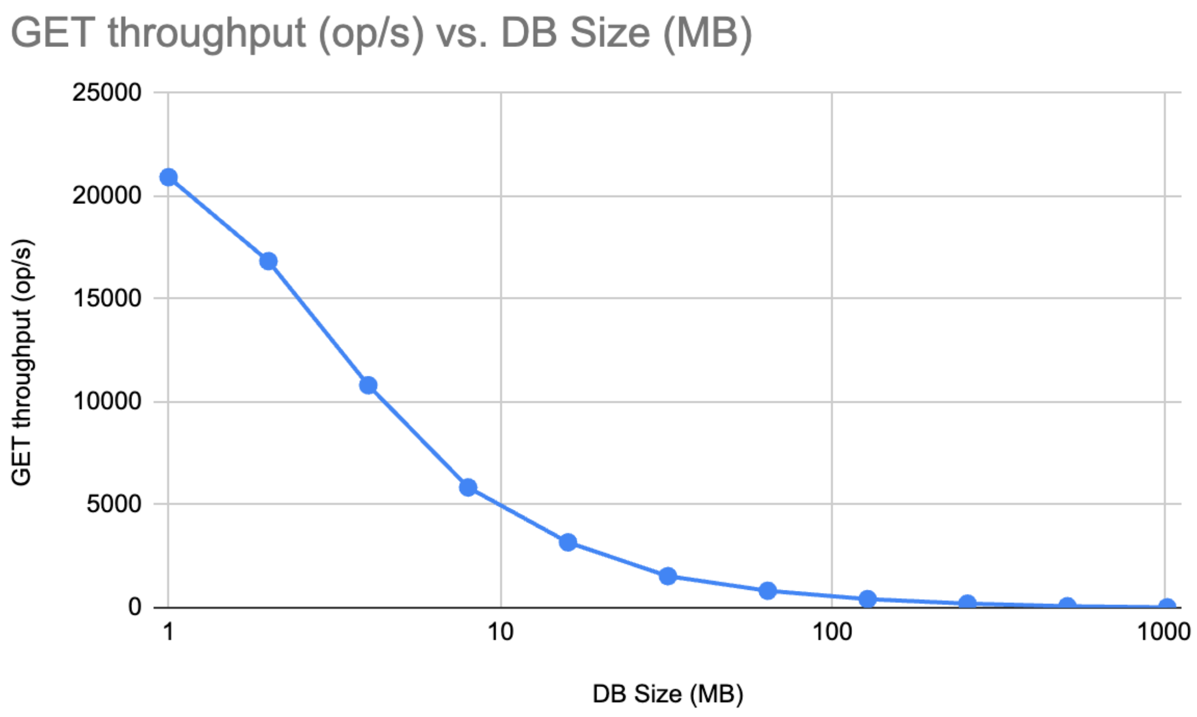
Our project is complete except for the following known issues:

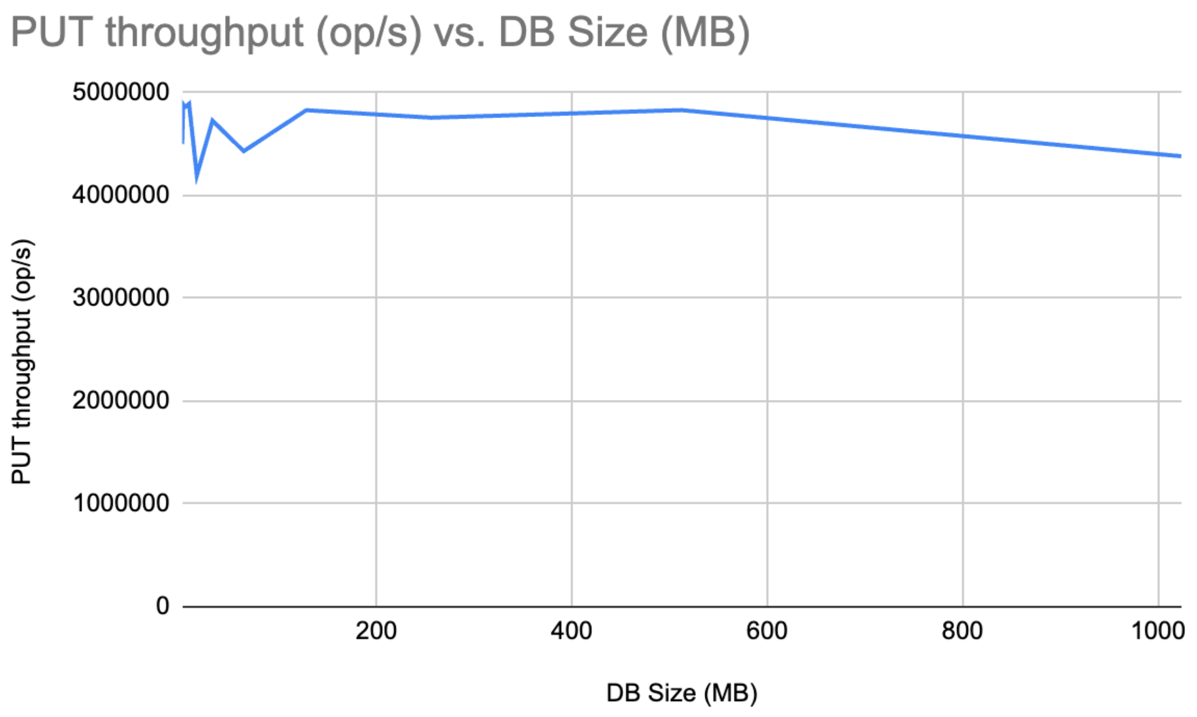
- Our LSM tree does not currently support re-opening after closing
- Our benchmark scripts do not create CSVs, but output the benchmark stats to stdout

Experiments

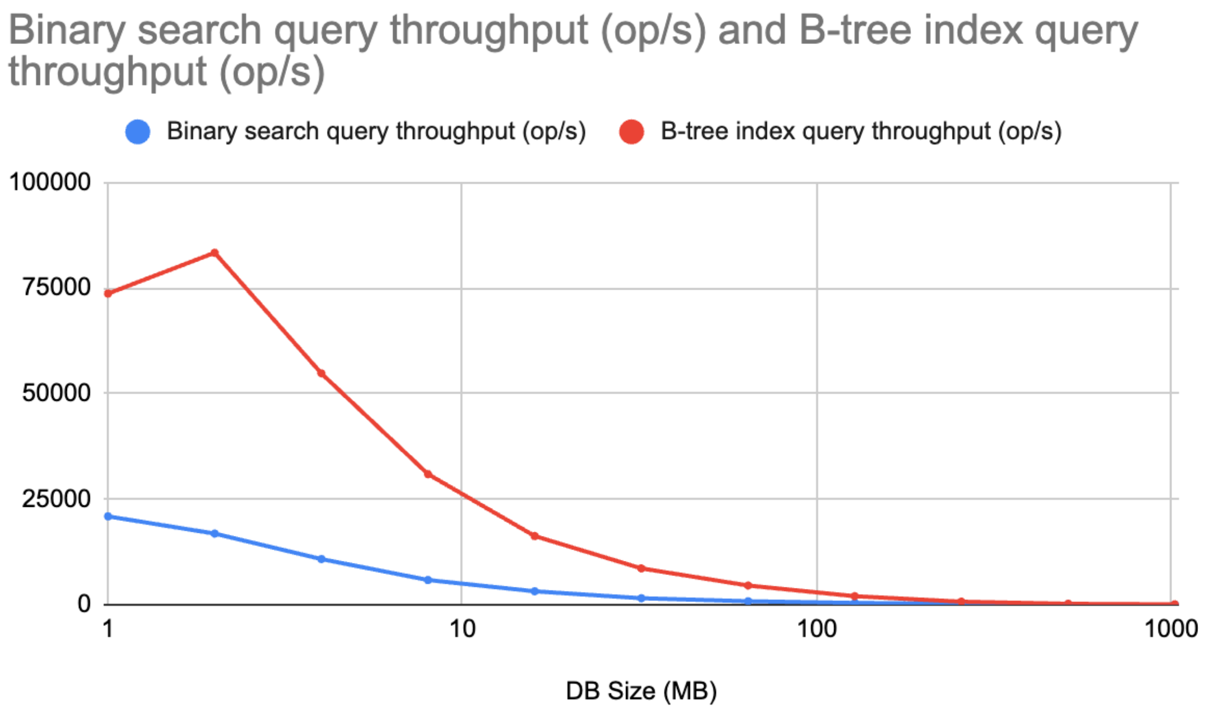
Part 1



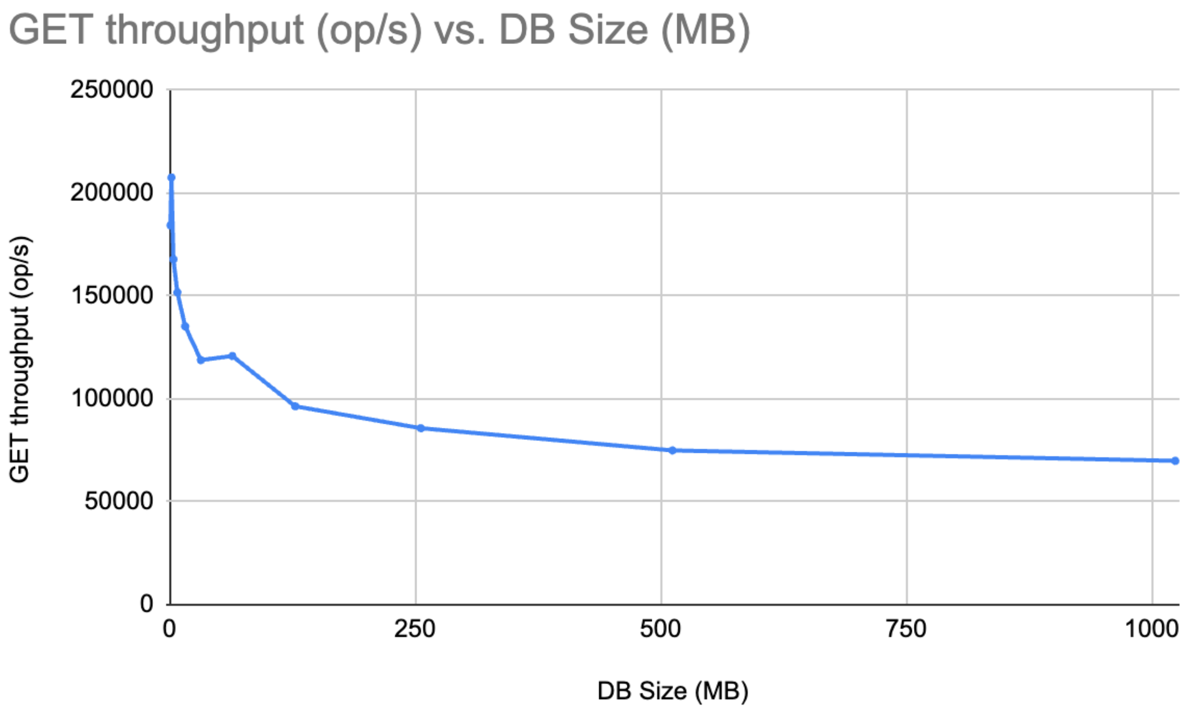

GET throughput (op/s) vs. DB Size (MB)
25000
20000
15000
10000
5000
0
1
10
100
1000
GET throughput (op/s)
DB Size (MB)

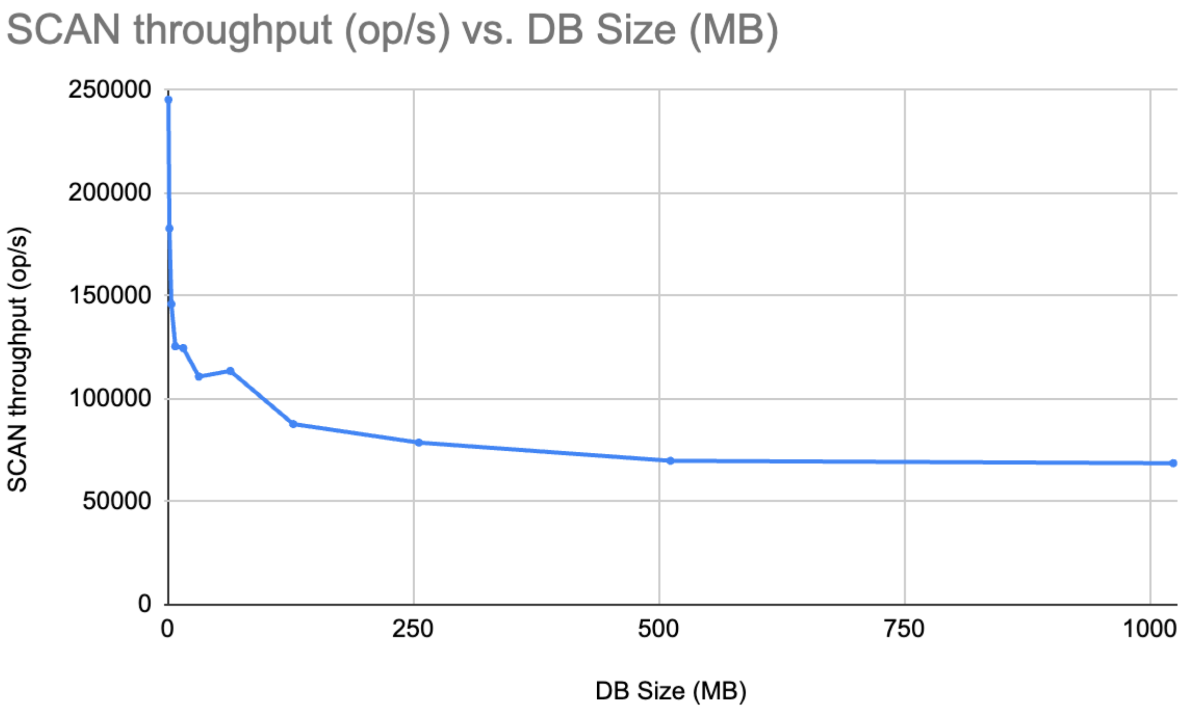

PUT throughput (op/s) vs. DB Size (MB)
5000000
4000000
3000000
2000000
1000000
0
200
400
600
800
1000
PUT throughput (op/s)
DB Size (MB)

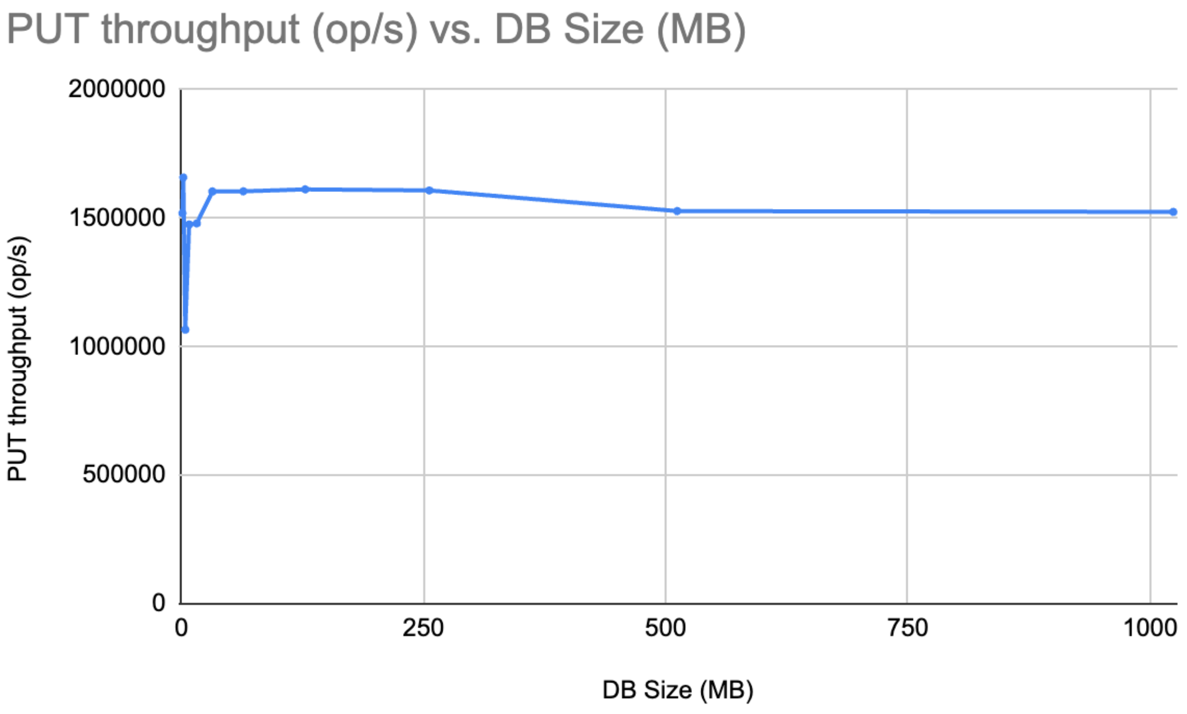
Part 2


Binary search query throughput (op/s) and B-tree index query throughput (op/s)
● Binary search query throughput (op/s)
● B-tree index query throughput (op/s)
100000
75000
50000
25000
0
1
10
100
1000
DB Size (MB)

Part 3


GET throughput (op/s) vs. DB Size (MB)
GET throughput (op/s)
250000
200000
150000
100000
50000
0
0
250
500
750
1000
DB Size (MB)


SCAN throughput (op/s) vs. DB Size (MB)
SCAN throughput (op/s)
250000
200000
150000
100000
50000
0
0
250
500
750
1000
DB Size (MB)


PUT throughput (op/s) vs. DB Size (MB)
PUT throughput (op/s)
2000000
1500000
1000000
500000
0
0
250
500
750
1000
DB Size (MB)

Testing

We have 35 tests (both unit and integration), which can be run using `cargo test`. Most of the test are co-located with the code that they test in a test module, but we also have some integration tests contained in the kv/tests directory.

Compilation & running Instructions

To run tests, run `cargo test`

To run part 1 experiments, run `cargo run –release –bin part1`

To run part 2 experiments, run `cargo run –release –bin part2`

To run part 3 experiments, run `cargo run –release –bin part3`