CSC443 Final Project Report

Michael Mao, Jian Jia Chen

Part 1:

Implementation:

Our memtab stores KV-pairs in an AVL tree implementation.
Upon filling, the memtab flushes its contents into a sorted list as our SST, which are assigned numbers beginning at 0 counting up.
The database class contains a memtab instance, which it manages by flushing upon filling it after a put.

The get method of the database first uses the memtab get method, which simply BST searches through the AVL tree. Then, it loops through each SST, newest-first, reading the file into memory as a sorted array of KV-pairs, and performs binary search. Get stops at the first matching key it finds and returns the value.

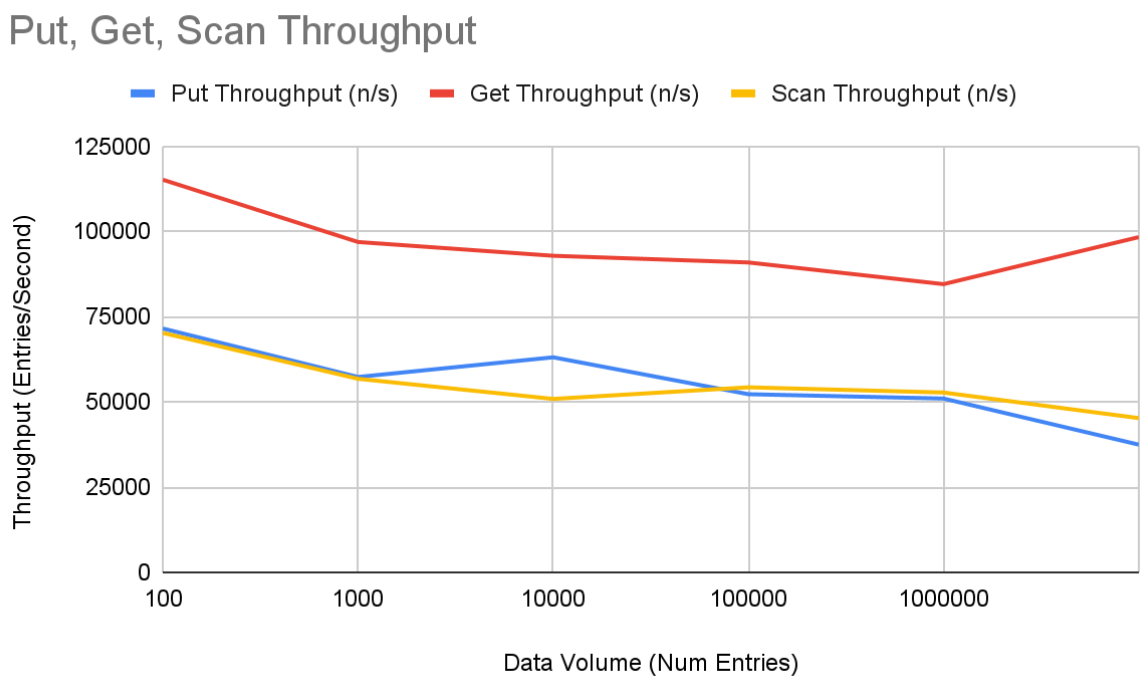
The scan method of the database also first scans the memtab using BST search to identify the values closest to the minimum and maximum, while also being within the given range. It performs a similar operation on the SSTs using a modified binary search. The Scan method will iterate through the memtab and every SST on a call.

On close, the database class will save some basic information such as the number of existing files as well as save the memtab to a special SST.

On open, the database will reconstruct its file counter and the memtab.

Experiments:

The part 1 experiments were performed with 256 byte pages.
Get and scan experiments were each averaged, with the number of operations being 10 per operation type per data volume, except for 10,000,000, which was performed with only 3.
Integers were inserted in reverse order.



Data Volume (n)	Put Throughput (n/s)	Get Throughput (n/s)	Scan Throughput (n/s)
100	71530.8	115181	70239.5
1000	57313.2	96915.2	56779.5
10000	63085.9	92874.5	50878.7
100000	52263.3	90910.9	54282
1000000	50986.2	84552.3	52753.7
10000000	37462.8	98344.2	45249.5

The throughput appears to decrease fairly slowly even on a logarithmic scale. Get appeared to have become faster with n = 10000000, but this could be due to a smaller number of gets being conducted.

Part 2:

Implementation:

Page Directory:
The database maintains a page directory that uses SST metadata on the Key-Value entries the page holds (min key, max key, size) to improve performance and better utilise the extendible buffer pool. It is implemented as a binary tree, whose leaves represent the SST in sorted order. It begins as one layer, and we insert all incoming KV into the root. Once any leaf becomes full, (data page filled), we split it down its median value into two new children leaves. The former leaf becomes an internal node and no longer corresponds to a SST, and instead directs the tree via the split median value. The left child inherits the bottom half of data and the right child the top half. The new leaves then insert incoming insertions until another split is necessary. New SSTs are purely created via splits, meaning that at any given point at least half of the allocated memory is holding active data.
As a sorted binary tree, this allows the SST's to correspond to a unique range of keys, and may find the SST's key in log of |SSTs| time for any get to the database. This is enabled by a unique 32bit hash key each SST is assigned on creation, which serves as their name in file as well as their hashkey in the buffer. Then after retrieving the key's SST via the directory, and that SST's hash, we may look into the buffer if the directory indicates that the page is present in memory using the retrieved hash key and find the page in expected constant time, and find the KV pair within that page in log time.

Extendible Buffer Pool:
The buffer pool uses a bit mask on the uniformly randomly generated hash keys for each SST to map it to a bucket. As a design choice, hashing on an already uniformly random key would imply that we only need to correctly preserve the distribution of keys as we map keys to a bucket. The chance of collision of any two keys is equal to the probability that their leading *n* bits where identical,
$P[h(k_1) = h(k_2)] = P[k_1[i] = k_2[i]\ ] \ for\ i \in [0, n]$. Since each bit $k[i]$ is uniformly and independently sampled, (using a time seeded random generator), we take the product sum of any single bit colliding *n* times. $P[k_1[i] = k_2[i]\ \forall i < n] = (\frac{1}{2})^n$, which is optimal collision probability for 2^n buckets.

B Trees:
The database currently uses a B-Tree class to build a B-tree instance given page data. Then for any retrieved page the database may construct and buffer a B-tree representation for that page and complete queries to that frame in $log_b n$ time. Currently we are looking to remove the step of preprocessing the tree by building it, and directly processing a contiguous array representation of the tree to cut away the non-sublinear time needed to insert each value into the tree. (As binary searching the sorted SSTs would give $log_2 n$ time, which is already sublinear.)

Experiments:

The first workload we tested for was 1000 contiguous page inserts, corresponding to the querying or insertion of evenly distributed keys.

Part 3:

Implementation:

A LSM tree is introduced, which stores a linked list, with nodes representing a row in the LSM tree, each with two entries for a filename attached to a SST. When a flush from the memtab occurs, it is inserted into the root, representing the top of the LSM tree, and merge operations are performed upon filling a row, and the combined SST is moved to the next row.

Merge operations are done by storing a KV-pair from each of the two SSTs, performing merge-sort, and filling a buffer with the results. Upon filling the buffer, the contents are written to the target SST. At the beginning of each SST, there is now a line of space dedicated to the filter bitmap, which is developed during the merge-sort, and written to the beginning at the end of the operation. Now, to read a page with the SSTs, there is a method which identifies the SST page name from a page number, so that the SSTs may still be iterated through by page.

Remove operations have been supported with the use of tombstones. MIN_INT is used to denote a tombstone, and gets and scans support this by

The bloom filter of the tree uses simple modulo operations on the key as hash functions. Bloom filter bitmaps are stored within SSTs with no buffer implementation, and they are loaded as the get method iterates through pages from newest to oldest.

Testing:

Used experiment.cpp with different settings memtab size, such as 4, 64, 100, 1000, and 250000 integers, using STAGE set to 1 and 3. Used simpletest.cpp to perform tests on closing and re-opening a database, as well as correctness checks on operations across stages, including deletes and updates for STAGE == 3. During early development, tested the b-tree independently with builds and gets and scans.

Project Status:

- Bug: Occasionally performs an invalid free operation during scan using the part 3 code.

Compilation & Running Instructions:

For work on parts 1 & 3, the main branch can be used.
Once the makefile is used, run *experiment1.out* with parameters:
<stage> <n> <num_gets_scans> <scan_length>

Where stage is either 1 or 3 depending on the part of the project to test,
n is the number of items to put in the database,
num_gets_scans is the number of gets and scans to perform for the average,
scan_length is the number of keys to include in the scan.

For work on part 2, the branch “12Failsafe” should be used.