Part 2 Re-designed)

Repo: https://github.com/furrystreamrock/443-Database-Project/tree/12Failsafe

Implementation:

SST_directory class:

The re-implementation of part 2) includes mainly the use of a page directory data structure. This is defined in *memtab.cpp* as *SST_directory* class. Its purpose is to improve our Database's ability to use the buffer pool by finding any insert, update, delete or get in $\log_2 \frac{N}{B}$ expected time. (Absolute worst case is in $\frac{N}{B}$ time.) The directory is implemented as a binary tree, that is 'balanced' in accordance to the distribution of inserted keys. The leaves of this tree represent every SST in the database in strictly sorted and ascending order. This means that an in-order traversal of the leaves will give a strict ascending key order of all K/V pairs, where there is no key overlap between SST nodes. The internal nodes serve to direct requests through the tree, similar to a B-Tree with factor two. However this is special from a B-Tree because the structure supports deletes and insertion updates to its leaves. It achieves this by 'splitting' any terminating leaf node into two children, $left$ and $right$, where each will inherit half the K/V pairs split down the median key value. As we insert into the database, the directory will dynamically expand, maintaining metadata on all SST's that are the tree's leaf nodes, and guide insertion, updates, gets as we use the database.

Dynamic Buffer Pool:

The buffer pool for is implemented as an extendible hash table in *database.cpp*. Given knowledge of the key for any SST, it may return the sorted K/V pairs in that table in expected constant time. Our implementation for extendible hashing uses a uniformly randomly generated unsigned long that is assigned to every SST as the page directory deems necessary to create. This is then hashed into a hash table index between $[0, 2^d)$ where d is the depth of the buffer by looking at the leading d bits of the key. The chance of collision is equal to the probability that every bit of two tables match, hence for uniformly independently chosen bits, is $\left(\frac{1}{2}\right)^d = \frac{1}{2^d} = \frac{1}{\#buckets}$, hence we match the conditions of universal or perfect hashing. The buffer table begins at 4 slots, and doubles up to a maximum depth that is user defined as we fill it. If ever pages are evicted such that 70% of the buffer pool is not utilized, it will shrink itself by halving its size to save memory.

It may evict pages either using LRU or Clock policies, the policy may be set by the user under the parameter $eviction\_policy$.

B-Trees:

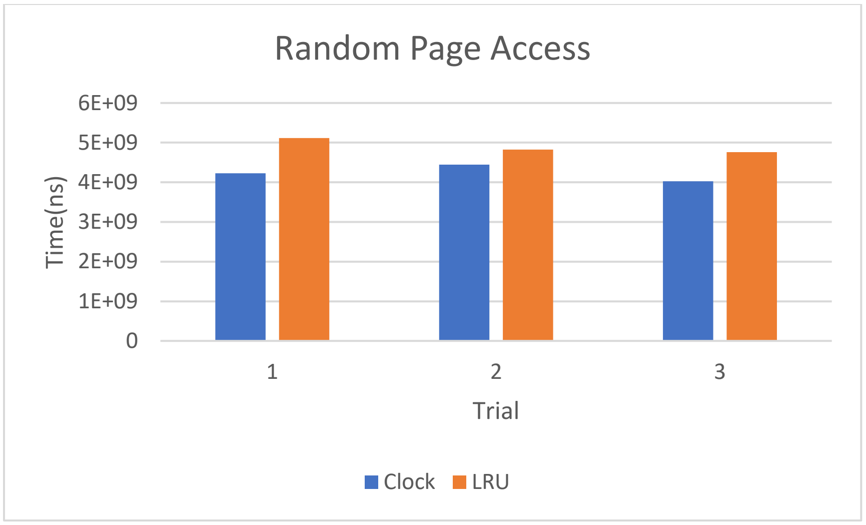
Unfortunately, I didn't have the capacity or time to have this redesign work with B-Trees.

Eviction Experiments.

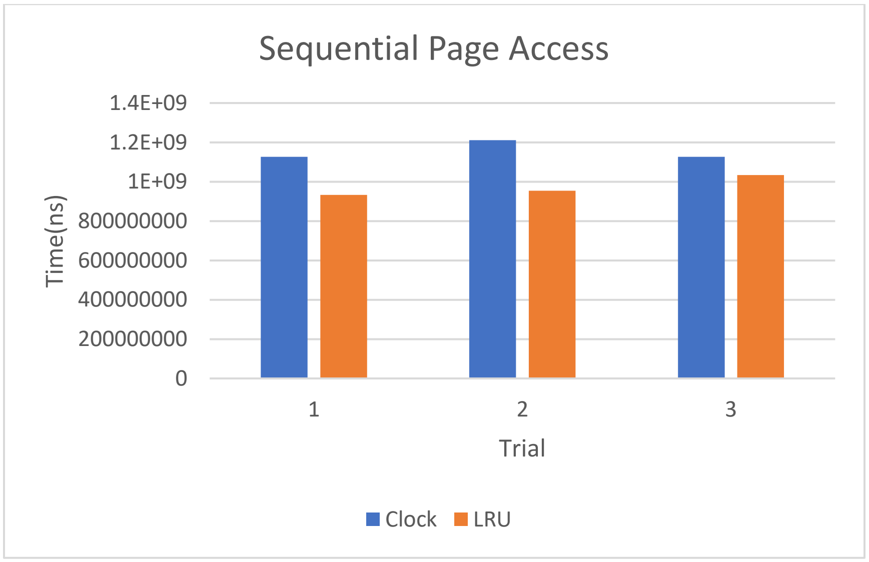
Workload 1: Uniformly random page accesses, 10 000 accesses. 16 maximum buffer capacity, 8 entries per page.

Run experiment by compiling and running *mike_experiment.cpp*. Set 'experiment' to 0, and eviction to 0 for clock, 1 for LRU.

Figure:



Workload 2: Sequential page access. 10 000 accesses. 16 maximum buffer capacity, 8 entries per page. Run by setting 'experiment' to 1, and eviction as needed.



From our experimental results, we see that Clock had superior performance for random accesses, but LRU did better for sequential. This aligns with our expectations, as LRU will perfectly evict pages in sequential while clock experiences flooding. On the other hand, with completely uniformly random accesses, clock does a better job at capturing 'hot' pages and retaining them in the buffer.

***when running experiments on my computer, I often get seg-faults where I wasn’t able to open a file for write. I don’t think there’s any pattern to it, and experiment would work just by running repeatedly until every write didn’t cause error. I hope its specific to my machine, but experiment may have to be run more than once.