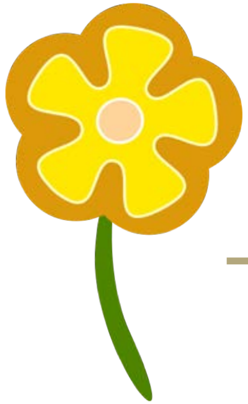
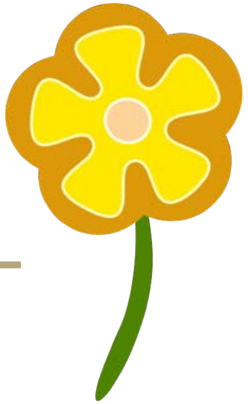
DaisyDB: Heuristic LSM Tree Read-Write Optimization





By Benjamin Borden

Motivation

The traditional LSM Tree implementation is divided between:

- Leveling: read optimized - better for large amounts of stationary data.
- Tiering: write optimized - better for frequent updates.

Who chooses the correct implementation?

Traditionally someone with knowledge of:

- the use case
- the correct implementation for the type of use case

Motivation

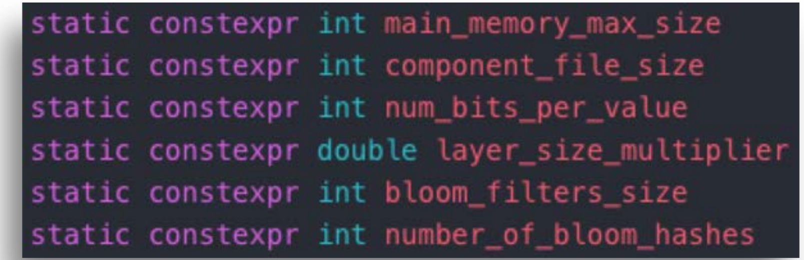
Why is this a problem?

- LSM Trees becoming more mainstream!
- Amateurs, students, and hobbyists are using said products!
- Non-professionals are expecting a plug-and-play solution!

Problem Statement

Can a heuristic toggle between tiering and leveling based systems create performance benefits on a read or write performance heavy workload? And can those benefits be created on a more balanced workload as well?

Knobs



```
static constexpr int main_memory_max_size
static constexpr int component_file_size
static constexpr int num_bits_per_value
static constexpr double layer_size_multiplier
static constexpr int bloom_filters_size
static constexpr int number_of_bloom_hashes
```

LSM Tree Knobs

- Main-memory_max_size - Number of values the main memory buffer can store.
- Component_file_size - Number of values allowed for in a single file.
- Num_bits_per_value - the number of bits in the bloom filter per value stored.
- Layer_size_multiplier - The number of times larger the next layer is than the previous.
- Number_of_bloom_hashes - The number of hashed values per insert.
- Mode - tier or level

Swap Based Knobs:

- Level_tier_swap_boundary - the point over which a swap system switches between leveled and tiered merges
- Swap_cushion - the buffer over the swap boundary to prevent constant swaps
- Running_average_size - number of operations to measure read-write heaviness

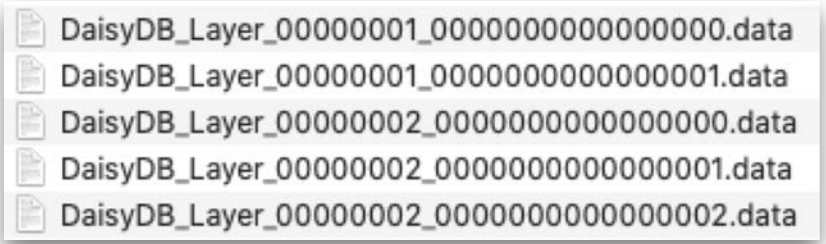
Buffer

- Stores values in main memory prior to being loaded to disk
- Implemented in the code as a binary tree based key-value map:
    - Keeps the keys sorted!
    - Gets are quick!
    - The tree isn't always balanced, but the buffer stores a relatively small number of values
- Once full, buffer is written to disk in a file and moved down into the disk hierarchy through offloadMainMemory()!

Difficulties: Data Structure Implementation, maintaining a sorted order, implementing deletes!

File Structure

- The disk storage is made of layers of organized .data files!
- The number of layers expands to accommodate the amount of data!
- Files are organized into ‘components’ which are sorted runs of data.
- Each file is static and sorted!
- Each layer can contain `Layer_size_multiplier` as many files as the prev



```
DaisyDB_Layer_00000001_0000000000000000.data
DaisyDB_Layer_00000001_0000000000000001.data
DaisyDB_Layer_00000002_0000000000000000.data
DaisyDB_Layer_00000002_0000000000000001.data
DaisyDB_Layer_00000002_0000000000000002.data
```

File Moving

- Groups of files make up layers of the LSM tree
- When a layer reaches a size capacity, determined by **layer_size_multiplier^layer_num**
  Then the layer is moved down to the next layer by retagging it

DaisyDB implements layering and tiering based on where merging the files occurs in the file moving

- Merging the layer prior to retagging the files = tiering
- Merging the layer after retagging the files = leveling

Fence Pointers & Bloom Filters

- Each file retains its own fence pointers and bloom filters
- stored in main memory and are assigned based on a key made of:
    - pair<int,int> = first int represents the layer, second int represents the file

Difficulties: How to reassign fence pointers after a merger? How to keep fence pointers and bloom filters persistent?

Heuristic Switching

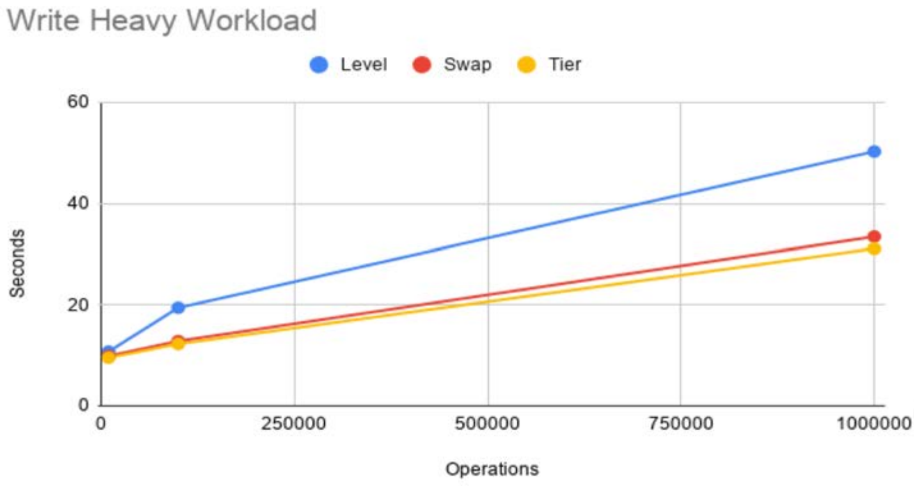
- A running average of the last number of operations is kept in memory
  - Measures the number of writes vs reads under the workload
  - If avg swings into the write heavy category, it switches to tiered & vice versa

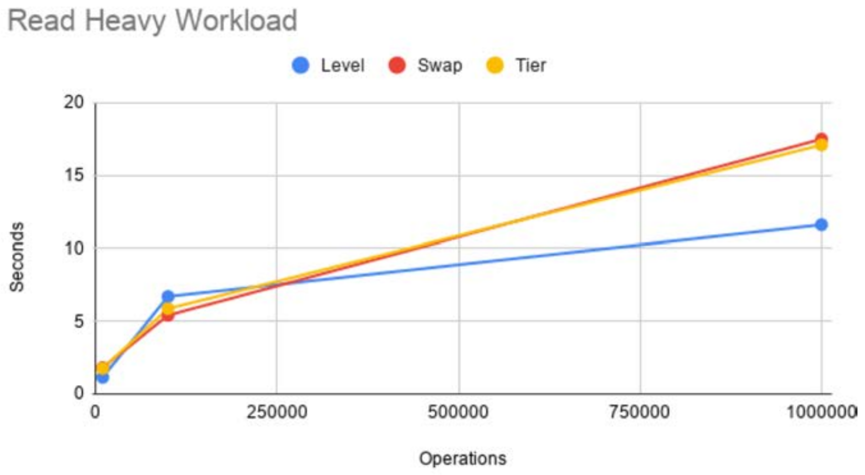
- Puts = Write
- Deletes = Write
- Scans = Read
- Gets = Read

Difficulties: Scans are very expensive, how do they factor into the avg?

Observations

- Extremely write heavy workloads see a lot of benefit from Tier-Level swapping when mis-implemented
- Read heavy workloads see much less drastic benefit, until enough writes are made to restructure the file tree.





Observations from along the way

- **Mixed read-write workloads see little benefit, but even reads and writes favor tiered systems due to the writes being more expensive than reads in this system**
- **Scans are difficult to adjust for due to being more expensive than normal reads**

Pending Items

- Increase testing to measure the rate of performance change based on workload
- Re-examine the file merging methodology, in attempt to optimize for semi-sorted data.

Contributions

- *LSM-based Storage Techniques: A Survey* - Luo, Carey
- *The Log-Structured Merge-Bush & the Wacky Continuum* - Dayan, Idreos
- *The Log-Structured Merge-Tree (LSM-Tree)* - O’Neil, Cheng, Gawlick, O’neil