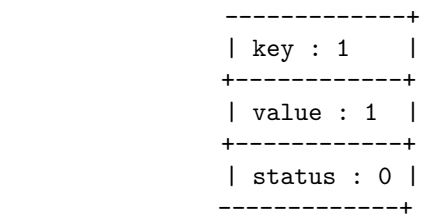
LSMTree

Implementation of the Log-Structured Merge Tree data structure

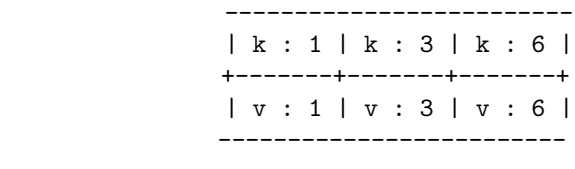
by Efstathios Karatsiolis & John C. Merfeld

Data model

Our LSM tree implementation stores pairs of key-value integers. Each key can only correspond to a single value. Each pair of key-value form an object of type Entry. An entry will also contain a boolean variable that will indicate if this entry in the LSM tree is a deletion. We might migrate our status metadata to some other part of our implementation for space and speed efficiency. You can see a high level representation of our entries below.



Multiple Entries will be stored in memory in a sorted array. We only allow for a specific number of `Entries` to be stored in memory before flushing the data to the disk. When we reach that threshold, the group of `Entries` in memory form a `Run`. Once a `Run` is formed, it is flushed to the disk. Main memory only contains a single `Run`, and it looks like this:



Or, on a macro level a run in main memory (r_main):

```
[r_main] = [1 3 6 7 32 96]
```

A `Run` consists of multiple pages of space. For each of these pages we will maintain a `FencePointer`. The `FencePointers` for each page will keep track of the minimum and maximum key existing in that page as well as the position of that page in the run. A list of all `FencePointers` for each level will be maintained in memory.

Each `Run` will also have a corresponding `BloomFilter` for its keys.

Entering data into our system will be done according to the following procedure.

Once `r_main` reaches capacity, we write it to disk. This involves three steps:

1. Record the highest and lowest values contained in `r_main` into a new `FencePointer`.
2. Create a new `BloomFilter` for the new `Run` in memory.
3. Insert `r_main` into the topmost `Level` of the `DiskTree`. Update `BloomFilter` and `FencePointers` metadata to refer to that level in the disk.

For each level we will maintain a metadata object containing parallel lists of the bloomfilters and fencePointers. Each entry of these lists will refer to 1 run.

We will focus on implementing a leveled version of the LSM tree, focusing on fast reads and potentially slower writes. After this, we will try to implement the leveling version of the LSM tree and try to add tunability to our code.

API support

`Get(int k)`

Returns either the value or some NULL result: First, check the `Run` in memory for `k`. Then, one by one, check the `Searchables`. First the fence pointer, then the bloom filter. If both of those give promising results, check the corresponding `Run` in disk and return when the entry is found.

`GetRange(int min, int max)`

Returns either all values in range or some NULL result: First, check for any values in the main memory `Run` that can be added to the return set. Then, go about a similar process as the one for point queries (this time we only need to check the fence pointers). The key difference is that we do not stop once we find the run that contains the values we want. We must continue to check each run for more values to add to the return set.

`Insert(int k, int val)`

Returns nothing: Inserts a new `Entry` into the `Run` in main memory with its `isRemove` bit set to false. If the main memory array is full: create a bloom filter and fence pointer for it, sort-flush it to disk, and check for merges in the lower `Runs`.

`Update(int k, int newVal)`

Returns nothing: Updates consist of an `Insert(k, newVal)` call. Whenever runs in the tree are merged, the hotter `Entry` takes precedence and removes the colder one.

`Delete(int k)`

Returns nothing: Inserts a new `Entry` into the `Run` in main memory with its `isRemove` bit set to 1. Whenever runs in the tree are merged, matching keys with different `isRemove` bits annihilate each other.

Experiments

Since we will start from implementing the leveling version of the LSM tree our experiments will focus on efficient and fast reads. Of course we will try to optimize memory management to optimize organization of the data and get the best possible performance. We will experiment with both dense and sparse data. We will also check how well will our system perform on increasing loads of data, thus if we will have an expected scalability. We will also try to identify any bottlenecks in the performance of our LSM so we can figure a way to deal with it.