Pebbles DB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees(II)

Pebble DB is virtually identical to another data base we covered in class LSM Trie, with one significant difference. Both help to decrease write amplification by reducing the amount of work needed to move sstables, fragments, into the next layer. Instead of merge sorting the sstables into the next level we simply place them on top of the next level after merge sorting with themselves. We also only move the sstables within a partitioned group, when the group is full, as opposed to compacting a whole layer. The allowance over overlapping tables on each layer does introduce some lookup cost. However, we can tune this tradeoff by adjusting how many overlapping layers there are in a partition. Essentially, how much of an ordered list each entire layer is. The big difference between these two databases is that, LSM Trie has predetermined positions for its partitions. However, Pebble DB tries to be clever by dynamically allocating its dividers, called guards, as will be explained in the first question. On the outset this seems like it should be a great idea, and possibly even place the dividers in the most optimal positions. However, we shall see in this paper some potential drawbacks with this method.

(1) “FLSM can be viewed as a generalization of the LSM data structure.”. Please explain this statement.

Both LSM and FLSM have similar structures. They can keep their lower levels in memory to improve performance. Both, use sstables to store ordered data and pass the sstables through layers as they become full. The layers on both structures become larger as one moves to higher levels on the tables. However, instead of making each layer one giant ordered sstable, FLSM splits up the tables with guards that are inserted into the layers as the table is used. The guards are chosen using MurmurHash from keys that are inserted into a layer. The probability of a key being chosen increases as a keys move to higher layers. Once a guard is chosen it is copied to the higher levels during compaction into those layers. Multiple overlapping sstables can be attached to each guard. When a guard is full the keys are merge sorted and then partitioned to fit into the guards at the higher level. As mentioned in the introduction, this looks very similar to the structure of LSM-Trie. Just hearing these differences may lead one to assume that FLSM could not be a generalization of LSM. However, upon closer inspection once can see that FLSM can be configured such that only one sstable is allowed per guard. Once this is done the performance would be virtually identical to LSM!

Once we make this realization, we can see that FLSM affords us a tuning nob, upon the original LSM, which enables us to achieve faster writes speeds, at the expense of slower read speeds through the changing of the number of sstables permitted on a single guard. This can be tuned per application to match the work load! This generalized LSM sounds too good to be true! Unfortunately, as we learned in class there is an issue to the use of these guard keys. Unlike its virtually identical cousin LSM-Trie, the fact that the guards are constantly being inserted, will actually defeat some of our write performance, as it will frequently split up sstables within a guard upon insertion, which is a very costly operation.
(2) “New sstables are simply added to the correct guard in the next level. There are two exceptions to the no-rewrite rule”. Describe these two exception compaction scenarios.

We have two cases where we will perform the costly operation of writing over an existing sstable. One is when we have reached the lowest level, level five. At this point we have no choice because there will be no subsequent layers to write to. Another, scenario where we will choose to perform this costly operation is when we are avoiding writing over layer five. It is preferable to write over the sstable in level four than to have to perform a second costly compaction into level five.

(3) “deleting a guard will involve a significant amount of compaction work” Please describe the operations involved in a guard deletion.

There are two times where it is beneficial to delete a key. One is when the guard is completely empty at every level. This should avoid the problems of compaction and should avoid having a clutter of what are essentially small empty files taking up disk space. Another case is when the guards become too unbalanced and the deletion would allow for the redistribution of the keys. The authors of the paper had not actually implemented deletion into PebbleDb. However, they speculated of a way to perform the operation. They proposed that the deletion be marked in memory, then at compaction time a guard could be removed from a level and its children would either be transferred to the neighbor guard or compacted to guards in the next level. Whenever, a guard is deleted at a level the guard would also have to be removed at all lower levels, to keep the definition of how guards must be copied all the way to the highest level from wherever a guard is place, which would be broken by introducing this gap. Also, the guards could optionally be removed at higher levels if it would prove beneficial. Metadata would be kept about the deletion, which could be useful in case there is a power outage, and the deletion needs to be subsequently rebuilt.

(4) Describe how a range query is served.

In order to achieve this goal we will need to collect all of the keys in the range. We will accomplish this using the following steps: First, the guards are identified that intersect with the range. Second, we find the sstables within the guards at each layer that intersect with the range. Third, we perform binary searches on the sstables to find their smallest key that fits within the range. Fourth, we perform a merge sort of the sstables starting from the smallest keys within the ranges until we reach the last key within our range. Finally, we return the merged list of keys and their values.

(5) “In Workload F, all writes are read-modify-writes: the workload does a get() before every put() operation. As a result, the full write throughput of PebblesDb is not utilized, resulting in performance similar to that of other key-value stores.” Explain why for this workload PebbleDB has similar performance.

Pebble DB works best with sequential writes. This is where it achieves its best performance. While this workload better than constant reads it is still slow. In order to find the information, it will have to go to each level and use bloom filters to find out what sstables to binary search. However, there will still be false positives and most likely going through multiple layers before finding the data that is being looked for. Workloads which give more time to writing will organize the data better organized, run faster, and yield better results. However, once again as we learned in class there is the problem of constantly inserting guards which may even hamper the write performance. We must call into question the results the authors of the paper presented.