Summary of The Log-Structured Merge-Tree (LSM-Tree)

By S. Xiao Fernández Marín

1 Summary

The Log-Structured Merge-tree (LSM-tree) is a disk-based data structure that keeps the most recent entries in memory. They are also useful as they keep the most recent entries in memory as they are optimized to provide low-cost indexing for a file experiencing a high rate of loads (inserts, updates and deletes), so the lookups are slower (but faster than table scans).
It also uses an algorithm that defers, which means that do not write immediately - just when there is time, and it does it in a sequential batch instead of random disk access.

The LSM-trees are composed of different trees. The most common is with two trees. One in disk (with a B-tree structure, C_1) and another one in memory (C_0), smaller than the C_1. The new rows are added in C_0 and after that, it will migrate to C_1.
For accessing C_1 (with cost, as it has to read from disk), first it has to look in C_0 (with no cost). Moving from C_0 to C_1 has also a cost, as it can batch rows and do sequential writes.

For migrating entries to C_1 in a two-component LSM-tree, a rolling merge process is used, a process is seen in Figure 1. The C_0 tree merges entries and C_1 creates a new leaf (added on the right side). The left side just gets deleted in block (with a clean up job).

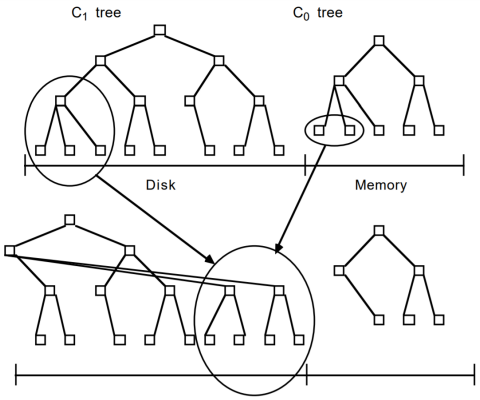

Figure 1: Rolling merge process steps

With closer examination, C_0 is optimized for in-memory so there is no need to store the nodes with the size of disk blocks, and everything is stored from left to right. The lowest values are moved to C_1 when merge, which stores the data on disks so they have more similarities with B-trees and instead of updating the nodes, it gets new versions of them as it uses sequential write, while readings can be random or sequential.

Sometimes C_1 is too big so the solution to this is to have more than two components LSM-trees. The C_0 tree will still have the same size as it is in memory, but for it not having a big different size than C_1, they put multi-components $C_1, C_2, ..., C_{k-1}, C_k$ so the merge does not take a long time.

To find some components in an LSM-tree, the algorithm starts looking in the C_0 tree and if does not find anything, it starts looking on the disk. When finding the goal, it stops, as there is just one value in the whole LSM-tree. New data is inserted in the left side of the C_0 tree. For deleting a value, if it is not in C_0, the value is inserted as a node entry and later when doing a merge, if it finds the same value in some tree of the disk, it deletes it from there. It does not search on disk by default because it is too expensive.

2 Questions not answered by this text

There is the challenge of the secondary index, whereas the index maintenance has not been systematically addressed,

3 What has changed since this was written

LSM Trees have been the pioneer of almost all NoSQL DB. Nowadays, they have more popularity as we have SSDs cards instead of HDDs, so we have more velocity to write and search (but more data to store in applications). Consequently, there is a need to redesign storage engines in DBMSs such as NOVA, a log-structured file system. Although LSM-trees are still used these days, used in DBs like Apache Cassandra, SQLite, Google Bigtable and HBase between others.