LSM-Trees Under (Memory) Pressure

Link : https://cs-people.bu.edu/mathan/publications/adms22-mun.pdf

SHaMBa achieves better lookup performance when large BFs do not fit in memory. Can we do something similar if we have large index blocks? Give a concrete example to explain why it could work or not.

This paper itself mentioned a strategy which RocksDB supports for partitioning index and filters, where each SST file is organized in smaller partitions and each one has its own index and filter blocks. We can use this approach to create multi-level indexes and load the first level in memory and load the next (Child) based on the first level index values. We can think of it as a B+ tree where first node tells us which is the next page (Node) we must pull from disk, whereas simultaneously we are also discard the other half of the tree. Similarly, we can store fence pointers into multiple blocks and keep the parents in memory. Once we check the bloom filter and gets a True positive, we will look up for the point query in parent and then pull the child based in the given values. This will akin like B+ tree and can easily solve the memory issue for large index blocks. This might also increase both the space amplification and the CPU utilization to allow for finer memory management.

Background

To have a finer granular memory management, systems like RocksDB support partitioned index and filters, where each SST file is organized in smaller partitions and each one has its own index and filter blocks. This approach creates the need for a two-level index and necessitates the use of the index prior to the filter, slightly increasing both the space amplification and the CPU utilization to allow for finer memory management. The design of MBF presented in this paper can co-exist with partitioned filters (if each filter is more than one disk page), however, it can also be used instead of partitioned filters, since it can allow for fine-grained memory usage without increasing space or CPU consumption. With partitioning, the index/filter of an SST file is partitioned into smaller blocks with an additional top-level index on them. When reading an index/filter, only top-level index is loaded into memory. The partitioned index/filter then uses the top-level index to load on demand into the block cache the partitions that are required to perform the index/filter query. The top-level index, which has much smaller memory footprint, can be stored in heap or block cache depending on the cache_index_and_filter_blocks setting.