Student ID: 1505022

Modern key-value stores use LSM-based storage, where unlike traditional index structures, these do not perform in-place updates. Rather, LSM tree first buffers all writes in main memory, and subsequently flushes the buffer as sorted run to disk whenever it fills up, and organizing the disk-runs into a number of levels of increasing sizes. LSM-tree later sort-merges these runs. This design has lots of benefits including superior write performance, high space utilization, tunability, and simplification of concurrency control and recovery. Two well known optimizations used by most LSM-trees include Bloom filters (a probabilistic data-structure designed to improve point lookup costs) and Partitioning. Recent LSM-trees typically organize their memory components using skip-list or B+ tree and organize their disk components using B+ trees or sorted-string tables (SSTables).

In this paper, the authors survey the recent research efforts on improving LSM-trees. The six major issues and some of their optimizations discussed by the authors are as follows:

Write Amplification, where it impacts both the write performance of LSM-tree and also the lifespan of SSDs due to frequent disk writes. Some possible improvements provided in the paper include Tiering, which has lower write amplification than levelling. Authors describe four structures which are variants of partitioned tiering with vertical grouping such as: WriteBuffer(WB) which uses hash-partitioning to achieve workload balance , light-weight compaction tree (LWC-tree) where if a SSTable group contains too many entries, it will shrink the key range after it is merged, PebblesDB (built using Fragmented Log-Structured Merge Tree, uses the idea of guards) and dCompaction which uses the idea of virtual SSTabls and virtual merges to reduce merge frequency. Merge skipping is also discussed as a way to improve write performance where main idea is, if possible, to push some entries directly from level 0 to a higher level by skipping some level-by-level merges then total write cost is reduced. Data skew is exploited where basic idea is to separate hot keys from cold keys in the memory so that only cold keys are flushed to the disk.

Merge Operations is the second issue where the can have negative impacts on the system including buffer cache misses after merges and write stalls during large merges. Some optimizations mentioned include improving merge performance, where VT-tree is described which presents a stitching operation to improve performance. Pipelined merge is also discussed which uses CPU and I/O parallelism. Reducing Buffer Cache Misses is discussed where an approach is to use a smart cache warmup algorithm to fetch new component incrementally, to smoothly redirect incoming queries from old components to new components. Another optimization discussed is Minimizing Writing Stalls, where bLSM proposes a spring-and-gear merge scheduler to minimize write stalls, where basic idea is to tolerate an extra component at each level so that merges at different levels can proceed parallelly.

Hardware is the third issue discussed, where LSM-tree implementation must be such as to fully utilize the underlying platforms. Previously hard disks were the main hardware platforms to be used, but recent underlying platforms include large memory which improves both write and lookup performances, multi-core which includes an optimistic concurrency control approach, SSDs which support efficient random I/Os (unlike traditional hard disks which only supported sequential I/Os) and NVMs which further provide efficient byte-addressable random accesses with persistence guarantees, and attempts to perform native management of storage devices (SSDs/HDDs) to optimize the performance of LSM-tree implementations.

Handling special workloads is the fourth issue, where certain special workloads such as temporal data, small data, semi-sorted data and append-mostly data can also be considered to achieve better performance. Basic LSM-tree structure is adapted to handle special workloads. LHAM more efficiently supports temporal workloads by attaching a range of timestamps to each component to facilitate temporal queries processing by pruning irrelevant components. LSM-trie manages a large number of key-value pairs where each key-value pair is small.