Utilizing Compression for LSM Trees

**Introduction**

Since the invention of the first relational database in the late 1970s, researchers have identified that different architectures and data structures can minimize latency for basic access methods. Given some architecture and set of data structures used for any given database, there is always an inherent performance trade-off across different access methods, such as *reads, writes, and updates.* Furthermore, researchers have also had to balance performance trade-offs for access methods with the amount of space used by the database system, whether that be in memory or on disk. Generally, different architectures and data structures have different trade-offs, for example, a traditional column-store with a B-Tree has optimized reads, sacrificing some of the efficiency for writes and updates.

When analyzing the performance of these access methods, traditionally the bottleneck has been I/O time – the amount of time it takes to move pages of data from disk to memory or to move data within the memory hierarchy (L3 to L2 cache, or L2 to L1 cache). However, with the exponential growth of hardware features such as Solid-State Drives (SSD), and larger Random-Access Memory (RAM) for most machines, I/O it not the only bottleneck we must consider anymore. Central Processing Unit (CPU) costs have grown more expensive with more computationally expensive queries due to having to operate on larger amounts of data (due to the advances in hardware). Latency, the amount of time it takes any system to complete a given operation, is how we are measuring performance for the rest of this paper. For simplicity, we can define latency as the sum of I/O cost and CPU cost for any given operation.

The last component of this tradeoff space is overall data space (either in memory or on disk). When we add new types of auxiliary data structures (such as B+ Trees, Bloom Filter, etc.) we will more data to hold in memory and more data to persist to disk once the database goes offline. Moreover, with the rapid growth of data through the last two decades, database researchers have to be conscientious of the method in which they store data to disk. Minimizing our memory and disk footprint is super beneficial for companies as it reduces their space footprint on the cloud or on their own servers. Based on this landscape of design tradeoffs, we have decided to focus on the following overarching question for this paper: *how can we design a system that minimizes* ***space****, while still maintaining strong read and write performances?*

**LSM Trees**

Since the early 2000s, Log Structured Merge Trees (LSM Trees) have emerged as one of the prominent data structures that are used for large NoSQL systems such as RocksDB, MongoDB, and DynamoDB.