Scaling Storage Engines for 100x Big Data

Niv Dayan

Department of Computer Science
University of Toronto
nivdayan@cs.toronto.edu

1 Introduction

Our society is creating and storing exponentially increasing amounts of data. What is often less thought of are the storage engines that maintain this data and facilitate the extraction of knowledge from it. In the late-2000s, a new class of storage engines emerged that prioritize the efficiency of ingesting new data. They include Google's BigTable, Amazon's DynamoDB, Facebook's RocksDB, as well as Apache Cassandra and HBase. These engines have become indispensable for a wide range of applications, including cloud storage, blockchain, machine learning, etc. Nevertheless, a lingering problem is that their performance deteriorates with respect to the amount of data they store. This, in turn, causes applications running on top to have to spend disproportionately more time, energy, and hardware in order to, say, transact on a blockchain, train a deep learning model, or add a photo to the cloud. This talk will discuss how to allow such storage engines to function more efficiently as the big data that they store continues to grow.

2 Background

LSM-Tree. Modern storage engines streamline new application data into storage (disk or SSD) as small sorted files, which are later merged into larger sorted files. This organization is known as a log-structured merge-tree (LSM-tree). With LSM-tree, merging files more eagerly creates higher overheads for writes but allows for faster queries as there are fewer files to search. This trade-off is controlled by a compaction policy, which dictates which files to merge under which conditions.

Filters. Each file of an LSM-tree is assigned a "filter" in fast memory (DRAM chips). A filter is a compressed approximate representation of a file that takes up little space. Filters can be quickly searched to rule out files that do not contain the target data. Thus, they eliminate unnecessary accesses to slower storage. The more space a filter is assigned, the more accurate it becomes thus allowing queries to rule out the file with a higher probability. An LSM-tree implements a filtering policy to decide how much memory to assign each filter. Together, the filtering and compaction policies govern a three-way trade-off between the overheads of queries, writes and space

3 Research Problem

As with most tree structures, LSM-tree's query and write overheads grow logarithmically with respect to the data size. The intuition is that as the data grows, the number of files that must be queried and merged grows too. In our current era of exponential data growth, logarithmic scalability implies linearly increasing overheads with respect to time. The outcome is rapidly deteriorating performance. While it is possible to offset one of the overheads growing by another (e.g., by merging more eagerly or allocating larger filters to prevent query overheads from increasing), it is impossible with existing designs to keep all three overheads steady at the same time as the data grows. This begs a question: is it possible to achieve sub-logarithmic query and write costs for an LSM-tree, all without hurting space?

4 Talk Content

This talk will discuss a series of papers that tackle the problem of how to better scale performance as the data grows in the context of LSM-trees. The talk can be given at different lengths, from 15 minutes to an hour, depending on the amount of time available. It will cover at least two and at most all of the following papers: Monkey (SIGMOD 2017), Dostoevsky (SIGMOD 2018), LSM-bush (SIGMOD 2019), Rosetta (SIGMOD 2020), Chucky (SIGMOD 2021), and Spooky (VLDB 2022). These papers show how to co-design the LSM tree's compaction policy with its filters in ways that lead to asymptotic improvements in both query and insertion throughput, meaning that performance deteriorates more slowly or not at all as the data grows. This talk will conclude with a vision towards amorphous storage engines and data structures, which self-design to optimize any application workload.