CS2241 Project Status Report: Enhancing LSM Trees with Predictive Modeling

Nico Fidalgo, Puyuan Ye

April 7, 2025

Project Overview

Our project investigates how machine learning can enhance the performance of Log-Structured Merge (LSM) Trees by predicting where data is likely to be found during read operations. Specifically, we aim to reduce the number of Bloom filter queries during point and range lookups by introducing a lightweight predictive model that suggests which levels of the LSM tree should be searched.

In traditional LSM Trees, point queries must traverse from the memtable through all on-disk levels, consulting Bloom filters at each level. This can be costly as the number of levels grows, particularly in large-scale systems. Our central hypothesis is that a predictive model can learn to map a key (or a derived feature such as a monotonically increasing ID) to the most likely level(s) where it resides, thereby reducing unnecessary Bloom filter and disk access.

The novelty of our approach lies in coupling each inserted key-value pair with a globally unique, monotonically increasing ID. Since older data migrates deeper into the tree over time (via compaction), and newer data resides closer to the top, the ID serves as a reliable proxy for data age and level placement. This introduces a learnable pattern between data ID and LSM level that we exploit using a machine learning model.

Current Progress

We have implemented a baseline LSM Tree in C++ that supports:

- A memtable and multi-level SSTable structure with leveling compaction.
- Per-level Bloom filters for efficient lookups.
- Insertion of tombstones to support key overwrites.
- An auxiliary map that records the mapping between keys and their assigned unique IDs.

This baseline provides a working and testable version of a traditional LSM Tree that we will compare against in our experiments.

Planned Work

Our next major goal is to integrate a predictive model that can estimate the LSM level for a given key based on its assigned ID. The model will be pre-trained on key-ID-level mappings collected during the operation of a training LSM Tree configured identically to the test system. Since we will use the same LSM Tree design parameters (e.g., size ratio, compaction policy, number of levels) in both training and deployment, the level distribution patterns learned by the model should generalize directly. This eliminates the need for online training or adaptation, allowing for faster development and stable performance expectations.

At query time (for GET or range operations), the model will produce a small subset of candidate levels based on the key's ID. Bloom filters will then be consulted only at those predicted levels. If a false negative from the model leads to a failed lookup, we will conservatively fall back to querying the remaining Bloom filters to ensure correctness. This hybrid approach is designed to maintain the accuracy of traditional LSM Tree lookups while reducing the average query cost.

We plan to evaluate this predictive LSM Tree along several dimensions:

Model Integration and Evaluation

- Train a decision tree or lightweight neural network to map IDs to likely LSM levels.
- Evaluate prediction accuracy (per-level precision/recall).
- Measure the model's runtime to ensure inference remains constant-time or near constant-time.
- Quantify the impact of false negatives and measure the frequency with which fallbacks are triggered.

Performance Testing

We will evaluate the predictive LSM Tree against the baseline across varying configurations:

- **Query Types:** Random and skewed point lookups, as well as range queries of varying sizes.
- **Data Volume:** Small (few levels), medium (moderate compaction), and large-scale datasets (many levels).
- **LSM Parameters:** Size ratio (2x, 5x, 10x), compaction policy (leveling vs. tiering).
- **Model Architectures:** Simple classifiers (decision trees), versus more expressive models (small MLPs).

Metrics Collected

For each experiment, we will collect:

- Average and percentile latency for GET and range queries.
- Total number of Bloom filter queries per operation.
- Bloom filter false positive rate at each level.
- Model prediction time and accuracy.
- Memory overhead introduced by the predictive model and auxiliary key-ID map.
- Number and cost of fallback operations due to incorrect model predictions.

Research Insights and Expected Contributions

By running the above tests, we aim to empirically and analytically answer the following questions:

1. Under what conditions (data size, number of levels, Bloom filter false positive rate) does a predictive model significantly reduce lookup time?
2. What is the "tipping point" where the model's inference time and potential mispredictions are outweighed by the benefit of reduced Bloom filter checks?
3. How do LSM Tree design parameters (e.g., size ratio, compaction strategy) interact with the predictive model's effectiveness?
4. What are the most appropriate model architectures and hyperparameters for deployment in real systems?
5. When does the predictive model become a liability rather than a benefit?

We also intend to derive an analytical expression for the tipping point based on system parameters such as the model inference time, Bloom filter query cost, false positive rate, and number of levels. This derivation, combined with our empirical benchmarks, will provide strong guidance for future deployment of predictive components in read-optimized storage systems.

Conclusion

Our project proposes a practical and theoretically grounded enhancement to LSM Trees using predictive modeling. With our baseline implementation complete, we are well positioned to begin model integration and experimental evaluation. The results of this work will contribute to the broader literature on learned data structures and provide insights into how hybrid predictive-traditional data systems can deliver better performance at scale.