Enhancing LSM Trees with Learned Components: Bloom Filters and Fence Pointers

Nico Fidalgo, Puyuan Ye

March 13, 2025

1 Research Questions

Log-structured merge trees (LSM Trees) are widely used data structures in modern database systems, offering efficient write performance at the cost of more complex read operations. The efficiency of read operations in LSM Trees heavily depends on auxiliary data structures such as Bloom filters and fence pointers. This project seeks to answer the following questions:

1. Can machine learning-based prediction algorithms be effectively implemented on Bloom filters and fence pointers to practically improve read query efficiency in LSM Trees?
2. What types of learning models yield the best performance improvements for these components?
3. How do different LSM Tree design parameters (size ratio, number of levels, etc.) interact with learned components?
4. Which classes of read queries benefit most from prediction-enhanced components, and are there query patterns where traditional implementations outperform learned approaches?

By exploring these questions, we aim to contribute to the growing body of research on learned data structures and potentially demonstrate practical performance improvements for real-world database workloads.

2 Background and Motivation

LSM Trees organize data across multiple levels, with each level increasing exponentially in size. New data enters a memory buffer (usually called memtable) before being flushed to disk as Sorted String Tables (SSTables) at the first level. When a level reaches capacity, a compaction process merges data into the next level.

For read operations, LSM Trees must search potentially multiple levels to find requested data. To improve efficiency, LSM Trees employ two critical components:

1. **Bloom filters**: Probabilistic data structures that quickly determine if data is definitely not in a level (avoiding unnecessary disk I/O).
2. **Fence pointers**: Metadata that indicates which pages within a level might contain the requested data.

Recent work on "algorithms with predictions" has shown that learned models can enhance traditional data structures by leveraging patterns in the data. For instance, learned indexes [Kraska et al., 2018] have demonstrated significant performance improvements over traditional B-trees by predicting data locations. This project explores whether similar techniques can benefit the critical components of LSM Trees.

The significance of this research lies in the potential to improve read performance in LSM Trees without sacrificing their write efficiency advantages, potentially benefiting numerous database systems that rely on this architecture.

3 Planned Approach

Step 1: Literature Review and System Design

- Read about LSM Tree implementations (RocksDB, LevelDB, etc.)
- Review existing work on learned Bloom filters and prediction-based index structures

Step 2: Implementation

- Implement a baseline LSM Tree in C++ with traditional Bloom filters and fence pointers
- Train machine learning models in Python for Bloom filters and fence pointers

Step 3: Experimentation

- Test different query patterns:
  - Point lookups (random, skewed distribution)
  - Range queries of varying sizes
  - Mixed read/write workloads
- Measure:
  - Query latency (average, percentiles)
  - I/O operations per query
  - Accuracy of predictions
  - Memory overhead of learned components
- Compare:
  - Traditional vs. learned Bloom filters
  - Traditional vs. learned fence pointers
  - Different model architectures and hyperparameters
  - Performance across varying LSM Tree configurations (size ratio, number of levels)
- Analyze results to identify when learned components provide the greatest benefits and when it's better off to not use learners

4 Resources Required

- Computational Resources
- Software and Libraries
  - C++ development environment for LSM Tree implementation
  - Python with machine learning frameworks (TensorFlow/PyTorch) for model development
  - GitHub for open source implementations of LSM Trees and Bloom filters/fence pointers with learning models
- Synthetic data with controllable distributions

5 Backup Plans

MVP 1: Learned Bloom Filters for LSM Trees

- Focus exclusively on implementing and evaluating learned Bloom filters
- Compare against traditional Bloom filters across different false positive rates and memory budgets
- Deliverable: Analysis of when learned Bloom filters outperform traditional implementations

MVP 2: Learned Fence Pointers

- If time permits after MVP 1, implement learned models for fence pointers
- Compare position prediction accuracy and resulting query performance
- Deliverable: Evaluation of learned fence pointer effectiveness

MVP 3: LSM Tree Design Space Exploration

- Analyze how different LSM Tree parameters interact with learned components
- Identify configurations where learned components provide maximum benefit
- Deliverable: Design guidelines for LSM Trees with learned components

MVP 4: Advanced Model Exploration

- Experiment with more sophisticated model architectures
- Investigate transfer learning and adaptation to changing data distributions
- Deliverable: Comparison of model architectures and training approaches

References

Idreos, S., Zoumpatianos, K., Athanassoulis, M., Dayan, N., Hentschel, B., Kester, M.S., Guo, D., Maas, L., Qin, W., Wasay, A. and Sun, Y. (2018). The Periodic Table of Data Structures. Harvard University.

Dayan, N., Athanassoulis, M., and Idreos, S. (2018). Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*.

Dayan, N. and Idreos, S. (2018). The Log-Structured Merge-Bush & the Wacky Continuum. Harvard University.

Dayan, N. and Idreos, S. (2018). Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*.

Dayan, N., Athanassoulis, M., and Idreos, S. (2018). Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems (TODS)*, 43(4):1–48.

Broder, A. and Mitzenmacher, M. (2004). Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509.

Kraska, T., Beutel, A., Chi, E.H., Dean, J., and Polyzotis, N. (2018). The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM.

Mitzenmacher, M. (2018). A Model for Learned Bloom Filters and Optimizing by Sandwiching. *Advances in Neural Information Processing Systems*, 31.

Dai, Z. and Shrivastava, A. (2020). Adaptive Learned Bloom Filter (Ada-BF): Efficient Utilization of the Classifier with Application to Real-Time Information Filtering on the Web. *Advances in Neural Information Processing Systems*, 33.

Rae, J.W., Bartunov, S., and Lillicrap, T.P. (2019). Meta-learning neural Bloom filters. *International Conference on Machine Learning*.