Optimizations in Part A - Storage Engine

Kumar Shresth 24CS60R51

March 31, 2025

1 Overview

This document outlines the optimizations implemented in the storage engine to enhance read performance. Since the workload is read-heavy (frequent `GET` requests with fewer `SET` and `DEL` operations), the goal was to minimize latency for reads.

2 Optimization Techniques

The following techniques were applied:

- **Caching Frequently Accessed Keys:** A Least Recently Used (LRU) cache was introduced to store hot keys in-memory, preventing redundant disk lookups.
- **File Access Optimization:** Instead of scanning `evicted_data.txt` line by line, a key-to-offset index was created for direct file access.
- **Structured Binary Storage:** Evicted data was stored in a structured binary format instead of plain text, reducing read overhead.

3 Details of Implementations

3.1 LRU Caching for Reads

- Implemented an LRU cache using:
  - `cacheMap`: A `std::unordered_map` storing key-value pairs.
  - `cacheList`: A `std::list` to maintain LRU order.
- When a key is accessed:
  - If found in `cacheMap`, return instantly.
  - If retrieved from `evicted_data.txt`, add it to the cache.

3.2 Optimized File Access Using Indexing

- Instead of scanning the eviction file line by line, an index (`fileIndex`) was maintained.
- The `fileIndex` stores mappings of keys to file offsets, allowing direct seeks instead of sequential reads.
- This significantly reduces retrieval time for evicted keys.

4 Conclusion

These optimizations ensure that `GET` operations are significantly faster, reducing unnecessary disk I/O and improving overall system performance.