Design Document for Key-Value Based Storage Engine

1. Overview

BLINK DB Part 1 implements a simple key-value storage engine in C++ that supports the following operations:

- **SET <key> <value>**: Stores the value under the specified key.
- **GET <key>**: Retrieves the value associated with the key.
- **DEL <key>**: Deletes the key-value pair.

In addition, the engine uses an LRU (Least Recently Used) eviction policy to prevent memory overflow. When the engine reaches its capacity, it flushes the least recently used data to disk and later restores it if needed.

This part of the project is optimized for a **balanced workload**, meaning that GET and SET operations occur at comparable rates—a scenario common in many real-world applications.

2. Workload Optimization

Workload:
Balanced (optimized for both GET and SET operations).

Reasoning:

- Many real-world applications involve a mix of reads and writes.
- A balanced approach ensures that neither reads (GET) nor writes (SET/DEL) become a bottleneck.

3. Data Structures

Primary Data Structure: Hash Table

- **Implementation:** `std::unordered_map`

- **Pros:**
  - Provides average O(1) time complexity for GET, SET, and DEL operations.
  - Simple and well-suited for in-memory key-value storage.
- **Cons:**
  - Can use high memory if the key-space is large.
  - Does not maintain any order of keys.

Secondary Data Structure: LRU Cache

- **Implementation:**
  - A `std::list` to maintain the order of key usage.
  - An auxiliary `std::unordered_map` mapping keys to their corresponding iterator in the LRU list.
- **Purpose:**
  - The LRU cache improves GET performance for frequently accessed keys.
  - It also manages memory by determining which keys should be evicted when capacity is reached.
- **Eviction Policy:**
  - When the in-memory store reaches its capacity, the key at the back of the LRU list (the least recently used) is flushed to disk and removed from memory.
  - If that key is later requested, it is restored from disk.

4. Optimizations & Trade-offs

LRU Cache & Eviction Policy

- **Optimization:**
  - Ensures that frequently accessed keys remain in memory, thereby speeding up GET operations.
  - Prevents memory overflow by evicting the least recently used key.
- **Trade-off:**
  - The additional bookkeeping (maintaining the list and iterator map) consumes extra memory and introduces some overhead during GET and SET operations.

Persistent Disk Storage

- **Optimization:**
  - When evicting a key, the engine flushes its value to a disk file.
  - If a key is requested later and it’s not found in memory, the engine can restore it from disk.

- **Trade-off:**
    - Disk I/O is slower than memory access, so there is a performance hit when restoring keys.
    - This design assumes that the rate of evictions is low relative to the overall workload.

Thread Safety and Concurrency

- **Implementation:**
    - The storage engine uses a `std::recursive_mutex` to protect shared resources.
- **Reasoning:**
    - Although Part 1 primarily demonstrates functionality via a REPL, the code is designed to be thread-safe to allow future extension (such as integration with a network server).
- **Trade-off:**
    - Mutex locks may introduce some overhead in highly concurrent environments, but they are necessary to prevent race conditions.

5. Implementation Details

Command Processing and REPL

- The engine provides a simple REPL (Read-Eval-Print Loop) that accepts commands:
    - `SET <key> <value>`
    - `GET <key>`
    - `DEL <key>`
- The REPL processes user commands and calls the corresponding methods on the storage engine.

Capacity Management and Data Restoration

- **Eviction:**
    - The engine checks its capacity on each SET operation.
    - If capacity is reached, it evicts the least recently used key, flushes its data to disk, and logs the operation.

- **Restoration:**
    - When a GET command is received for a key not in memory, the engine attempts to restore it from disk.
    - If the key is found on disk, it is re-inserted into the in-memory store and its usage is updated in the LRU cache.

6. Compilation and Execution

Compilation:`g++ part1.cpp -o part1`
Execution:`./part1 part1`

7. Summary

This design for Part 1 of BLINK DB uses a hash table for efficient storage, enhanced by an LRU cache mechanism for memory management. The balanced workload optimization ensures that both GET and SET operations are performed efficiently. Thread safety is maintained using mutexes, and persistence is achieved by flushing evicted keys to disk. The design decisions were made to meet the needs of a balanced real-world workload while keeping the implementation straightforward and extensible for future network integration.