CS 744 : DECS Project : Phase 1

Key-Value Server With Database and LRU Cache Integration

[Github Repository]

System Overview

This project implements a **client–server key–value storage system** that provides both **fast in-memory access** and **persistent database storage**. The system is designed to efficiently handle frequent read and write operations using an **LRU (Least Recently Used) cache** for hot data and **PostgreSQL** for long-term persistence.

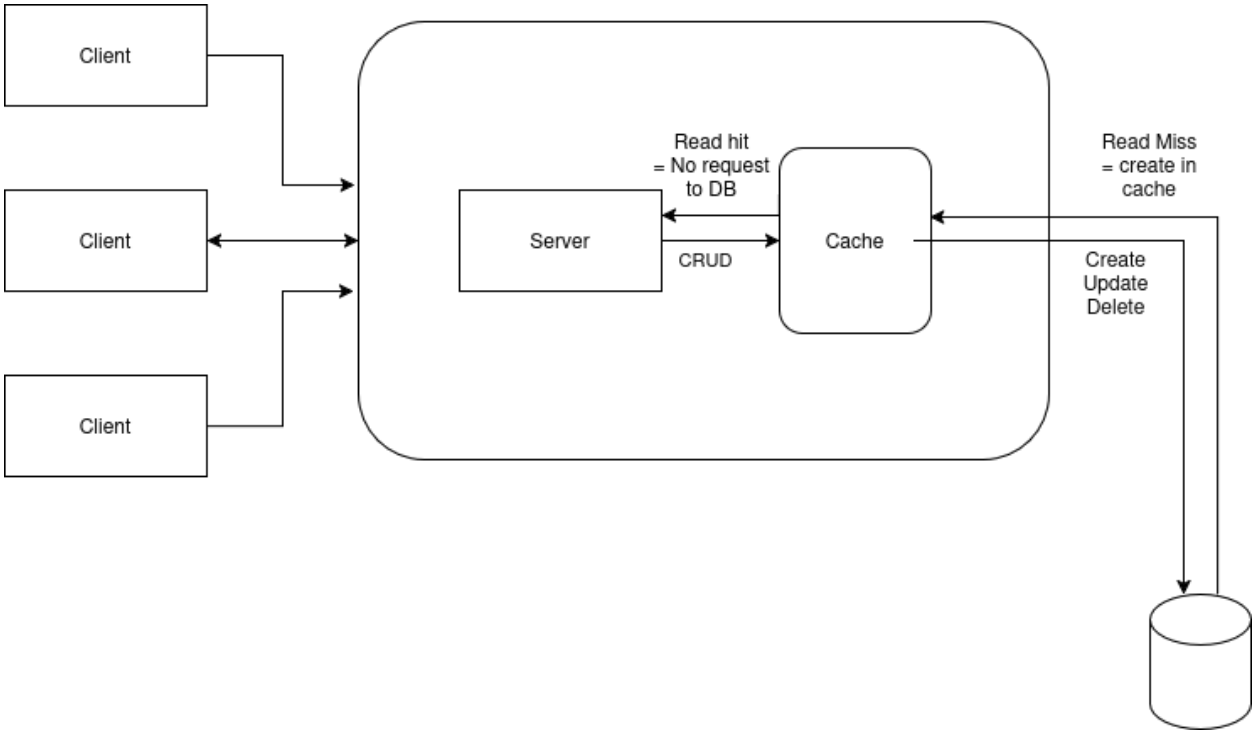
System Architecture

The architecture follows a modular design consisting of three core components:

1. **Client**
   The client module provides the interface for users or external applications to send key–value operations (e.g., `GET`, `PUT`, `DELETE`) to the server. It communicates over TCP sockets using a simple request–response protocol.

2. **Server Core**
   The server listens for client connections, parses incoming commands, and coordinates data access between the cache and the database.

   - On a **read request**, it first queries the **LRU cache**. If the key exists in the cache, the value is returned immediately.
   - On a **cache miss**, it fetches the value from **PostgreSQL**, updates the cache, and responds to the client.
   - On a **write request**, it updates both the cache and the database to maintain consistency.



3. **LRU Cache Module**
   This in-memory cache is implemented using a combination of a **hash map** and a **doubly-linked list**, enabling O(1) access, insertion, and eviction operations. When the cache reaches capacity, the least recently used item is evicted automatically.

4. **Database Layer (PostgreSQL)**
   Acts as the persistent storage backend. All key–value pairs are stored in a `kv_table` with schema `(key TEXT PRIMARY KEY, value TEXT)`. This ensures data durability across restarts and enables recovery from cache losses.

Data Flow Summary

1. **Client → Server:** Sends request (e.g., `PUT key value` or `GET key`).
2. **Server → Cache:** Checks or updates the LRU cache.
3. **Server → Database:** Syncs persistent data if necessary.
4. **Server → Client:** Sends back the response.

Key Features

- **Hybrid storage:** Combines in-memory speed with persistent reliability.
- **Scalable and modular:** Components can be extended or replaced independently.
- **Efficient caching:** LRU policy ensures frequently accessed data remains hot.
- **Crash recovery:** Persistent PostgreSQL storage guarantees data integrity.