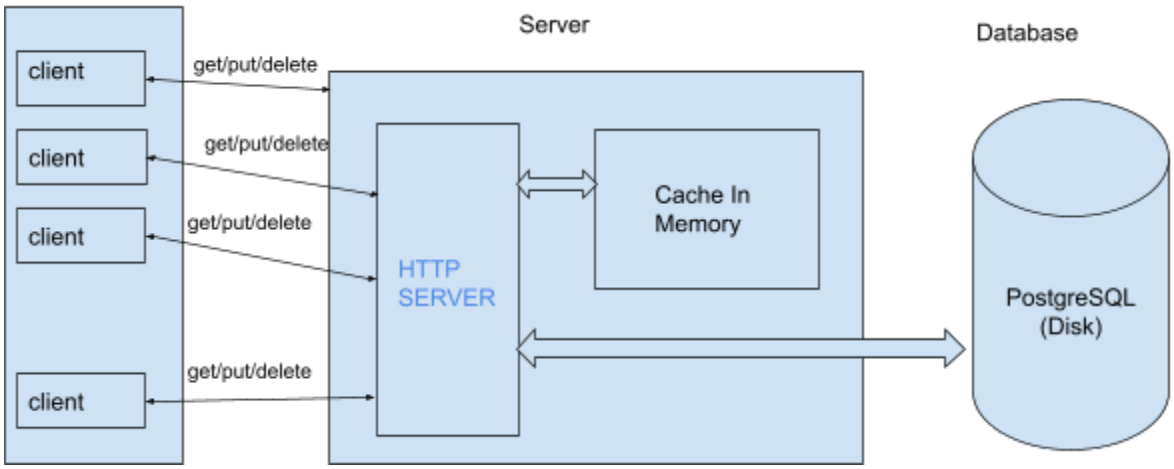
HTTP-based Key-Value Server

Roll no- 25m0750
Paper code-CS744 Autumn 2025

Description:
A high-performance, multi-threaded **HTTP Key-Value Store** in C++. It uses **PostgreSQL** for persistent storage and an in-memory **LRU (Least Recently Used) Cache** to speed up data retrieval.



1. Architectural Overview

The server acts as a middleware between HTTP clients and a PostgreSQL database.

- **Clients** send HTTP requests (GET, PUT, DELETE) to endpoints like `http://localhost:8080/kv/myKey`.
- **Server** checks its fast in-memory cache first.
- **Database** (PostgreSQL) is used for persistent data.

2. Core Components

A. `PGPool` (Database Connection Pool)

Connecting to a database is expensive and slow. This class maintains a pool of open connections to PostgreSQL to be reused by different threads.

- **Initialization:** On startup, it opens `pool_size` (default 4) connections.
- **Preparation:** It creates the `kv_store` table if it doesn't exist and "prepares" SQL statements (`kv_get`, `kv_put`, `kv_del`) for efficiency.
- **Thread Safety:** Uses a `mutex` and `condition_variable` to allow multiple HTTP worker threads to safely `acquire()` and `release()` connections.

B. `LRUCache` (In-Memory Cache)

This is a classic implementation of a Least Recently Used cache. It keeps the most frequently accessed data in RAM for instant retrieval.

- **Mechanism:** It uses two data structures simultaneously:
    1. `std::list`: Maintains the order of use. The front is the Most Recently Used (MRU), and the back is the Least Recently Used (LRU).
    2. `std::unordered_map`: Provides O(1) fast lookups, mapping a "key" to an iterator pointing to that item's location in the `std::list`.
- **Eviction Policy:** When the cache is full and a new item is added, it removes the item at the back of the list (the LRU item) to make space.
- **Thread Safety:** Protected by a `std::mutex` because multiple HTTP threads might try to read/update the cache at the same time.

C. `KVHandler` (HTTP Request Handler)

This class connects the web server (`CivetServer`), the database (`PGPool`), and the cache (`LRUCache`).

- **Cache Warming:** When the server starts, `warmUpCache()` queries the *entire* database and loads it into the cache (up to the cache's max size). This ensures the cache is useful immediately upon startup.
- **Request Routing:** It parses URI paths (e.g., extracting `myKey` from `/kv/myKey`) and routes them to `doGet`, `doPut`, or `doDelete`.

---

3. How Caching is Implemented

The code uses standard caching strategies to ensure data consistency between memory and the database.

Read Path (GET Request) - "Cache-Aside" / "Read-Through"

When a client requests a key:

1. **Check Cache:** The handler first asks the `LRUCache`.
    - *HIT:* If found, returns the value immediately (super fast). The cache marks this item as "most recently used."
2. **Check DB (Miss):** If not in cache, it acquires a DB connection and queries PostgreSQL.
    - *Not found in DB:* Returns 404.
    - *Found in DB:*
        1. Value is fetched from DB.
        2. Value is **inserted into the Cache** so future requests for this key will HIT.
        3. Value is returned to the client.

Write Path (PUT Request) - "Write-Through"

When a client updates or inserts a key:

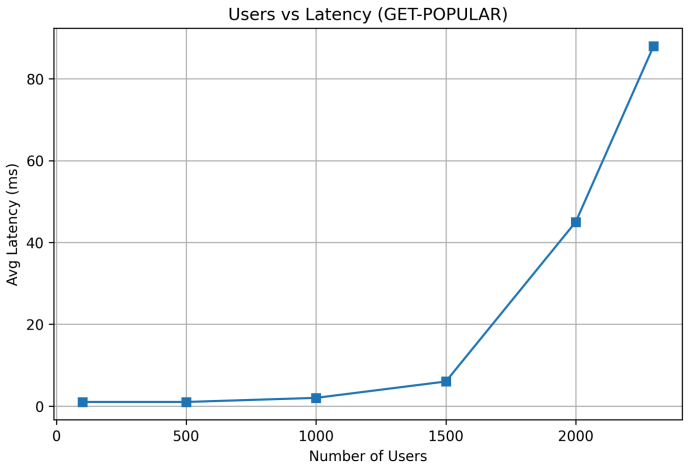
1. **Write to DB:** It first persists the data to PostgreSQL to ensure durability.
2. **Update Cache:** If the DB write succeeds, it immediately updates (put) the item in the `LRUCache`. This ensures subsequent reads get the new value.

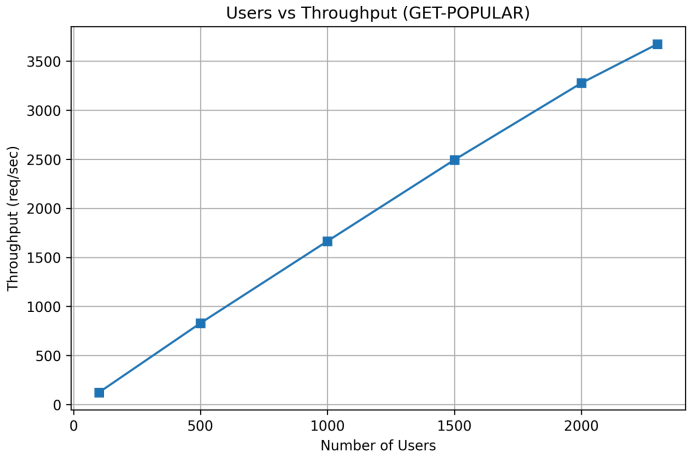
Delete Path (DELETE Request)

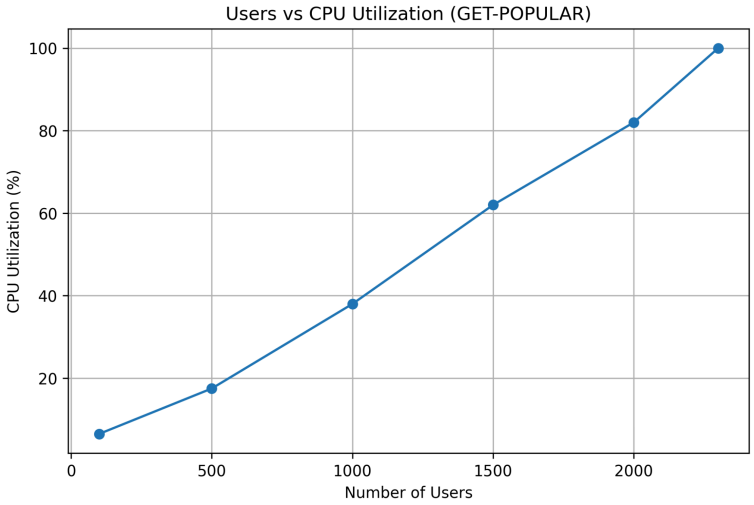
1. **Delete from DB:** Removes the row from PostgreSQL.
2. **Invalidate Cache:** Calls `cache_.erase(key)` to remove it from memory, ensuring no one reads stale data.

Results:

Workload GET-popular:





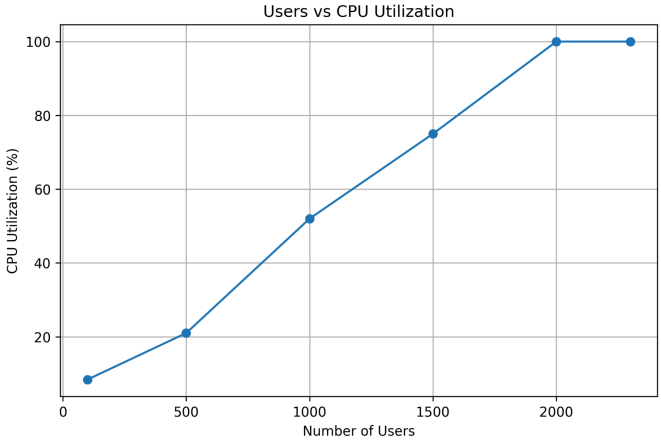


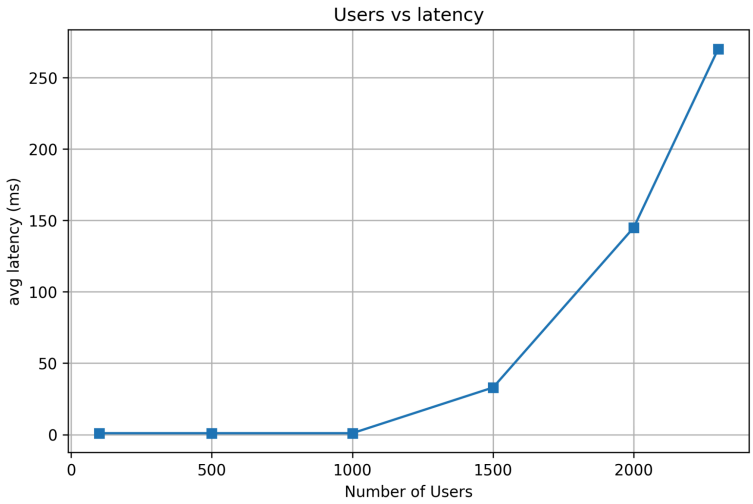
This workload access data from cache only , no io over head.
For this workload cpu in server is bottleneck

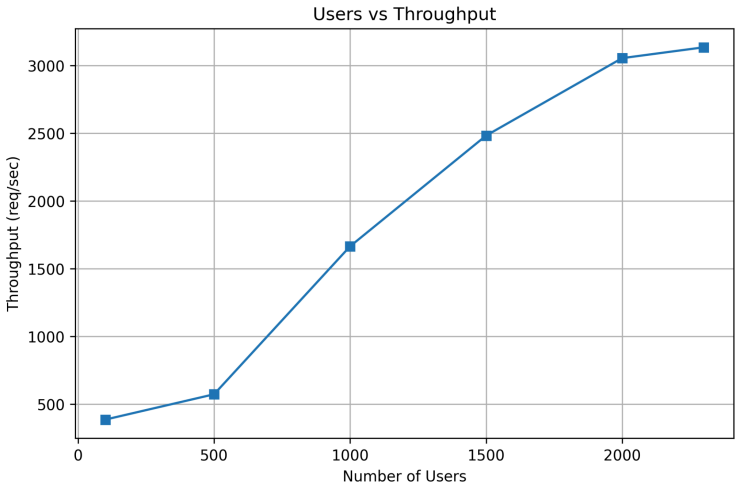
CPU Utilization Trend

- CPU increases steadily from 8% → 100%.
- System becomes fully saturated at 2300 users

Workload get-all:





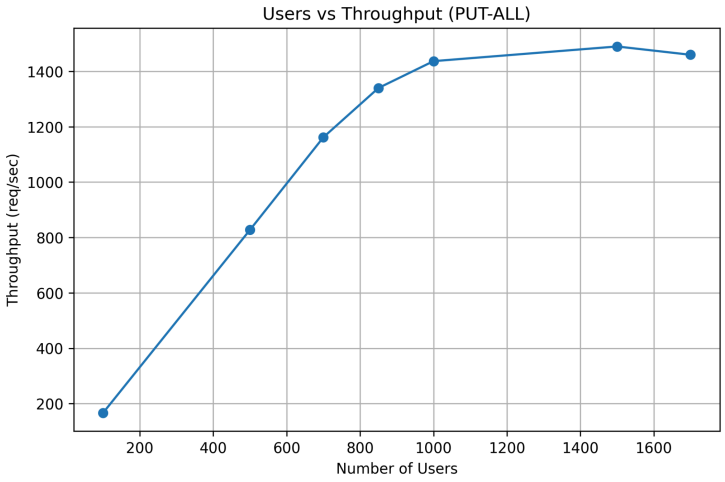


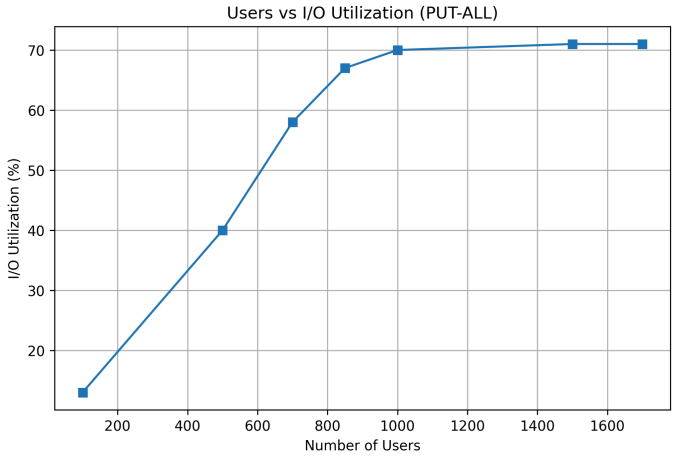
This workload simulates get request from cache if miss then request served by database , only read overhead. But no write in db table;

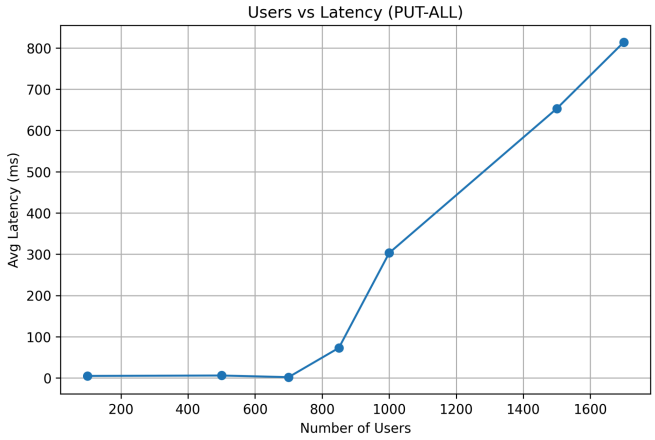
CPU Utilization Trend

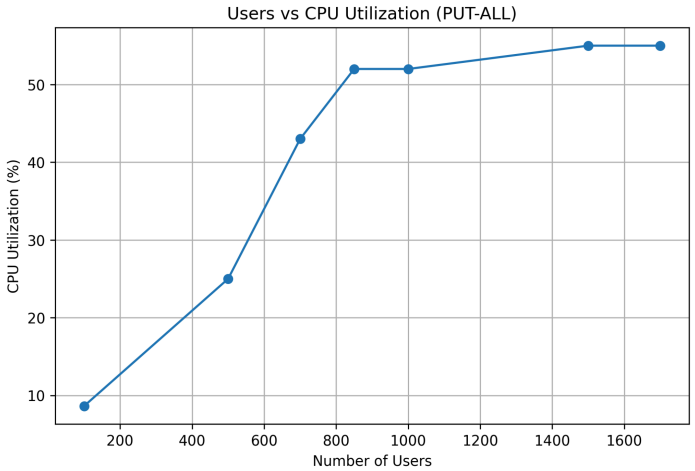
- CPU increases steadily from 8% → 100%.
- System becomes fully saturated at 2000 users

Workload put_all:









This workload simulates only put request ,served by database , read overhead for finding the key and write value on db table;

CPU Utilization Trend

- CPU is not the bottleneck as only 50-55% is utilized.
- I/O operation in database is main bottleneck here. After 70 % utilization saturates its capacity.

Github Link:

https://github.com/Blooming2003sun/decs_kvserver