Real-Time Leaderboard System

Name: Abhay Kumar Mittal

Roll Number: 22M0822

Course: CS744 - Design and Engineering of Computing Systems

Instructor: Prof. Mythili Vutukuru

GitHub Repository:
https://github.com/MitAbhay/DECS_Project

Date of Submission: November 7, 2025

1 Overview

This project implements a real-time leaderboard system that handles score updates and top-rank queries under different performance conditions. The key goal is to demonstrate how execution paths differ when working with data entirely in memory versus interacting with a persistent disk-backed database. In high-performance systems such as gaming platforms, financial leaderboards, or real-time scoring dashboards, both speed and durability must be considered. This project showcases these trade-offs clearly by supporting three distinct operational modes: Cache-only, Database-only, and Hybrid mode. The system also includes built-in request logging to capture latency, cache hit rates, and data path usage, providing valuable insight for performance evaluation.

2 System Components

HTTP Server

The HTTP server is implemented in C using the `libmicrohttpd` lightweight web server library. It handles incoming REST API requests and directs them through appropriate execution paths based on the configured mode. The server parses parameters, performs score updates, retrieves leaderboard data, and formats responses in JSON. It also measures latency per request, which helps in comparing performance across modes. The server acts as the central coordinator of the system, connecting clients to either the cache or the database backend.

In-Memory Cache

The cache is implemented as a simple array-based structure in memory that stores player IDs and scores. Accessing data from RAM provides very low latency and makes leaderboard retrieval significantly faster. However, because this data is not persisted, any server restart results in full data loss. The cache is most useful in systems where speed is important and persistence is either optional or handled through periodic checkpointing. In hybrid mode, the cache supplements the database by serving leaderboard reads extremely quickly while still maintaining data consistency.

PostgreSQL Database

The PostgreSQL database stores leaderboard data persistently, ensuring durability and consistency. Every write in DB-only or Hybrid mode commits new scores to disk, making data available across system restarts. However, database access is slower than RAM due

to disk I/O and query processing overhead. This demonstrates how persistent storage introduces latency. In real-world deployments, indexing, query tuning, and caching layers like Redis are often used to mitigate latency concerns, mirroring the exact behaviors showcased in this project.

Load Generator (loadgen)

The load generator is a separate client tool designed to send high-frequency concurrent requests to the server. It simulates real-world usage scenarios, making it possible to observe system performance under load. By running tests across different modes, we can gather throughput, latency, and cache hit/miss data. This enables direct comparison between CPU-bound and I/O-bound execution paths. The load generator plays a crucial role in validating performance trade-offs and makes the project evaluative rather than purely functional.

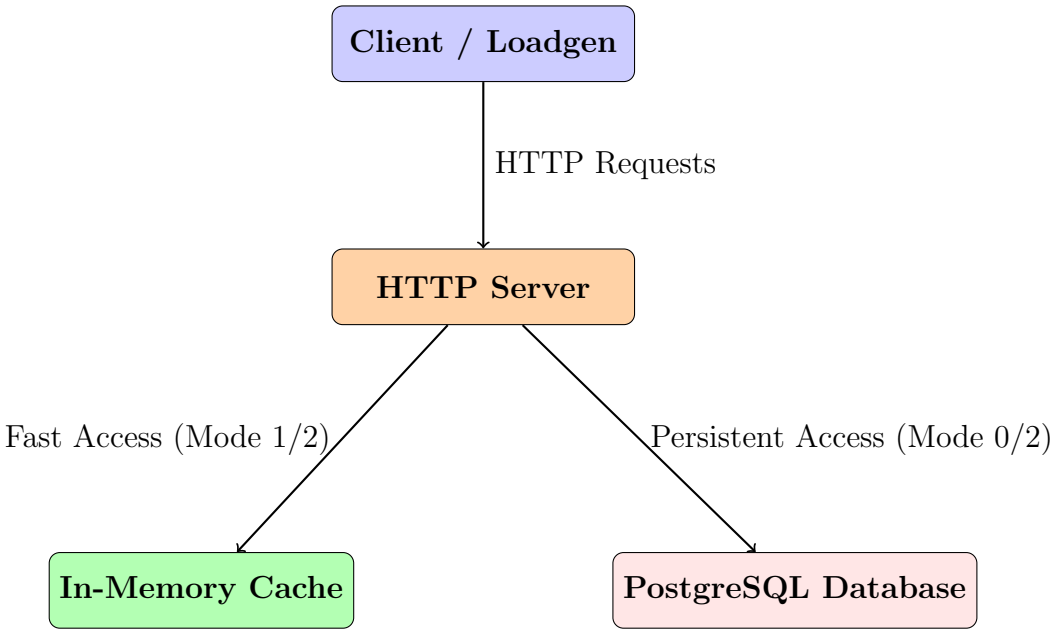
3 Execution Modes

- **Mode 0 (DB-only):** All read and write operations go directly to PostgreSQL. This mode ensures full durability and accuracy of results regardless of server failure. However, it also introduces high latency due to disk access and SQL execution. This mode represents typical persistent storage-based systems and is useful for observing I/O-bound bottlenecks.

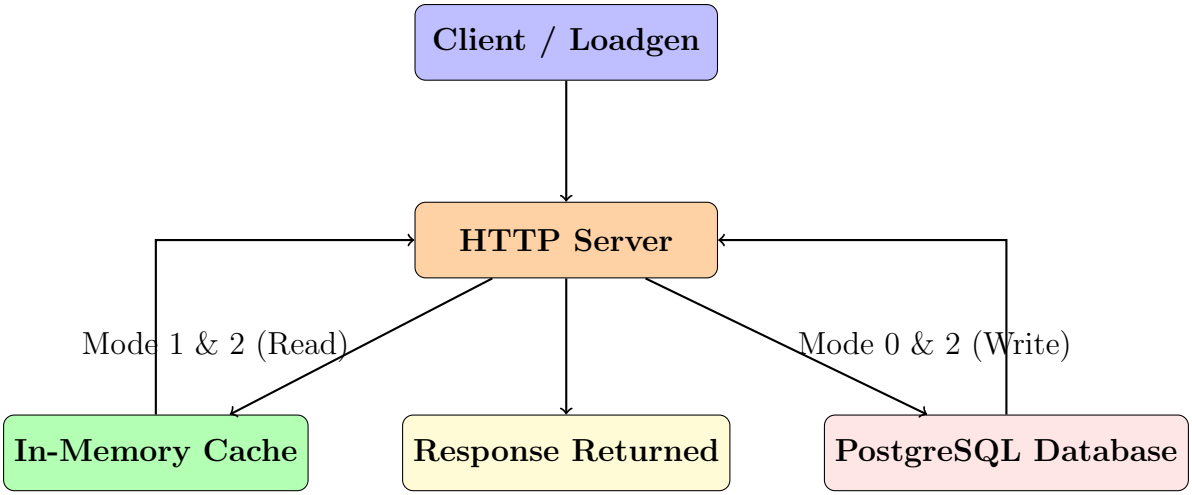
- **Mode 1 (Cache-only):** All operations happen entirely in RAM. This mode offers the fastest performance with extremely low latency, making it ideal for performance-critical environments. However, no writes are persisted, meaning data is lost on restart. This mode is useful to demonstrate CPU-bound execution and the trade-off between speed and durability.

- **Mode 2 (Hybrid):** Write requests update both cache and database, ensuring persistence, while read requests are served from cache for low latency. This mode provides a real-world balanced approach similar to how modern systems use Redis alongside SQL databases. It achieves high performance without sacrificing durability.

4 Architecture



5 Data Flow



6 Execution Paths: Memory vs Disk

This project is designed to clearly demonstrate how the same API request can follow two different execution paths depending on where the data is stored. The two request types used to show this are:

- **/update_score** – modifies the score of a player.
- **/leaderboard** – retrieves the top–N players.

These requests behave differently across the three modes:

Example Request: Update Player Score

```
curl -X POST "http://127.0.0.1:8080/update_score?player_id=7&score=350"
```

- **Mode 0 (DB-only):** The server sends an SQL query to PostgreSQL. This involves disk access, transaction processing, and is therefore **I/O-bound**.
- **Mode 1 (Cache-only):** The score is updated directly in the in-memory array. There is no disk or database interaction, making this **CPU-bound and extremely fast**.
- **Mode 2 (Hybrid):** The score is written to both the cache *and* the database. This mode ensures durability while keeping reads fast.

Example Request: Retrieve Leaderboard

```
curl "http://127.0.0.1:8080/leaderboard?top=5"
```

- **Mode 0 (DB-only):** Runs a SQL query to fetch and sort scores from PostgreSQL. Sorting and I/O make this the **slowest path**.
- **Mode 1 (Cache-only):** Scores are sorted in memory and returned immediately. This path demonstrates **pure CPU-bound processing**.
- **Mode 2 (Hybrid):** Reads come from the cache, so the leaderboard is returned **as fast as Mode 1**, while data remains persistent due to the write-through database updates.

Key Insight

These two request types directly demonstrate the performance difference between:

- **Memory access (nanoseconds–microseconds)** — fast, but volatile.
- **Disk/DB access (milliseconds)** — persistent, but slower.

This fulfills the core learning objective of evaluating **CPU-bound vs I/O-bound execution paths**.

7 Repository Link

https://github.com/MitAbhay/DECS_Project