Database Management System Laboratory

Distributed Databases

G M M Prabhash	**22CS30027**
Sadda Suchith Reddy	**22CS10063**
Emandi Devi Lakshman	**22CS10022**
Sai Deepak Reddy Mara	**22CS10066**
Gedda Sai Shasank	**22CS10025**

Team Titans, Department of Computer Science and Engineering, IIT Kharagpur

Course Instructors: Prof. Pabitra Mitra, Prof. K. S. Rao

Abstract—This project presents the design and implementation of a scalable, distributed load balancer for sharded databases using consistent hashing. The system ensures fault tolerance with heartbeat monitoring and automatic replica spawning, and is built using Docker for container orchestration and Quart for asynchronous client handling. We demonstrate effective data distribution, dynamic replica management, and strong recovery mechanisms, making the system suitable for real-world distributed environments.

Keywords—*Distributed Databases, Load Balancer, Consistent Hashing, Quart, Docker, Fault Tolerance, Sharding, Replication*

1. Problem Statement

Objective

The objective of this assignment is to design and implement a scalable, distributed database system that leverages sharding and replication to achieve high performance, fault tolerance, and elasticity. The system architecture must utilize Docker containers to simulate a cluster of database servers, each responsible for managing one or more shards—logical partitions of the main table (`StudT`: `Stud_id`, `Stud_name`, `Stud_marks`). Each shard must be replicated across multiple server containers to enable parallel read operations and ensure data availability in the event of server failures.

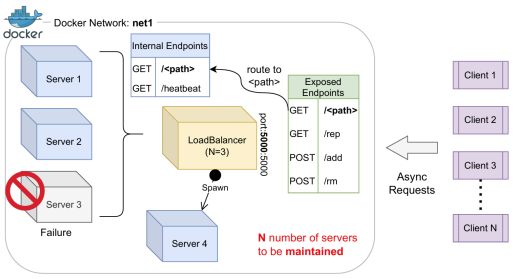


Figure 1. Load Balancer Architecture

A central load balancer component must be implemented to coordinate the distribution of client requests among the available server containers. This load balancer is responsible for:

- Maintaining metadata tables that map `Stud_id` ranges to shard identifiers and track the placement of shard replicas across server containers.
- Employing a consistent hashing algorithm with virtual nodes to evenly distribute requests and minimize data movement during scaling events.
- Routing read requests in parallel to all available replicas of a shard, thereby maximizing throughput.
- Ensuring strong consistency for write, update, and delete operations by synchronously propagating changes to all replicas of the relevant shard, using mutex locks to serialize concurrent writes to the same shard.
- Dynamically handling the addition and removal of server containers and shards, including automatic data redistribution and recovery from failures by copying shard data from surviving replicas.

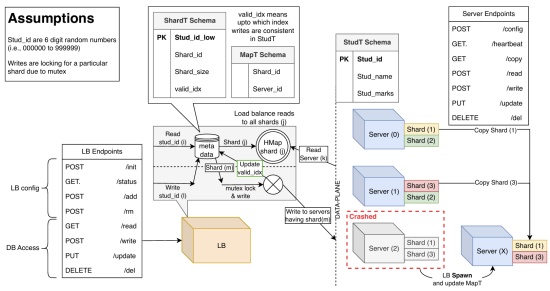


Figure 2. Sharded and Replicated Database Architecture

The system must expose a comprehensive set of HTTP REST API endpoints for:

- Initializing and configuring of the distributed database, including schema definition, shard partitioning, and server-shard assignments.
- Adding or removing servers and shards at runtime, with appropriate validation and error handling.
- Performing CRUD operations (Create, Read, Update, Delete) on student records, supporting range queries and ensuring atomicity and durability of write operations.
- Monitoring system health and status, including real-time reporting of server and shard configurations.

The implementation should provide robust fault tolerance by detecting server failures through heartbeat mechanisms and automatically spawning replacement containers, restoring their shard data from existing replicas. Performance analysis must be conducted to evaluate the system's scalability, read and write throughput under varying configurations (e.g., increasing the number of shards, replicas, and servers), and the effectiveness of the load balancer in distributing requests and recovering from failures.

The solution must be developed using Python (or another approved language), containerized with Docker, and include all necessary orchestration scripts (`Dockerfiles`, `docker-compose`, `Makefile`) for reproducible deployment. The codebase should be well-documented and version-controlled, with a `README` detailing design decisions, testing methodology, and performance observations. The final system should demonstrate the principles of distributed databases, including horizontal scaling, high availability, and efficient load balancing using consistent hashing and sharding techniques.

2. Methodology

Architecture Overview

Our system architecture is designed for scalability, fault tolerance, and efficient load distribution. Clients interact with the system via asynchronous HTTP requests sent to a central load balancer. The load balancer handles routing using consistent hashing to ensure minimal key reassignment during dynamic server changes.

The consistent hash ring uses 512 slots with 9 virtual nodes per physical server to enhance uniformity in distribution. Each server container exposes minimal REST endpoints such as `/home`, `/heartbeat`, and data-specific routes, enabling health checks and processing of client queries.

Implementation Details

- **Language & Framework:** The backend is implemented in Python using the Quart asynchronous web framework, leveraging `asyncpg` for PostgreSQL communication and `aiohttp` for internal HTTP requests.
- **Containerization:** Docker is used to encapsulate each server and database shard. Docker Compose defines the complete topology and facilitates network isolation and service orchestration.
- **Data Management:**
  - `StudT` table is partitioned across shards using modulo-based sharding logic.
  - `ShardT` maintains shard-server mappings.
  - `MapT` tracks virtual to physical node mappings for consistent hashing.
- **Fault Tolerance:** Server containers periodically send heartbeat signals to the load balancer. If a server misses successive heartbeats, the system automatically respawns a new replica and reintegrates it into the hash ring.

Important Internal Functions: Heartbeat and Flatline

To maintain high availability and ensure fault tolerance in the distributed system, the load balancer internally uses two key monitoring functions — `get_heartbeat` and `flatline`.

1. `get_heartbeat`

The `get_heartbeat` function is a periodic monitoring routine that sends a lightweight HTTP request (heartbeat) to each server container at regular intervals of t seconds. The purpose is to confirm that the server is alive and functioning. If the server responds, it is considered healthy and remains active in the consistent hash map.

- Monitors all registered server instances.
- Executes every t seconds (as defined in system parameters).
- If a server responds with a valid status code, it is marked as active.

2. `flatline`

The `flatline` function works in conjunction with `get_heartbeat` to detect and respond to server failures. If a server fails to respond to heartbeat requests for N consecutive intervals (set to $N = 5$ in our implementation), it is presumed to have failed.

Upon detection, the `flatline` function performs the following steps:

- Removes the failed server and its virtual nodes from the consistent hash map.
- Launches a new Docker container to act as a replacement server.
- Retrieves the shard data that the failed server was responsible for from other existing replicas.
- Initializes the new container with the copied shard data and updates the metadata tables.

This proactive fault recovery mechanism ensures that the system can quickly self-heal without manual intervention, maintaining data availability and consistency even in the event of node failures.

2.1. Consistent Hashing

Consistent hashing is a strategy used to distribute data or requests across a set of servers in a way that minimizes data movement when servers are added or removed. Unlike traditional hashing, where a small change in the number of servers leads to major reshuffling of keys, consistent hashing organizes keys and servers on a circular hash ring, ensuring only a small portion of keys need to be remapped during such changes.

Basic Structure (Without Virtual Servers):

- The consistent hash map is represented as a circular ring of $M = 512$ slots.
- Each client request is assigned a request ID (Rid), which is mapped to a slot using a hash function: $H(i) = i^2 + 2i + 17$.
- Each physical server has a unique ID (Sid) and is also placed on the ring using the hash function: $\Phi(i) = i^2 + 25 \mod M$.
- Requests are routed to the first server encountered while moving clockwise from their hashed slot.

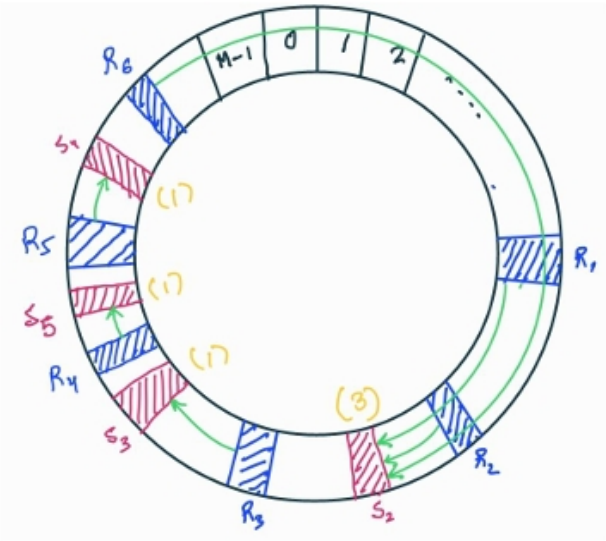


Figure 3. Fig. 2: Data Structure to implement a Consistent Hash Map

This approach offers high availability and easy scalability. However, it has a drawback: if servers are unevenly distributed or if a server fails, the load may become highly imbalanced.

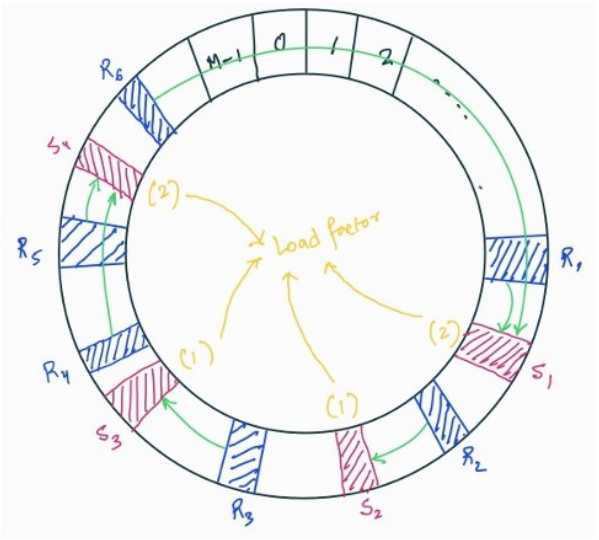
Challenges without Virtual Servers:

- Uneven distribution of requests, especially with a small number of servers.
- In case of a server failure, all requests mapped to that server shift to the next server, creating hotspots.

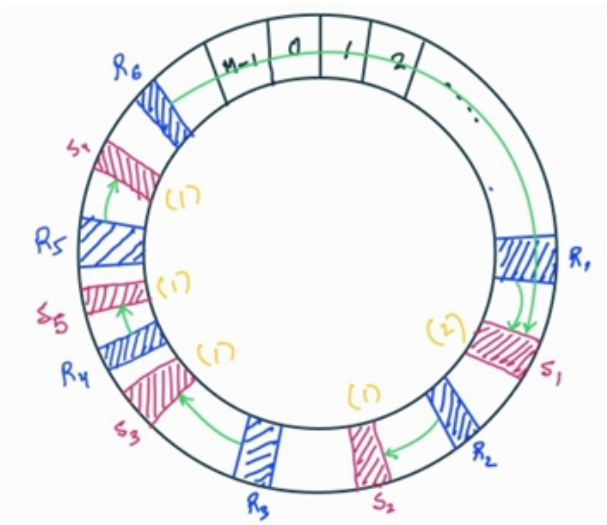
Introduction of Virtual Servers:

To overcome these issues, each physical server is represented by multiple virtual servers on the hash ring. This results in more uniform distribution and better fault tolerance.

- For each physical server S_i, $K = \log_2(M) = 9$ virtual servers are created, identified as $S(i, j)$ where j ranges from 0 to $K - 1$.
- Each virtual server is placed on the ring using a refined hash function: $\Phi(i, j) = i^2 + j^2 + 2j + 25$.
- Requests are routed to the nearest clockwise virtual server slot.



(a) (a) Server addition in system



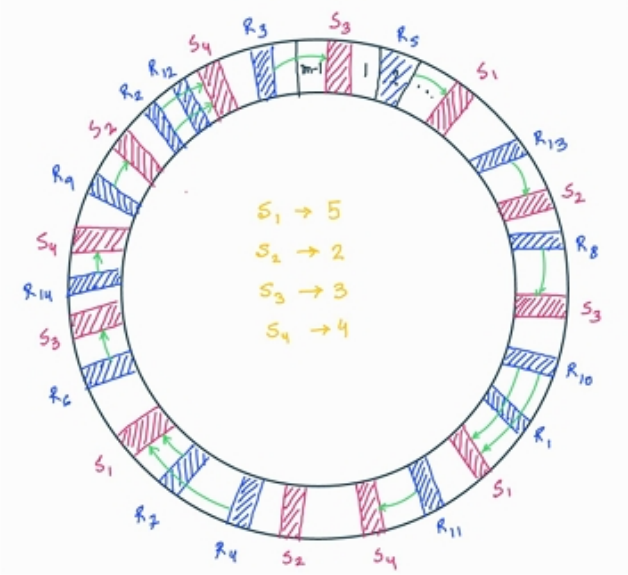
(b) (b) Server failure in system

Figure 4. Fig. 3: Consistent Hash-map in different scenarios

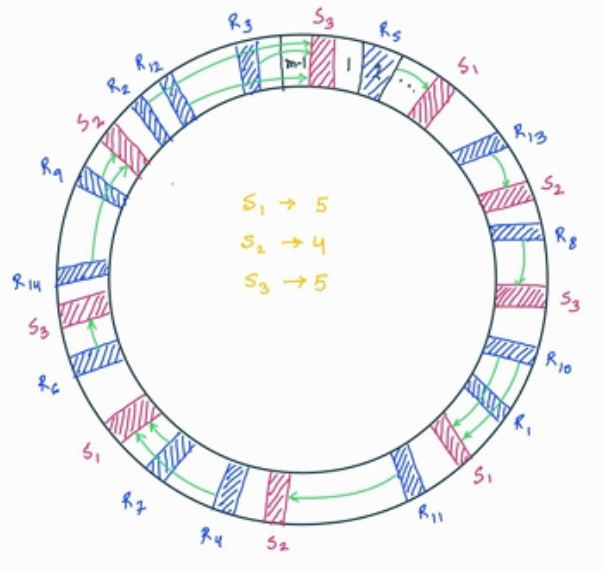
Advantages of Virtual Servers:

- Requests are more evenly distributed across all physical servers.

- When a server fails, its virtual servers are removed from the ring, and their responsibilities are picked up evenly by other virtual servers on the ring.
- Improves load balancing and fault tolerance significantly, especially when the number of servers is small.



(a) Virtual Servers



(b) Failure with Virtual Server

Figure 5. Fig. 3 (contd.): Virtual Servers in Consistent Hash Map

Statistical analysis suggests that using $K = \log_2(M)$ virtual servers for each physical server yields the best load distribution and system resilience, making it an industry-standard approach in distributed systems and caching frameworks.

2.2. Route Definitions

In this section, we describe the REST API endpoints exposed by both the load balancer and the individual server containers in the distributed database system. These routes support system initialization, configuration, and all CRUD operations on student records across shards and replicas.

Load Balancer API Endpoints

The load balancer is the central coordinating unit that handles client requests, manages metadata, performs consistent hashing, and routes requests to appropriate server replicas. It is also responsible for managing fault tolerance and scaling operations such as adding/removing servers or shards.

Table 1. Load Balancer Route Definitions

Route	Method	Description
`/init`	POST	Initialize the distributed database with schema, shard partitions, and server-shard assignments.
`/status`	GET	Retrieve current status of the system including shard placement and metadata tables.
`/add`	POST	Dynamically add server containers and update shard mappings accordingly.
`/rm`	DELETE	Remove a server and adjust shard mappings, triggering replication if necessary.
`/read`	POST	Read student records from the appropriate shards for a given ID range.
`/write`	POST	Insert new student records into the appropriate shard and propagate to all replicas.
`/update`	PUT	Update an existing student entry across all replicas of the respective shard.
`/del`	DELETE	Delete a student record from all replicas of the relevant shard.
`/help`	GET	To get the use of all the end points present.

Server API Endpoints

Each server container stores one or more shard replicas. It exposes a minimal set of routes to process read/write requests, respond to health checks, support replication, and allow recovery. These endpoints are primarily invoked internally by the load balancer.

Table 2. Server Route Definitions

Route	Method	Description
`/config`	POST	Initialize shard tables and replica configuration when the container is launched.
`/heartbeat`	GET	Health check endpoint used by the load balancer for fault detection.
`/copy`	GET	Export shard data for replication or recovery when a new replica is spawned.
`/read`	POST	Read entries from a specific shard for a given student ID range.
`/write`	POST	Write new student entries into a specified shard.
`/update`	PUT	Update existing student data by student ID in the assigned shard.
`/del`	DELETE	Delete a student entry from a shard replica based on student ID.

CAT-DAT-VAT Protocol for Consistency

To ensure **strong eventual consistency** in our distributed database system, we introduce the **CAT-DAT-VAT protocol**. The system architecture comprises a central *Load Balancer (LB)* and multiple backend *Servers* (each hosting shards and handling queries). The protocol ensures that all replicas across shards maintain a consistent view of data, even under concurrent operations or partial server failures.

The protocol uses three logical timestamps to manage the lifecycle of entries:

- **CAT (Create At Term)** – logical time when an entry is created.
- **DAT (Delete At Term)** – logical time when an entry is marked for deletion (set to ∞ if not deleted).
- **VAT (Valid At Term)** – the version acknowledged as committed by the system.

The flow of the algorithm is as follows:

- Lines **1–3** are executed by the **Load Balancer**, which receives the client request and initiates consistency handling.
- Lines **4–28** are executed at each **Server**, where entries are validated and operations are applied using the logical timestamps.
- Lines **29–33** return control to the **Load Balancer**, which finalizes the VAT update and sends the result back to the client.

This structured protocol guarantees consistency and correctness during reads, creates, updates, and deletions. See **Algorithm 1** for the full pseudocode.

Timestamps used:

- **CAT (Create At Term)** — logical time when the entry was created.
- **DAT (Delete At Term)** — logical time when the entry was deleted (set to ∞ if active).
- **VAT (Valid At Term)** — highest known consistent term per server.

3. Demonstration

This section showcases the working of all HTTP REST API routes developed as part of the project. Each route's functionality was validated by sending appropriate client queries and observing the corresponding server responses. Screenshots were captured for both the request and the response for each route.

Route Demonstrations

- `POST /init` – Initializes schema and shard assignments.

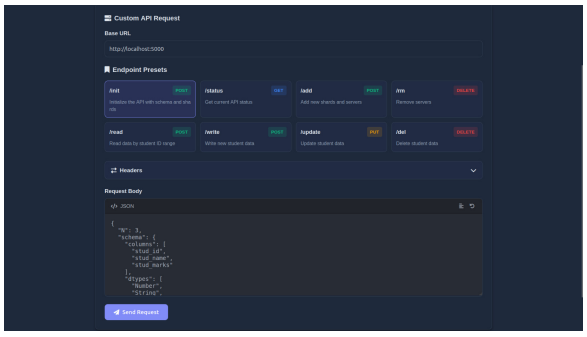


Figure 6. Query for /init Endpoint

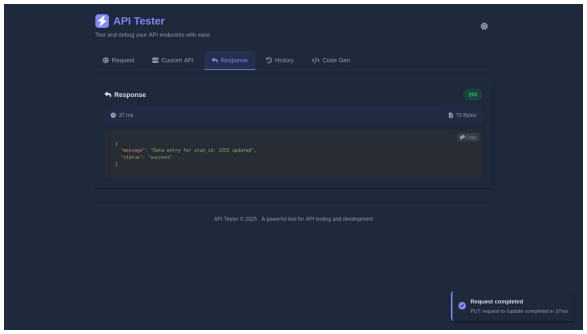


Figure 7. Response for /init Endpoint

- `GET /status` – Displays current load balancer and server state.

Algorithm 1 CAT-DAT-VAT Protocol for Distributed Consistency

```
procedure ProcessRequest(op, shard_id, stud_id, VAT)        ▷ Executed at Load Balancer
    max_VAT ← VAT
    for each server in server_list do                        ▷ Executed at each server

        Remove entries where DAT ≤ VAT
        Remove entries where CAT > VAT
        Set DAT ← ∞ for entries where DAT > VAT              ▷ Mark as deleted
        if op = READ then
            entry ← find entry for stud_id with CAT ≤ VAT
            response ← { data: entry }
        else
            term ← max(currentTerm[shard_id], VAT) + 1
            if op = CREATE then
                insertEntry(stud_id, CAT = term, DAT = ∞)
                currentTerm[shard_id] ← term
                response ← { vat: term }
            else if op = DELETE then
                updateEntry(stud_id, set DAT ← term)
                currentTerm[shard_id] ← term
                response ← { vat: term }
            else if op = UPDATE then
                updateEntry(stud_id, set DAT ← term)        ▷ Mark old as deleted
                term ← term + 1
                insertEntry(stud_id, CAT = term, DAT = ∞)
                currentTerm[shard_id] ← term
                response ← { vat: term }
            end if
        end if
        sendResponseToLoadBalancer(response)
        max_VAT ← max(max_VAT, response.vat)

    end for
    updateVAT(shard_id, max_VAT)
    sendResponseToClient()
end procedure
```

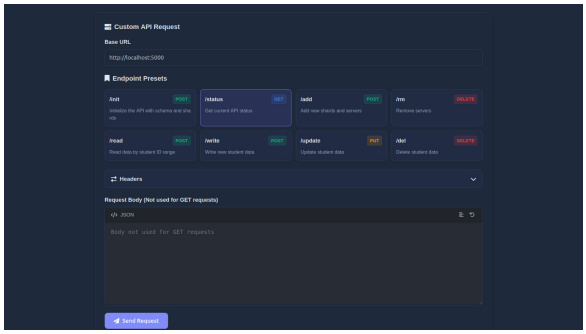


Figure 8. Query for /status Endpoint

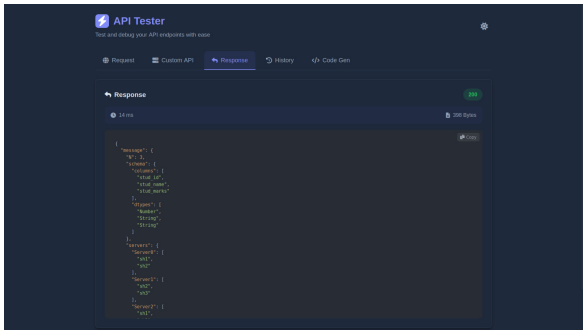


Figure 9. Response for /status Endpoint

- `POST /add` – Adds new shards and/or servers.

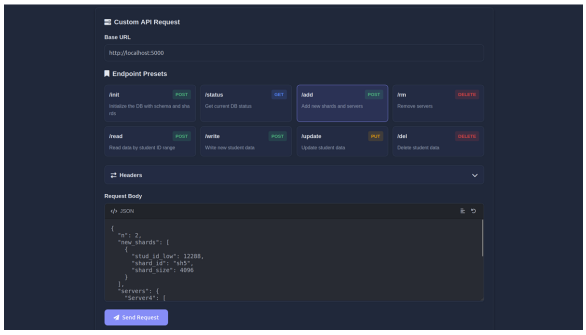


Figure 10. Query for /add Endpoint

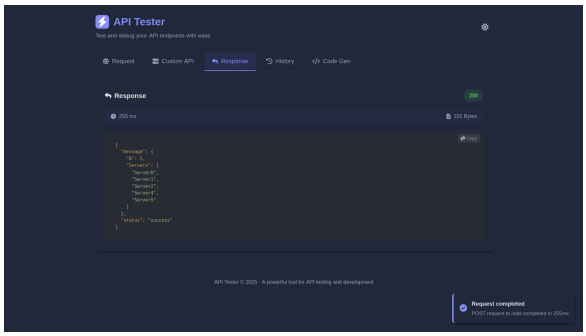


Figure 11. Response for /add Endpoint

- `DELETE /rm` – Removes a server instance.

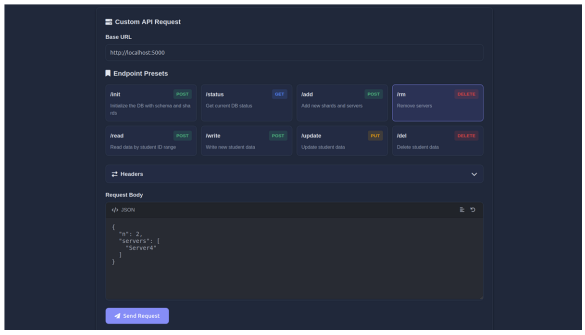


Figure 12. Query for /rm Endpoint

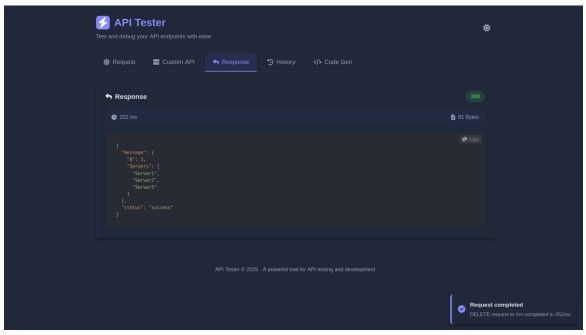


Figure 13. Response for /rm Endpoint

- `POST /read` – Reads data based on student ID range.

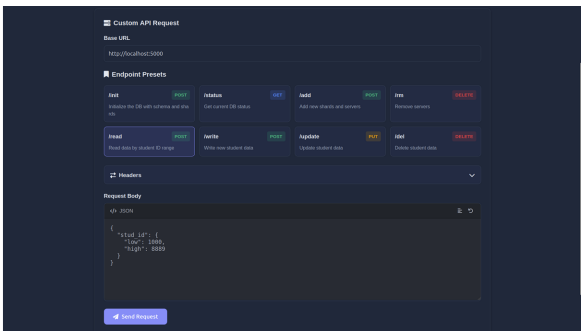


Figure 14. Query for /read Endpoint

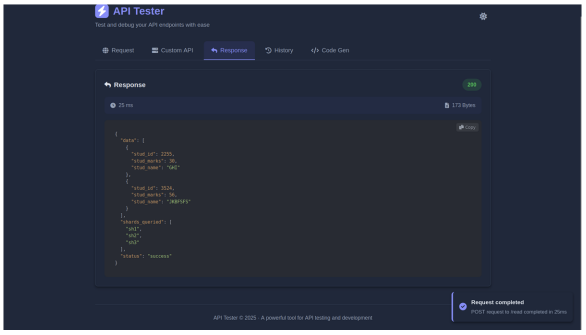


Figure 15. Response for /read Endpoint

- `POST /write` – Inserts new student data.

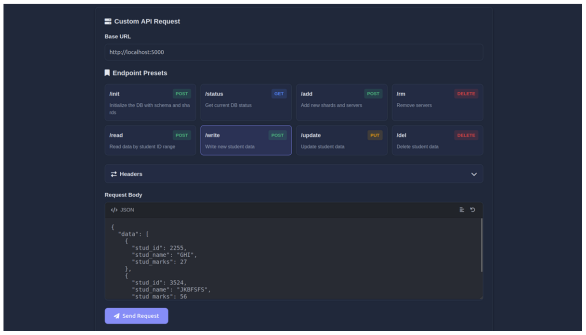


Figure 16. Query for /write Endpoint

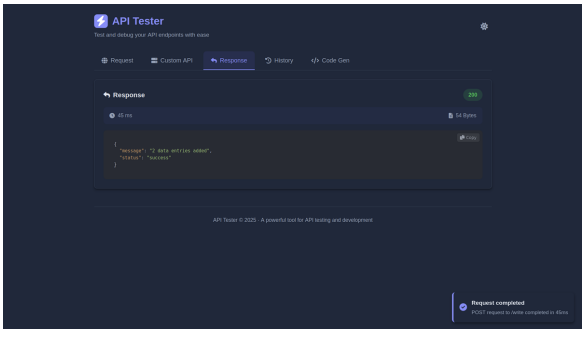


Figure 17. Response for /write Endpoint

- `PUT /update` – Updates existing student data.

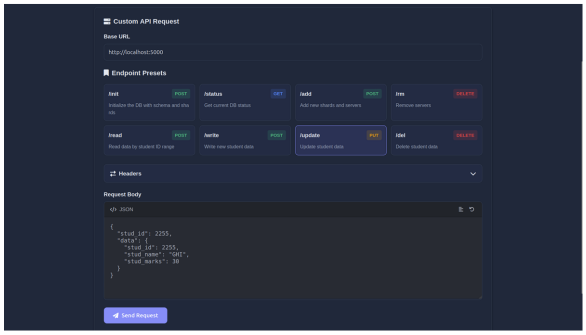


Figure 18. Query for /update Endpoint

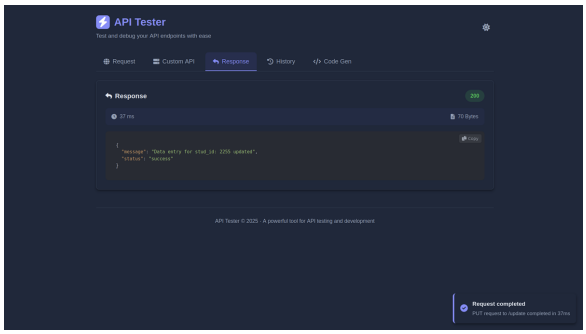


Figure 19. Response for /update Endpoint

- `DELETE /del` – Deletes student data.

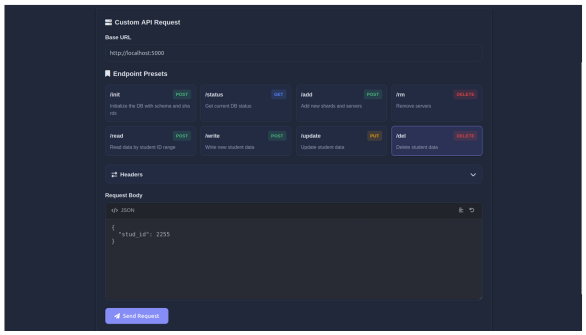


Figure 20. Query for /del Endpoint

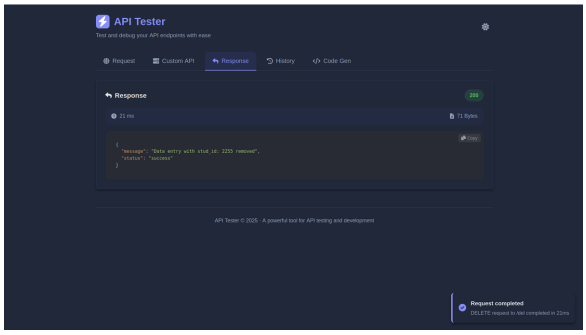


Figure 21. Response for /del Endpoint

4. Results

This section evaluates the performance and behavior of both the load balancer and the sharded distributed database under different configurations.

4.1. Part 1: Load Balancer Performance

Experiment A-1: Load Distribution Across 3 Servers 10000 asynchronous requests were sent to 3 server instances. The load distribution is visualized below:

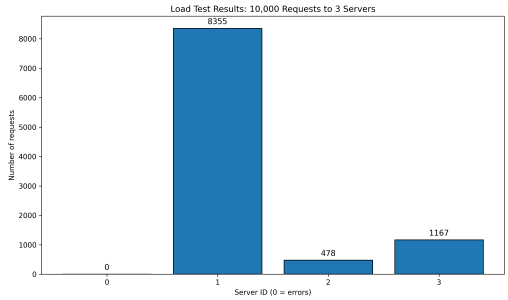


Figure 22. Request Distribution across 3 Servers

Experiment A-2: Scalability Test Incrementing server count from 2 to 6 and launching 10000 requests per configuration, the each server load was recorded:

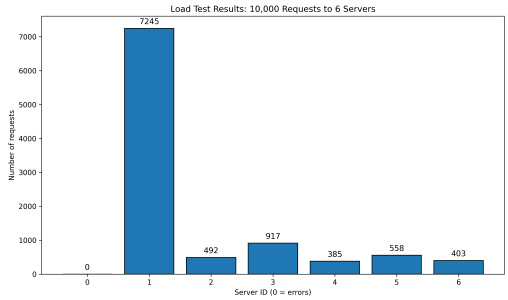


Figure 23. Number of requests vs. Server ID

Experiment A-3: Fault Recovery This experiment demonstrates the fault tolerance mechanism implemented via periodic heartbeats and the `flatline` recovery process. When a server fails to respond to 5 consecutive heartbeat checks, the load balancer detects the failure and launches a new replacement server container. The system then redistributes and restores the shard data from other replicas to maintain consistency and availability.

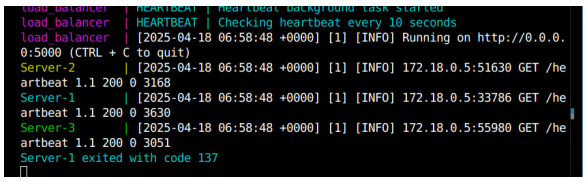


Figure 24. A server fails to respond to heartbeats — triggering the `flatline` mechanism

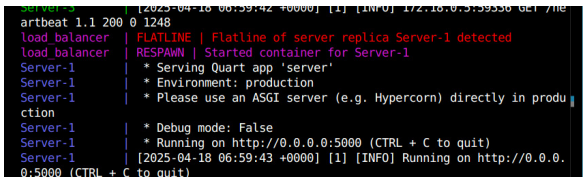


Figure 25. New server container launched by load balancer and shard data restored

Experiment A-4: Hash Function Variation In this experiment, we evaluated the effect of different hash functions on request distribution. Specifically, we implemented and tested the MD5 hash function to map incoming requests to server instances. The MD5 hash of each request was used to determine the server index by taking the modulo with the number of available servers. This method ensures a more uniform distribution compared to given hashing strategies.

The distribution of 10000 requests across 3 and 6 servers using the MD5 hash function is shown below:

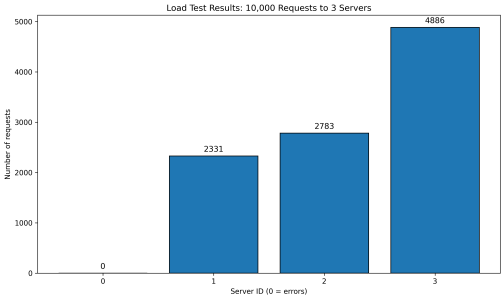


Figure 26. Request Distribution using MD5 Hash Function across 3 Servers

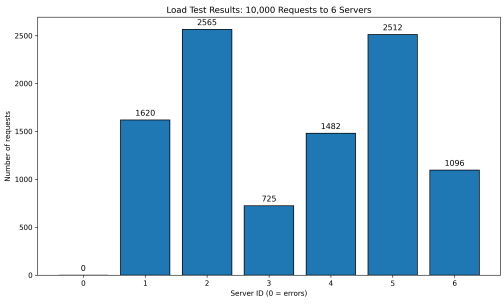


Figure 27. Request Distribution using MD5 Hash Function across 6 Servers

4.2. Part 2: Scalable Database with Sharding

Experiment A-1: Baseline Performance 10000 reads and writes were executed under the default configuration:

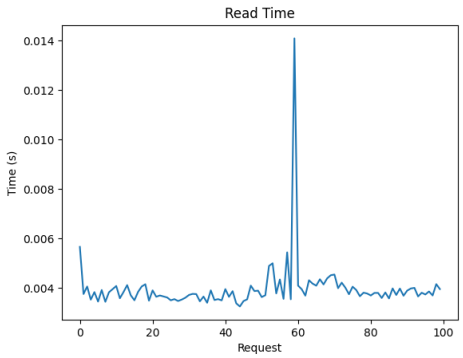


Figure 28. Read time analysis

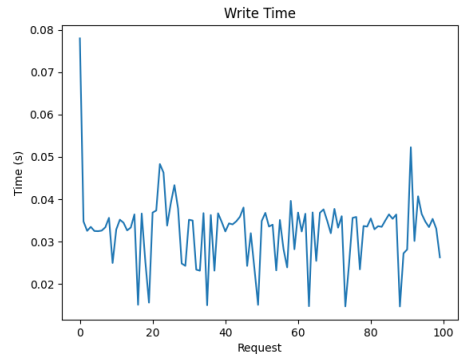


Figure 29. Write time analysis

Experiment A-2: Increased Shard Replicas Replicas were increased to 7. Read speed improved, write speed decreased:

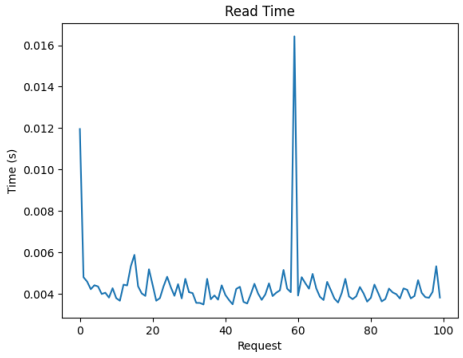
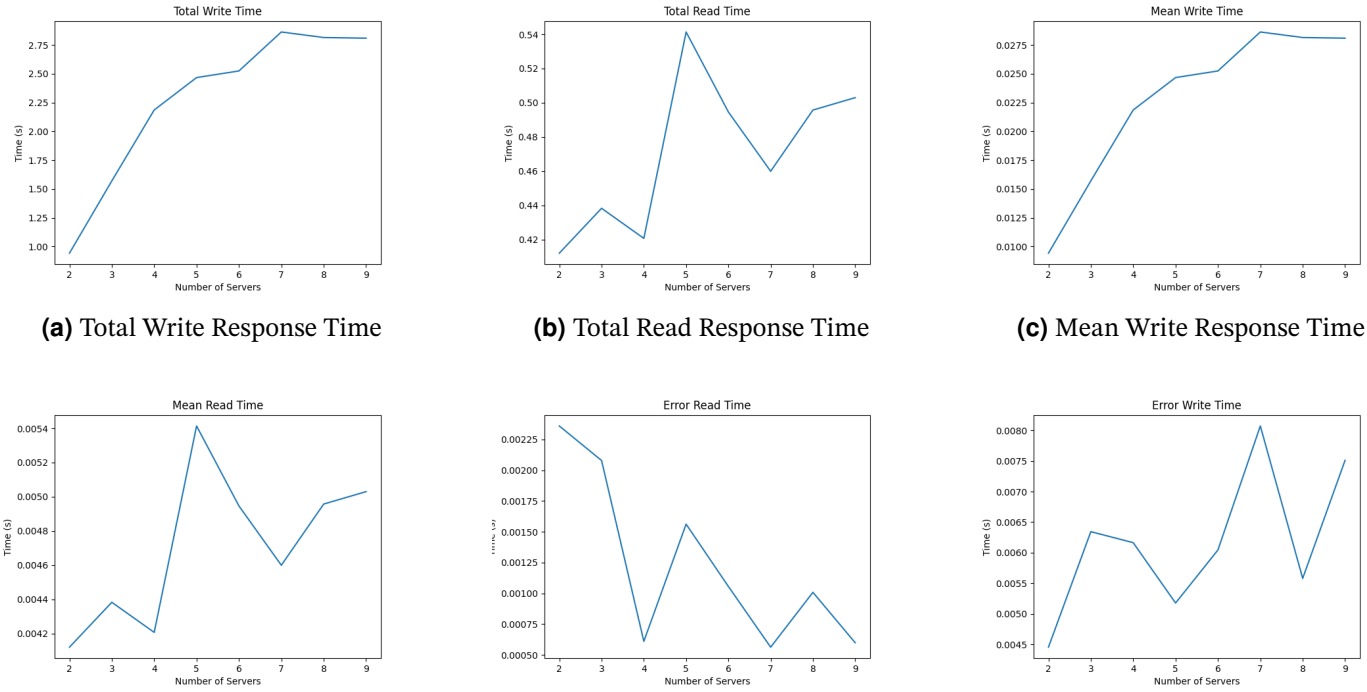


Figure 30. Performance with More Replicas on reads



(a) Total Write Response Time **(b)** Total Read Response Time **(c)** Mean Write Response Time

(d) Mean Read Response Time **(e)** Error Read Response Time **(f)** Error Write Response Time

Figure 32. Performance metrics with increasing number of servers (up to 10), with 6 shards and 8 replicas.

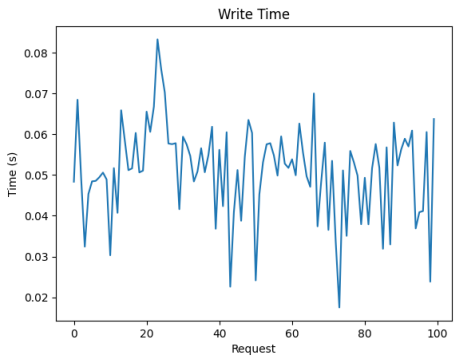


Figure 31. Performance with More Replicas on writes

Experiment A-3: Scaling Servers and Shards We increased the number of servers to 10, the number of shards to 6, and the number of shard replicas to 8. The write and read performance were measured for 10,000 operations each. The results (see Fig. 32) illustrate the impact on various timing metrics as the number of servers scales.

Write Speedup: Compared to the baseline configuration (with fewer servers), the total write time increased as more servers were added, indicating overhead in coordination. However, mean write time per server stabilized after a certain point.

Read Speedup: The total and mean read times fluctuated slightly, but overall remained consistent, suggesting a more balanced read load distribution.

Experiment A-4: Failure Recovery In this experiment, we manually terminated a running server container to simulate a failure scenario. The system successfully detected the failure via heartbeat timeouts (see Fig. 33) and invoked the `flatline` function. This triggered the launch of a replacement server container (see Fig. 34) and restored the shard data by copying it from other replicas (see Fig. 35).

```
Load-Balancer  | HEARTBEAT | Checking heartbeat every 10 seconds
Load-Balancer  | [2025-04-18 12:33:07 +0530] [61] [INFO] 172.18.0.1:40794 POST /
init 1.1 200 53 4375409
Server2        | [2025-04-18 12:33:07 +0530] [61] [INFO] 172.18.0.2:35174 GET /h
eartbeat 1.1 200 - 1415
Server0        | [2025-04-18 12:33:07 +0530] [61] [INFO] 172.18.0.2:53704 GET /h
eartbeat 1.1 200 - 1248
Server1        | [2025-04-18 12:33:07 +0530] [62] [INFO] 172.18.0.2:49256 GET /h
eartbeat 1.1 200 - 1348
Server1 exited with code 137
Load-Balancer  | HEARTBEAT | Checking heartbeat every 10 seconds
```

Figure 33. Server fails to respond to heartbeat and is marked as down

```
Server0        | [2025-04-18 12:34:01 +0530] [61] [INFO] 172.18.0.2:35398 GET /h
eartbeat 1.1 200 - 937
Server2        | [2025-04-18 12:34:01 +0530] [61] [INFO] 172.18.0.2:33964 GET /h
eartbeat 1.1 200 - 913
Load-Balancer  | FLATLINE | Flatline of server replica Server1 detected
Load-Balancer  | RESPAWN | Started container for Server1
Server1        | The files belonging to this database system will be owned by us
er "postgres".
Server1        | This user must also own the server process.
Server1        |
Server1        | The database cluster will be initialized with locale "en_US.utf
```

Figure 34. New server container is automatically launched by the load balancer

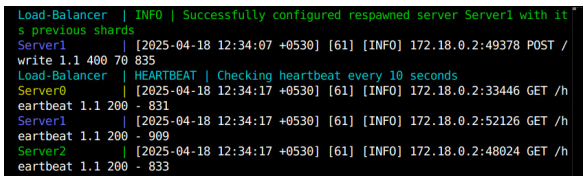


Figure 35. Shard data successfully copied to the new server from existing replicas

5. Conclusion and Future Scope

This project presents a distributed load balancer with support for sharding, replication, and fault tolerance using consistent hashing and heartbeats. It integrates asynchronous request handling via Quart and utilizes Docker containers for orchestration. Our system supports dynamic server addition and removal, ensuring robustness in real-world, large-scale environments.

Future Work

The current implementation serves as a strong foundation for a scalable distributed database proxy. However, several enhancements can further strengthen its capabilities:

- **Enhanced WAL Mechanism:** Extend the existing Write-Ahead Logging (WAL) system to handle distributed transactions. Incorporate consensus protocols such as Paxos or RAFT for improved fault tolerance and automatic primary server selection.
- **Scalability Improvements:** Experiment with a larger number of shards and replica servers to test scalability under high-load conditions. Investigate adaptive load balancing strategies and support automatic reconfiguration of the consistent hashing map.
- **Multi-Database Support:** Extend compatibility to include NoSQL databases like MongoDB or distributed systems like Cassandra. Compare performance, consistency guarantees, and recovery mechanisms across different backends, and evaluate storage solutions like RAID or cloud-based volumes.
- **Advanced Failure Recovery:** Improve fault detection with more granular health checks and implement self-healing behaviors. Utilize distributed leader election algorithms for faster and more reliable primary recovery.
- **Real-Time Monitoring and Analytics:** Build integrated dashboards to track key metrics such as throughput, latency, and request distribution. Visualization tools can provide valuable insights during scaling and failure events.
- **Code and Performance Optimizations:** Refactor core logic to better leverage Python's asynchronous capabilities. Optimize container startup, interprocess communication, and data serialization. Perform extensive profiling to identify and eliminate bottlenecks.

References

[1] Docker Documentation. *Docker Getting Started Guide*. Available at: https://docs.docker.com/get-started/

[2] Docker Compose Documentation. *Overview and Usage*. Available at: https://docs.docker.com/compose/

[3] Quart Framework Documentation. *Quart: An Async Python Web Framework*. Available at: https://quart.palletsprojects.com/

[4] PostgreSQL Documentation. *Trust Authentication*. Available at: https://www.postgresql.org/docs/current/auth-trust.html

[5] Roughgarden, T., Valiant, G. *CS168: The Modern Algorithmic Toolbox*, Lecture Notes, Stanford University. Available at: https://web.stanford.edu/class/cs168/

[6] Li, X., Nie, Q., Zhou, H. (2018). A dynamic load balancing algorithm based on consistent hash. In *IEEE IMCEC*, 2018. DOI: https://doi.org/10.1109/IMCEC.2018.8469393

[7] shardQ GitHub Repository. Available at: https://github.com/prasenjit52282/shardQ

[8] PhoenixKing2501. *Distributed Assignment 2 – CAT-DAT-VAT Algorithm Implementation*. GitHub repository, 2024. Available at: https://github.com/PhoenixKing2501/Distributed-Assignment-2

Project Repository

The full source code and instructions to run the project are available on GitHub:
https://github.com/prabhash-golla/DBMS_LAB/tree/main/Assignment5

Project Website

The frontend hosted version of the project is available at:
https://distributed-databases.w3spaces.com/