Final Report – Distributed Database System

1. Introduction

This project presents a simplified distributed database system implemented in Go (Golang) with a master-slave architecture. The system supports basic database operations such as creating databases and tables, executing CRUD operations, and synchronizing write operations across nodes through replication.

The system consists of one **Master Node** and two **Slave Nodes**, communicating via HTTP. All nodes are connected to a central MySQL server. This setup simulates a basic real-world distributed architecture used in scalable and fault-tolerant systems.

---

2. Architecture Overview

The system follows a **Master-Slave topology**:

- **Master Node:**
  - Handles all *write operations* (CREATE, INSERT, UPDATE, DELETE).
  - Performs *replication* to all connected slaves.
  - Allows read and write requests.
- **Slave Nodes:**
  - Accept *read operations* (SELECT).
  - Receive *replication commands* from the master.
  - Do not accept direct write operations.

All nodes communicate using HTTP endpoints (`/execute` and `/replicate`). The master replicates all write requests to slaves for consistency.

3. Design Choices

A. Language and Tools

- **Go (Golang):** Chosen for its performance, concurrency support, and simplicity in writing HTTP servers.
- **MySQL:** Lightweight, widely adopted RDBMS with native support for SQL syntax.
- **HTTP Protocol:** Used for communication between nodes for simplicity and ease of testing.

B. Node Structure

Each node is initialized with:

- `Node ID`
- `Port`
- `Master IP` and `Port` (for slaves)
- `MySQL Connection`

The master maintains a list of peer URLs for all slaves to replicate write operations automatically.

C. Replication Mechanism

- When a write request is handled by the master, it is immediately sent to the `/replicate` endpoint of all slaves.
- Each slave decodes the request and executes it against its local MySQL instance.
- Mutex (`sync.Mutex`) is used to handle concurrent database access safely.

D. Command Abstraction

A unified `Command` struct is used to:

- Represent the action (`create_db`, `insert`, `select`, etc.)
- Include metadata like database, table, columns, values, and conditions.

This struct simplifies communication and reuse across both `/execute` and `/replicate`.

---

4. Features

- ✅ Create & drop databases and tables
- ✅ Insert, update, delete, and select data
- ✅ Master-to-slaves replication for write operations
- ✅ Thread-safe database operations
- ✅ Easy deployment using command-line arguments

---

5. Challenges & Limitations

A. Network Communication

- Initially, the nodes failed to reach each other due to firewall or IP misconfiguration.
- Fixed by ensuring nodes used accessible local IPs and appropriate ports were opened.

B. Data Consistency

- Replication is done asynchronously. If a slave is down during replication, it will miss the command.
- No mechanism for retrying failed replications or syncing missed commands yet.

C. Error Handling

- Limited error propagation from slaves back to the master.
- No acknowledgment system implemented for replication success.

---

6. Future Improvements

- 🔁 Implement write-ahead logs for better fault tolerance
- 🔀 Add leader election for failover support
- 🎁 Add persistent message queues to ensure replication delivery
- Implement unit and integration testing
- Improve query parsing for more complex SQL operations

---

7. Conclusion

This project demonstrates a working distributed database prototype with basic replication and concurrency support using Go and MySQL. It provides a strong foundation to build more complex features like sharding, replication logs, and automatic failover. The simplicity of the design makes it an ideal starting point for understanding distributed systems and database replication