Distributed Database System - Final Report

1. Introduction

This report presents the design and implementation of a basic distributed database system built using the Go programming language. The system aims to simulate key principles of distributed systems, including master-slave architecture, data replication, and basic fault tolerance.

2. System Architecture

The architecture consists of three nodes: one master and two slaves. Each node runs independently and communicates via HTTP using RESTful APIs. The master node is responsible for creating tables and managing the database schema. All nodes (including slaves) can handle operations like insert, update, delete, and search. Each node maintains its own in-memory copy of the data. A configuration file (`config.json`) defines node roles and addresses.

3. Key Features and Functionality

- Master creates tables dynamically and distributes schema.
- Data replication: insert/update/delete operations on the master are automatically propagated to all slaves.
- Fault tolerance: if the master node fails, the node with the lowest ID promotes itself as a temporary master.
- RESTful API support for CRUD operations.
- Lightweight and easy-to-run Go application with no external dependencies.

4. Design Decisions

- Chose HTTP over TCP for simplicity and human-readable testing (using Postman or curl).
- In-memory storage used for faster performance and easier replication.
- JSON was selected as the communication format due to its readability and compatibility.
- A simple static leader election strategy was used (lowest node ID), avoiding complexity from consensus algorithms.

5. Challenges Faced

- Ensuring all nodes remained in sync after creation or update operations.
- Handling master failure and preventing split-brain scenarios where multiple nodes act as master simultaneously.
- Deciding on a minimal but effective replication strategy.
- Debugging inter-node communication and race conditions when sending concurrent requests.

6. Conclusion and Future Work

This distributed database prototype successfully demonstrates the fundamentals of replication and coordination across nodes. While simple, it opens the path to more advanced features like persistent storage, full leader election protocols (e.g., Raft), GUI dashboards, and even horizontal scaling. The system can be extended to support client authentication and real-time synchronization in larger environments.

7. Sample API Usage

Below are examples of API requests used during the testing and validation of the system:

• Create Table

```
POST /create_table
{
  "table_name": "students",
  "columns": ["id", "name", "grade"]
}
```

• Insert Record

```
POST /insert
{
  "table_name": "students",
  "row_id": "1",
  "data": {
    "id": "1",
    "name": "Ahmed",
    "grade": "A"
  }
}
```

• Search Record

```
GET /search?table=students&column=name&value=Ahmed
```

8. Testing Strategy

The system was tested by running multiple nodes on different ports and performing insert, update, delete, and search operations from different nodes. The consistency of data across all nodes was validated after each operation. Fault tolerance was tested by manually stopping the master node and observing the promotion of a slave node to master.

9. Screenshots / Logs (Optional)

Sample logs from node consoles:

```
This node is MASTER
Sent to http://localhost:8081/insert - Status: 200 OK
Master is down! Trying to become master...
[WARNING] I am the new TEMP MASTER
```