Distributed Database System: Architecture, Design, and Challenges

May 15, 2025

1 Introduction

This report details the architecture, design choices, and challenges of a distributed database system developed to provide robust data management with high availability. The system employs a master-slave architecture, using Go for backend nodes (master and slaves) and Python with Tkinter for a user-friendly GUI. It supports standard database operations (CREATE_DB, CREATE_TABLE, INSERT, UPDATE, DELETE, SEARCH, DROP_DB) executed on SQL Server instances. Designed for flexibility, the system operates on a single laptop (IP: 192.168.149.137) or multiple laptops connected via a phone hotspot, addressing requirements for team collaboration and portability.

2 System Architecture

The system follows a master-slave architecture with one master node (192.168.149.137:8000) and two slave nodes (:8001, :8002). Each node runs a Go-based TCP server connected to a local SQL Server instance, ensuring persistent data storage. The master coordinates operations, broadcasting them to slaves for replication, while slaves can process operations independently when the master is unavailable, enhancing fault tolerance.

2.1 Components

- Master Node (master.go): Handles client requests from the GUI, executes operations on its SQL Server, and broadcasts operations to slaves using TCP. It uses a mutex for thread-safe SQL execution.
- Slave Nodes (slave.go): Process operations directly on their SQL Server instances, accepting requests from the GUI or master. Slaves operate autonomously if the master is down.
- GUI (gui.py): A Tkinter-based interface allowing users to select nodes, choose operations, and input parameters (database, table, data, condition). It features dynamic fields, tooltips, and a status panel showing node connectivity (green for connected, red for disconnected).
- SQL Server: Each node connects to a local SQL Server Express instance using Windows Authentication, storing data persistently. Operations are executed directly, eliminating in-memory state.

2.2 Communication

Nodes communicate via TCP, with JSON-encoded operations sent between the GUI, master, and slaves. The master broadcasts operations to slaves asynchronously, ensuring replication when connected. The GUI uses TCP sockets to send operations to the selected node and receive responses, such as SEARCH results or error messages.

3 Design Choices

The system's design prioritizes simplicity, availability, and portability, balancing functionality with ease of use.

3.1 Technology Stack

- Go for Backend: Chosen for its concurrency model (goroutines) and simplicity in building TCP servers. The go-mssqldb driver integrates seamlessly with SQL Server.
- Python Tkinter for GUI: Selected for rapid development and cross-platform compatibility, providing an intuitive interface without external dependencies.
- SQL Server Express: Used for persistent storage, leveraging Windows Authentication for ease of setup. Its robustness supports enterprise-level data management.

3.2 Persistence and Replication

Initially, slaves used an in-memory map for operations, causing errors like "table does not exist" due to lack of persistence. The design shifted to direct SQL Server execution, with each node maintaining its database. The master broadcasts operations to slaves for replication, ensuring consistency when connected. Slaves' ability to operate independently during master downtime enhances availability, though it may lead to temporary data divergence.

3.3 GUI Design

The GUI dynamically adjusts input fields based on the selected operation, using placeholders (e.g., {"id": 1, "name": "Alice"} for INSERT) and tooltips for guidance. A status panel pings nodes every 5 seconds, updating connectivity indicators. This design improves usability, especially for non-technical users, and supports debugging by visualizing node states.

3.4 Portability

To support team collaboration, the system includes configuration comments in master.go and gui.py for updating slave IPs (e.g., 192.168.149.138:8001). This enables deployment on a phone hotspot, critical for ad-hoc testing environments.

4 Challenges and Solutions

The development faced several challenges, addressed through iterative refinements.

4.1 Slave Operation Failures

Challenge: Slaves initially stored data in an in-memory map, causing errors (e.g., “table users does not exist”) when executing operations like INSERT or SEARCH, as the map wasn’t synced with SQL Server.

Solution: Removed the in-memory map, updating slaves to execute all operations directly on SQL Server. The processOperation function was modified to generate SQL queries dynamically, using INFORMATION_SCHEMA for metadata (e.g., column names for SEARCH).

4.2 Master Downtime

Challenge: The system needed to remain operational when the master was down, as slaves originally relied on master coordination for some operations.

Solution: Enhanced slave.go to process all operations (CREATE_DB, INSERT, etc.) independently, connecting to local SQL Server instances. This required removing restrictions (e.g., DROP_DB on slaves) and ensuring robust error handling.

4.3 SEARCH Operation Errors

Challenge: The SEARCH operation failed with “table users does not exist” due to issues in the getTableColumns function, which queried INFORMATION_SCHEMA.COLUMNS.

Solution: Updated getTableColumns to check database existence (sys.databases) and table existence (sys.tables) explicitly, adding a fallback query. Detailed logging was introduced to trace query execution, and TABLE_SCHEMA = ’dbo’ was specified to avoid schema mismatches.

4.4 GUI Usability

Challenge: The GUI’s SEARCH fields were confusing, with an unused Data field and vague placeholders (e.g., {}).

Solution: Removed the Data field for SEARCH, updated the Condition placeholder to {”id”: 1}, and added a clear tooltip: “Filter rows, e.g., {”id”: 1} for specific ID, or {} for all rows.” Dynamic field adjustments were preserved for other operations.

4.5 Network Portability

Challenge: Ensuring the system worked across multiple laptops on a phone hotspot required flexible configuration and network reliability.

Solution: Included IP configuration comments in source files and ensured TCP communication used timeouts (5 seconds) to handle unreliable networks. Firewall instructions were provided to open ports 8000-8002.

5 Conclusion

The distributed database system successfully delivers a robust, user-friendly solution for data management, with high availability through slave independence and a clear GUI.

Design choices like SQL Server persistence, Go's concurrency, and Tkinter's simplicity balanced functionality with ease of use. Challenges such as slave failures, SEARCH errors, and network portability were addressed through direct SQL execution, enhanced metadata queries, and flexible configurations. Future improvements could include automatic data synchronization when the master reconnects and advanced query support for complex SEARCH operations.