Database week 5

Khaula Molapo

1.

In a banking system, a distributed database is used to manage customer accounts across multiple branches and ATMs worldwide. Instead of storing all data on a single server, account information is partitioned and replicated across multiple nodes. When a customer makes a transaction, the system ensures that updates are propagated to all relevant nodes to maintain consistency. For example, a deposit made in New York must reflect in the customer's balance if they check it in London. Distributed databases improve reliability, as the failure of one node does not halt the system; other nodes continue processing transactions. They also reduce latency by storing data closer to users. Security is maintained through authentication and encrypted connections. The system can handle high loads because read and write requests are balanced across nodes. This scenario demonstrates why banks rely on distributed databases for performance, scalability, and fault tolerance in handling sensitive financial data.

2.

Two-Phase Commit (2PC) is a protocol used in distributed databases to ensure atomic transactions across multiple nodes. It works in two phases: the prepare phase, where a coordinator asks participating nodes if they can commit a transaction, and the commit phase, where nodes finalize the transaction if all agree. If any node cannot commit, the transaction is rolled back across all nodes. 2PC ensures consistency and reliability, preventing partial updates that could lead to data corruption. It is widely used in banking, e-commerce, and other systems where correctness is critical, despite introducing some overhead and potential blocking during failures.

3.

For practice, I designed a simple partitioning scheme for a bank's customer database. Accounts are divided based on customer regions: North, South, East, and West. Each region is assigned a node that stores all accounts from that area. Transactions from a branch in a region are handled by the local node, reducing latency and improving efficiency. Reflection: Designing this scheme highlighted the benefits of partitioning: it balances load, speeds up queries, and enhances fault tolerance. However, cross-region transactions require careful coordination to maintain consistency. This exercise emphasized how logical partitioning can simplify distributed database management while supporting scalability.

4.

In Draw.io, I created a distributed banking database diagram. At the top, a Coordinator Node manages transactions across the system. Below, four regional nodes—North, South, East, West—store partitioned account data. Each regional node is connected to local ATMs and branch servers, showing how transactions flow from users to the database. Arrows indicate communication for updates, replication, and transaction coordination. The diagram highlights

how distributed architecture ensures fault tolerance, load balancing, and efficient data access. It clearly shows the relationship between the coordinator and nodes, as well as the flow of data in a geographically distributed banking system.

```
banking_db/
├── coordinator/
│   ├── transaction_manager.py
│   └── log.txt
│
├── branches/
│   ├── branch_A/
│   │   ├── accounts.db
│   │   └── customers.db
│   ├── branch_B/
│   │   ├── accounts.db
│   │   └── customers.db
│   └── branch_C/
│       ├── accounts.db
│       └── customers.db
│
├── replication/
│   ├── sync_service.py
│   └── backup/
│       └── branch_A_backup.db
│
├── clients/
│   ├── mobile_app/
│   │   └── app_interface.java
│   └── web_app/
│       └── dashboard.html
│
├── logs/
│   ├── coordinator.log
│   ├── branch_A.log
│   ├── branch_B.log
│   └── branch_C.log
```