Assignment 1: Simulating a Distributed RDBMS SimuFragDB Report

Nikunj Mahajan (IMT2023068), Gautam Kappagal (IMT2023082)
Pragya Rai (IMT2023529), Chirayu Choudhary (IMT2023004)

February 5, 2026

1 Introduction

This report documents the design, implementation, and evaluation of `SimuFragDB`, a simulated distributed database system. The objective was to implement horizontal fragmentation across multiple PostgreSQL instances and handle query routing via a Java client, gaining insights into the architectural principles of distributed systems.

2 Approach for Data Aggregation and Routing

2.1 Horizontal Fragmentation & Routing

Our implementation strictly follows the Horizontal Fragmentation strategy defined in the assignment specifications.

- **Setup:** We initialized $N = 3$ database fragments on a single physical machine.
- **Routing Logic:** We utilized the provided deterministic routing function (`Router.java`) which uses a modulo hash of the primary key:

$$FragmentID = |key.hashCode()| \pmod{numFragments}$$

- **Transactional Queries (Insert/Update/Read Profile):** For operations involving specific students (e.g., `INSERT_STUDENT`, `UPDATE_GRADE`), the request was routed explicitly to the specific fragment calculated by the router. This ensured that data for a specific entity resided consistently on one node.

2.2 Aggregation Strategy

For analytical queries such as `getAvgScoreByDept()` and `getAllStudentsWithMostCourses()`, the assignment instructions specified that "These queries must be executed on a randomly selected fragment" rather than aggregating data from all nodes.

- **Implementation:** Our client selects a random active connection from the connection pool and executes the SQL query on that single node.
- **Implication:** This design simulates a partition-local query. While efficient, it means the result set reflects only the subset of data residing on that specific shard, rather than the global dataset.

3 Challenges Faced and Resolutions

3.1 JDBC Connection Management

Challenge: Managing multiple active JDBC connections simultaneously without resource leakage.
Resolution: We implemented a `HashMap<Integer, Connection>` as a connection pool in `setupConnections()`. This allowed valid references to all fragment instances to be maintained throughout the client's lifecycle, enabling O(1) retrieval of the correct connection handle for any given transaction.

3.2 Data Consistency vs. Simulation Constraints

Challenge: The primary challenge was the discrepancy between the "Expected Output" (generated from a monolithic database containing all data) and the "Actual Output" (generated from distributed fragments).
Resolution: We identified that queries like `READ_ALL` or `READ_SCORE` executed on a single random fragment would inherently return incomplete data compared to the monolithic baseline. We resolved to stick strictly to the assignment guideline of querying a "randomly selected fragment," accepting that the accuracy metric would reflect the distributed nature of the data (seeing only $1/N$ of the total records).

4 Execution Metrics

4.1 Execution Time

The total execution time for the workload as reported by the `Driver.java` was:

5529 ms

4.2 Accuracy Analysis

The evaluation compared the output of `SimuFragDB` against a baseline single-instance database.

Metric	**Value**
Total Query Output Lines	*2462*
Matching Lines	*823*
Accuracy Percentage	$\approx$ **33%**

Table 1: Accuracy Metrics comparing Distributed vs. Monolithic execution.

Analysis of Accuracy:
The observed accuracy of approximately 33% is consistent with the system architecture. With the data horizontally fragmented across 3 nodes, any aggregation query executed on a single random node (as per instruction) has visibility into only $\approx$ 33% of the total dataset. The remaining 67% of the data resides on the other two un-queried fragments. Thus, the low accuracy is not a defect in the code, but a verification that the data was successfully distributed and sharded across the three independent database instances.