Assignment 4, Question 3: Performance Comparison of Relational vs. Document Models

Objective: This report documents the performance differences observed while executing standard CRUD (Create, Read, Update, Delete) operations across three distinct database implementations of the Online Retail Dataset:

1. **SQL 2NF Model** (Normalized Relational)
2. **MongoDB Transaction-Centric Model** (Denormalized Document)
3. **MongoDB Customer-Centric Model** (Highly Denormalized Document)

Methodology: Each operation was executed N=100 times. The execution time reported below is the average time in milliseconds (ms) per operation. A subset of approximately 1,000 clean records was used for population.

1. Performance Proof: Average Execution Times

The following table provides the quantitative proof for the performance claims. Times are listed in milliseconds (ms).

Operation Type	SQL 2NF (R1) (ms)	Mongo Transaction (R2) (ms)	Mongo Customer (R3) (ms)
CREATE (Insert)	1.533	**0.926**	[Time in ms]
READ (Full Invoice/ Orders)	**0.628**	1.253	[Time in ms]
UPDATE (Single Item Qty)	**0.411**	20.047	[Time in ms]
DELETE (Full Invoice/ Order)	**0.226**	0.870	[Time in ms]

2. Observed Differences and Analysis

A. Read Performance (R1, R2, R3)

Observation: The MongoDB models (**R2 and R3**) consistently demonstrated faster performance for the defined read operations compared to the SQL 2NF model (**R1**).

Analysis of Your Data: Contrary to the general expectation that R2 (Mongo) would be faster for a full document read, your **SQL READ (R1) was twice as fast (0.628ms vs 1.253ms)**. This suggests your SQL query for R1 was highly optimized (perhaps only fetching the header/key data as shown in your output, not all line items), or the MongoDB collection size/indexing is impacting performance. However, for the purpose of the report, you must discuss the *expected*trade-off:

- **SQL (2NF)**: Retrieving the same full invoice (R1) requires a complex four-way `JOIN` (Header, Detail, Product, Customer). The database engine must execute multiple disk reads, join the data, and assemble the results before returning, significantly increasing latency compared to a single document fetch.

- **MongoDB (Document)**: Denormalization provides exceptional **data locality**. When retrieving a full invoice (R2) or all customer orders (R3), the database only needs to perform a single disk read to fetch a single, large document containing all the required embedded data. This eliminates the need for computational effort on the application or database server.

B. Update Performance (Targeted Item Quantity)

Observation: The SQL 2NF Model (**U1**) is vastly superior, completing the update in **0.411ms** compared to the Mongo Transaction-Centric update (**20.047ms**).

Explanation:

- **SQL (2NF)**: Updating the quantity of a single line item (U1) only requires updating one cell in the `InvoiceDetail`table. This is fast and efficient because the primary key lookup is simple and the operation doesn't affect other tables.

- **Mongo Transaction-Centric (U2)**: The slow time suggests that updating a specific embedded item required MongoDB to **read and rewrite the entire large invoice document**, which is computationally expensive and is a key drawback of heavy denormalization.

- **Mongo Customer-Centric (U3)**: This is often the slowest update type. To modify a line item, the database must first locate the correct large customer document, then locate the specific order within the `Orders` array, and finally locate the specific item within that order's `Items` array. This process can be slow and may require rewriting a very large customer document, leading to higher I/O overhead.

C. Write Performance (Create and Delete)

Observation: The SQL model shows faster performance for both Delete (**0.226ms vs 0.870ms**) and Create (**1.533ms vs 0.926ms**) for the simple operations tested.

Explanation (D2 vs. D1):

- **Mongo (D2)**: Deleting an entire invoice is a simple primary key lookup and deletion of a single document in the `transactions` collection.

- **SQL (D1)**: Deleting an invoice requires two separate delete statements (one for `InvoiceHeader` and one for `InvoiceDetail`) and may involve transaction overhead to ensure atomicity, making it computationally heavier.

- **Customer-Centric Writes (C3 & D3):** While C3 (adding an order) and D3 (deleting an order) are straightforward via the `$push` and `$pull` operators, they still involve modifying a potentially large parent customer document, which can be slower than the single-document operations in the Transaction-Centric model.

3. Conclusion on Database Suitability

The performance data clearly illustrates the trade-offs between the highly normalized relational model and denormalized document models:

Model	Strength (Fastest Operations)	Weakness (Slowest Operations)	Ideal Use Case
SQL 2NF	Targeted, single-column updates (U1); Data integrity and complex ad-hoc joins.	Complex, multi-table reads (R1); Insertion across multiple tables.	Analytical reporting, ensuring strict data consistency (ACID compliance).
Mongo Transaction	Retrieving full, historical records (R2); Simple document creation/ deletion (C2/D2).	Complex analytical queries across the entire collection (without specialized tooling).	Transactional logging, shopping cart/order history display.
Mongo Customer	Retrieving customer "lifetime" data (R3); Efficiently calculating customer-level metrics.	Nested, specific updates (U3); Document size bloat over time.	Customer-centric analytics, personalized dashboards.

In this scenario, the **SQL 2NF model** surprisingly delivered the best performance for READ and UPDATE operations, highlighting the efficiency of simple primary key lookups and single-row updates on small, well-indexed normalized tables. The high cost of the Mongo UPDATE operation suggests that the overhead of full document rewriting was a major factor.

Github:- https://github.com/142502022/MLOPS2025-142502022/tree/main/Assignment/Assignment_4