Assignment 4 - Database Implementation

Name: Vedachalam Geetheswar
Roll Number: 142201025
Github Url: mlops2025w_142201025

Question 3: CRUD Operations Performance Analysis

Customer-Centric Approach Overview

Before going into the transaction-centric approaches, I implemented a customer-centric approach where all customer data and their invoices are embedded in a single document.

Customer-Centric Structure:

```
{
  "CustomerID": 17850,
  "Country": "United Kingdom",
  "Invoices": [
    {
      "InvoiceNo": "536365",
      "InvoiceDate": "2010-12-01",
      "Items": [...]
    }
  ]
}
```

When Customer-Centric is Better:

- Getting all invoices for a specific customer (single document read)
- Customer analytics and reporting (complete customer view)
- CRM systems where customer is the primary entity
- Scenarios where you frequently need all customer transaction history together

Customer-Centric Disadvantages:

- Slow for individual invoice lookups (must scan invoice arrays)
- Document size can grow large for active customers (MongoDB limit)
- Complex nested updates when modifying specific invoice items
- Not efficient for product-level or invoice-level analytics

In the following two approaches, I compare transaction-centric designs against this customer-centric approach to analyze which performs better for different operations.

.

.

Approach 1: Transaction-Centric with Embedded Customer Data

In this first approach, I embedded customer data (Country) directly in each transaction document to keep everything in one place.

Document Structure:

```
{
  "InvoiceNo": "536365",
  "CustomerID": 17850,
  "Country": "United Kingdom",
  "InvoiceDate": "2010-12-01",
  "Items": [...]
}
```

Performance Results

Operation	Transaction-Centric	Customer-Centric	Winner
Create Invoice	0.44 ms	0.63 ms	Transaction (1.42x)
Find by ID	0.34 ms	0.67 ms	Transaction (1.99x)
Single Invoice	0.25 ms	0.42 ms	Transaction (1.68x)
Customer Invoices	8.79 ms	1.04 ms	Customer (8.48x)
Customer Invoices by Month/Year	9.71 ms	0.75 ms	Customer (12.87x)
Product Sales	145.80 ms	170.08 ms	Transaction (1.17x)
Top Customers	139.11 ms	152.95 ms	Transaction (1.10x)
Sales by Country	159.22 ms	186.23 ms	Transaction (1.17x)
Product Search	0.99 ms	6.49 ms	Transaction (6.53x)
Update Item Quantity	0.88 ms	0.65 ms	Customer (1.35x)
Delete Invoice	0.35 ms	0.85 ms	Transaction (2.39x)

Approach 2: Transaction-Centric with Separated Customer Collection

I realized that embedding Country in every transaction causes data redundancy. If a customer has 100 transactions, "United Kingdom" is stored 100 times. This violates normalization principles.

So I separated customer data into its own collection, similar to how we would have a separate Users table in SQL.

.

.

Transaction Collection:

```
{
  "InvoiceNo": "536365",
  "CustomerID": 17850,
  "InvoiceDate": "2010-12-01",
  "Items": [...]
}
```

Transactions_Customers Collection:

```
{
  "CustomerID": 17850,
  "Country": "United Kingdom"
}
```

Indexes Added:

```
transactions.create_index('InvoiceNo', unique=True)
transactions.create_index('CustomerID')
customers.create_index('CustomerID', unique=True)
```

Performance Results

Operation	Transaction-Centric	Customer-Centric	Winner
Create Invoice	0.51 ms	0.79 ms	Transaction (1.56x)
Find by ID	0.43 ms	1.12 ms	Transaction (2.60x)
Single Invoice	0.33 ms	0.83 ms	Transaction (2.53x)
Customer Invoices	1.25 ms	0.87 ms	Customer (1.43x)
Customer Invoices by Month/Year	0.85 ms	0.93 ms	Transaction (1.10x)
Product Sales	128.23 ms	166.72 ms	Transaction (1.30x)
Top Customers	154.98 ms	147.35 ms	Customer (1.05x)
Sales by Country	389.28 ms	182.22 ms	Customer (2.14x)
Product Search	1.07 ms	6.37 ms	Transaction (5.94x)
Update Item Quantity	0.99 ms	0.81 ms	Customer (1.22x)
Delete Invoice	0.51 ms	0.69 ms	Transaction (1.35x)

Major Advantages of Approach 2

1. **No Data Redundancy:** Customer country stored only once instead of in every transaction
2. **Consistency:** If customer country changes, update in one place
3. **Better Indexing Performance:** CustomerID index dramatically improved customer queries from 8.79ms to 1.25ms (7x faster)
4. **Normalized Design:** Follows database best practices, similar to SQL foreign keys
5. **Scalability:** Easy to add more customer fields without affecting transactions

---

Major Disadvantage of Approach 2

Sales by Country became 2.4x slower (159ms to 389ms)

Why this happens:

- In Approach 1, Country is directly available in each transaction document
- In Approach 2, we need to use MongoDB's $lookup operation to join transactions with transactions_customers collection
- The $lookup performs a left outer join which adds overhead:
  - Scans the transactions_customers collection
  - Creates temporary joined documents in memory
  - Increases query execution time

Code comparison:

Approach 1 (fast):

```
db.transactions.aggregate([
  {$unwind: '$Items'},
  {$group: {_id: '$Country', totalRevenue: ...}}
])
```

Approach 2 (slower):

```
db.transactions.aggregate([
  {$lookup: {
    from: 'transactions_customers',
    localField: 'CustomerID',
    foreignField: 'CustomerID',
    as: 'customer_info'
  }},
  {$unwind: '$customer_info'},
  {$unwind: '$Items'},
  {$group: {_id: '$customer_info.Country', totalRevenue: ...}}
])
```

Conclusion

The choice between approaches depends on our use case:

Customer-Centric Approach:

- Best for CRM systems and customer analytics
- Excels at customer-based queries (8.48x faster for customer invoices in Approach 1)
- Provides complete customer view in single read
- Ideal when customer is the primary focus

Transaction-Centric Approach 1 (Embedded Country):

- Best for invoice-level operations and product analytics
- Fast for country-based queries
- Simple structure but has data redundancy

Transaction-Centric Approach 2 (Normalized):

- Best for production systems requiring data consistency
- Eliminates redundancy and follows normalization principles
- Dramatically improves customer queries with proper indexing (7x faster)
- Trade-off: slower country queries due to $lookup overhead

In production:

For most production systems, use a hybrid strategy:

- **Transaction-centric (Approach 2)** for transactional data and order management
- **Customer-centric** for customer analytics and CRM dashboards
- Cache frequently accessed data at application level to mitigate performance trade-offs

The key insight is that MongoDB's flexibility allows us to maintain both approaches simultaneously if needed, choosing the optimal structure for each specific use case rather than forcing everything into a single design pattern.

In My MLOps project also:

- when I am using ny db sql(Our team used rdbms) or nosql, We mostly normalize things, like user, media, ratings, etc.
- Normalizing is a best approch to save data in scale on clouds since it is expensive

So, At last there is not a certain way which is best, But according to Principles follwed by Systems and Clouds. Mostly Databases follows normalization at deeper levels for saving data in clouds which is expensive.