MLOps Assignment-4

Data Modeling and Performance Analysis with SQLite and MongoDB

Chakradhar Reddy
ID: 112201022

October 2, 2025

Contents

Abstract

This report details the process of designing, implementing, and analyzing database solutions for the UCI Online Retail Dataset. The assignment covers relational database modeling using SQLite in 2nd Normal Form, document-oriented modeling using MongoDB with two distinct approaches (transaction-centric and customer-centric), and a comparative performance analysis of CRUD operations on both MongoDB models. Finally, the customer-centric model is deployed to a cloud-hosted MongoDB Atlas cluster, and its successful implementation is verified. The findings highlight the critical trade-offs between relational and NoSQL databases and demonstrate how data modeling choices directly impact application performance.

1 Introduction & Data Preparation

The objective of this assignment is to work with the UCI Online Retail Dataset, a transactional dataset from a UK-based online retailer. The tasks involve modeling this data in both a relational (SQLite) and a document (MongoDB) database, performing a comparative analysis, and deploying the solution to a cloud service (MongoDB Atlas).

1.1 Dataset

The original dataset, the UCI Online Retail Dataset (https://archive.ics.uci.edu/dataset/352/online+r contains over 540,000 records with key attributes such as `InvoiceNo`, `StockCode`, `CustomerID`, and `Country`.

1.2 Initial Data Preparation

The original Excel (`.xlsx`) file was converted into a CSV format using the `convert_data.py` script. Rows with missing `CustomerID` were dropped to ensure data integrity for customer-centric modeling.

2 Question 1: Relational Database Design (SQLite)

A 2nd Normal Form (2NF) relational database was designed and implemented in SQLite.

2.1 Database Design (2NF)

The data was split into two tables to eliminate partial dependencies:

1. **Product Table**: `StockCode` (PK), `Description`, `UnitPrice`.
2. **Invoice Table**: `InvoiceNo`, `StockCode` (FK), `Quantity`, `InvoiceDate`, `CustomerID`, `Country`.

This design reduces data redundancy. The implementation in `question1.py` uses `executemany()` for efficient bulk insertion.

3 Question 2: Document-Oriented Design (MongoDB)

The data was modeled in MongoDB using two different document-oriented approaches.

3.1 Transaction-Centric Model

Each document represents a single invoice, embedding an array of items purchased in that transaction. This model is ideal for processing individual orders.

3.2 Customer-Centric Model

Each document represents a single customer, embedding an array of all their invoices. This model is optimized for retrieving a complete customer history in a single operation.

4 Question 3: Performance Analysis (CRUD Operations)

This section details how quickly each design performs standard database operations: Create, Read, Update, and Delete across both MongoDB collections.

4.1 Observations

- **Create Operations:**
  - **Bulk Insert:** The customer-centric design is significantly faster (`0.005815s` vs `0.011436s`) as it creates fewer overall documents, despite each one being larger.
  - **Single Insert:** Both designs perform similarly, with the transaction-centric model being slightly faster (`0.000514s` vs `0.000628s`) due to the smaller, simpler document structure.
- **Read Operations:**
  - **Customer Data Lookup:** The customer-centric design is **56% faster** (`0.000440s` vs `0.000688s`) because all data for a customer is retrieved in a single document read.
  - **Sales by Country:** The transaction-centric design is **48% faster** (`0.000398s` vs `0.000771s`) as aggregation on a flat document structure is more direct than on nested data.
  - **Invoice Lookup:** The transaction-centric design is **27% faster** (`0.000405s` vs `0.000556s`) since invoices are top-level documents, allowing for a direct and indexed lookup.
  - **Transaction Count:** The transaction-centric design is **52% faster** (`0.000702s` vs `0.001469s`) for simple counting operations.
- **Update Operations:**
  - **Customer Data:** The customer-centric design is **31% faster** (`0.000433s` vs `0.000628s`) because it only needs to update a single document.
  - **By Country:** The customer-centric design is **21% faster** (`0.004028s` vs `0.005107s`) for bulk updates affecting multiple customers within a country.
- **Delete Operations:**
  - **Customer Data:** The customer-centric design is **17% faster** (`0.000337s` vs `0.000406s`) as it can remove all of a customer's data in a single operation.
  - **By Country:** The customer-centric design is **15% faster** (`0.000934s` vs `0.001101s`) for bulk deletion tasks.

4.2 Conclusion

The optimal design depends entirely on the application's primary access patterns:

- Choose the **transaction-centric design** if your application frequently performs invoice-level lookups, transaction counting, or country-based sales analysis.

- Choose the **customer-centric design** if your application is focused on customer profiles, requires fast access to complete purchase histories, or performs frequent bulk updates/deletions on a per-customer basis.

4.3 Performance Results (Average Time in Seconds)

Table 1: Detailed benchmark results for CRUD operations.

Operation	Transaction-Centric	Customer-Centric
CREATE - Bulk Insert	0.011436	0.005815
CREATE - Single Insert	0.000514	0.000628
READ - Customer Data Lookup	0.000688	0.000440
READ - Sales by Country	0.000398	0.000771
READ - Invoice Lookup	0.000405	0.000556
READ - Transaction Count	0.000702	0.001469
UPDATE - Customer Data	0.000628	0.000433
UPDATE - By Country	0.005107	0.004028
DELETE - Customer Data	0.000406	0.000337
DELETE - By Country	0.001101	0.000934

5 Question 4: MongoDB Atlas Deployment

The customer-centric model was deployed to a MongoDB Atlas cluster to demonstrate cloud readiness.

5.1 Deployment Process and Verification

The `question4.py` script connects to the Atlas cluster using a secure connection string from an environment variable (`MONGODB_URI`). It processes a subset of the data and inserts it in batches.

To confirm a successful deployment, the `test_question4.py` script was executed. This script connects to the cluster and performs several checks. The output below serves as proof of a successful data load.

```
Total customer-centric documents in cluster: 20

Sample document:
{'country': 'Germany',
 'customer_id': '12472',
 'invoices': [{'invoice_date': '2010-12-01T14:33:00',
               'invoice_no': 'C536548',
               'items': [{'Description': '3 HOOK HANGER MAGIC GARDEN',
                          'Quantity': -4,
                          'StockCode': '22244',
                          'UnitPrice': 1.95},
                         {'Description': '5 HOOK HANGER MAGIC TOADSTOOL',
                          'Quantity': -5,
                          'StockCode': '22242',
                          'UnitPrice': 1.65},
                         {'Description': 'SET/5 RED RETROSPOT LID GLASS BOWLS',
                          'Quantity': -1,
```

```
                        'StockCode': '20914',
                        'UnitPrice': 2.95}],
              'total_amount': -122.3}],
 'total_spent': -122.3}

First 2 customers from India:

Distinct countries (up to 10): ['EIRE', 'Germany', 'Norway']
```

The output confirms that **20 unique customer documents** were inserted. The sample document shows a cancelled invoice (indicated by the 'C' prefix and negative values), demonstrating the model's ability to handle returns. The distinct country list is accurate for the data subset used, and the connection to the MongoDB instance is functional. The entire process is validated.