Big Data Storage for E-Commerce: Data Modeling, Analysis, and Benchmarking

David Daniel Chukwuemeka
Artificial Intelligence and Data Engineering
Innopolis University
Email: d.david@innopolis.university

***Abstract*—This report presents a comprehensive solution for managing big data within a global e-commerce context. It addresses the challenges posed by high data volume, variety, and the need to extract valuable insights for targeted marketing and operational efficiency. Our approach involves designing and implementing optimal data models using SQL and NoSQL databases — specifically PostgreSQL, MongoDB, and Neo4J — to support both real-time and batch processing. We discuss the data modeling process, experimental results from performance benchmarking, and critically analyze the trade-offs between the models, with the aim of suggesting the most scalable solution for future growth.**

I. Introduction

In this project, we address the data storage challenges faced by a global e-commerce company. The objective is to design and implement efficient data models for handling a large volume of heterogeneous data derived from customer transactions, social networks, and marketing campaigns. The project involves the use of Python, PostgreSQL, MongoDB, Neo4J and OrioeDB with the aim of benchmarking and comparing the performance of each model under various query loads.

II. Data Modeling

In this section, we describe the data models developed for the project across different databases. Each subsection details the specific modeling approach tailored to the respective database system.

A. PostgreSQL

The PostgreSQL schema is designed using a normalized relational model within the `ecommerce` namespace to ensure data integrity and support complex queries. Key design decisions include:

- **Users:** A minimal table containing only the unique `user_id` to simplify foreign key relationships.
- **UserFriends:** Captures mutual friendships with a unique composite key to prevent duplicates.
- **Events:** Logs user interactions with products including event type, timestamp, and session details.
- **Categories & Products:** `Categories` store metadata, while `Products` uses a merged key (product and category ID) for uniqueness.
- **Campaigns:** Contains general campaign data, with campaign-specific attributes divided into two tables: `BulkCampaignAttributes` (timing and recipient info) and `CampaignSubjectAttributes` (subject details).
- **UserDevices:** Connects users with their device information and purchase history.
- **Messages:** Stores campaign messages; boolean states (e.g., opened) are inferred from the presence of corresponding timestamp fields.

This design emphasizes normalization, efficient storage, and flexible querying, making it well-suited for handling high-volume e-commerce data.

Figure 1 illustrates the PostgreSQL data model.

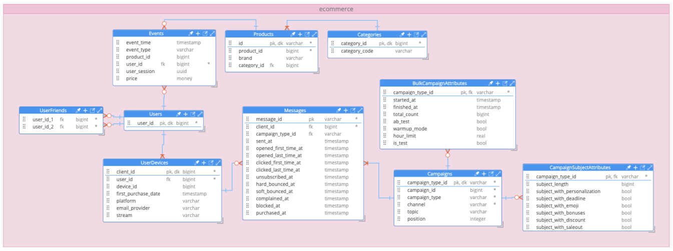

Fig. 1. PostgreSQL Data Model

B. MongoDB

The MongoDB data model adopts a document-oriented approach by storing data in JSON-like structures. The data was split into the following colections:

- **Users Collection:** Each document contains a unique `user_id`, an array of `friends`, and a nested array of `devices` with fields such as `device_id`, `first_purchase_date`, `platform`, `email_provider`, and `stream`.
- **Products Collection:** Documents embed the `category_code` from the `Categories` collection along with product details like `product_id`, `brand`, and `category_id`.
- **Events Collection:** Each document records a user event with fields including `event_time`, `event_type`, `product_id`, `user_id`, `user_session`, and `price`.
- **Campaigns Collection:** Campaign documents use the `campaign_type_id` as the identifier and

embed attributes from two nested sub-documents: `bulk_attributes` and `subject_attributes`.

- **Messages Collection:** Each document contains detailed timestamp information for each message (e.g., `sent_at`, `opened_first_time_at`, etc.), with date values stored in ISO string format.

This design is ideal for handling semi-structured data and rapid schema evolution, as well as for scaling horizontally in a distributed environment.

Figure 2 shows the MongoDB data model.

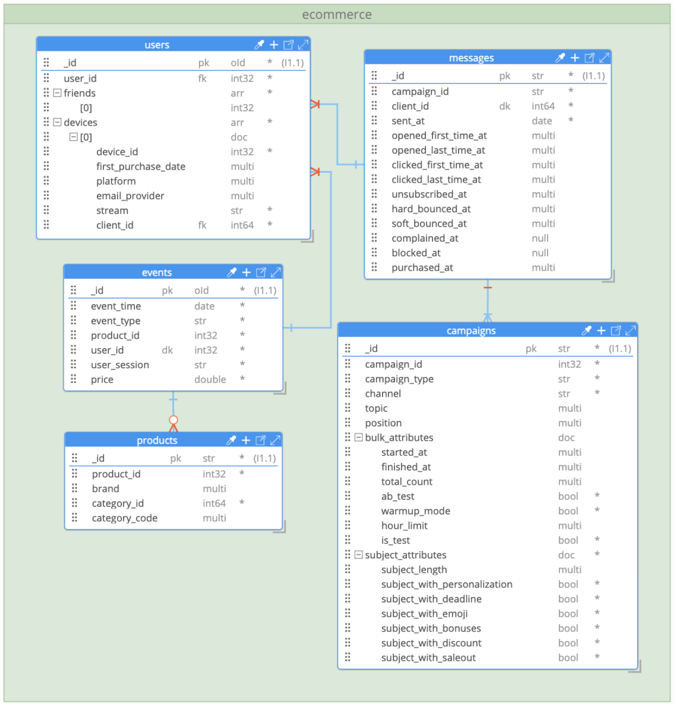

Fig. 2. MongoDB Data Model (JSON Schema)

C. Neo4J

The Neo4J data model is built on a graph-based paradigm that naturally represents relationships between entities. In our implementation:

- **Nodes:** Primary entities such as `User`, `Product`, `Campaign`, `Message`, `Category`, and `Device` are modeled as nodes. Unique constraints are enforced on key properties (e.g., `user_id` for Users, `product_id` for Products, `campaign_type_id` for Campaigns, and `message_id` for Messages) to maintain data integrity.
- **Relationships:** Explicit relationships are created to capture interactions and associations:
  - **FRIEND_WITH:** Connects Users to each other, modeling social networks.
  - **HAS_DEVICE:** Links Users to their Devices.
  - **RECEIVED:** Connects Devices and Users to Messages.
  - **SENT_MESSAGE:** Represents the action of a Campaign sending a Message.
  - **BELONGS_TO:** Connects Products to Categories.
  - **PERFORMED_EVENT:** Captures events where Users interact with Products.
- **Schema Constraints:** Our schema also includes constraints and indexes to ensure fast lookups and consistent data. These constraints prevent duplicate nodes and ensure that each entity's key identifier remains unique throughout the database.

Figure 3 presents the Neo4J graph model, which visually summarizes the nodes, relationships, and constraints used in the system.

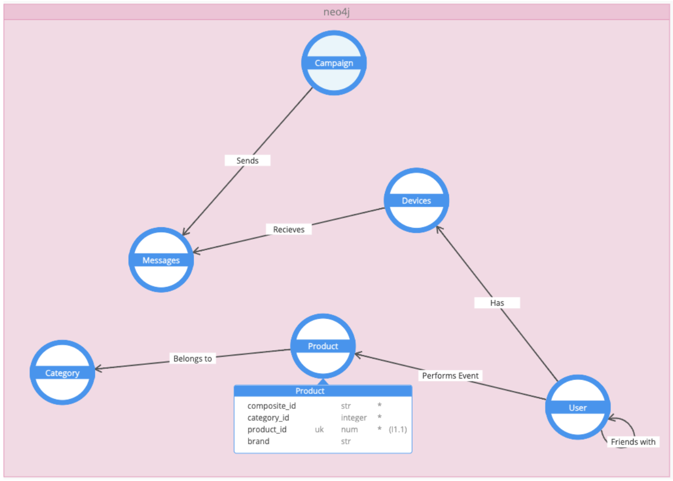

Fig. 3. Neo4J Data Model

D. Orioeldb

The Orioeldb model is a hybrid approach that combines features of relational and NoSQL systems to support both structured and semi-structured data. It balances normalization and denormalization to optimize for transactional consistency while enabling flexible and fast analytical queries. Key entities are indexed to accelerate performance and the model is designed to handle large data volumes without compromising query efficiency.

For Orioeldb data model, we use the same data model as PostgreSQL because OrioleDB is a storage extension for PostgreSQL which uses PostgreSQL's pluggable storage system. It is designed to be a drop-in replacement for PostgreSQL's existing storage engine

Each model has been developed using Hackolade to ensure optimal structure and scalability. The Hackolade files (.hck.json) have also been uploaded to the project's github repository.

III. TASK 1

The goal of this analysis is to determine whether marketing campaigns effectively led customers to purchase products.To solve this task we focus on:

- Counting the number of users who received each campaign (“users_received”).
- Counting how many users made a purchase after receiving the campaign (“users_purchased”).
- Calculating the purchase percentage (“purchase_percentage”) as a ratio of the above.

We then produce visualizations (pie charts) to depict these metrics (purchase ratio, campaign types, and channels). These insights help identify which campaign channel performed best and guide future engagement strategies.

A. *Database-Specific Approaches*

PostgreSQL: A single SQL query aggregates data from `Messages`, `UserDevices`, and `Campaigns`. It calculates the number of distinct users receiving campaigns and those who made a purchase, then computes the percentage of users who purchased. script- /scripts/q1.sql

MongoDB: An aggregation pipeline uses `$lookup` (to join `messages` with `campaigns`), `$group` (to compute distinct users for “received” and “purchased”), and a projection step to calculate the purchase percentage. script - /scripts/q1.js

Neo4j: A Cypher query `MATCH`es from `Campaign` nodes through `Message` relationships to `User` nodes. It collects unique users who received and purchased, then calculates a purchase percentage for each campaign. script - /scripts/q1.cypher

OrioeDB: Since OrioeDB is an extension of PostgreSQL, it uses the same SQL-based approach as PostgreSQL for Task 2, benefiting from similar performance and indexing features.

B. *Pie Chart Visualisations*

Figure 4 shows the overall purchase ratio. We can visualize what percentage of campaigns led to purchases.

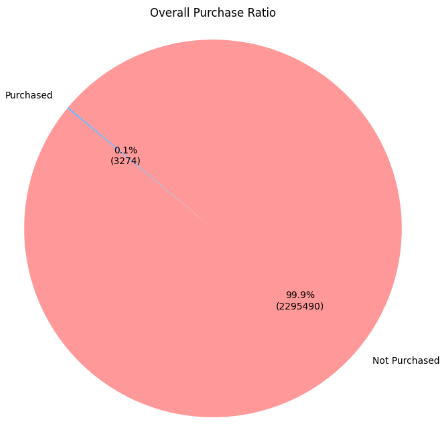

Fig. 4. Overall Purchase Ratio (Purchased vs. Not Purchased)

Figure 5 shows the message channel usage by the count of campaigns, with this we can visualize what channel was mostly used to send messages.

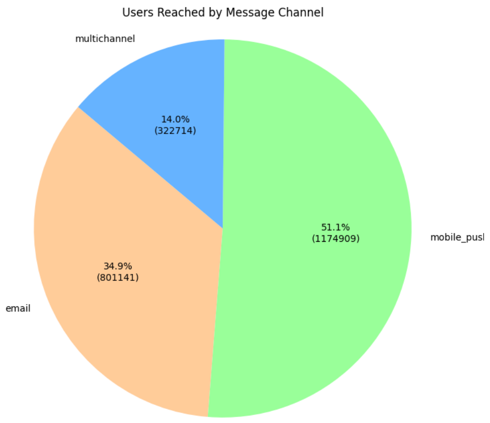

Fig. 5. Channel Usage (by Campaign Count)

Figure 6 illustrates which message channel lead to the most purchases.

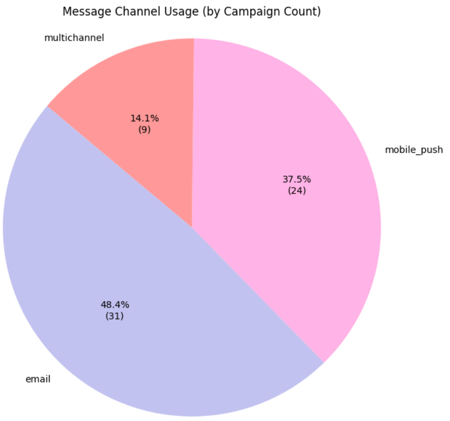

Fig. 6. Users that made purchases grouped by campaign channel

C. *Conclusion*

All three approaches return the same rows and produce similar aggregate results, confirming the reliability of the methodology across different data stores (PostgreSQL, MongoDB, and OrioelDB. It is evident from the visualisations that the channel which reached the most users (`opened_at` not null) is **mobile push**.

Thus, for upcoming campaigns, engaging customers through social networks—especially by sending mobile push notifications—is a highly recommended strategy for enhancing customer engagement and driving product purchases.

IV. Task 2: Personalized Recommendations and Full Text Search

A. General Approach

For Task 2, the objective is to generate personalized product recommendations for users based on their interactions ("view" and "purchase" events) and to enable full text search on product categories. The general steps include:

1) **Event Aggregation:** Aggregate user events by counting the number of views and purchases per product.
2) **Ranking:** Rank products per client based on the interaction counts using window functions (in SQL) or equivalent accumulators (in NoSQL).
3) **Enrichment:** Join the aggregated data with product and category details.

This approach is implemented across different database systems by exploiting each system's unique query and indexing capabilities.

B. Database-Specific Approaches

PostgreSQL: Utilizes Common Table Expressions (CTEs), window functions (e.g., `ROW_NUMBER()`) and joins to aggregate and rank user events. The full text search is performed using PostgreSQL's native text search functions applied to the `category_code` field. Scripts for Task 2 are located in `/scripts/q2.sql`.

MongoDB: Employs an aggregation pipeline with stages such as `$match`, `$group`, `$lookup`, and the `$topN` accumulator to derive the top product recommendations per client. For full text search, a text index on the `category_code` field is used in conjunction with the `$text` operator. Scripts for Task 2 are available in `/scripts/q2.py`.

Neo4J: Uses Cypher queries to traverse from user nodes through event relationships to product nodes. Pattern matching and ranking techniques are applied to determine the top recommendations per client, while full text search is enabled through indexing and text matching on category properties. The corresponding scripts are provided in `/scripts/q2.cypher`.

OrioeDB: Since OrioeDB is an extension of PostgreSQL, it uses the same SQL-based approach as PostgreSQL for Task 2.

C. Conclusion

V. Experimental Setup and Benchmarking Results

The data analysis queries were run several times to evaluate the performance of the three data models. The experimental setup includes details on the system specifications, the testing environment, and the execution time of each query.

A. Test Machine and Software Specifications

- **Test Machine:** MacBook Pro (Apple M1) with 8GB RAM, CPU: Apple M1 (8-core) with a base clock of approximately 3.2 GHz, running macOS Sequoia (15.3.1 (24D70)).

B. Docker Environment Setup

- **Containerization:** All databases are deployed using Docker containers.
- **Docker Engine Specifications:**
  - Docker Desktop Version: 4.37.2 (179585)
  - Docker Engine Version: 27.4.0
  - Docker Compose Version: v2.31.0-desktop.2
  - Operating System: macOS (Apple Silicon)
  - Resource Allocation
    - CPU Architecture: arm64
    - Memory limit: 8GB
    - Swap : 4GB
    - CPU limit: 8
    - Disk usage limit: 59.6GB
- **Images Used:**
  - **PostgreSQL:** `postgres:latest` (17.4)
  - **MongoDB:** `mongo:latest` (8.0)
  - **Neo4J:** `neo4j:latest` (2025.02.0)
  - **OrioeDB:** `postures:latest` (latest-pg17)

C. Benchmarking Procedure

Each query was executed 5 times per database using the `time` command on a Linux subsystem. All database connections were verified prior to benchmarking. Screenshots of the query results for each database are provided in the Appendix.

D. Benchmark Results

Table I summarizes the average execution times and standard deviations for each query across the databases.

TABLE I
Benchmark Query Execution Times

Database	Query	Avg. Time (s)	Std. Dev. (s)
PostgreSQL	Q1	8.800	2.050
	Q2	2.071	0.238
	Q3	-	-
MongoDB	Q1	44.568	3.035
	Q2	0.004	0.007
	Q3	-	-
OrioeDB	Q1	18.037	2.251
	Q2	4.573	0.195
	Q3	-	-
Neo4J	Q1	-	-
	Q2	-	-
	Q3	-	-

E. Query Execution Times Chart

Figure 7 shows the chart comparing query execution times in seconds across the different databases.

VI. Discussion

In this section, we critically discuss and compare the performance of the implemented data models. We analyze the advantages and disadvantages of each approach in terms of data retrieval speed, scalability, and overall system performance.

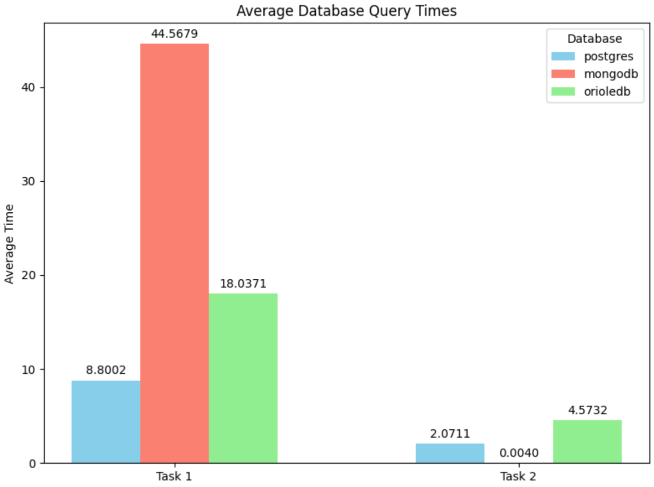

Fig. 7. Query Execution Times Comparison (in seconds)

VII. Conclusion

The report concludes by summarizing key insights from data modeling, experimental analysis, and benchmarking efforts. Recommendations are provided for the most efficient and scalable data model for real-time and batch processing in a high-volume e-commerce environment. Unfortunately I couldn't upload the data from the csv files to Neo4j because it requires about 10gb of RAM, therefore I couldn't benchmark this database.