Assignment №2

Anastassiya Luzinsan
a.luzinsan@innopolis.university

Abstract

This report compares data models for e-commerce data storage and analysis, focusing on PSQL, MongoDB, Neo4j. We design and implement models for each, perform data cleaning and loading, and execute analytical queries. The report details experimental setup, benchmark results (execution times, performance), and a discussion of each model's strengths and weaknesses for e-commerce needs.

1 Introduction

E-commerce data volume and complexity demand efficient data storage and analysis solutions. This report investigates PSQL, MongoDB, and Neo4j, to determine their suitability for key e-commerce tasks. The goal is to compare data models and technologies based on these specific analysis requirements.

2 Motivation

E-commerce businesses require efficient management and analysis of diverse data for personalized recommendations, targeted marketing, and product discovery. This comparison of database systems and models informs architectural decisions to meet evolving e-commerce platform demands.

3 Data

The dataset for this assignment was compiled from multiple sources on Kaggle (Kechinov, 2024), combining e-commerce behavioral and marketing campaign data. It consists of the following files:

- events.csv: captures user behavior (views, purchases, etc.) from October to December 2019, linking users and products with timestamps and session identifiers.
- campaigns.csv: contains marketing campaign details (type, channel, topic) with properties including subject length, presence of subject content such as personalization, deadlines, emojis, bonuses, discounts, or sales announcements.
- messages.csv: details messages sent in campaigns, linking to campaign ID, message type, channel, recipient client ID, and device platform. Includes behavior metrics with corresponding timestamps.
- client_first_purchase_date.csv: specifies the date of the first purchase for each client.
- `friends.csv`: Stores mutual friendships between users via user ID pairs.

4 Data Modeling

This section describes the data models designed for each database system: PSQL, MongoDB, and Neo4j. The models were created using Hackolade.

4.1 PSQL Data Model

The PSQL data model, provided in Figure 1, follows a relational schema designed for efficient querying and data integrity. The schema is organized into 13 tables within the e-commerce schema, reflecting the different entities and relationships within the e-commerce dataset.

The tables and their relationships are as follows:

- **products** and **product_cards**: The separation of `products` and `product_cards` driven by the business logic of e-commerce product listings. The `products` table stores core, brand-agnostic product information: a unique `product_id`, `category_id`, and `category_code`. This represents the abstract concept of a product category, irrespective of brand. Then, the `product_cards` table introduces the brand dimension. A single abstract `product` (e.g., a specific model of vacuum cleaner) can have multiple `product_cards`, each representing a listing from a different `brand`. This one-to-many relationship, enforced by the foreign key from `product_cards` to `products`, is crucial for handling scenarios where the same product is offered by various brands, each with potentially different pricing or marketing.
- **clients**. Uniquely identified by `client_id`, this table not only stores this primary key, but implicitly contains the `user_id` and `user_device_id`, also including the nullable ‘first_purchase_date‘. Implicitly means that user-related IDs are not stored as separate columns for normalization purposes, they are encoded within the `client_id` itself.
- **users**. The `users` table tracks all platform users, providing a central link for analyzing user behavior across browsing, purchasing, and campaign interactions.
- **friends**. The `friends` table models the social network aspect of the e-commerce platform, representing mutual friendships between `users`.
- **events**. Each event record signifies an interaction (view, purchase, etc.) between a `user` and a `product_card` at a specific `event_time`. By linking to `product_cards` rather than directly to `products`, we accurately record user interaction with a specific *branded* product listing. This entity captures the `price`

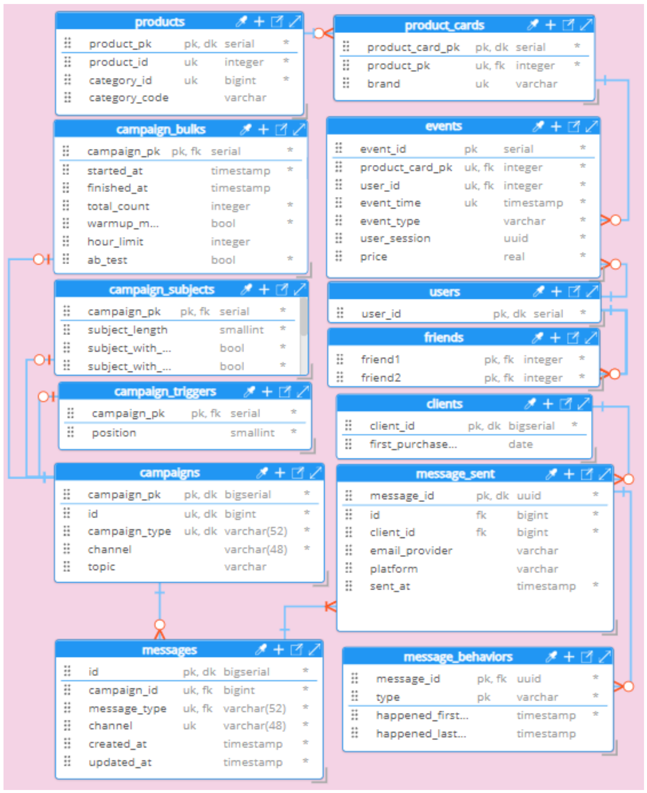

Figure 1. Relational model for e-commerce

at the time of the event. This reflects the business reality of dynamic pricing and allows for analysis of price sensitivity in user behavior and purchase decisions.

- **campaigns**, **messages**, **message_sent**, and **message_behaviors** are the set of tables models the marketing campaign aspect of the e-commerce platform. `campaigns` defines the overall marketing initiatives (bulk, trigger, transactional), characterized by an `id`, `campaign_type`, `channel`, and `topic`. `messages` then acts as an abstraction, defining the *type* and *channel* of message within a campaign (e.g., a specific email template). This separation is important for scalability as a single campaign can have multiple message types, and message types can be reused across campaigns. The one-to-many relationship from `campaigns` to `messages` reflects this. `message_sent` tracks the *actual sending* of individual messages to `clients`. This table is crucial for tracking message delivery and engagement at the individual recipient level. Finally, `message_behaviors` records user interactions *with* those sent messages (opens, clicks, etc.).
- **campaign subtypes** (**campaign_bulks**, **campaign_subjects**, **campaign_triggers**). Using these tables linked one-to-one with `campaigns` is a design pattern to handle the varying attributes of different `campaign_types` and `channels`. Bulk campaigns have attributes like ‘started_at‘, ‘finished_at‘, and ‘total_count‘ that are irrelevant to trigger or transactional campaigns. Similarly, subject-related attributes are specific to email and push channels. This vertical partitioning ensures that the main `campaigns` table remains generalized, while subtype tables hold specific attributes, reducing null values and improving storage efficiency.

This relational model emphasizes data normalization and referential integrity, enabling complex queries and analysis across different aspects of the e-commerce data. Its strengths are evident in its robust handling of the product catalog through separate `products` and `product_cards` tables for granular product and brand analysis, the unified customer view provided by the consolidated `clients` table, and the comprehensive marketing analysis framework enabled by the normalized `campaigns` suite. Data integrity is a core strength, ensured by the relational structure and normalization. However, weaknesses include a potential lack of schema flexibility for rapidly evolving e-commerce needs, potentially lower performance for graph-heavy queries related to social features, and the complexities associated with horizontal scaling for massive data volumes. Furthermore, the inherent complexity of relational model design can be a consideration during initial setup. Therefore, the PSQL model is a powerful and reliable choice, especially for businesses prioritizing data integrity and complex relational analysis, but its limitations regarding agility and extreme scalability must be carefully evaluated based on specific e-commerce platform requirements.

4.2 MongoDB Data Model

The MongoDB data model, provided in Figure 2, leverages a document-oriented approach to provide flexibility and scalability for e-commerce data, prioritizing denormalization and embedded structures for efficient querying of typical e-commerce workloads.

- **events collection**. Storing `event_time`, `event_type`, `product_pk`, `user_id`, `user_session`, and `price` within a single document optimizes write operations for high-volume event streams. The schema-less nature of MongoDB allows for easy addition of new event types or attributes in the future without schema migrations, crucial for adapting to evolving business needs and tracking new user behaviors. Indexing on `event_time`, `product_pk`, and `user_id` ensures efficient retrieval for time-series analysis and user-centric behavior queries.
- **campaigns collection**. The `campaigns` collection stores campaign details with embedded sub-documents for campaign-type-specific attributes (`bulk_details`, `trigger_details`, `subject_details`). This flexible schema directly maps

to the varying nature of marketing campaigns, where different types have distinct properties. Embedding these details within the main campaign document reduces the need for joins and simplifies retrieval of complete campaign information. MongoDB's document structure readily accommodates the addition of new campaign types or channel-specific attributes, offering agility in managing diverse marketing initiatives. The unique index on `id` and `campaign_type` enforces campaign uniqueness within the system.

- **products collection**. The `products` collection stores essential product catalog data, including `product_pk`, `product_id`, `brand`, `category_id`, and `category_code`. This collection focuses on core product attributes, optimized for fast retrieval of product information by `product_id` or `category`. The unique index on `product_id`, `brand`, and `category_id` maintains product uniqueness, mirroring business rules for product identification.
- **users collection**. The `users` collection embeds device information and first purchase date within each user document using the `devices` array. This denormalization groups all relevant user context into a single document, minimizing read operations for user profile retrieval. MongoDB's flexible array structure allows for users to have multiple devices associated with their profile. Indexing on `user_id` and a compound index on `devices.client_id` and `devices.client_device_id` ensures efficient user lookup and client device retrieval.
- **messages collection**. The `messages` collection stores detailed information for each sent message, including campaign linkages, client details, channel information, timestamps, and an embedded array of `behaviors`. Embedding the `behaviors` array within the message document is a key design choice for MongoDB, enabling efficient retrieval of complete message engagement data without joins. This structure optimizes queries related to message performance and user interactions with marketing communications. The unique index on `message_id` ensures each message is uniquely tracked.
- **friends collection**. The `friends` collection directly represents user friendships as documents, storing pairs of `friend1` and `friend2` user IDs. This simple structure, indexed with a unique index on `friend1` and `friend2`, is optimized for fast retrieval of friendship connections, supporting social feature implementations within MongoDB's document model.

In conclusion, the MongoDB data model prioritizes schema flexibility and denormalization to optimize read and write performance for typical e-commerce operations. Compared to the PSQL relational model, MongoDB offers greater agility

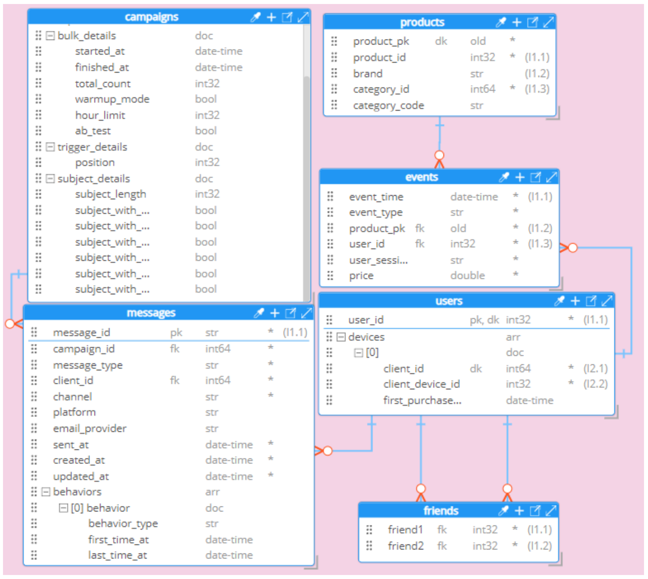

Figure 2. MongoDB e-commerce model

in adapting to evolving data structures and can excel in handling high-volume, schema-flexible data. However, it sacrifices the strong data integrity constraints and complex relational query capabilities inherent in PSQL. Compared to a graph database like Neo4j, MongoDB provides a more document-centric approach, potentially less optimized for deep graph traversals but simpler for general e-commerce data management and queries focused on individual entities and their immediate attributes and embedded relationships. The choice between these models depends heavily on the specific priorities of the e-commerce platform: data agility and scalability versus strong data integrity and complex relational analysis.

4.3 Neo4j Data Model

The Neo4j data model, visualized in Figure 3, is designed to represent the e-commerce data as a graph, emphasizing relationships between entities. This approach is fundamentally different from the relational (PSQL) and document-oriented (MongoDB) models, focusing on connections and traversals rather than tables or documents.

Nodes - the core entities are represented as nodes, with labels indicating their type:

- **message**: Represents individual messages sent to clients. Properties include ‘message_id‘ (Node Key), ‘campaign_id‘, ‘message_type‘, and ‘channel‘.
- **user**: Represents individual users of the platform. The key property is ‘user_id‘ (Node Key).
- **client**: Represents clients who receive messages. Properties include ‘client_id‘ (Node Key) and ‘first_purchase_date‘.

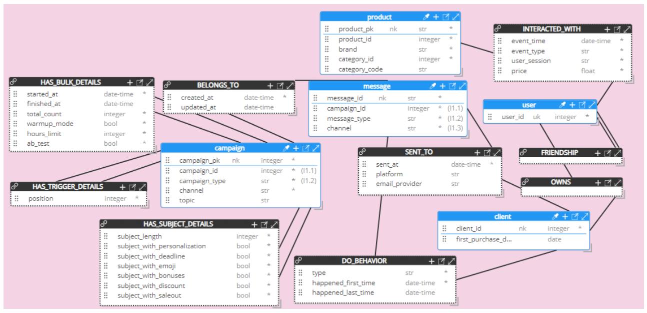

Figure 3. Neo4J data model

- **campaign**: Represents marketing campaigns. Properties include ‘campaign_pk‘ (Node Key), ‘campaign_id‘, ‘campaign_type‘, ‘channel‘, and ‘topic‘.
- **product**: Represents products. Properties include ‘product_pk‘ (Node Key), ‘product_id‘, ‘brand‘, ‘category_id‘, and ‘category_code‘.

Relationships - connections between nodes are represented by *typed* relationships, each with a specific meaning:

- **BELONGS_TO**: Connects a ‘message‘ to the ‘campaign‘ it belongs to. Properties: ‘created_at‘, ‘updated_at‘.
- **SENT_TO**: Connects a ‘message‘ to the ‘client‘ who received it. Properties: ‘sent_at‘, ‘platform‘, ‘email_provider‘.
- **OWNS**: Connects a ‘user‘ to a ‘client‘, representing the user’s ownership of a client profile (if they are distinct entities).
- **INTERACTED_WITH**: Connects a ‘user‘ to a ‘product‘, representing a user interaction. Properties: ‘event_time‘, ‘event_type‘, ‘user_session‘, ‘price‘.
- **FRIENDSHIP**: Connects two ‘user‘ nodes, representing a mutual friendship.
- **DO_BEHAVIOR**: Connects a ‘client‘ to a ‘message‘, indicating a specific behavior (e.g., "click", "open", "purchased"). Properties: ‘type‘, ‘happened_first_time‘, ‘happened_last_time‘.
- **HAS_BULK_DETAILS**, **HAS_TRIGGER_DETAILS**, **HAS_SUBJECT_DETAILS**: These relationships connect a ‘campaign‘ node to nodes containing additional, type-specific details.

In conclusion, the Neo4j data model excels at representing and querying relationships within the e-commerce data. Compared to the PSQL model, Neo4j offers significantly better performance for graph traversal and relationship-based queries, such as finding friends of purchasing users or identifying complex patterns of user behavior. Compared to the MongoDB model, Neo4j provides a more natural and efficient way to represent and query highly interconnected data, making it ideal for social network analysis and recommendation engines. However, Neo4j is not optimized for the kinds of structured, transactional operations or aggregations where PSQL excels, and it lacks the schema flexibility of MongoDB for rapidly evolving data structures. The best choice depends on the dominant query patterns and the importance of relationships in the e-commerce data analysis.

5 Data Cleaning and Loading

The data cleaning and preprocessing formed a foundation for this assignment, ensuring data quality, consistency, and compatibility across the three target database systems: PSQL, MongoDB, and Neo4j. All data cleaning and transformation logic is encapsulated within the Python script `clean_data.py`, located in the `/scripts` folder.

The `clean_data.py` script performs a series of operations to transform the raw CSV data into formats tailored for each database. The process begins with loading the raw data files (`messages.csv`, `campaigns.csv`, `events.csv`, `client_first_purchase_date.csv`, and `friends.csv`) using the `pandas` library. During this initial data ingestion, `clean_data.py` applies appropriate data type mappings and date parsing, ensuring that timestamps, boolean values, and categorical data are correctly interpreted. Specifically, date and time columns are parsed using the `parse_dates` and `date_format` parameters of the `pandas.read_csv` function, and boolean values, represented as ’t’ and ’f’ in the raw data, are converted to Python’s boolean type. The script also addresses formatting inconsistencies, such as correcting the non-standard UUID format in the `message_id` column using regular expressions.

For the PSQL relational model, `clean_data.py` undertakes several normalization steps. Unique, auto-incrementing IDs are generated for messages based on a combination of `campaign_id`, `message_type`, and `channel` providing a primary key for the `messages` table. The `campaigns` data undergoes vertical partitioning, separating campaign details into distinct tables (`campaign_bulks`, `campaign_subjects`, `campaign_triggers`) based on campaign type, minimizing null values and storage redundancy. Additionally, surrogate keys, namely `product_pk` and `product_card_pk`, are generated for products, facilitating the relational structure.

For MongoDB, a document-oriented database, the script prepares the data for denormalization and embedding. Behavior data related to messages (clicks, opens, purchases) is transformed into a list of embedded documents within each message document, optimizing for retrieval of message and related behavior in a single query. Similarly, details from the `campaigns` dataset, such as bulk, trigger, and subject-specific attributes, are embedded directly within campaign documents, further reducing the need for joins in typical query patterns.

For Neo4j, a graph database, `clean_data.py` prepares data for CSV import using a specialized format. This involves creating separate CSV files for each node type (e.g., message, user, client, campaign, product) and relationship type (e.g., BELONGS_TO, SENT_TO, FRIENDSHIP). Helper functions within the script transform the pandas DataFrames into the required format, which includes adding a :LABEL column for node types and :ID, :START_ID, :END_ID, and :TYPE columns for relationships, as required by Neo4j's import tools. The script also handles the specific requirements of the Neo4j model, ensuring that relationship directions are correctly represented and preparing composite keys as needed for certain relationships.

Finally, the script ensures data type consistency across all three databases and handles missing values appropriately. The processed data is then saved to CSV files for PSQL and Neo4j, and to JSON files for MongoDB, placing them in their respective output directories: output/psql, output/mongo, and output/neo4j.

5.1 PSQL Data Loading

The SQL script load_data_psql.sql, found in the /scripts folder, provides the instructions for both creating the PSQL database schema and populating it with the cleaned data. It begins by creating the e_commerce schema and setting it as the default search path. Subsequently, the script defines all tables within the schema (products, product_cards, users, events, campaigns, messages, clients, friends, campaign_bulks, campaign_subjects, campaign_triggers, message_sent, and message_behaviors). Each table definition includes column names, data types, primary key constraints, foreign key constraints (to maintain referential integrity), and unique constraints and check constraints. The script uses \COPY to import data from the CSV files created by clean_data.py into each corresponding table.

5.2 MongoDB Data Loading

The JavaScript script load_data_mongodb.js, located in the /scripts folder, defines the structure of the MongoDB collections, including schema validation rules and indexes, but the actual loading process is handled separately via the mongoimport utility. The script creates the collections: events, campaigns, products, users, messages, and friends.

For each collection, the script includes an optional JSON Schema validator using the $jsonSchema operator. This validator enforces basic data type and structure requirements, contributing to data quality within MongoDB's flexible schema environment. Additionally, the script creates indexes on frequently queried fields, such as a unique index on the combination of event_time, product_pk, and user_id in the events collection.

The actual data loading into MongoDB is accomplished using a series of mongoimport commands, executed from the command line. These commands, found in the shell script load_mongo_data.sh, import the JSON files generated by the clean_data.py script into their respective collections.

5.3 Neo4j Data Loading

The Cypher script load_data_neo4j.cypher, located in the /scripts folder, defines the constraints and indexes for the Neo4j database. The actual loading of data is then performed using the neo4j-admin import tool from the command line. This approach is chosen for its efficiency in handling large-scale imports into Neo4j. The shell script load_data_neo4j.sh includes the command line for running neo4j-admin utility.

The load_data_neo4j.cypher script first creates the required constraints and indexes in the Neo4j database, defining the overall schema structure. This ensures uniqueness for key node properties (using Node Key constraints) and enforces NOT NULL constraints where appropriate. Indexes are created on fields that are commonly used in queries.

6 Data Analysis Tasks

This section details the analysis performed to address the first research question: determining campaign effectiveness and exploring the potential of leveraging social network data. The analysis was conducted using Python scripts that interact with the PSQL database via its respective Python driver psycopg2-binary. The Python script along with the query are located in the /scripts/analysis/ folder. Due to time constraints, only Task 1 for psql was fully implemented.

6.1 Task 1: Campaign Effectiveness

The primary goal of this task was to analyze whether marketing campaigns effectively attracted customers to purchase products. A secondary goal was to explore how to incorporate social network information to improve future campaign targeting.

6.1.1 PSQL Analysis.

The PSQL analysis was performed using the SQL query located in /scripts/analysis/q1.sql. This query, reproduced below, leverages SQL's strengths in joining and aggregating data across multiple tables:

```
WITH CampaignInteractions AS (
    SELECT DISTINCT
        ms.client_id,
        m.campaign_id,
  c.campaign_type,
      mb.happened_first_time AS interaction_time,
        mb.type AS interaction_type
    FROM e_commerce.message_sent ms
    JOIN e_commerce.messages m ON ms.id = m.id
    JOIN e_commerce.message_behaviors mb
        ON ms.message_id = mb.message_id
    JOIN e_commerce.campaigns c
        ON m.campaign_id = c.id
        AND m.message_type = c.campaign_type
    LEFT JOIN e_commerce.campaign_bulks cb
        ON c.campaign_pk = cb.campaign_pk
  WHERE mb.happened_first_time BETWEEN ms.sent_at
        AND (ms.sent_at + INTERVAL '30 days')
  AND (c.campaign_type <> 'bulk' OR
     mb.happened_first_time BETWEEN cb.started_at
            AND COALESCE(cb.finished_at, NOW()))
),
PurchasingUsers AS (
    SELECT DISTINCT
     SUBSTRING(client_id::varchar(19), 10, 9)::int
           AS user_id,
        campaign_id,
        campaign_type
    FROM CampaignInteractions
 WHERE interaction_type = 'purchased'
),
FriendPurchases AS (
    SELECT DISTINCT
        f.friend2 AS friend_user_id,
        pu.campaign_id,
        pu.campaign_type
    FROM PurchasingUsers pu
    JOIN e_commerce.friends f
        ON pu.user_id = f.friend1
    JOIN PurchasingUsers pu2
        ON f.friend2 = pu2.user_id
        AND pu.campaign_id = pu2.campaign_id
)
SELECT
    c.id AS campaign_id,
    c.campaign_type,
   COUNT(DISTINCT ms.client_id) AS total_messages,
    COUNT(DISTINCT CASE WHEN ci.interaction_type
        IN ('clicked', 'opened')
        THEN ci.client_id END)
        AS clients_with_interaction,
   COUNT(DISTINCT pu.user_id) AS users_purchased,
    COUNT(DISTINCT fp.friend_user_id)
        AS friends_who_also_purchased,
 CASE
        WHEN COUNT(DISTINCT ms.client_id) > 0
         THEN (COUNT(DISTINCT pu.user_id) * 100.0)
                / COUNT(DISTINCT ms.client_id)
        ELSE 0
    END AS conversion_rate
FROM e_commerce.campaigns c
JOIN e_commerce.messages m ON c.id = m.campaign_id
    AND c.campaign_type = m.message_type
JOIN e_commerce.message_sent ms ON m.id = ms.id
LEFT JOIN CampaignInteractions ci
    ON ms.client_id = ci.client_id
    AND m.campaign_id = ci.campaign_id
LEFT JOIN PurchasingUsers pu
    ON SUBSTRING(ms.client_id::varchar(19),
                       10, 9)::int = pu.user_id
    AND m.campaign_id = pu.campaign_id
LEFT JOIN FriendPurchases fp
    ON pu.user_id = fp.friend_user_id
    AND pu.campaign_id = fp.campaign_id
GROUP BY c.id, c.campaign_type
ORDER BY conversion_rate DESC
LIMIT 10;
```

The query uses Common Table Expressions (CTEs) to improve readability and modularity. ‘CampaignInteractions‘ identifies clients who interacted with messages within a 30-day attribution window after the message was sent, correctly accounting for the ‘finished_at‘ date of bulk campaigns (if available) and handling ongoing campaigns. ‘PurchasingUsers‘ extracts the ‘user_id‘ from the ‘client_id‘ using ‘SUBSTRING‘ and identifies users who made purchases after interacting with a campaign. ‘FriendPurchases‘ then identifies friends of these purchasing users who *also* made purchases within the same campaign, leveraging the ‘friends‘ table. The final ‘SELECT‘ statement aggregates these results by ‘campaign_id‘ and ‘campaign_type‘, calculating the total number of messages, the number of clients with interactions, the number of users who purchased, the number of friends of purchasing users who also purchased, and a conversion rate.

The corresponding Python script, also in `/scripts/analysis/q1.py`, connects to the PSQL database using psycopg2, executes the SQL query, and prints the results in a formatted table.

The approach demonstrates how social network data can be incorporated into campaign effectiveness analysis within a relational database, even without native graph database features. It highlights the strengths of PSQL in handling structured data and complex SQL joins.

7 Experimental Results

This section presents the experimental setup and the results of executing the campaign effectiveness analysis query (Task 1) on the PSQL database. Due to performance issues with the initial MongoDB aggregation query and the incomplete execution of the Neo4j query, we were unable to obtain comparable timing results for those databases at this stage. This section focuses on presenting and analyzing the PSQL results, while acknowledging the limitations in comparing across all three database systems.

7.1 Experimental Setup

The experiments were conducted on a single machine with the following specifications:

- **Operating System:** Windows 11 Pro, Version 21H2, Build 22000.2538
- **Processor:** Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
- **RAM:** 40.0 GB (39.9 GB usable)
- **System Type:** 64-bit Operating System, x64-based processor

The following software versions were used:

- **PSQL:** PostgreSQL 17.4
- **MongoDB:** 8.0.4 (mongosh 2.3.9)
- **Neo4j:** 5.24.0
- **Python:** 3.11.0
- **Python Libraries:** `psycopg2-binary` (2.9.10), `pymongo` (4.11.2), `neo4j` (5.28.1)

The databases were accessed through Python scripts using their respective drivers. The development environment was Visual Studio Code, running directly on the host machine (no virtualization or containerization was used).

7.2 Benchmarking Methodology

The PSQL query for Task 1 (analyzing campaign effectiveness) was executed five times. Execution times were measured. Average execution time and standard deviation were calculated from these five runs. Due to the aforementioned issues, comparable timing data was not collected for MongoDB and Neo4j.

7.3 Results

The results of the PSQL query execution are summarized in Tables 1 and 2. The average execution time for the PSQL query was 23.2581 seconds, with a standard deviation of 1.5918 seconds. This indicates some variability in execution times, likely due to system load and caching effects.

Table 1. PSQL Query Execution Times (Task 1)

Run	1 (s)	2 (s)	3 (s)	4 (s)	5 (s)
PSQL	24.9133	24.6906	22.9823	21.0416	22.6628

Table 2. PSQL Query Performance Summary (Task 1)

Database	**Avg. Exec. Time (s)**	**STD (s)**
PSQL	23.2581	1.5918

Output for campaign effectiveness analysis:
The output, presented in Table 3, shows the effectiveness metrics for the top 10 campaigns, sorted by conversion rate. Key metrics reported include:

- **campaign_id**: The unique identifier of the campaign.
- **campaign_type**: The type of the campaign (e.g., 'transactional').
- **total_messages**: The total number of messages sent for that campaign.
- **clients_with_interaction**: The number of distinct clients who interacted (clicked or opened) with messages from the campaign.

Table 3. PSQL Campaign Effectiveness Results

Camp ID	Type	Tot.Msgs	Inters.	Purchs	FrsPurs	Conv(%)
33	trans.	319	188	9	0	2.82
29	trans.	21121	7980	393	0	1.86
31	trans.	647	349	12	0	1.85
32	trans.	24717	11716	425	1	1.72
58	trans.	350	235	6	0	1.71
59	trans.	255	152	4	0	1.57
27	trans.	26450	12392	365	0	1.38
28	trans.	951	512	13	0	1.37
122	trans.	369	240	5	0	1.36
55	trans.	1352	755	13	0	0.96

- **users_purchased**: The number of distinct users who made a purchase after interacting with a campaign message.
- **friends_who_also_purchased**: The number of distinct friends of purchasing users who also made purchases related to the *same* campaign.
- **conversion_rate**: A simple conversion rate, calculated as (users_purchased / total_messages) * 100.

From this output, we can observe variations in campaign performance. For example, campaign ID 33 has the highest conversion rate, while ID 55 has the lowest. We can also observe that the number of messages sent and clients with interaction will vary, but very few to none friends of buyers from one campaign made the purchase as well.

8 Discussion

The experimental results, primarily focused on the PSQL data model due to performance and implementation challenges with MongoDB and Neo4j, provide valuable insights into campaign effectiveness and the potential of leveraging social connections. However, it's crucial to interpret these results in the context of the limitations of the current analysis and the inherent differences between the database systems.

8.1 PSQL Analysis and Findings

The PSQL query (q1.sql), designed to measure campaign effectiveness and incorporate social network influence, successfully executed and yielded quantifiable results. The output, presented in Table 3, reveals significant variations in performance across different campaigns. Campaign ID 33 demonstrates the highest conversion rate (2.82%), while campaign ID 55 shows the lowest (0.96%). This suggests that certain campaigns, even within the same 'transactional' type, are significantly more effective at driving purchases than others. This finding immediately highlights the need for further investigation into the factors contributing to these performance differences. Are there differences in message content, targeting, timing, or other campaign parameters that explain the variations?

The query also provides data on the total number of messages sent, the number of clients who interacted with messages (clicked or opened), and the number of users who made purchases after interacting with a campaign. These metrics, when considered together, paint a more complete picture of campaign performance than conversion rate alone. For example, a campaign with a high number of messages sent but a low conversion rate might indicate issues with targeting or message relevance. Conversely, a campaign with a high conversion rate but a low number of messages sent might suggest a highly effective campaign that could benefit from increased reach.

The "friends_who_also_purchased" metric, while currently showing very low values (mostly 0 or 1 in the provided output), is a crucial aspect of the analysis. Its low values in the current data *do not* necessarily mean that social influence is unimportant. It suggests either: a) the social network data in the ‘friends‘ table is sparse or incomplete, b) the campaigns analyzed did not effectively reach interconnected users, or c) the 30-day attribution window might be too short to capture the full effect of social influence (which can be more delayed than direct campaign influence).

The use of CTEs (Common Table Expressions) in the PSQL query significantly improved its readability and maintainability. By breaking down the complex logic into smaller, named blocks (‘CampaignInteractions‘, ‘PurchasingUsers‘, ‘FriendPurchases‘), the query becomes easier to understand, debug, and modify.

8.2 MongoDB and Neo4j: Limitations

It’s important to acknowledge the limitations of the current analysis regarding MongoDB and Neo4j. The attempted MongoDB aggregation query suffered from severe performance issues, preventing its successful execution within a reasonable timeframe. This highlights a key difference between relational databases and document databases like MongoDB: while MongoDB excels at storing and retrieving flexible, semi-structured data, it is generally less efficient at performing complex relational joins, especially across multiple collections. The multiple ‘$lookup‘ operations and the need to unwind arrays contribute to the performance bottleneck. This finding underscores the importance of choosing the right database technology for the specific analytical task. While MongoDB’s schema flexibility is beneficial for storing diverse e-commerce data, complex relational analysis is better suited to PSQL’s structured approach.

For Neo4j, a data model was successfully implemented, and the database was populated with the preprocessed data. Basic connectivity and query execution were verified using the Neo4j Python driver and Cypher shell. A Cypher query for Task 1, designed to analyze campaign effectiveness and incorporate social network data, wasn’t developed due to time constraints within the assignment deadline, a comprehensive performance evaluation and analysis of the results for this specific query were not completed. While simple test queries against the Neo4j database were successful, confirming the correct setup and data loading, the more complex analysis required for Task 1, involving multiple node types and relationships, could not be fully executed and benchmarked before the submission deadline. Therefore, while the Neo4j database is operational and populated, comparable performance metrics and detailed results for Task 1 are not available at this time.