**Honors Project-Draft 1**

**CS 350 – Database Management Systems**

**Camila Fernandez Noguchi**

Supervised by: Professor Heikki Topi

**1-Introduction**

Today’s world consists of rapidly changing business models and an increase in data generation like never before. The current relational model used in SQL databases is great for transaction processing, creating reports and showing data in useful ways, but it lacks scalability and flexibility. To meet these demands, a new model called NoSQL has emerged. As a CIS major and honors student, I would like to explore a NoSQL database to deepen my knowledge in an area of database management that is becoming more relevant in the current years.

I plan to focus on MongoDB as my NoSQL database because of its wide usage in the industry and compare it to PostgreSQL, a traditional open-source relational DBMS. I will be assessing how they differ in terms of inserting large numbers of instances, creating reports, and modifying large amounts of data. I will set this project in a business scenario where the instances I insert, the reports I make, and the modifications I'll implement are relevant in the business sector.

**2-Characteristics of the test environment**

**2.1-PostgreSQL**

Version: PostgreSQL 13.0

Management Tool: pgAdmin 4

**2.2-MongoDB**

Version: MongoDB Community Edition 4.4

MongoDB Atlas

PyMongo 3.11

**2.3-Test Platform**

Operating system: macOS Catalina Version 10.15.7

RAM: 16 GB 2400 MHz DDR4

CPU: 2.3 GHz 8-Core Intel Core i9

SSD Capacity: 500 GB

**3-Database creation and population for MongoDB and PostgreSQL**

**3.1-Data Sourcing**

The data used for this project is based on a Peruvian furniture company, Drimer. Drimer produces mattresses, furniture, pillows, and other home décor. The database model used is based on a simplified version of the company’s selling strategy where we assume that they only sell three different products (mattresses, furniture, and pillows) on their own store. In reality, their product variety is much wider, and they are sold in a number of department stores around the country such as Ripley, Saga Falabella, and Oeschle.

Although the data entries on products, factories and stores are real data from the company as of 2018, the customer’s personal information is protected by law and could not be shared. To make up for it, customer data was generated to fill the fields of DNI (Peruvian identification number), name, surname, address and telephone number, and, in the case of juridical people, RUC (Peruvian identification number for companies).

I managed to get this information through a good friend whose family owns Drimer and has done database project using this model and data before. I have permission to use this information from both my friend who put this database together, and executives from the company.

Below is the contact information of Drimer’s Plant Superintendent Fernando Yi:

**Fernando Yi**

Drimer Plant Superintendent

Phone: +51 981292478

Email: [fyi@drimer.com.pe](mailto:fyi@drimer.com.pe)

**3.2-Data Model**

3.2.1-Relational Model for PostgreSQL

The model has been translated from Spanish to English but the primary key from “Juridical Person” has maintained its original name. RUC is the identification number for juridical people in Peru.

Diagram

Description automatically generated

3.2.2-BSON Model for MongoDB

Client

{

\_id:

name:

surname:

address:

phone:

}

JuridicalPerson

{

\_id:

name:

address:

representative\_id:

phone:

}

Order

{

\_id:

date:

client\_id:

product: [

{

product\_id:

store\_id:

quantity:

},

{

product\_id:

store\_id:

quantity:

}

]

}

Product

*In case the product is a mattress*

{

\_id:

detail:

price:

type: mattress

filling:

size:

}

*In case the product is a couch*

{

\_id:

detail:

price:

type: couch

model:

}

*In case the product is a pillow*

{

\_id:

detail:

price:

type: pillow

filling:

}

Store

{

\_id:

Address:

Product: [

{

product\_id:

stock:

},

{

product\_id:

stock:

}

]

}

Notes: In order to adapt the relational model previously showed to a dictionary/JSON format, I chose to embed tables with the criteria that the most common queries will be benefited by having the data embedded. For example, a query that will probably be used a lot is checking the products bought in an order. With the previous model, you had to join two tables to be able to see the products bought in each order. Nonetheless, I chose to not embed some tables like, for example, Product since I believe that a lot of queries will have to consult that collection independently and having it on its own makes it more efficient.

**3.3-Creation of Structure and Data Upload**

3.3.1-MongoDB

The database is stored in MongoDB’s cloud service, MongoDB Atlas, for free. I used PyMongo to connect to my cluster instead of the Mongo Shell because of the refactoring that had to be done to transform the original data in csv to the dictionary/JSON format required by MongoDB. I created a series of Python scripts for each collection (the equivalent of tables in MongoDB) to refactor the data and upload it to my cluster.

The python scripts can be found in GitHub through this link:

<https://github.com/camilaferno/CS350HonorsProject/tree/main/PyMongoScripts>

(See Exhibit A for Screenshots of “DrimerDB” in MongoDB Atlas)

3.3.2-PostgreSQL

The SQL scripts used for the table creation and population of the data can be found on GitHub through this link:

<https://github.com/camilaferno/CS350HonorsProject/blob/main/create.sql>

**3.4-Time Comparison Chart**

3.4.1-MongoDB

Note: Cluster is located in Oregon and the data was uploaded from Lima, Peru.

|  |  |
| --- | --- |
| **Collection** | **Time** |
| Client | 3.0548 seconds |
| JuridicalPerson | 3.05577 |
| Store | 3.13276 |
| Product | 3.64139 |
| Order | 5.93216 |
| **Total** | 18.81688 |

3.4.2-PostgreSQL

Connected via localhost

|  |  |
| --- | --- |
| **Table** | **Time** |
| Client | 0.05 seconds |
| JuridicalPerson | 0.051 |
| Store | 0.037 |
| Product | 2.312 |
| Order | 0.075 |
| Order\_Product | 0.138 |
| Couch | 0.036 |
| Natural\_Person | 0.035 |
| Pillow | 0.045 |
| Legal\_Representative | 0.045 |
| Juridical\_Person | 0.065 |
| Catalogue | 0.082 |
| **Total** | **2.971** |

PostgreSQL was considerably faster probably because of two reasons:

1. The database is hosted locally on my computer
2. There was no need of changing format. PostgreSQL lets you import a csv file without further modifications while for MongoDB the script needed to change the csv to a dictionary form.

**4-Experimentation**

**4.1-Reporting**

4.1.1-Report Description

We will be executing three queries: The first one will be a control query which in theory should be equally efficient in both SQL and MongoDB, the second one should run theoretically faster in SQL, and the third one should run theoretically faster on MongoDB.

Query #1 (control query):

Find all clients who have made an order after 2016.

This query should be efficient in both databases since it will only require to look for the value in a single table/collection.

Query #2 (favorable on SQL):

Find all clients who have bought more than 1 mattress in the same order.

This query should be more efficient in SQL because it requires joining two table/collections. In MongoDB joins are often discouraged because of its document format.

Query #3 (favorable on MongoDB):

Find which store has the most stock the product with id = 2115020691.

This query takes advantage of MongoDB’s flexible data model. While in SQL we will be to execute JOIN statement, MongoDB has *product\_id* nested on the *Store* collection.

4.1.2-Report Scripts

For SQL:

Query #1

SELECT client\_id

FROM Order

WHERE order\_date > "2016-12-31";

Query #2

SELECT "Order".client\_id

FROM "Order"

WHERE 1 < (

SELECT COUNT(Mattress.product\_id)

FROM Mattress, Order\_Product, Client

WHERE Mattress.product\_id = Order\_Product.product\_id

AND Order\_Product.order\_id = "Order".order\_id

AND "Order".client\_id = Client.client\_id)

Order by "Order".client\_id;

Query #3

SELECT Store.store\_id, address

FROM Store

JOIN Catalogue ON Store.store\_id = Catalogue.store\_id

WHERE product\_id = '2115020691' AND stock = (

SELECT MAX(stock)

FROM Store

JOIN Catalogue ON Store.store\_id = Catalogue.store\_id

WHERE product\_id = '2115020691'

);

The queries can also be found on GitHub with the following link:

For MongoDB:

Query #1

collection = db["Order"]

start = datetime(2016, 12, 31, 0, 0, 0)

for order in collection.find({'date': {'$gt': start}}, {'client\_id': 1}):

print("client\_id: ", order['client\_id'])

Query #2

OrderCollection = db["Order"]

resultlist = []

counter = 0

for result in OrderCollection.aggregate([

{

'$lookup':

{

'from': "Product",

'localField': "product.product\_id",

'foreignField': "\_id",

'as': "result\_docs"

}

}

]):

mattress\_perorder = 0

for product in result['result\_docs']:

if product['type'] == 'mattress':

mattress\_perorder += 1

if mattress\_perorder > 1:

counter += 1

print(result['client\_id'])

break

print("Total Rows: ", counter)

Query #3

collection = db["Store"]

results = {}

for store in collection.find(

{"product.product\_id": "2115020691"},

{"product": {"$elemMatch": {"product\_id": "2115020691"}}

}):

results[store["\_id"]] = store["product"][0]["stock"]

maximum = max(results, key=results.get)

print("store\_id: ", maximum)

The complete script can be found on GitHub through this link:

4.1.3-Assessment

|  |  |  |
| --- | --- | --- |
| **Execution Time** | | |
|  | **MongoDB** | **PostgreSQL** |
| **Query #1** | 2.17099 seconds | 0.043 seconds |
| **Query #2** | 4.16519 seconds | 0.102 seconds |
| **Query #3** | 1.88648 seconds | 0.073 seconds |

**Note: PostgreSQL’s server is hosted locally while MongoDB’s server is in Oregon. The test took place from a computer in Lima, Peru. That may be why MongoDB queries take substantially longer than the SQL ones.**

Analysis:

Starting with MongoDB, we see that the query that took the longest is the second one. This was supposed to be costly for MongoDB because of the joins required to execute the query. Joins are in theory more difficult in MongoDB. However, Query #2 also took the longest on PostgreSQL. It is most probably because of the multiple joins that it required. While on MongoDB we just needed to join tables “Order” and “Product”, on the SQL model we had to connect the tables “Order”, “Mattress”, “Order\_Product”, and “Client”.

Another observation is that Query #3 took the least amount of time amongst the other queries on MongoDB while in PostgreSQL was the second to fastest. We had proposed that the third query will take less time on MongoDB than on PostgreSQL because in PostgreSQL we had to execute a join on tables “Store” and “Catalogue” while in MongoDB Catalogue’s data is nested on the collection “Store”. In conclusion, this query worked as expected.

Query #1 for MongoDB was in the middle of Query #2, and 3 in terms of speed. In my opinion, the difference in execution time between Query #1 and Query #2 is not great enough to mean something. In PostgreSQL it was the fastest because it did not require a JOIN statement unlike the other queries.

**Exhibits**

**Exhibit A**

**Graphical user interface, application

Description automatically generated**

**Graphical user interface, application

Description automatically generated**

**Graphical user interface, application, Teams

Description automatically generated**

**Graphical user interface, application, Teams

Description automatically generated**

**Graphical user interface, application, email

Description automatically generated**