2141 Assignment Part 3

Name: Nishika Solanki
Banner Id: B00953260
Topic: Brazilian E-commerce Dataset

Brief Summary of the database

- The Brazilian E-Commerce database is designed to manage information related to an online retail scenario. Key components include:
- **Data Source:** The Brazilian E-commerce Dataset is taken from Kaggle.com.
- **License information:** Here are the data sources: taken from Kaggle.com. (License CC BY-NC-SA 4.0)

  locations:URL[https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce/?select=olist_geolocation_dataset.csv] Date accessed: December 03, 2023.
  customers:URL[https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce/?select=olist_customers_dataset.csv] Date accessed: December 03, 2023.
  orders:URL[https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce/?select=olist_orders_dataset.csv] Date accessed: December 03, 2023.
  products:URL[https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce/?select=olist_products_dataset.csv] Date accessed: December 03, 2023.
  sellers:URL[https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce/?select=olist_sellers_dataset.csv] Date accessed: December 03, 2023.
  order_items:URL[https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce/?select=olist_order_items_dataset.csv] Date accessed: December 03, 2023.

- **Number of tables & Number of attributes per table:** This dataset has a total of 6 tables.
  - **Table 1:** location- includes information about geographic location.
  - **Number of attributes:** 5, i.e., zip_code_prefix, latitude, longitude, city, state.
  - **Number of rows:** 49

  - **Table 2:** customers- includes customer information such as id, zip code.
  - **Number of attributes:** 3, i.e., customer_id, customer_unique_id, zip_code_prefix
  - **Number of rows:** 930
  -

  - **Table 3:** orders- includes information about the order.
  - **Number of attributes**: 8, i.e., order_id, customer_id, order_status, order_purchase_timestamp, order_approved_at, order_delivered_carrier_date, order_delivered_customer_date, order_estimated_delivery_date.
  - **Number of rows:** 46

  - **Table 4:** products – includes products information which has its dimensions.
  - **Number of attributes:** 9, i.e., product_id, product_category_name, product_name_lenght, product_description_lenght, product_photos_qty, product_weight_g, product_length_cm, product_height_cm, product_width_cm,
  - **Number of rows:** 163

- **Table 5:** sellers- includes information about the sellers.
- **Number of attributes:** 2, i.e., seller_id, zip_code_prefix
- **Number of rows:** 193

- **Table 6:** order_items- includes order_items information given the attributes below.
- **Number of attributes:** 6, i.e., order_item_id, order_id, product_id, seller_id, price, freight_value.
- **Number of rows:** 54

Three Business Rule

- **Below are the three business rules enforced by Brazilian E-commerce Database:**

1. **Order-Customer Relationship (Referential Integrity):**
   - **Business Rule:** Every order in the orders table must be associated with an existing customer in the customer's table.
   - **Expressed Constraint:** Foreign Key constraint on orders.customer_id references customer.customer_id.

2. **Maintain Customer Location Integrity**
   - **Business Rule:** A customer's location (zip code prefix) in the customers table must correspond to a valid entry in the location table.
   - **Expressed constraint:** Foreign Key constraint on customer.zip_code_prefix references location.zip_code_prefix attribute.

3. **Consistency in Order Item**
   - **Business Rule:** Each entry in the order_items tale must refer to a valid order in the orders table and a valid product and seller in the products and sellers table, respectively.
   - **Expressed Constraints:** Composite primary key constraint on (order_item_id, order_id) in order items table. Foreign key constraints on order_items.order_id, order_items.product_id, and order_items.sellers_id references orders.order_id, products.product_id, and sellers.seller_id respectively.

- **These constraints ensure data integrity and business rule compliance by avoiding inconsistent or invalid database entries, maintaining table relationships, and ensuring that references to entities (customers, orders, products) are valid.**

Brief explanation of the five queries

Query 1: Selects products with weight over 500 grams.

```
SELECT *, (product_weight_g / 1000) AS product_weight_kg
FROM products
WHERE product_weight_g > 500
ORDER BY product_weight_g;
```

Explanation and Purpose: This query filters products based on the weight, calculates the weight in kilograms, and presents the results in ascending order of the original weight. The purpose of this query is to retrieve product information for items weighing over 500 grams, calculate a derived attribute "product_weight_kg" by converting grams to kilograms, and orders the results by the original weight.

Query 2: Retrieves order details with customer information.

```
SELECT order_id, customer_id, order_status, zip_code_prefix
FROM orders LEFT OUTER JOIN customers
USING(customer_id)
ORDER BY zip_code_prefix;
```

Explanation and Purpose: The query combines order and customer data, ensuring that all orders are included, even if there is no matching customer. It orders the results by the zip code prefix. It fetches details of orders along with associated customer information, using a LEFT OUTER JOIN on the customer_id, and orders the results by the zip code prefix

Query 3: Retrieves order details with item count greater than one.

```
SELECT o.order_id ,COUNT(o.order_item_id) AS order_item_Count, p.customer_id, p.order_status
FROM orders p JOIN order_items o
ON p.order_id = o.order_id
GROUP BY o.order_id
HAVING order_item_Count > 1
ORDER BY order_item_Count DESC;
```

Explanation and Purpose: This query joins orders and order_items, counts the number of items for each order, filters for orders with more than one item, and orders the results by the item count. It retrieves order details for orders with more than one item, including the count of order items, and orders the results by the item count in descending order.

Query 4: Retrieves customer information with total orders and average order value

```sql
SELECT c.customer_id, c.customer_unique_id, c.zip_code_prefix,
       customer_order.total_orders, customer_order.average_order_value
FROM customers c
INNER JOIN (
    -- Subquery in the FROM clause to calculate total orders and average order value for each customer
    SELECT o.customer_id,
           COUNT(o.order_id) AS total_orders,
           AVG(oi.price + oi.freight_value) AS average_order_value
    FROM orders o
    LEFT JOIN order_items oi ON o.order_id = oi.order_id
    GROUP BY o.customer_id
) AS customer_order ON c.customer_id = customer_order.customer_id
ORDER BY customer_order.total_orders DESC;
```

Explanation and Purpose: The query calculates total orders and average order value for each customer using a subquery in the FROM clause, then joins this information with customer details and orders the results by total orders. This query retrieves customer information along with the total number of orders and the average order value, ordered by total orders in descending order.

Query 5: Creates a view and reflects changes in the underlying tables

```sql
DROP VIEW IF EXISTS customer_order_view;
CREATE VIEW customer_order_view AS(SELECT
    cu.customer_id,
    cu.customer_unique_id,
    cu.zip_code_prefix,
    o.order_id,
    o.order_status,
    COUNT(oi.order_item_id) AS total_order_items
FROM
    customers cu
JOIN
    orders o ON cu.customer_id = o.customer_id
LEFT JOIN
    order_items oi ON o.order_id = oi.order_id
GROUP BY
    cu.customer_id, o.order_id
);
-- Run a SELECT query on the created VIEW
SELECT * FROM customer_order_view;
-- Modify one of the underlying tables ('orders' table)
UPDATE orders
SET order_status = "Shipped"
WHERE order_id = "3e27135b0c650634ca397ea4c2943e1e";
-- Re-run the SELECT query on the VIEW to reflect changes in the underlying tables
SELECT * FROM customer_order_view
ORDER BY total_order_items DESC;
```

Explanation and Purpose: The script first creates a view combining customer and order details. It creates a virtual table called customer_order_view. It then runs a SELECT query on the view, updates the “orders” table, and re-runs the SELECT query on the view to reflect changes, ordering the results by the count of order items in descending order. The purpose of this query is to create a view “customer_order_view” combining customer and order details with the count of order items and demonstrates the impact of modifying an underlying table on the view.

Explanation on the use of Stored Procedures

Stored Procedure 1: AddNewLocation

```
DELIMITER $$

-- Drop the procedure if it already exists
DROP PROCEDURE IF EXISTS AddNewLocation$$

-- Create a new stored procedure AddNewLocation
CREATE PROCEDURE AddNewLocation (
    IN zip_code_prefix INT,
    IN latitude DECIMAL (50, 20),
    IN longitude DECIMAL (50, 20),
    IN city VARCHAR(30),
    IN state CHAR(2)
)
BEGIN

    -- Start a new transaction to ensure atomicity
    START TRANSACTION;

   -- Insert new location into the 'location' table
    INSERT INTO location (zip_code_prefix, latitude, longitude,city,state)
    VALUES (zip_code_prefix, latitude, longitude,city,state);

    -- Commit the transaction to make the changes permanent
    COMMIT;
END $$

DELIMITER;

-- Call the AddNewLocation procedure with values
CALL AddNewLocation(99950,-28.07010363,-52.01865773,"tapejara","RS");
```

Purpose of this procedure is to add new location to the “location” table. Below is the information needed in Call ():
Zip_code_prefix: An integer representing the ZIP code prefix of the new location.
Latitude: decimal value representing the latitude of the new location
Longitude: decimal value representing the longitude of the new location
City: String representing the city of the new location
State: String representing the state of the new location

This procedure initiates a database transaction using “START TRANSACTION”. It then inserts a new location into the ‘location’ table with the provided information. Using “COMMIT”, it commits the transaction, meaning it add the location permanently to the location table.

This stored procedure does not explicitly return any information. It just adds new locations. This procedure is primarily focused on inserting data. The CALL() adds a new location with the specified details.

Stored Procedure 2: UpdateCustomerLocation

```
DELIMITER $$

-- Drop the procedure if it already exists
DROP PROCEDURE IF EXISTS UpdateCustomerLocation$$

-- Create a new stored procedure UpdateCustomerLocation
CREATE PROCEDURE UpdateCustomerLocation (
    IN customerId VARCHAR(50),
    IN newZipCodePrefix INT
)
BEGIN
    -- Start a new transaction to ensure atomicity
    START TRANSACTION;

    -- Update the zip_code_prefix for a specific customer in the 'customers' table
    UPDATE customers
    SET zip_code_prefix = newZipCodePrefix
    WHERE customer_id = customerId;

    -- Commit the transaction to make the changes permanent
    COMMIT;

END $$

DELIMITER ;

-- Call the UpdateCustomerLocation procedure with values
CALL UpdateCustomerLocation('00a39528c677a55852f57235f988b837', 99950);
```

The second procedure “UpdateCustomerLocation” updates the zip code prefix of a customer in the “customers” table. Below is the information needed in Call ():
customerId: String representing the ID of the customer whose location needs to be updated.
newZipCodePrefix: An integer representing the new ZIP code prefix for the customer.

This procedure initiates a database transaction using “START TRANSACTION”. It then updates zip_code_prefix for the specified customer in the ‘customers’ table. It then commits the transaction using “COMMIT.”

This procedure does not return any new information. It only updates the zip code for the specific customer. It focuses on updating data. The CALL () updates the ZIP code prefix for the specified customer.

These stored procedures are invoked using the “CALL” statement, providing the required input parameters. They are designed to perform specific database operations and do not explicitly return values. The use of transactions ensures data integrity during the execution of these procedures.