MAHMOUD AMIN

OLIST COMPANY

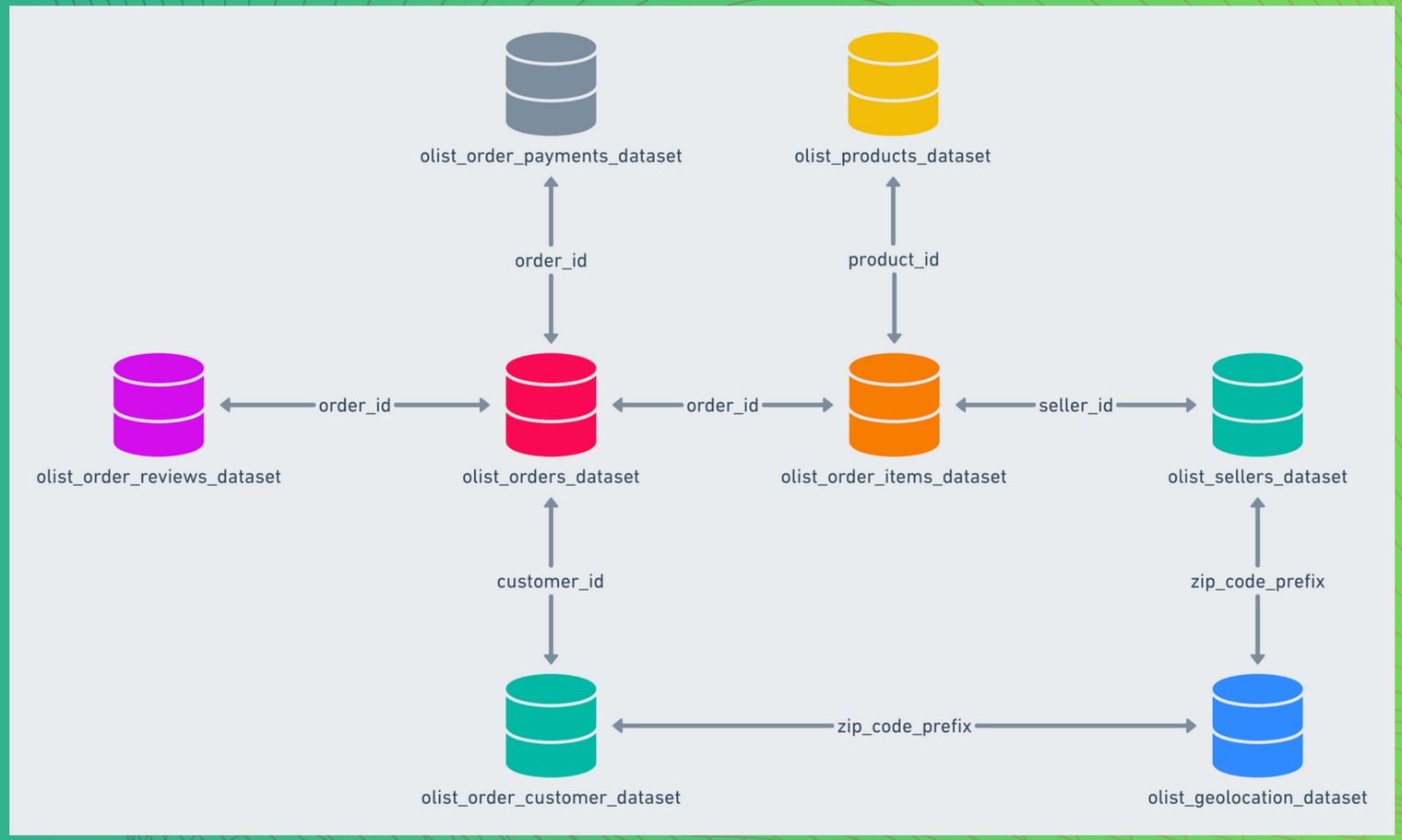
DB MODELLING

PART 1 / 3 DB MODELLING
PART 2 / 3 EDA USING SQL
PART 3 / 3 DASHBOARDING

HTTPS://GITHUB.COM/MAHMOUDAMINTAHA/OLIST_DATABASE

Olist, the largest department store in Brazilian marketplaces, generously provided this dataset. It serves as a bridge connecting small businesses from various regions of Brazil with the customers.

In the following slides, I'll offer a sneak peek into the modeling stage of the 9 CSV files from the Olist database using SQL.



Utilizing SQL (Concat - Replace) to write a query for renaming all the database columns.

```sql
SELECT
    CONCAT('ALTER TABLE ', TABLE_NAME, ' RENAME COLUMN ', '`',COLUMN_NAME,'`', ' TO ', REPLACE(COLUMN_NAME, ' ', '_'), ';')
FROM
    INFORMATION_SCHEMA.COLUMNS
WHERE
    TABLE_SCHEMA = 'brazil';
```

The result set of the previous query is a new query that renames all columns replacing spaces with underscores.

Here is a part of it:

```sql
ALTER TABLE order_payments RENAME COLUMN `order id` TO order_id;
ALTER TABLE order_payments RENAME COLUMN `payment installments` TO payment_installments;
ALTER TABLE order_payments RENAME COLUMN `payment sequential` TO payment_sequential;
ALTER TABLE order_payments RENAME COLUMN `payment type` TO payment_type;
ALTER TABLE order_payments RENAME COLUMN `payment value` TO payment_value;
ALTER TABLE order_reviews RENAME COLUMN `order id` TO order_id;
ALTER TABLE order_reviews RENAME COLUMN `review answer timestamp` TO review_answer_timestamp;
ALTER TABLE order_reviews RENAME COLUMN `review comment message` TO review_comment_message;
ALTER TABLE order_reviews RENAME COLUMN `review comment title` TO review_comment_title;
ALTER TABLE order_reviews RENAME COLUMN `review creation date` TO review_creation_date;
ALTER TABLE order_reviews RENAME COLUMN `review id` TO review_id;
ALTER TABLE order_reviews RENAME COLUMN `review score` TO review_score;
ALTER TABLE orders RENAME COLUMN `customer id` TO customer_id;
ALTER TABLE orders RENAME COLUMN `order approved at` TO order_approved_at;
```

After selecting values from customer table, Duplicate ‘Customer id’ are noticed. While ‘Customer unique id’ is distinct.

```
SELECT * FROM customers LIMIT 10;
```

WHY?

After some online investigation I found out that OLIST system generates a unique id `Customer_unique_id` for each customer right after signing up so it's permanent. While generates a random ID every time a customer places an order which is stored in `Customer id` column.

```
SELECT
    *
FROM
    customers
WHERE customer_unique_id IN (
    SELECT customer_unique_id
    FROM customers
    GROUP BY customer_unique_id
    HAVING COUNT(customer_unique_id) > 5
)
ORDER BY customer_unique_id
LIMIT 25;
```

```sql
SELECT
    Count(geolocation_zip_code_prefix),
    COUNT(distinct geolocation_zip_code_prefix),
    COUNT(distinct geolocation_zip_code_prefix,
 geolocation_city, geolocation_state)
FROM
    geolocation;
```

our geolocation table includes over one million rows, To check it out I wrote this query.

I found only about 20,000 unique entries only, So it was necessary to drop all the duplicates.
Luckily we have location data in customers and sellers data, And I'll use this data to compensate the geolocation table

```sql
CREATE TABLE geo AS
            (
                SELECT   geolocation_zip_code_prefix,
                        max(geolocation_city),
                        max(geolocation_state)
                FROM     geolocation
                GROUP BY geolocation_zip_code_prefix
            );
```

```sql
INSERT INTO geolocation
(
    SELECT
        customer_zip_code_prefix,
        MAX(customer_city) AS customer_city,
        MAX(customer_state) AS customer_state
    FROM
        CUSTOMERS
    WHERE
        customer_zip_code_prefix NOT IN (SELECT
                                             geolocation_zip_co
de_prefix
                                         FROM
                                             geolocation)

    GROUP BY
        customer_zip_code_prefix
);
```

```sql
INSERT INTO geolocation
(
    SELECT
        seller_zip_code_prefix,
        MAX(seller_city) AS seller_city,
        MAX(seller_state) AS seller_state
    FROM
        SELLERS
    WHERE
        seller_zip_code_prefix NOT IN (SELECT
                                           geolocation_zip_co
de_prefix
                                       FROM
                                           geolocation)
    GROUP BY
        seller_zip_code_prefix
);
```

Check the order_items table for duplicates

```sql
SELECT count(order_id),
       count(DISTINCT order_id, product_id),
       count(DISTINCT order_id, product_id, seller_id
)
FROM   order_items
```

To drop duplicates and prepare the table to have a composite primary key consisting of (order_id, product_id), I wrote this query.

```sql
CREATE TABLE oi AS
  (SELECT *
   FROM   (SELECT order_id,
                  order_item_id,
                  product_id,
                  seller_id,
                  shipping_limit_date,
                  price,
                  freight_value,
                  Row_number()
                    OVER (
                      partition BY order_id, product_id
                      ORDER BY order_id, product_id) AS rn
           FROM   order_items) AS desired_table
   WHERE  rn = 1)
```

```sql
SELECT *
FROM   order_payments
WHERE  order_id IN (SELECT order_id
                    FROM   order_payments
                    GROUP  BY order_id
                    HAVING Count(order_id) > 1)
ORDER  BY order_id
LIMIT  10;
```

```sql
SELECT count(*),
       count( DISTINCT order_id, payment_sequential)
FROM   order_payments;
```

Checking order_payments table IDs for duplicates.

By counting all rows versus distinct combinations, No difference found.

By further investigating the table it was found that every order has many payments some of them by credit, Others by vouchers. Which explains duplicate order IDs. For this table the primary key should consist of (order_id, payment_sequential).

```sql
SELECT DISTINCT(product_category_name)
FROM products
WHERE product_category_name NOT IN (
                SELECT product_category_name
                FROM category_translation
                                )
```

```sql
INSERT INTO category_translation
VALUES
    ('pc_gamer', 'pc_gamer'),
    ('portateis_cozinha_e_preparadores_de_alimentos', '
portable_kitchen_food_preparators');
```

Official language in Brazil is Portuguese,
Category_translation table have the english translation for the product_category_name column in products table.

After founding two missing translations in the translation table by the upper query, I translated and added the translations to it by the lower query.

Final Schema

Manipulation mentioned in this slide deck is just a glimpse of what I did the main jupyter notebook to prepare all tables for primary and foreign keys.

Check the link of the repository https://github.com/mahmoudamintaha/olist_database

Finally after creating all relations, The desired schema is achieved.