# CIS 550 Final Project: Oeda Platform

A Business Analytics Tool for E-Commerce Retailers

## Introduction

1. Project goals and target problems

   The rapid development of the Internet has made online shopping more and more common. People's shopping habits are also gradually shifting from offline to online. Given the gradual growth of e-commerce, insights from online retail sales data are helpful for businesses. Our project will focus on the sales data of online merchants to create a user-friendly, intuitive Business Intelligence or Business Analytics tool for e-commerce. This project will explore the relationship between people's preferences and Brazilian e-commerce markets and generate relevant graphs based on the data. By analyzing the data of people's online shopping preferences, we can help merchants better understand the entire market, formulate sales strategies that are more in line with market trends, and develop products that meet consumer needs. Typically, analytics for sales and marketing insights only include company sales data. Our analysis will retain geographical data for more in-depth market analysis. Since our data combines geographic information, users can understand the consumption habits of different regions based on the characteristics of areas, which provides convenience for regional market research.

2. Application functionality

   Oeda Platform is a web application where e-commerce merchants can perform data analysis and data visualization to gain sales and marketing insights from their business and customer behavior. Main functions of our platform include:

   - Display trending/featured visual analysis: display an overview of pre-built analytics that provide key sales and marketing insights. For example, a dashboard of trending product categories of the month or KPI dashboards.

   - Search data: users can search product, order, or payment information using multiple different search keywords and filters. For example, they can query for certain orders based on transaction dates, price range, or product categories. Users can also choose the density level for the display of query results (compact, standard, comfortable).

   - Create insightful reports and dashboards: users can generate a variety of charts, widgets, pivot, summary or tabular views on sales and geographical data based on their search input. For example, they can generate a report of top 10 selling product types by certain product specification, or a line graph that compares customer review scores of two different product categories.

   - Export/download reports: users can export queried results as .csv files to their local device

   - Display geographic visualization: provides visualization for geographic distribution of customer/sellers/sales data and display Brazilian city demographics

   - Perform market analysis: users can search and generate sales results and product analysis based on Brazilian city demographics

   - Interactive geographic visualization: users can analyze geographical data with interactive map charts, which allows users to compare and measure key metrics across regions, states, cities

3. List of group members (name, email address, Github username)


   - Huifang Ye | huifangy@seas.upenn.edu | Ye-Huifang

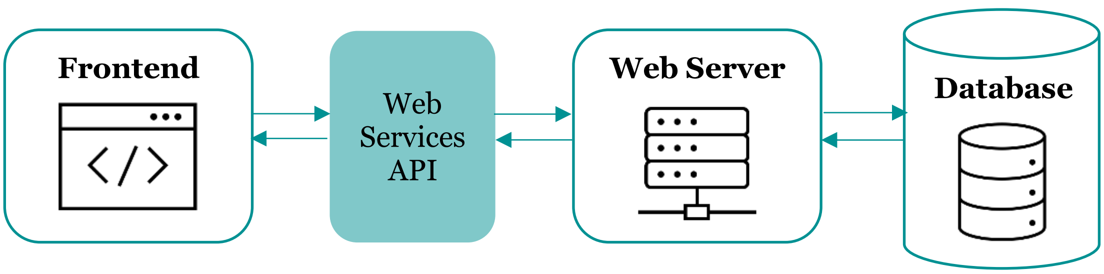
- Joyce An-Jie Wang | joycew3@seas.upenn.edu | joycewang3
- Zhihui Zhang | clarazh@seas.upenn.edu | zhihuizhang0625
- Zhuoran Hu | zhuohu@seas.upenn.edu | zhuoranhu


**Architecture**

1. List of technologies

- Data cleaning: Python (Pandas, PandaSQL) and Colab
- Frontend: HTML/CSS, JavaScript, React.js
- Libraries: nivo charts, datagrip, mui, react-brazil-map
- Backend: SQL, Node.js
- Database: MySQL, AWS RDS
- Testing: Jest

2. System architecture diagram




3. Description of application

- Dashboard (Homepage): This page displays an overview of featured dashboards which include recent transactions, top 10 highest reviewed products, overall sales increment by year, and etc.

- Analytics (Search): This section consists of two pages, Transaction Info and Market Info, where both pages enable a search bar to query input years, months, product specification, or customer/seller location. It returns a table with the specified query from the user. Users can search for transaction related information at Transaction Info page and sales information of different cities at Market Info page.

- Report: This section consists of two pages, Market Analysis and Geographic Distribution. Market Analysis page performs an analysis on the number of Walmart stores in each customer city and how that potentially affects the sales and products preferences of users' e-commerce platforms. Users can specify a certain year of the data and it will return the queried result. Geographic Distribution page return an interactive geographic map that shows the different payment information (payment amount and payment methods) across different states where customers are located. In both pages, users can export/download the reports to their local device.

- Visualization: This section includes three different interactive charts that visualize the trend of total number of orders and top selling products of each year. It also includes the popularity of different product categories based on customer reviews.

**Data**

1. Brazilian E-Commerce Public Dataset by Olist

- Source Link: [Link]

- Description: A Brazilian ecommerce public dataset of orders made at the Olist store. There are 6 different datasets in total (refer to the ER diagram in Appendix 1). The dataset has information of 100k orders from 2016 to 2018 made at multiple marketplaces in Brazil. Its features allow viewing an order from multiple dimensions: from order status, price, payment and freight performance to customer location and product attributes, and finally reviews written by customers. It also contains a geolocation dataset that relates Brazilian zip codes to geolocational coordinates. (https://olist.com/pt-br/)

- Summary Statistics: Refer to ***Appendix 0***

- How we used the datasets: We used these datasets to perform data analysis jointly on orders, products, payments, and customer and seller information in order to gain useful sales and marketing insights for e-commerce retailers. We then generate reports and dashboards of the analysis on these datasets. Main fields that are used in our analysis refer to ***Appendix 0***.

2. Brazilian Cities

- Source Link: Link

- Description: Brazil is the world's fifth-largest country by area, with 8.5 million square kilometers, and the fifth most populous, with over 208 million people. The Federative Republic of Brazil is composed of the union of the 26 states, the Federal District, and the 5,570 municipalities. This dataset is a compilation of several publicly available demographic information about Brazilian Municipalities. There are in total 79 fields for each city, which includes city, state, resident population, resident population by ages, Human Development Index (HDI), number of Pay TV users, Gross Domestic Product (GDP), total number of companies, number of companies by industries, number of Walmart stores, and etc.

- Summary Statistics: Refer to ***Appendix 0***

- How we used the dataset: We used this dataset to perform demographic analysis on customers and get insights into how the population and business activities in different cities may affect the sales of e-commerce retail. Some of the fields that are used in our analysis are listed as follows:

    - City: Name of the City
    - State: Name of the State
    - res_population: Resident Population
    - walmart: Total number of Walmart Stores

**Database**

1. Data ingestion procedure

   We created an AWS RDS instance and connected to the MySQL instance via DataGrip. We then created a database named "Brazil" and created 10 tables in the database: Category, City, Customer, Geolocation, Item, OrderInfo, Payment, Product, and Review, for each of our dataset using SQL DDL. We used the import wizard in DataGrip to load our datasets (CSV files) to the database.

2. ER diagram: Refer to ***Appendix 1***

3. Number of instances in each table (based on cleaned datasets)

| No | Table Name | Number of Instances |
|---|---|---|
| 1. | Category | 71 |

| No | Table Name | Number of Instances |
|---|---|---|
| 2. | City | 5,574 |
| 3. | Customer | 99,441 |
| 4. | Geolocation | 1,000,163 |
| 5. | Item | 112,650 |
| 6. | OrderInfo | 96,461 |
| 7. | Payment | 103,886 |
| 8. | Product | 32,340 |
| 9. | Review | 9,839 |

4. Normal form and justification

Below are the schemas of the 9 tables in our database and their functional dependencies:

**Category** (product_category_name, product_category_name_english)
$F_{category}$ = {product_category_name → product_category_name_english}

***Proof***: product_category_name is the primary key, therefore, a superkey of Category. Since for every functional dependency, X → A holds over Category, X is a superkey of Category, we know this relation is in Boyce-Codd Normal Form (BCNF).

**City** (City, State, Longitude, Latitude, Altitude, Area, Taxes, Gdp, res_population, pop_below1, pop_1_4, pop_5_9, pop_10_14, pop_15_59, pop_60, HDI, pay_TV, COMP_TOT, COMP_A, Wal-Mart)

$F_{City}$ = {City → State, City → Longitude, City → Latitude, City → Altitude, City → Area, City → Taxes, City → Gdp, City → res_population , City → pop_below1 , City → pop_1_4, City → pop_5_9, City → pop_10_14, City → pop_15_59, City → pop_60, City → HDI, City → pay_TV, City → COMP_TOT, City → COMP_A, City → Wal-Mart}

*Note: We omitted attributes COMP_B, COMP_C, ... COMP_U due to the space limit. They are of the same nature as attribute COMP_A, and are functionally determined by City only*

***Proof:*** City is the primary key, therefore, a superkey of City. Since for every functional dependency, X → A holds over City, X is a superkey of City, we know this relation is in BCNF.

**Customer** (customer_id, customer_unique_id, customer_zip_code_prefix, customer_city, customer_state)

$F_{Customer}$ = {customer_id → customer_unique_id, customer_id → customer_zip_code_prefix, customer_id → customer_city, customer_id →customer_state}

***Proof:*** customer_id is the primary key, therefore, a superkey of Customer. Since for every functional dependency, X → A holds over Customer, X is a superkey of Customer, we know this relation is in BCNF.

**Geolocation** (geolocation_zip_code_prefix, geolocation_lat, geolocation_lng, geolocation_city, geolocation_state)

$F_{Geolocation}$ = {geolocation_zip_code_prefix → geolocation_lat, geolocation_zip_code_prefix → geolocation_lng, geolocation_zip_code_prefix → geolocation_city, geolocation_zip_code_prefix → geolocation_state}

***Proof:*** `geolocation_zip_code_prefix` is the primary key, therefore, a superkey of Geolocation Since for every functional dependency, X → A holds over Geolocation, X is a superkey of Geolocation, we know this relation is in BCNF.

**Item** (order_id, order_item_id, product_id, seller_id, shipping_limit_date, price, freight_value)

$F_{Item}$ = {order_id, order_item_id → product_id, order_id, order_item_id → seller_id, order_id, order_item_id → shipping_limit_date, order_id, order_item_id → price, order_id, order_item_id → freight_value}

***Proof:*** (order_id, order_item_id) is the primary key, therefore, a superkey of Item. Since for every functional dependency, X → A holds over Item, X is a superkey of Item, we know this relation is in BCNF.

**OrderInfo** (order_id, customer_id, order_purchase_timestamp, order_approved_at, order_delivered_carrier_date, order_delivered_customer_date, order_estimated_delivery_date, order_purchase_year, order_purchase_month, order_purchase_day, order_approve_year, order_approve_month, order_approve_day, order_deliver_carrier_year, order_deliver_carrier_month, order_deliver_carrier_day, order_deliver_customer_year, order_deliver_customer_month, order_deliver_customer_day, order_estimate_delivery_year, order_estimate_delivery_month, order_estimate_delivery_day)

$F_{Order}$ = {order_id → customer_id, order_id → order_purchase_timestamp, order_id → order_approved_at, order_id → order_delivered_carrier_date, order_id → order_delivered_customer_date, order_id → order_estimated_delivery_date, order_id → order_purchase_year, order_id → order_purchase_month, order_id → order_purchase_day, order_id → order_approve_year, order_id → order_approve_month, order_id → order_approve_day, order_id → order_deliver_carrier_year, order_id → order_deliver_carrier_month, order_id → order_deliver_carrier_day, order_id → order_deliver_customer_year, order_id → order_deliver_customer_month, order_id → order_deliver_customer_day, order_id → order_estimate_delivery_year, order_id → order_estimate_delivery_month, order_id → order_estimate_delivery_day}

***Proof:*** order_id is the primary key, therefore, a superkey of OrderInfo. Since for every functional dependency, X → A holds over OrderInfo, X is a superkey of OrderInfo, we know this relation is in BCNF.

**Payment** (order_id, payment_sequential, payment_type, payment_installments, payment_value)

$F_{Payment}$ = {order_id → payment_sequential, order_id → payment_type, order_id → payment_installments, order_id → payment_value}

***Proof:*** order_id is the primary key, therefore, a superkey of Payment Since for every functional dependency, X → A holds over Payment, X is a superkey of Payment, we know this relation is in BCNF.

**Product** (product_id, product_category_name, product_name_length, product_description_length, product_photos_qty, product_weight_g, product_length_cm, product_height_cm, product_width_cm)

$F_{Product}$ = {product_id → product_category_name, product_id → product_name_length, product_id → product_photos_qty, product_id → product_weight_g, product_id →

product_length_cm, product_id → product_height_cm, product_id → product_width_cm}

***Proof:*** `product_id` is the primary key, therefore, a superkey of Product. Since for every functional dependency, X → A holds over Product, X is a superkey of Product, we know this relation is in BCNF.

**Review** (review_id, order_id, review_score, review_comment_title, review_comment_message, review_creation_date, review_answer_timestamp)

$F_{Review}$ = {review_id → order_id, review_id → review_score, review_id → review_comment_title , review_id → review_comment_message, review_id → review_creation_date, review_id → review_answer_timestamp}

***Proof:*** `review_id` is the primary key, therefore, a superkey of Review. Since for every functional dependency, X → A holds over Review, X is a superkey of Review, we know this relation is in BCNF.

## Queries

Below are 5 example queries of our application and how we used them (actual SQL queries refer to ***Appendix 2***):

1. Show the average rating and the number of review for each product category/each seller so that customers can easily evaluate the product quality from each category/seller

   ***Query 1:*** join OrderInfo, Item, Review, Category and Product, group by product category and averages the review scores and sums up the total review number of each group.

2. Query the differences of total orders between 2016, 2017, and 2018 for each product category. We want to know - what kind of products improved in sales (in terms of orders) in 2018, and what kind of products did not?

   ***Query 2:*** join order_items and products, group by product category, and sum the sales amount of the orders. Create subqueries for different years or different months within a year. Create a new column to calculate the difference between each year/month. Select the product categories with increments.

3. As part of market analysis, users may want to know whether the sales of their online shopping platform are affected by the number of Walmart stores in the areas. Users may query and create a sales report for the top 5 cities with the most Walmart stores.

   ***Query 3:*** For each of the five cities, list the total number of Walmart stores, total number of orders (based on customer location) by year, total sales (based on product price) by year, and the most popular product category based on sales (in English) by year. The result is ordered by the number of Walmart stores.

4. Our users may want to see the review scores (1-5) of their sold products. This query will retrieve the top 10 products with highest average review scores. Our application will create a review report using this query. We also allow users to filter reviews by specific order year/month.

   ***Query 4:*** The query first creates an index (order_id) on order_data. We make use of views to temporarily store frequently used data and fetch required information by joining fours table. After we acquire the review score for a specific category, we group by product category and calculate its average review score. Finally, we sort the query result by average scores in a descending order. The filtered 10 products are the ones with highest average review scores. We only consider products reviewed by at least 3 customers, so the review score is more representative.

5. As part of sales analysis, retail companies may want to learn about different payment habits of customers from different states. This query compares the differences in total, average, max, and min payment values by credit card users and bank ticket users from each state. Businesses may gain geographical insights on customers' paying habits and make decisions on future investments and plans.

   ***Query 5:*** join Customer, OrderInfo, Item, and Payment, group by customer state, and calculate the sum, average, max, and min of the payment values of the orders. Create subqueries for customers using credit cards and bank tickets respectively. Create a new column to calculate the difference between different types of customers. Select the product categories and differences in total, average, max, and min payment values.

**Performance Evaluation**

Key optimization strategies we used (or attempted to use) refer to ***Appendix 4.***

Our optimization for the five example queries in above ***Queries*** section are summarized in the table below:

| Query No. | Optimizations made | Timings Before | Timings After | Analysis |
|---|---|---|---|---|
| 1. *Type 1* | ***Strategy 1:*** when using `COUNT()` to get the total number of reviews in 2018, we used `COUNT(I.product_id)` instead of `COUNT(*)` every column to push down the selection for count operation | 16 rows retrieved starting from 1 in **769 ms** (execution: 174 ms, fetching: 595 ms) | 16 rows retrieved starting from 1 in **396 ms** (execution: 158 ms, fetching: 238 ms) | Executed **48.50%** faster applying this strategy. This strategy succeeded because it helps to decrease the count execution for multiple columns. |
| | ***Strategy 6 & 8:*** we used NATURAL JOIN for joining all five tables in this query since there are common fields between these five tables | **396 ms** (execution: 158 ms, fetching: 238 ms) | **264 ms** (execution: 143 ms, fetching: 121 ms) | Executed **33.33%** faster applying this strategy. This strategy succeeded because this query requires a join of three large tables: OrderInfo (96,461 rows), Item (112,650 rows), Product (32,340 rows), and two other small tables. A more efficient join strategy makes a difference. |
| | ***Strategy 2:*** created and used the index `item_idx` on Item table. | **264 ms** (execution: 143 ms, fetching: 121 ms) | **214 ms** (execution: 134 ms, fetching: 80 ms) | There was minor improvement (1.89%) from applying this strategy.<br><br>*<u>Note:</u> Creating this index took 351 ms. However, it also optimized many other following queries in our app, so we still consider it optimized our queries overall.* |
| | ***Summary*** | ***769 ms*** | ***214 ms*** | ***Total optimization: 72.17%*** |
| 1. *Type 2* | ***Strategy 1:*** when using `COUNT()` to get the total number of reviews in 2018, we used `COUNT(I.product_id)` instead of `COUNT(*)` every column to push down the selection for count operation | 500 rows retrieved starting from 1 in **577 ms** (execution: 162 ms, fetching: 415 ms) | 500 rows retrieved starting from 1 in **344 ms** (execution: 121 ms, fetching: 223 ms) | Executed **40.38%** faster applying this strategy. This strategy succeeded because it helps to decrease the count execution for multiple columns. |
| | ***Strategy 6 & 8:*** we used NATURAL JOIN for joining all five tables in this query | **344 ms** (execution: 121 ms, | **274 ms** (execution: 95 ms, | Executed **20.35%** faster applying this strategy. This strategy succeeded because this query requires a join of three large tables: OrderInfo (96,461 |

| Query No. | Optimizations made | Timings Before | Timings After | Analysis |
|---|---|---|---|---|
| | since there are common fields between these five tables | fetching: 223 ms) | fetching: 141 ms) | rows), Item (112,650 rows), Product (32,340 rows), and two other small tables. A more efficient join strategy makes a difference. |
| | ***Strategy 2:*** used the index `item_idx` on Item table. | **274 ms** (execution: 95 ms, fetching: 141 ms) | **225 ms** (execution: 184 ms, fetching: 41 ms) | There was minor improvement (1.79%) from applying this strategy. |
| | ***Summary*** | ***577 ms*** | ***225 ms*** | ***Total optimization: 61.01%*** |
| 2. | ***Strategy 3:*** we used 3 temporary tables to obtain the intermediate information we need for our final query result instead of multiple subqueries | 4 rows retrieved starting from 1 in **812 ms** (execution: 638 ms, fetching: 174 ms) | 4 rows retrieved starting from 1 in **500 ms** (execution: 400 ms, fetching: 100 ms) | Executed **28.42%** faster applying this strategy. This strategy succeeded because this query required the join of three large tables OrderInfo (96,461 rows), Item (112,650 rows), Product (32,340 rows) over three times. Temporarily storing frequently used data leads to the improvement in performance. |
| | ***Strategy 2:*** used the index `item_idx` on Item table | **500 ms** (execution: 400 ms, fetching: 100 ms) | **409 ms** (execution: 352 ms, fetching: 57 ms) | There was minor improvement (1.82%) from applying this strategy. |
| | ***Summary*** | ***812 ms*** | ***409 ms*** | ***Total optimization: 49.63 %*** |
| 3. | ***Strategy 1***: for each of the temporary relation created using WITH AS clause, we specified the SELECT fields that are necessary to get the query result. For example, we selected `city` and `walmart` fields instead of using `SELECT*` for the `top_cities` temp table | 10 rows retrieved starting from 1 in **934 ms** (execution: 644 ms, fetching: 290 ms) | 10 rows retrieved starting from 1 in **627 ms** (execution: 466 ms, fetching: 161 ms) | Executed **32.87%** faster applying this strategy. This strategy succeeded because there are relatively large numbers of fields in the city dataset (81 columns) and therefore pushing projection down to the base query works. |
| | ***Strategy 3:*** we used 5 temporary tables to obtain the intermediate information we need for our final query result instead of multiple correlated subqueries | NA | NA | We adopted this strategy from the very first beginning when we started developing this query. |
| | ***Strategy 4:*** we limit the result of our query in the very first temporary relation `top_cities` instead of selecting the top 5 cities in the following associated temporary relations `total_orders`, `total_sales`, and `total_product` and the final query | **627 ms** (execution: 466 ms, fetching: 161 ms) | **615 ms** (execution: 514 ms, fetching: 101 ms) | There was only minor improvement (< 1%) from applying this strategy because the size of data from the city dataset is relatively small (5,573 rows) |
| | ***Strategy 6 & 8:*** we used INNER JOIN or NATURAL JOIN for every single | **615 ms** (execution: 514 ms, | **551 ms** (execution: 474 ms, | Executed **10.41%** faster applying this strategy. This strategy works because the query requires multiple joins on |

| Query No. | Optimizations made | Timings Before | Timings After | Analysis |
|---|---|---|---|---|
| | temporary table created in this query that needs joined information: `orders_products`, `top_product`, and final query | fetching: 101 ms) | fetching: 77 ms) | multiple tables with large data. For example, a join of OrderInfo (96,461 rows), Item (112,650 rows), Product (32,340 rows), and Customer (99,441 rows) datasets was used to create the `orders_products` temp table, and a join of three temp tables `total_orders`, `total_sales`, and `top_product` was used to get the final query result. |
| | ***Summary*** | ***934 ms*** | ***551 ms*** | ***Total optimization: 41%*** |
| 4. | ***Strategy 2:*** created and used the index `order_idx` on the Item table. | 10 rows retrieved starting from 1 in **148 ms** (execution: 105 ms, fetching: 43 ms) | 10 rows retrieved starting from 1 in **106 ms** (execution: 68 ms, fetching: 38 ms) | Executed **28.38%** faster applying this strategy.<br><br>*<u>Note:</u> Creating this index took 1s 152ms. However, it also optimized many other following queries in our app, so we still consider it more efficient overall.* |
| | ***Strategy 6 & 8:*** we used INNER JOIN or NATURAL JOIN for every single temporary table. We used NATURAL JOIN for Item, Product, Category, and Review | **106 ms** (execution: 68 ms, fetching: 38 ms) | **93 ms** (execution: 69 ms, fetching: 24 ms) | Executed **28.38%** faster applying this strategy. This strategy works because the query requires multiple joins on multiple tables with large data. Using NATURAL JOIN largely reduces execution time. |
| | ***Strategy 7:*** we made sure smaller tables are on the left side of join syntax when joining tables. We changed the order of join to: Category (71 rows) → Product (32,951 rows) → Item (112,650 rows). | **93 ms** (execution: 69 ms, fetching: 24 ms) | **86 ms** (execution: 69 ms, fetching: 17 ms) | Executed **12.26%** faster applying this strategy. This strategy helps because this query requires join over large table such as Item (112,650 rows) with much smaller table Category (71 rows) |
| | ***Summary*** | ***148 ms*** | ***86 ms*** | ***Total optimization: 41.89 %*** |
| 5. | ***Strategy 5:*** we removed DISTINCT clause and used GROUP BY to query payment information by each `customer_state` | 27 rows retrieved starting from 1 in **442 ms** (execution: 309 ms, fetching: 133 ms) | 27 rows retrieved starting from 1 in **381 ms** (execution: 327 ms, fetching: 54 ms) | Executed **13.80%** faster applying this strategy. Using GROUP BY reduces the cost of eliminating data from grouped fields, and thus increases performance. This strategy helps a bit since there were not many rows in the resultant table that need to be eliminated using DISTINCT. |
| | ***Strategy 7:*** we made sure smaller tables are on the left side of join syntax when joining tables. We changed the order of tables to Product (32,951 rows) → Customer (99,441 rows) → OrderInfo (99,441 rows) → Payment (1,033,886 rows). | **381 ms** (execution: 327 ms, fetching: 54 ms) | **280 ms** (execution: 227 ms, fetching: 53 ms) | Executed **26.51%** faster applying this strategy. This strategy succeeded because this query requires join over large table such as Customer (99,441 rows), OrderInfo (99,441 rows), and Payment (1,033,886 rows) with relatively smaller table Product (32,951 rows). Similarly, we joined tables based on table size for all other |

| Query No. | Optimizations made | Timings Before | Timings After | Analysis |
|---|---|---|---|---|
| | | | | temporary tables. Re-ording tables remarkably improves the execution time. |
| | ***Strategy 6 & 8:*** we used INNER JOIN or NATURAL JOIN for every single temporary table created in this query that needs joined information. | NA | NA | We applied NATURAL JOIN when we first developed this query. Hence, the joining method for this query is optimized. |
| | ***Summary*** | ***442 ms*** | ***280 ms*** | ***Total optimization: 36.65 %*** |

***Note:*** *The Query No. column refers to the query number given in the above Queries section. For example, 1. refers to Query 1*

Refer to ***Appendix 2*** and ***3*** for above complete SQL queries before and after optimization.

### Technical Challenges

For some of us it was our first time creating a full stack web application, so it was a challenge searching for useful tutorials to learn more about JavaScript, React, Node.js on our own, and applying new libraries to create interactive dashboards for our application. For others, understanding the backend functionalities, and how to correctly connect them to the frontend and our database was new and challenging. It was also our first time configuring some relevant setup in a different operating system, or handling errors in interacting with APIs and so on. Some of the technical challenges we faced and how we overcame them are as follows:

1. Setup for Node.js installation on Windows is different from on Mac, and we encountered an error when executing `npm run start-win`. There were some extra configurations that need to be done before executing `npm start` on Windows system. For example, the following script needs to be added to the `package.json` file:

   ```
   "scripts": {
      "start-win": "node main.js"
   }
   ```

   Solution source: Link

2. In many instances when we tried to make requests to our API from the frontend, the error "Error: read ETIMEDOUT" occurred. After some research, we found that this error is caused when our request response was not received in each time. The solution is to catch that error first by registering a handler on the error, so the unhandled error won't be thrown later.

   Solution source: Link

3. We used React Data Grid component to develop our reporting and analytics pages for frontend. At the beginning we didn't realize that the Data Grid component requires all data rows to have a unique "id" property, so we didn't incorporate an id column in most of our data or designed it in our table schema. Therefore, we had problem connecting the data to the frontend even our API was working well. Eventually, we create a function to generate random id and assign it to each row using `getRowId={(row:any)=> row.column}` to solve this problem.

   Solution source: Link

## *Appendix 0* – Summary Statistics of Datasets

*Brazilian E-Commerce Dataset*

a. Relevant size statistics

| Name of dataset | Number of rows | Number of columns | Size (MB) |
|---|---|---|---|
| olist_customers | 99,441 | 5 | 9.03 |
| olist_geolocation | 1,000,163 | 5 | 61.27 |
| olist_order_items | 112,650 | 7 | 15.44 |
| olist_order_payments | 99,440 | 5 | 5.78 |
| olist_order_reviews | 98,410 | 7 | 14.45 |
| olist_orders | 99,441 | 8 | 17.65 |
| olist_products | 32,340 | 9 | 2.38 |
| olist_sellers | 3,095 | 4 | 0.175 |
| product_category _name_translation | 71 | 2 | 0.027 |

b. Summary statistics of relevant attributes

- olist_customers dataset:

| Top 10 customer cities | Number of customers |
|---|---|
| Sao Paulo | 15,540 |
| Rio de Janeiro | 6,882 |
| Belo Horizonte | 2,773 |
| Brasilia | 2,131 |
| Curitiba | 1,521 |
| Campinas | 1,444 |
| Porto Alegre | 1,379 |
| Salvador | 1,245 |
| Guarulhos | 1,189 |
| Sao Bernardo do campo | 938 |

- olist_order_items dataset:

| | price | freight_value |
|---|---|---|
| **min** | 0.850000 | 0.000000 |
| **max** | 6735.000000 | 409.680000 |
| **median** | 74.990000 | 16.260000 |
| **mean** | 120.653739 | 19.990320 |
| **std** | 183.633928 | 15.806405 |
| **skew** | 7.923208 | 5.639870 |

- olist_order_payments dataset:

| | payment_sequential | payment_installments | payment_value |
|---|---|---|---|
| count | 103886.000000 | 103886.000000 | 103886.000000 |
| mean | 1.092679 | 2.853349 | 154.100380 |
| std | 0.706584 | 2.687051 | 217.494064 |
| min | 1.000000 | 0.000000 | 0.000000 |
| 25% | 1.000000 | 1.000000 | 56.790000 |
| 50% | 1.000000 | 1.000000 | 100.000000 |
| 75% | 1.000000 | 4.000000 | 171.837500 |
| max | 29.000000 | 24.000000 | 13664.080000 |

- olist_order_reviews dataset:

| | review_score |
|---|---|
| count | 9839.000000 |
| mean | 3.837585 |
| std | 1.556435 |
| min | 1.000000 |
| 25% | 3.000000 |
| 50% | 5.000000 |
| 75% | 5.000000 |
| max | 5.000000 |

- olist_products dataset:

| | product_name_lenght | product_description_lenght | product_photos_qty | product_weight_g | product_length_cm | product_height_cm | product_width_cm |
|---|---|---|---|---|---|---|---|
| min | 5.000000 | 4.000000 | 1.000000 | 0.000000 | 7.000000 | 2.000000 | 6.000000 |
| max | 76.000000 | 3992.000000 | 20.000000 | 40425.000000 | 105.000000 | 105.000000 | 118.000000 |
| median | 51.000000 | 595.000000 | 1.000000 | 700.000000 | 25.000000 | 13.000000 | 20.000000 |
| mean | 48.476592 | 771.492393 | 2.188961 | 2276.956586 | 30.854545 | 16.958813 | 23.208596 |
| std | 10.245699 | 635.124831 | 1.736787 | 4279.291845 | 16.955965 | 13.636115 | 12.078762 |
| skew | -0.903200 | 1.962078 | 2.193438 | 3.607632 | 1.750295 | 2.148907 | 1.678041 |

- olist_sellers dataset:

| Top 10 seller cities | Number of sellers |
|---|---|
| Sao Paulo | 694 |
| Curitiba | 127 |
| Rio de Janeiro | 96 |
| Belo Horizonte | 68 |
| Ribeirao Preto | 52 |
| Guarulhos | 50 |
| Ibitinga | 49 |

| | |
|---|---|
| Santo Andre | 45 |
| Campinas | 41 |
| Maringa | 40 |

Main fields used in each dataset:

olist_customers:
- customer_id
- customer_city

olist_order_items:
- order_id
- product_id
- price

olist_order_payments:
- payment_type
- payment_value

olist_order_reviews:
- review_id
- review_score

olist_orders:
- order_id
- order_purchase_timestamp

olist_products:
- product_id
- product_category_name

olist_sellers:
- seller_id
- seller_city

product_category_name_translation:
- product_category_name
- product_category_name_english

*Brazilian Cities Dataset*

a. Relevant size statistics:

| **Name of dataset** | **Number of rows** | **Number of columns** | **Size (MB)** |
|---|---|---|---|
| Brazilian_Cities | 5,573 | 81 | 2.36 |

b. Top 10 Brazilian cities with highest GDP:

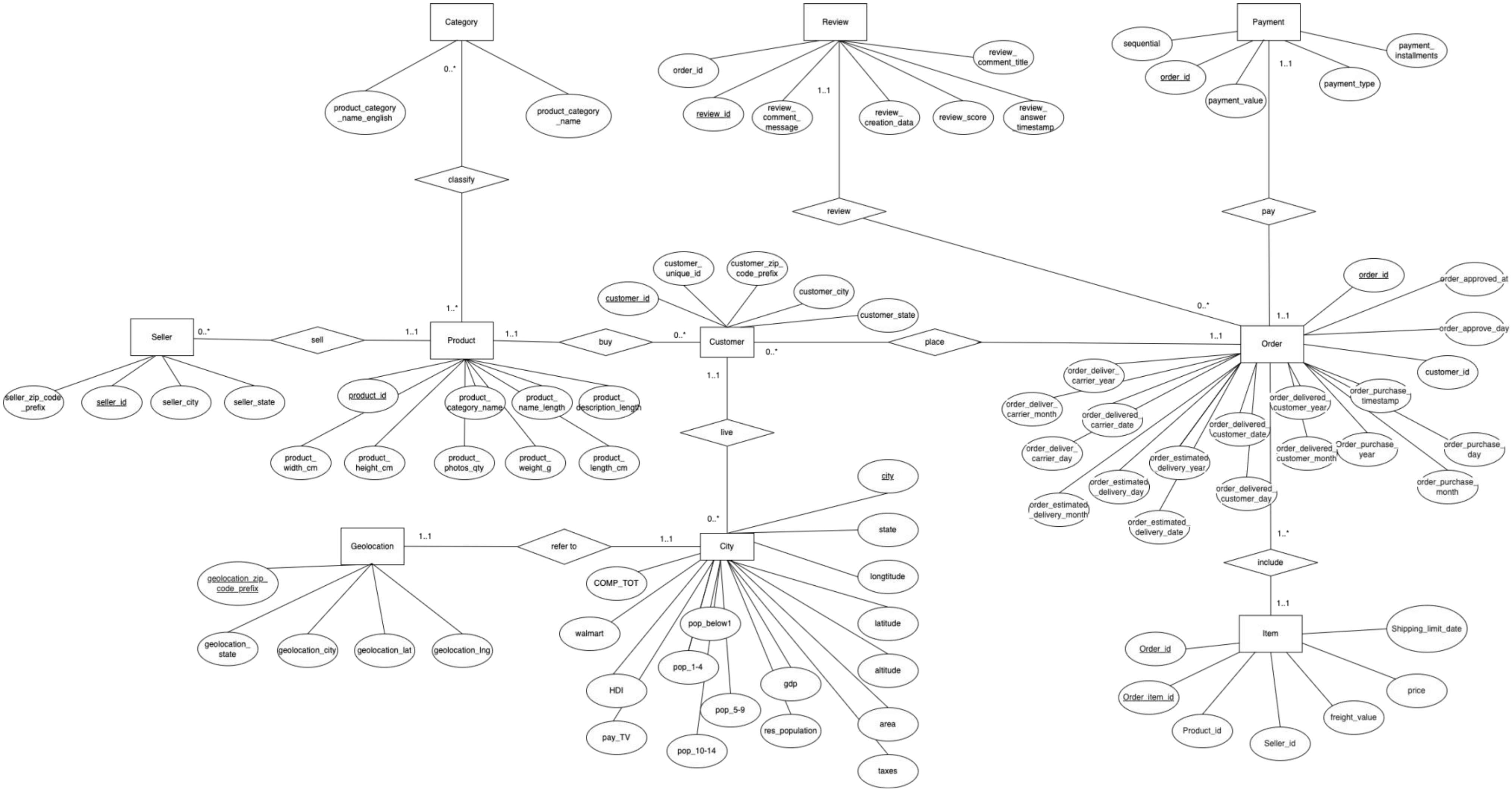
| city | state | gdp |
|---|---|---|
| são paulo | SP | 687035889.6 |
| rio de janeiro | RJ | 329431359.9 |
| brasilia | DF | 235497106.6 |
| belo horizonte | MG | 88277462.53 |
| osasco | SP | 74402691.05 |
| manaus | AM | 70296364.35 |
| salvador | BA | 61102372.82 |
| fortaleza | CE | 60141145.2 |
| campinas | SP | 58523732.73 |
| guarulhos | SP | 53974918.69 |

c. Summary statistics of relevant attributes:

| index | gdp | res_population | HDI | COMP_TOT | pay_TV |
|---|---|---|---|---|---|
| **count** | 5570.0 | 5565.0 | 5565.0 | 5570.0 | 5570.0 |
| **mean** | 954583.6306283664 | 34277.771608265946 | 0.659185804132974 | 906.7531418312387 | 3094.209156193896 |
| **std** | 11219549.873402538 | 203112.62242404092 | 0.07196084370221587 | 8333.768198888903 | 35794.78784506603 |
| **min** | 14.96 | 805.0 | 0.418 | 6.0 | 1.0 |
| **25%** | 43709.4625 | 5235.0 | 0.599 | 68.0 | 88.0 |
| **50%** | 125153.425 | 10934.0 | 0.665 | 162.0 | 247.0 |
| **75%** | 329539.2825 | 23424.0 | 0.718 | 448.0 | 815.0 |
| **max** | 687035889.6 | 11253503.0 | 0.862 | 530446.0 | 2047668.0 |

*Note: res_population – resident population, COMP_TOT – total number of companies in the city, pay_TV – number of Pay TV users in the city*

## *Appendix 1* - ER diagram




*Note: the following attributes are omitted from the City table in the above ER diagram due to space limit: COMP_A, COMP_B, COMP_C, COMP_D, COMP_E, COMP_F, COMP_G, COMP_H, COMP_I, COMP_J, COMP_K, COMP_L, COMP_M, COMP_N, COMP_O, COMP_P, COMP_Q, COMP_R, COMP_S, COMP_T, COMP_U. There are a total of 40 attributes in City table.*

## *Appendix 2* – SQL Queries (Optimized)

### Query 1:

```
CREATE INDEX item_idx
ON Item(order_item_id);

# Type 1: (group by product category)
SELECT product_category_name_english, AVG(review_score) as avgreview_2018,
count(I.product_id) AS review_num
              FROM Item I
                       NATURAL JOIN Product
                       NATURAL JOIN OrderInfo
                       NATURAL JOIN Review
                       Natural JOIN Category
              WHERE order_purchase_year = 2018
              GROUP BY product_category_name
              ORDER BY avgreview_2018 DESC;

# Type 2: (group by each seller)
SELECT seller_id, AVG(review_score) as avgreview_2018, count(I.product_id) AS
review_num
              FROM Item I
                       NATURAL JOIN Product
                       NATURAL JOIN OrderInfo
                       NATURAL JOIN Review
              WHERE order_purchase_year = 2018
              GROUP BY seller_id
              ORDER BY avgreview_2018 DESC;
```

### Query 2:

```
WITH TEMP1
    AS (SELECT product_category_name, COUNT(order_id) AS order_2016
        FROM Item
           NATURAL JOIN Product
           NATURAL JOIN OrderInfo
        WHERE order_purchase_year = 2016
        GROUP BY product_category_name),
TEMP2
    AS (SELECT product_category_name, COUNT(order_id) AS order_2017
         FROM Item
          NATURAL JOIN Product
          NATURAL JOIN OrderInfo
         WHERE order_purchase_year = 2017
         GROUP BY product_category_name),
TEMP3
    AS (SELECT product_category_name, COUNT(order_id) AS order_2018
        FROM Item
          NATURAL JOIN Product
          NATURAL JOIN OrderInfo
         WHERE order_purchase_year = 2018
        GROUP BY product_category_name)
SELECT product_category_name_english AS Product,
       order_2016,
       order_2017,
       order_2018,
       (order_2017 - order_2016) AS difference_2016_2017,
       (order_2018 - order_2017) AS difference_2017_2018
```

```
FROM TEMP1
    NATURAL JOIN TEMP2
    NATURAL JOIN TEMP3
    NATURAL JOIN Category;
```

**Query 3:**

```
WITH top_cities
       AS (SELECT city, walmart
           FROM City
           ORDER BY walmart DESC
           LIMIT 5),
    orders_products
       AS (SELECT O.order_id,
               O.order_deliver_customer_year AS year,
               C.customer_id,
               C.customer_city AS city,
               P.product_id,
               P.product_category_name,
               I.price
           FROM OrderInfo O
           JOIN Item I ON O.order_id = I.order_id
           JOIN Product P ON I.product_id = P.product_id
           JOIN Customer C ON O.customer_id = C.customer_id),
   total_orders
       AS (SELECT city, year, COUNT(DISTINCT order_id) AS count
           FROM orders_products
           WHERE city IN (SELECT city FROM top_cities)
           GROUP BY city, year),
   total_sales
       AS (SELECT city, year, SUM(price) AS sales
           FROM orders_products
           WHERE city IN (SELECT city FROM top_cities)
           GROUP BY city, year),
   top_product
       AS (SELECT city, year, c.product_category_name_english, SUM(price) AS sales
           FROM orders_products op
           JOIN Category c ON c.product_category_name = op.product_category_name
           WHERE city IN (SELECT city FROM top_cities)
           GROUP BY city, year, c.product_category_name_english)
SELECT tc.city AS City,
       tc.walmart AS 'Number of Walmart Stores',
       tto.year AS Year,
       tto.count AS 'Number of Orders',
       ts.sales AS Sales,
       tp.product_category_name_english AS 'Top Selling Product'
FROM top_cities tc
NATURAL JOIN total_orders tto
NATURAL JOIN total_sales ts
JOIN top_product tp ON tc.city = tp.city
WHERE tto.year = tp.year AND tp.sales >= ALL (SELECT sales
                                              FROM top_product tp
                                              WHERE tp.city = tc.city
                                              AND tp.year = tto.year)
ORDER BY tc.walmart DESC, tto.year;
```

**Query 4:**

```
CREATE INDEX order_id_idx
ON OrderInfo(order_id);

WITH temp
   AS (SELECT R.review_id AS review_id,
              R.review_score AS review_score,
              P.product_id AS product_id,
              C.product_category_name_english AS product_category
        FROM Review R
        Natural JOIN Category C
        NATURAL JOIN Product P
        NATURAL JOIN Item I)
SELECT product_category,
       AVG(review_score) AS avg_review_score,
       COUNT(*) AS review_num
FROM temp
GROUP BY product_category
HAVING COUNT(*) > 3
ORDER BY avg_review_score DESC
LIMIT 10;
```

**Query 5:**

```
WITH TEMP1
  AS (SELECT customer_state, SUM(payment_value) AS total_payment_credit,
             AVG(payment_value) AS avg_payment_credit,
             MIN(payment_value) AS min_payment_credit,
             MAX(payment_value) AS max_payment_credit
        FROM Item
                 NATURAL JOIN Product
                 NATURAL JOIN Customer
                 NATURAL JOIN OrderInfo
                 NATURAL JOIN Payment
        WHERE payment_type = 'credit_card'
        GROUP BY customer_state),
TEMP2
  AS (SELECT customer_state, SUM(payment_value) as total_payment_boleto,
             AVG(payment_value) AS avg_payment_boleto,
             MIN(payment_value) AS min_payment_boleto,
             MAX(payment_value) AS max_payment_boleto
        FROM Item
                 NATURAL JOIN Product
                 NATURAL JOIN Customer
                 NATURAL JOIN OrderInfo
                 NATURAL JOIN Payment
        WHERE payment_type = 'boleto'
        GROUP BY customer_state)

SELECT customer_state,
     ROUND((total_payment_credit - total_payment_boleto), 2) AS total_paydiff,
     ROUND((avg_payment_credit - avg_payment_boleto), 2) AS avg_paydiff,
     ROUND((max_payment_credit - max_payment_boleto), 2) AS max_paydiff,
     ROUND((min_payment_credit - min_payment_boleto), 2) AS min_paydiff
FROM TEMP1 NATURAL JOIN TEMP2
GROUP BY customer_state
ORDER BY total_paydiff DESC;
```

### *Appendix 3* – SQL Queries (Before Optimization)

Below highlighted in red are the parts where major optimization were made:

**Query 1:**

```
Type 1: (group by product category)
SELECT product_category_name_english, AVG(review_score) as avgreview_2018,
count(*) AS review_num
              FROM Item I, Product P, OrderInfo O, Review R, Category C
              WHERE O.order_id = I.order_id
                AND I.product_id = P.product_id
                AND R.order_id = O.order_id
                AND P.product_category_name = C.product_category_name
                AND order_purchase_year = 2018
              GROUP BY P.product_category_name
              ORDER BY avgreview_2018 DESC;

Type 2: (group by each seller)
SELECT seller_id, AVG(review_score) as avgreview_2018, count(*) AS review_num
              FROM Item I, Product P, OrderInfo O, Review R
              WHERE O.order_id = I.order_id
                AND I.product_id = P.product_id
                AND R.order_id = O.order_id
                 AND order_purchase_year = 2018
              GROUP BY seller_id
              ORDER BY avgreview_2018 DESC;
```

**Query 2:**

```
SELECT product_category_name_english AS Product,
        order_2016,
        order_2017,
        order_2018,
        (order_2017 - order_2016) AS difference_2016_2017,
        (order_2018 - order_2017) AS difference_2017_2018
FROM (SELECT product_category_name, COUNT(order_id) AS order_2016
      FROM Item
         NATURAL JOIN Product
         NATURAL JOIN OrderInfo
      WHERE order_purchase_year = 2016
      GROUP BY product_category_name) temp1
NATURAL JOIN (SELECT product_category_name, COUNT(order_id) AS order_2017
               FROM Item
                     NATURAL JOIN Product
                     NATURAL JOIN OrderInfo
               WHERE order_purchase_year = 2017
               GROUP BY product_category_name) temp2
NATURAL JOIN (SELECT product_category_name, COUNT(order_id) AS order_2018
               FROM Item
                     NATURAL JOIN Product
                     NATURAL JOIN OrderInfo
               WHERE order_purchase_year = 2018
               GROUP BY product_category_name) temp3
NATURAL JOIN Category;
```

**Query 3:**

```
WITH top_cities
       AS (SELECT *
           FROM City
           ORDER BY walmart DESC),
    orders_products
       AS (SELECT O.order_id,
                  O.order_deliver_customer_year AS year,
                  C.customer_id,
                  C.customer_city AS city,
                  P.product_id,
                  P.product_category_name,
                  I.price
           FROM OrderInfo O, Item I, Product P, Customer C
           WHERE O.order_id = I.order_id
           AND I.product_id = P.product_id
           AND O.customer_id = C.customer_id),
   total_orders
       AS (SELECT city, year, COUNT(DISTINCT order_id) AS count
           FROM orders_products
           WHERE city IN (SELECT * FROM (SELECT city FROM top_cities LIMIT 5) temp)
           GROUP BY city, year),
   total_sales
       AS (SELECT city, year, SUM(price) AS sales
           FROM orders_products
           WHERE city IN (SELECT * FROM (SELECT city FROM top_cities LIMIT 5) temp)
           GROUP BY city, year),
   top_product
       AS (SELECT city, year, c.product_category_name_english, SUM(price) AS sales
           FROM orders_products op, Category c
           WHERE c.product_category_name = op.product_category_name
           AND city IN (SELECT * FROM (SELECT city FROM top_cities LIMIT 5) temp)
           GROUP BY city, year, c.product_category_name_english)
SELECT tc.city AS City,
       tc.walmart AS 'Number of Walmart Stores',
       tto.year AS Year,
       tto.count AS 'Number of Orders',
       ts.sales AS Sales,
       tp.product_category_name_english AS 'Top Selling Product'
FROM top_cities tc
NATURAL JOIN total_orders tto
NATURAL JOIN total_sales ts
JOIN top_product tp ON tc.city = tp.city
WHERE tto.year = tp.year AND tp.sales >= ALL (SELECT sales
                                              FROM top_product tp
                                              WHERE tp.city = tc.city
                                              AND tp.year = tto.year)
ORDER BY tc.walmart DESC, tto.year;
```

## Query 4:

```
WITH temp
   AS (SELECT R.review_id AS review_id,
              R.review_score AS review_score,
              P.product_id AS product_id,
              C.product_category_name_english AS product_category
   FROM Review R
   JOIN Item I
   ON I.order_id = R.order_id
```

```
   JOIN Product P
   ON P.product_id = I.product_id
   JOIN Category C
   ON C.product_category_name = P.product_category_name)
SELECT product_category,
       AVG(review_score) AS avg_review_score,
       COUNT(*) AS review_num
FROM temp
GROUP BY product_category
HAVING COUNT(*) > 3
ORDER BY avg_review_score DESC
LIMIT 10;
```

**Query 5:**

```
WITH TEMP1
  AS (SELECT customer_state, SUM(payment_value) AS total_payment_credit,
             AVG(payment_value) AS avg_payment_credit,
             MIN(payment_value) AS min_payment_credit,
             MAX(payment_value) AS max_payment_credit
        FROM Item
                 NATURAL JOIN Product
                 NATURAL JOIN OrderInfo
                 NATURAL JOIN Payment
                 NATURAL JOIN Customer
        WHERE payment_type = 'credit_card'
        GROUP BY customer_state),
TEMP2
  AS (SELECT customer_state, SUM(payment_value) as total_payment_boleto,
             AVG(payment_value) AS avg_payment_boleto,
             MIN(payment_value) AS min_payment_boleto,
             MAX(payment_value) AS max_payment_boleto
        FROM Item
                 NATURAL JOIN Product
                 NATURAL JOIN OrderInfo
                 NATURAL JOIN Payment
                 NATURAL JOIN Customer
        WHERE payment_type = 'boleto'
        GROUP BY customer_state)

SELECT DISTINCT customer_state,
     ROUND((total_payment_credit - total_payment_boleto), 2) AS total_paydiff,
     ROUND((avg_payment_credit - avg_payment_boleto), 2) AS avg_paydiff,
     ROUND((max_payment_credit - max_payment_boleto), 2) AS max_paydiff,
     ROUND((min_payment_credit - min_payment_boleto), 2) AS min_paydiff
FROM TEMP1 NATURAL JOIN TEMP2
ORDER BY total_paydiff DESC;
```

## *Appendix 4* – Query Optimization Strategies

We developed the following optimization strategies for our queries from the resources provided by the course:

Source links: [Link 1](), [Link 2]()

- ***Strategy 1:*** Push selection and projection down to the base query. Specify necessary column names in the SQL query instead of selecting all columns (using SELECT * FROM).

- ***Strategy 2:*** Create and use indexes. We base indexes on the columns that are being queried, especially those in joins. We also create indexes on primary keys to boost any computation involving the primary key field.

- ***Strategy 3:*** Use views or temporary tables to temporarily store frequently used data, which leads to improvement in query performance. We try to avoid correlated subqueries because these could decrease the speed of execution.

- ***Strategy 4:*** Limit the results obtained by the query. In case only limited results are required, it is better to use the LIMIT statement. This statement limits the records and only displays the number of records specified.

- ***Strategy 5:*** Remove the DISTINCT clause if not required. The DISTINCT clause is used to obtain distinct results from a query by eliminating the duplicates. However, this increases the execution time of the query as all the duplicate fields are grouped together. So, it is better to avoid the DISTINCT clause as much as possible. As an alternative, the GROUP BY clause can be used to obtain distinct results.

- ***Strategy 6:*** Use INNER JOIN instead of WHERE clause for creating joins or correlated subqueries because using the WHERE clause for creating joins will result in a Cartesian Product of the number of rows of two tables. This will slow down our query performance as our database uses several larger datasets.

- ***Strategy 7:*** Reduce the data before any joins as much as possible. When joining, make sure smaller tables are on the left side of join syntax, which makes this data set to be in memory/ broadcasted to all the vertical nodes and makes join faster.

- ***Strategy 8:*** Using INNER JOIN or NATURAL JOIN whenever possible.

## *Appendix 5* – API Specification

---------------------------------------------------------------------------------------

**ROUTE 1**
Request Path: /hello
Description : Returns a user name to fill in the greeting on our homepage. Greet the user and welcome the user to our Oeda Platform
Request Parameter(s) : None
Request Description: N/A
Query Parameter(s) : None
Route Handler : hello(req, res)
Response Type : string
Response Parameters : None
Response description :

- default: general welcome
- with name param: general welcome concatenated with user name

---------------------------------------------------------------------------------------

**Sample**
Path: http://localhost:8080/hello
Response: Hi! Welcome to the Oeda Platform!

Path: http://localhost:8080/hello?name=Clara
Response: Hi, Clara! Welcome to the Oeda Platform!

## Dashboard Section

---------------------------------------------------------------------------------------

**ROUTE 2**
Request Path: /YearlyOrder
Description : Returns an array of total numbers of orders in 2016, 2017, and 2018, and the differences/increments between two years (2016 v.s. 2017, 2017 v.s. 2018). Display the result in our homepage dashboards
Request Parameter(s) : year(int)
Request Description: select from year 2016, 2017, 2018
Query Parameter(s) : None
Route Handler : yearly_order(req, res)
Response Type : JSON
Response Parameters : { results (JSON array of {order_2018 (int), difference_2017_2018(string)}) }

Response description :

- default: total numbers of order in 2018, and the growth percentage compared to previous year
- with year param: total numbers of order in the chosen year, and the growth percentage compared to previous year

---------------------------------------------------------------------------------------

**Sample**
Path: http://localhost:8080/YearlyOrder?year=2018 (default)
http://localhost:8080/YearlyOrder (with param)

Response:


```
{
   "results": [
       {
           "order_2018": 52778,
           "difference_2017_2018": "21.58%"
       }
   ]
}
```

---------------------------------------------------------------------------------
**ROUTE 3**
Request Path: /YearlySales
Description : Returns an array of total sales in 2016, 2017, and 2018, and the differences/increments between two years (2016 v.s. 2017, 2017 v.s. 2018). Display the result in our homepage dashboards
Request Parameter(s) : year(int)
Request Description: select from year 2016, 2017, 2018
Query Parameter(s) : None
Route Handler : yearly_sales(req, res)
Response Type : JSON
Response Parameters : { results (JSON array of {sales_2018 (int), difference_2017_2018(string)}) }
Response description :

- default: total amount of sales in 2018, and the growth percentage compared to previous year
- with year param: total amount of sales in the chosen year, and the growth percentage compared to previous year

---------------------------------------------------------------------------------
**Sample**
Path: http://localhost:8080/YearlySales (default)
http://localhost:8080/YearlyOrder?year=2018 (with param)

Response:


```
{
   "results": [
       {
           "sales_2018": 3785320.43,
           "difference_2017_2018": "22.72%"
       }
   ]
}
```


---------------------------------------------------------------------------------
**ROUTE 4**
Request Path: /YearlyReview
Description : Returns an array of average review in 2016, 2017, and 2018, and the differences/increments between two years (2016 v.s. 2017, 2017 v.s. 2018). Display the result in our homepage dashboards
Request Parameter(s) : year(int)
Request Description: select from year 2016, 2017, 2018
Query Parameter(s) : None
Route Handler : yearly_review(req, res)
Response Type : JSON
Response Parameters : { results (JSON array of {review_2018 (int), difference_2017_2018(string)}) }
Response description :

- default: average review score in 2018, and the growth percentage compared to previous year
- with year param: average review score in the chosen year, and the growth percentage compared to previous year

---------------------------------------------------------------------------------
**Sample**
Path: http://localhost:8080/YearlyReview?year=2017(default)
http://localhost:8080/YearlyReview(with param)

Response:


```
{
   "results": [
       {
           "review_2017": 3.89,
```

```
        "difference_2016_2017": null
    }
  ]
}
```

--------------------------------------------------------------------------------------
**ROUTE 5**
Request Path: /YearlyState
Description : Returns an array of the number of states participating in the ecommerce in 2016, 2017, and 2018, and the differences/increments between two years (2016 v.s. 2017, 2017 v.s. 2018). Display the result in our homepage dashboards
Request Parameter(s) : year(int)
Request Description: select from year 2016, 2017, 2018
Query Parameter(s) : None
Route Handler : yearly_state(req, res)
Response Type : JSON
Response Parameters : { results (JSON array of {state_2018 (int), difference_2017_2018(string)}) }
Response description :
- default: total number of states participated in 2018, and the growth percentage compared to previous year
- with year param: total number of states in the chosen year, and the growth percentage compared to previous year

--------------------------------------------------------------------------------------
**Sample**
Path: http://localhost:8080/YearlyState (default)
http://localhost:8080/YearlyState?year=2017 (with param)

Response:

```
{
  "results": [
    {
        "state_2017": 27,
        "difference_2016_2017": "28.57%"
    }
  ]
}
```

## Analytics Section

--------------------------------------------------------------------------------------
**ROUTE 6**
Request Path: /search
Description : Returns details for the transactions that can be filtered by users based on price range, category, and time range
Request Parameter(s) : category(string), low (int), high(int), year(int), month(int)
Request Description: select from year 2016, 2017, 2018
Query Parameter(s) : None
Route Handler : search(req, res)
Response Type : JSON
Response Parameters : { results (JSON array of {id (int), order_id(int), price (float), category (string), year (int), month(int), review_score (float), customer_city (string)}) }
Response description :
- default: matching results based on default param values
- with year param: results matching param inputs

--------------------------------------------------------------------------------------

**Sample**
Path: http://localhost:8080/search (default category"", low 0 high 10000000 year "" month "")
http://localhost:8080/search?category=perf&low=0&high=500&year=2018&month=5&page=1&pagesize=2

( with param)

Response:


```
{
   "results": [
       {
           "id": "fff5169e583fd07fac9fec88962f189d",
           "order_id": "000aed2e25dbad2f9ddb70584c5a2ded",
           "price": 144,
           "category": "perfumery",
           "year": 2018,
           "month": 5,
           "review_score": 1,
           "customer_city": "santa barbara d'oeste"
       },
       {
           "id": "de0c1a4d8c367c58d66e61dfa379f4cf",
           "order_id": "0094bd07f49fed90209ffa62d1ef26d6",
           "price": 11.53,
           "category": "perfumery",
           "year": 2018,
           "month": 5,
           "review_score": 1,
           "customer_city": "praia grande"
       }
]
}
```


## Report Section

-------------------------------------------------------------------------------------

**Route 7**
Request Path: /all_market
Description : Returns a sales report for the top 5 cities with the most Walmart stores
Request Parameter(s) : None
Query Parameter(s) : None
Route Handler : all_market(req, res)
Return Type : JSON
Return Parameters : { results (JSON array of {City (string), Number of Walmart Store (int), Year (int)Number of Orders (int), Sales (float), Top Selling Product (string)}) }
Response description :

- An array of 5 cities

-------------------------------------------------------------------------------------

**Sample**
Path: http://localhost:8080/allmarket

Response:


```
{
   "results": [
       {
           "City": "salvador",
           "Number of Walmart Stores": 26,
           "Year": 2018,
           "Number of Orders": 125,
           "Sales": 14312.609946250916,
           "Top Selling Product": "auto"
       },
```

```
        {
            "City": "curitiba",
            "Number of Walmart Stores": 16,
            "Year": 2018,
            "Number of Orders": 122,
            "Sales": 14875.089968681335,
            "Top Selling Product": "small appliances"
        },
        {
            "City": "recife",
            "Number of Walmart Stores": 14,
            "Year": 2018,
            "Number of Orders": 49,
            "Sales": 6038.710014343262,
            "Top Selling Product": "auto"
        },
        {
            "City": "porto alegre",
            "Number of Walmart Stores": 12,
            "Year": 2018,
            "Number of Orders": 122,
            "Sales": 12946.850032806396,
            "Top Selling Product": "telephony"
        },
        {
            "City": "são paulo",
            "Number of Walmart Stores": 7,
            "Year": 2018,
            "Number of Orders": 1515,
            "Sales": 164246.76998853683,
            "Top Selling Product": "auto"
        }
    ]
}
```


-------------------------------------------------------------------------------------**Route: 8**

Request Path: /market
Description : Returns the market info for a specific city in Brazil
Route Parameter(s) : city(string), year(int)
Query Parameter(s) : None
Route Handler : market(req, res)
Return Type : JSON
Return Parameters : { results (JSON array of {City (string), Number of Walmart Store (int), Year (int), Number of Orders (int), Sales (float), Top Selling Product (string)}) }
Response description :

- Case 1: If the query parameter ( city, year ) is defined
  - Return match entries from the specific city and year.
- Case 2: If the query parameter ( city, year) is not defined
  - Return the query parameter will be set as "" and ""

-------------------------------------------------------------------------------------

**Sample**
Path: .http://localhost:8080/market (default city"", year "")
http://localhost:8080/market?city=salvador&year=2017 (with param)

Response:


```
{
   "results": [
      {
```

```
            "City": "salvador",
            "Number of Walmart Stores": 26,
            "Year": 2017,
            "Number of Orders": 82,
            "Sales": 9299.600039958954,
            "Top Selling Product": "perfumery"
        }
    ]
}
```

---------------------------------------------------------------------------------
**ROUTE 9**
Request Path: /allreview
Description : Return list the top 10 highest average review score product.Users can query and create a review report for the top 10 products with highest average review score and standard deviation score. We also allow users to filter the review by a specific duration of time.
Request Parameter(s) : None
Request Description: N/A
Query Parameter(s) : None
Route Handler : all_review(req, res)
Response Type : JSON
Response Parameters : { results (JSON array of {product_category (string), avg_review_score(float)}) }
Response description :

- default: return top 10 categories and their average and standard deviation results

---------------------------------------------------------------------------------
**Sample**
Path: .http://localhost:8080/allreview

Response:

```
{
   "results": [
       {
           "productCategory": "home appliances",
           "ReviewScore": 4.33
       },
       {
           "productCategory": "toys",
           "ReviewScore": 4.31
       },
       {
           "productCategory": "audio",
           "ReviewScore": 4.29
       },
       {
           "productCategory": "drinks",
           "ReviewScore": 4.22
       },
       {
           "productCategory": "electronics",
           "ReviewScore": 4.13
       },
       {
           "productCategory": "food",
           "ReviewScore": 4.12
       },
```

```
        {
            "productCategory": "stationery",
            "ReviewScore": 4.1
        },
        {
            "productCategory": "perfumery",
            "ReviewScore": 4.03
        },
        {
            "productCategory": "small appliances",
            "ReviewScore": 4
        },
        {
            "productCategory": "computers",
            "ReviewScore": 4
        }
    ]
}
```

-----------------------------------------------------------------------------------------
**ROUTE 10**
Request Path: /review
Description : We list the top 10 highest average review score product.Users can query and create a review report for the top 10 products with highest average review score and standard deviation score. We also allow users to filter the review by a specific duration of time.
Request Parameter(s) : category(string)
Request Description: select from product category
Query Parameter(s) : None
Route Handler : review(req, res)
Response Type : JSON
Response Parameters : { results (JSON array of {product_category (string), avg_review_score(float), std_dev_review_score (float), review_num (int)}) }
Response description :

- default: select default category "home appliances"
- with category param: return product review results for the selected category

-----------------------------------------------------------------------------------------
**Sample**
Path:http://localhost:8080/review (default category "home appliances")
     http://localhost:8080/review?category=home appliances (with category param)

Response:

```
{
    "results": [
        {
            "product_category": "home appliances",
            "avg_review_score": 4.333333333333333,
            "std_dev_review_score": 1.1785113019775793,
            "review_num": 12
        }
    ]
}
```

-----------------------------------------------------------------------------------------
**ROUTE 11**
Request Path: /allhabit
Description : Returns an array of the payment habits including the differences in total, average, max, and min payment values by credit card users and boleto (bank tickets) users from each state.
Request Parameter(s) : page(int), pagesize(int)
Request Description: 27 results in total
Query Parameter(s) : None

Route Handler : all_habit(req, res)
Response Type : JSON
Response Parameters : { results (JSON array of {customer_id (int), total_paydiff(float), avg_paydiff (float), max_paydiff (float), min_paydiff (float)}) }
Response description :

- default: return the first page of states results (page size = 10)
- with page and pagesize param: return the selected page and pagesize results

-------------------------------------------------------------------------------------

**Sample**
Path: http://localhost:8080/allhabit (default, page = 1, pagesize = 10)
http://localhost:8080/allhabit?page=1&pagesize=2 (with page and pagesize param)

Response:


```
{
    "results": [
        {
            "customer_state": "SP",
            "total_paydiff": 266370.54,
            "avg_paydiff": 28.19,
            "max_paydiff": 2561.94,
            "min_paydiff": -14.38
        },
        {
            "customer_state": "RJ",
            "total_paydiff": 102473.09,
            "avg_paydiff": 41.7,
            "max_paydiff": 943.99,
            "min_paydiff": -26
        }
    ]
}
```


-------------------------------------------------------------------------------------

**ROUTE 12**
Request Path: /habit
Description : Returns an array of the payment habits including the differences in total, average, max, and min payment values by credit card users and boleto (bank tickets) users from each state.
Request Parameter(s) : state(string)
Request Description: select from Brazil’s state using initials
Query Parameter(s) : None
Route Handler : habit(req, res)
Response Type : JSON
Response Parameters : { results (JSON array of {customer_id (int), total_paydiff(float), avg_paydiff (float), max_paydiff (float), min_paydiff (float)}) }
Response description :

- default: return the result for state SP
- with state param: return the habit result for the specific state input

-------------------------------------------------------------------------------------

**Sample**
Path: http://localhost:8080/habit (default state “”)
http://localhost:8080/habit?state=SP (with param)

Response:


```
{
    "results": [
        {
            "customer_state": "SP",
            "total_paydiff": 266370.54,
```

```
            "avg_paydiff": 28.19,
            "max_paydiff": 2561.94,
            "min_paydiff": -14.38
        }
    ]
}
```


--------------------------------------------------------------------------------------

**ROUTE 13**

Request Path: /transaction
Description : Returns 10 of the most recent transactions
Request Parameter(s) : None
Request Description: N/A
Query Parameter(s) : None
Route Handler : transaction(req, res)
Response Type : JSON
Response Parameters : { results (JSON array of {order_id(int),customer_id (int), order_purchase_timestamp (timestamp), payment_value (float)}) }
Response description :

- return the order id, customer id, transaction time and the transaction value for 10 of the most recent transactions happened in Brazil ecommerce platform.

--------------------------------------------------------------------------------------

**Sample**

Path: http://localhost:8080/transaction

Response:


```
{
    "results": [
        {
            "order_id": "35a972d7f8436f405b56e36add1a7140",
            "customer_id": "898b7fee99c4e42170ab69ba59be0a8b",
            "order_purchase_timestamp": "2018-08-29T19:00:37.000Z",
            "Payment_value": 93.75
        },
        {
            "order_id": "912859fef5a0bd5059b6d48fa79d121a",
            "customer_id": "b8c19e70d00f6927388e4f31c923d785",
            "order_purchase_timestamp": "2018-08-29T13:48:09.000Z",
            "Payment_value": 178.25
        },
        {
            "order_id": "fb393211459aac00af932cd7ab4fa2cc",
            "customer_id": "54365416b7ef5599f54a6c7821d5d290",
            "order_purchase_timestamp": "2018-08-29T13:14:11.000Z",
            "Payment_value": 106.95
        },
        {
            "order_id": "bee12e8653a04e76786e8891cfb6330a",
            "customer_id": "448945bc713d98b6726e82eda6249b9e",
            "order_purchase_timestamp": "2018-08-29T12:46:11.000Z",
            "Payment_value": 497.25
        },
        {
            "order_id": "4c867c2aac81653679eff07e922441a0",
            "customer_id": "8ffe52ac7d480a27180338e697eec534",
            "order_purchase_timestamp": "2018-08-29T02:51:54.000Z",
            "Payment_value": 38.36
        },
        {
            "order_id": "e7c290bfc31d7eed478c3d3d2d4d2953",
            "customer_id": "004440537b68545ca3c341d7279bc4c0",
            "order_purchase_timestamp": "2018-08-29T02:30:32.000Z",
```

```
            "Payment_value": 98.04
        },
        {
            "order_id": "dbb786f88b6d4e52fe3cb5d771b979d6",
            "customer_id": "478778636c75019554439f75286a22e3",
            "order_purchase_timestamp": "2018-08-29T01:56:30.000Z",
            "Payment_value": 161.97
        },
        {
            "order_id": "00b44ba3d7c4a5e9a9ebafef9150781d",
            "customer_id": "0b5f6687d659478f1747caed607c4ec5",
            "order_purchase_timestamp": "2018-08-29T01:10:46.000Z",
            "Payment_value": 67.58
        },
        {
            "order_id": "0ac69790e2a6e4c075edbd78648a3594",
            "customer_id": "1c3d8766b5f8b24d7e95001ce31d1d38",
            "order_purchase_timestamp": "2018-08-28T22:49:20.000Z",
            "Payment_value": 14.89
        },
        {
            "order_id": "93d7e1f0c5415c1a0679635187700f1d",
            "customer_id": "1288df1d9da5e124c94b06a6ba3ca50b",
            "order_purchase_timestamp": "2018-08-28T20:33:22.000Z",
            "Payment_value": 216.58
        }
    ]
}
```

------------------------------------------------------------------------------------

**ROUTE 14**

Request Path: /topOrder

Description : Returns 10 product category with the highest number of order

Request Parameter(s) : year(int)

Request Description: N/A

Query Parameter(s) : None

Route Handler : top_order(req, res)

Response Type : JSON

Response Parameters : { results (JSON array of {product_category(string), order_count(int)}) }

Response description :

- return the product category and order count for top 10 category with highest number of order

------------------------------------------------------------------------------------

**Sample**

Path: http://localhost:8080/topOrder (default)

http://localhost:8080/topOrder?year=2016 (with param)

Response:

```
{
    "results": [
        {
            "product_category": "toys",
            "order_count": 680
        },
        {
            "product_category": "telephony",
            "order_count": 662
        },
        {
            "product_category": "auto",
            "order_count": 617
        },
```

```
        {
            "product_category": "perfumery",
            "order_count": 530
        },
        {
            "product_category": "electronics",
            "order_count": 448
        },
        {
            "product_category": "baby",
            "order_count": 440
        },
        {
            "product_category": "stationery",
            "order_count": 380
        },
        {
            "product_category": "home appliances",
            "order_count": 109
        },
        {
            "product_category": "small appliances",
            "order_count": 104
        },
        {
            "product_category": "food",
            "order_count": 72
        }
    ]
}
```


---------------------------------------------------------------------------------------

**ROUTE 15**
Request Path: /topSales
Description : Returns 10 product category with the highest amount of sales
Request Parameter(s) : year(int)
Request Description: N/A
Query Parameter(s) : None
Route Handler : top_sales(req, res)
Response Type : JSON
Response Parameters : { results (JSON array of {product_category(string), sales_sum(int)}) }
Response description :

- return the product category and sales sum for top 10 category with the highest amount of sales

---------------------------------------------------------------------------------------

**Sample**
Path: http://localhost:8080/topSales (default)
http://localhost:8080/topSales?year=2016 (with param)

Response:


```
{
    "results": [
        {
            "product_category": "auto",
            "sales_sum": 85113.81
        },
        {
            "product_category": "toys",
            "sales_sum": 79925.39
        },
        {
            "product_category": "perfumery",
```

```
            "sales_sum": 64456.28
        },
        {
            "product_category": "baby",
            "sales_sum": 58005.15
        },
        {
            "product_category": "telephony",
            "sales_sum": 48028.98
        },
        {
            "product_category": "computers",
            "sales_sum": 36291.93
        },
        {
            "product_category": "stationery",
            "sales_sum": 35706.28
        },
        {
            "product_category": "small appliances",
            "sales_sum": 32287.96
        },
        {
            "product_category": "electronics",
            "sales_sum": 23526.72
        },
        {
            "product_category": "home appliances",
            "sales_sum": 10591.49
        }
    ]
}
```

---------------------------------------------------------------------------------------

**ROUTE 16**

Request Path: /topReview

Description : Returns op 10 category with highest average review

Request Parameter(s) : year(int)

Request Description: N/A

Query Parameter(s) : None

Route Handler : top_review(req, res)

Response Type : JSON

Response Parameters : { results (JSON array of {product_category(string), avg_review(int)}) }

Response description :

- return the product category and order count for top 10 category with highest average review

---------------------------------------------------------------------------------------

**Sample**

Path: http://localhost:8080/topReview (default)

http://localhost:8080/topReview?year=2018 (with param)

Response:

```
{
    "results": [
        {
            "product_category": "home appliances",
            "avg_review": 4.33
        },
        {
            "product_category": "toys",
            "avg_review": 4.31
        },
        {
```

```
            "product_category": "audio",
            "avg_review": 4.29
        },
        {
            "product_category": "drinks",
            "avg_review": 4.22
        },
        {
            "product_category": "electronics",
            "avg_review": 4.13
        },
        {
            "product_category": "food",
            "avg_review": 4.12
        },
        {
            "product_category": "stationery",
            "avg_review": 4.1
        },
        {
            "product_category": "perfumery",
            "avg_review": 4.03
        },
        {
            "product_category": "music",
            "avg_review": 4
        },
        {
            "product_category": "small appliances",
            "avg_review": 4
        }
    ]
}
```