Table Of Content

1. Background

The purpose of this document is to specify the overall requirements for the final project. It explains the outline of the development process within the development environment and defines the scope of the project, including problem statement, assumptions, objectives, use cases, the source of data, tools used, methodology for the ETL process, analysis of data, challenges or limitations we have encountered, how we circumvent it and, finally our recommendations for Olist, on how they can better their offerings towards their sellers.

2. Problem Statement

In Brazil, small and medium-sized businesses (SMBs) need help accessing financial services, limiting their growth potential. Despite the growing eCommerce sector, most SMB transactions still need to be physical, highlighting a disconnect between retail players and traditional banks. To bridge this gap and support SMB growth, Olist plans to expand its in-house banking services by 2024 to provide its sellers with integrated payment processing, tap-to-pay solutions, and better financial support [1].

3. Project Objectives

Our project objectives aim to discover patterns & trends behind the payment methods used on the Olist platform and how Olist can better their in-house banking products/offerings via the following analysis:

- **Payment Type Analysis:** Determine payment preferences on Olist Platforms & amount based on state
- **Sales Frequency Analysis:** Examine transaction patterns during specific periods, such as holidays or year-ends.
- **Seller CLV Analysis:** By categorizing sellers into low, med, high, and very high values based on the total amount transacted on Olist.

4. Use Cases

The project objectives are designed to address the financial access challenges SMBs face using Olist's platform and aid in their plans to better drive in-house banking services. By analyzing geospatial data, we can identify regions with high seller or transaction density, helping Olist better target high-density and underserved financial services. Sales frequency analysis will reveal periods of high transaction activity, enabling tailored banking support during peak times.

Furthermore, understanding customer payment preferences will help Olist expand the most convenient and preferred payment options, boosting SMBs' transaction rates. Lastly, analyzing sellers' Customer Lifetime Value (CLV) allows Olist to identify high-value clients who may benefit from enhanced financial support or low-value clients by offering them training programmes to help them grow their businesses. This will foster growth for these businesses and improve their long-term selling stability on Olist.

5. Source of Data

The Brazilian e-commerce dataset from Olist Store can be retrieved from Kaggle. The link is https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce. The dataset consists of 9 flat files (.csv), deemed accurate commercial data by Olist but anonymized, which are:

1. olist_customers_dataset.csv
2. olist_geolocation_dataset.csv
3. olist_order_items_dataset.csv
4. olist_order_payments_dataset.csv
5. olist_order_reviews_dataset.csv
6. olist_orders_dataset.csv
7. olist_products_dataset.csv
8. olist_sellers_dataset.csv
9. product_category_name_translation.csv

In addition, Olist has also released its marketing funnel dataset, which can be retrieved here: https://www.kaggle.com/datasets/olistbr/marketing-funnel-olist. This allows us to join both

datasets for the project analysis and gain a better perspective on their marketing efforts. It consists of 2 flat files (.csv), which are also anonymized:

1. olist_closed_deals_dataset.csv
2. olist_marketing_qualified_leads_dataset.csv

6. Assumptions

- An order may have multiple lists of items, each from a different seller.
    - This means that due to the unique nature of data collection from Olist, some datasets may have to repeat order_ids even if it is intentional. However, we have methods to work around this as we will be required to have a primary key for each of the dataframe/tables.
- Null values assumptions:
    - Most of the datasets/data frames will have their NULL/NaN/empty values replaced with either: 'Unknown' or 0, depending on the context (elaborated more under Section 8. Methodology)
- Business context assumptions:
    - The first payment instance of payment_type determines the entire order, so if an order_id was labelled with multiple instances of payment but was first transacted as a credit card payment, it will be grouped together and labelled as a credit card payment.

7. Tools Used

The following tools were instrumental in setting up, transforming our datasets and finally loading them into different cloud services such as storage, compute and data warehouses:

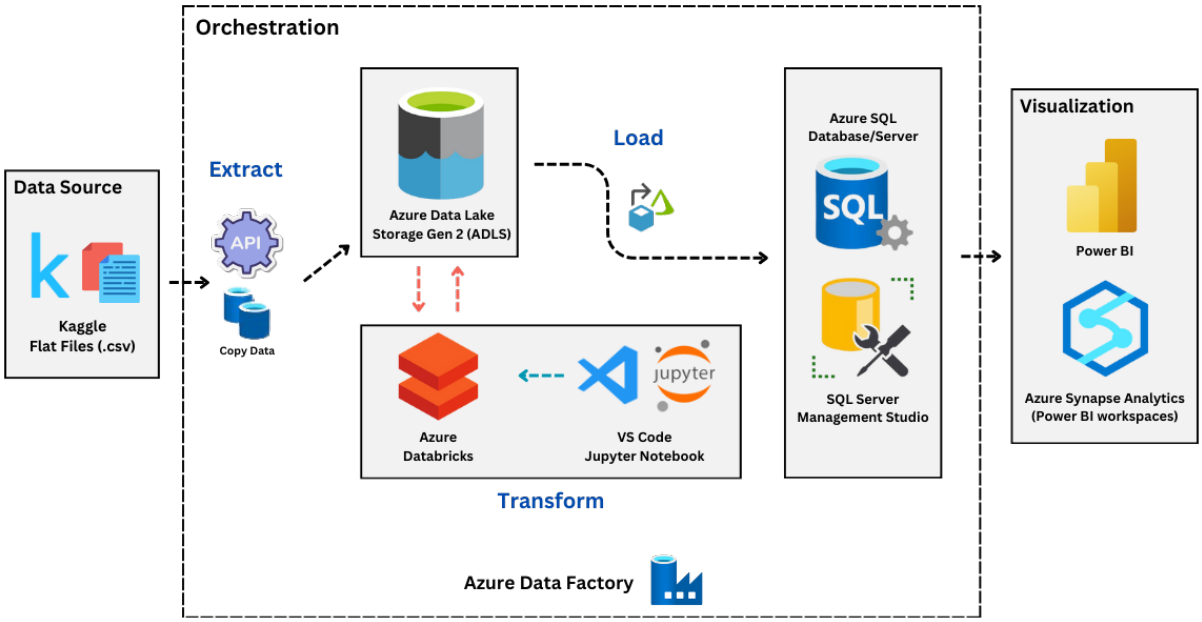
- Kaggle API
    - To extract our data, we must use the API provided by Kaggle. This will allow us to call the API and fetch datasets from the URL string provided.
- Jupyter Notebook
    - The notebook makes it easy to write our cleaning and transformation code locally and distribute it among teams. Its main function in this pipeline is for us to quickly get the code up and running and then streamline and simplify it into the actual transformation tool we will be using in the cloud later (Azure Databricks).
- Visual Studio Code
    - The IDE acts as a "temporary" solution for fetching data via Kaggle API and then storing it in our cloud storage of choice—Azure Data Lake Gen 2/Storage (ADLS) provided by Microsoft Azure. A script was written to automate the whole process of unpacking the datasets from their .zip files and then uploading them onto ADLS.
- Azure Data Lake Gen 2
    - Our cloud storage of choice. Which main purpose is to hold the raw data in their flat files (.csv) format after unpacking from the Kaggle API. And then, also store the clean flat files (.csv) format after transforming and cleaning them.
- Azure Data Factory (Azure Data Flow)
    - Our cloud storage ETL/User-interface tool for managing the entire pipeline on the cloud. While there are many ways to operate Azure Data Factory (ADF), we mainly use it to execute the extract process, transform and load (which was used in conjunction with Azure Data Flow).
- Azure Data Flow
    - It is part of ADF and during the loading phase. As we have multiple datasets to load into our data warehouse (Azure SQL Server), it is optimal to use Azure Data Flow. We can set the source files to be the cleaned data stored in ADLS

and the sink to be the Azure SQL server. The whole process can be easily debugged.

- Azure Databricks
    - Azure Databricks is our tool of choice for the transformation phase. After writing our cleaning and transformation code within Jupyter Notebook, we will deploy it onto Azure Databricks. Further streamlining and truncating the code in user-defined functions reduces the running time code and allows Azure Databricks to run our cleaning and transformation code within 1 minute.
- Azure SQL Database/Server
    - The Azure tool of choice for loading data after it has been cleaned and transformed was the Azure SQL Database (with Azure SQL Server). It allows us to perform T-SQL queries if required, set primary and foreign key constraints, and act as a data warehouse in the presentation layer of the pipeline.
- Power BI
    - For data visualisation, performing DAX operations and creating new columns within the dataset to make our visualisations able to answer our objectives. We also used Power BI workspace to cross-share semantic model reports/dashboards across the team for effective collaboration. Power BI also imports finalised datasets from the data warehouse (Azure SQL Database).
- Azure Synapse Analytics
    - Azure Synapse Analytics (ASA) was used as an additional tool to sync with the Power BI workspace and work on the data visualisations while on the cloud, tapping on Azure services.
- SQL Server Management Studio (SSMS)
    - SSMS was used for T-SQL, setting up foreign keys and primary keys, and some final debugging when the transformed/cleaned data was loaded into the Azure SQL Database/server. It was also used to automatically generate the entity relationship diagram (ERD) after relationships and constraints had been set.

8. Methodology

Project Workflow/Pipeline



1. Data is first extracted from Kaggle API via Azure Data Factory (ADF)'s copy data functionality.
2. It is then stored in our cloud storage of choice: Azure Data Lake Storage Gen 2 (ADLS).
3. The raw data is then transformed and cleaned using Azure Databricks and stored back into ADLS again for storage, but in a separate folder labelled cleaned data for Olist.
4. The cleaned datasets are then loaded into Azure SQL Database/Server using ADF's data flow functionality.
5. SQL server management studio is used to establish the tables and create primary keys and constraints such as foreign keys/referencing. It is also used to generate the entity relationship diagram (ERD).
6. Finally, the cleaned data, with relationships established, can be imported in Power BI by connecting to an Azure SQL database/server and used for visualisations.

7. As an extra step, we have also utilised Azure Synapse Analytics to tap on Power BI workspace, so team members can still work on basic visualisation on the cloud easily or share semantic models and reports on Azure.

Cleaning, Transforming & Modeling

Data Cleaning process:

- Changing 'customer_city', 'seller_city', 'order_status', 'payment_type', 'product_category_name', 'geolocation_city' and for the marketing dataset, the columns 'business_segment', 'lead_type', 'lead_behaviour_profile' and 'business_type' will be changed to lower cases for consistency.
- Changing 'customer_state', 'seller_city', 'geolocation_state' to upper cases
- Converting the following columns from different dataframes: 'order_purchase_timestamp', 'order_approved_at', 'order_delivered_carrier_date', 'order_delivered_customer_date', 'order_estimated_delivery_date', 'shipping_limit_date', 'review_creation_date' and 'review_answer_timestamp' to datetime.

Handling of NULLs or empty values:

- 'Order_approved_at' will be +1 business day from 'order_purchase_timestamp'.
- For orders which are cancelled, the 'order_delivered_carrier_date' and 'order_delivered_customer_date' will be replaced with placeholders: '2222-12-31 00:00:00'.
- For another type of 'order_status,' we will set the respective values to '2222-12-31 00:00:00'.
    - The rationale behind having the placeholders for datetime columns is that we are not planning to analyze any datetime objects, but in the event that we need to, we can easily identify these records by searching for these placeholders.
- To make it more meaningful, the 'review_comment_title' and 'review_description' labelled 'NaN' will be replaced with 'No Title' and 'No Review Provided'.
- For products, the 'product_category_name' will be 'uncategorized' if it is NaN. For the rest of the columns of the product dataset, it will be labelled as 0:
    - product_name_length

        - product_description_length
        - product_photos_qty
        - product_weight_g
        - product_length_cm
        - product_height_cm
        - product_width_cm
- For the marketing dataset (closed_deals), the following business logic applies:
    - If it is NaN, 'business_segment', 'lead_type', 'lead_behaviour_profile', and 'business_type' will be labelled as 'Unknown'.
    - 'has_company' and 'has_gtin' will be converted to a boolean datatype and labelled as 'FALSE' if the default value is 'Unknown', NaN, 0 or anything else besides 'TRUE'.
    - 'average_stock' and 'declared_product_catalog_size' will be changed to 0 if their default values are 'Unknown' or NaN.
- For the marketing dataset (qualified_leads), the following business logic applies:
    - 'origin' will be changed to 'Unknown' if it is NaN.
    - 'origin' will be changed to lowercase for consistency.
    - 'first_contact_date' will be converted to datetime data type.
- For most of the data sets, we also created a user-defined function that utilises the 'Unicode data' module to normalize accented characters found in 'sellers_city', 'customers_city', and 'geolocation_city'. This resolves the problem of cases such as **são Paulo to sao paulo** and prevents them both from showing up as unique values.

Transforming

Steps involved in transforming

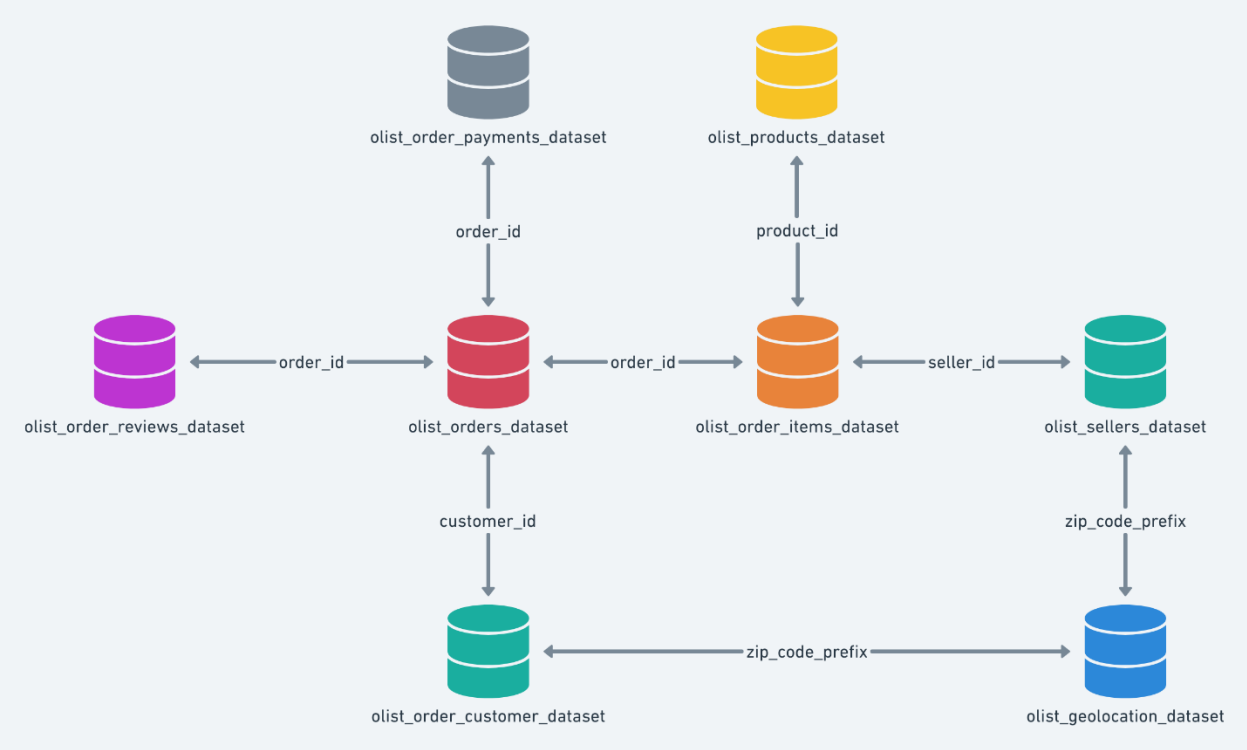
- For the products dataset, the column 'product_category_name', which by default is in Portuguese, will be replaced with the product_category_name_translation dataset for its English product category names.
- Using a custom user-defined function to get all possible combinations and permutations of 'geolocation_zip_code_prefix', 'geolocation_lat' and 'geolocation_lng' and remove the duplicated values. This results in 411,716 records being dropped, drastically reducing the size of the dataframe.
- Using a custom user-defined function to fix duplicated values of the geolocation data frame by grouping by 'zip_code_prefix' and then averaging by its latitude and longitude values, followed by using the first occurrences of 'geolocation_city' and 'geolocation_state' as its default value for each record resulting from the groupby.
- Using a custom user-defined function to get the dataset order_items, a new unique row identifier, so that we do not have to drop "duplicates" of the 'order_id' columns within the original data frame. This is because multiple 'order_id' are occurring, but it is tracked by the 'order_item_id' as the number of occurrences. However, since we still need a primary key for the dataframe/table, we decided to implement this function.
- As our analysis involved knowing more about the payment types and preferences of the users of Olist, for the dataframe order_payments, we used a custom user-defined function which, grouped by their 'order_id', performed aggregation by taking the sum of 'payment_value' and using lambda to check if the record has multiple payments by looking at the value of the column 'payment_installments' if it is > 2 then it will be TRUE. Otherwise, the default will be FALSE, taking the first occurrence of the payment type.
    - The order_reviews dataset also poses the same problem, where there can be multiple instances of similar 'order_id', but it is how Olist tracks the data. For example, an 'order_id' may have multiple 'review_id' due to having multiple items purchased. Similarly, a 'review_id' may have multiple 'order_id' because the same review can be given across all orders. This can be resolved by grouping the 'review_id' and then 'order_id'. And taking the mean 'review_score' and the latest

occurrence for 'review_comment_title', 'review_comment_message', 'review_creation_date' and 'review_answer_timestamp'.

- Using a custom user-defined function to fix missing data in the geolocation dataset, as sellers and customers are not present in the geolocation's zip code. To do this, we find the distinct zip codes and create a new dataframe for them, setting the latitude and longitude values to NaN and city and state to Unknown. After which, it is coated to the geolocation dataframe, with the latitude and longitude values replaced with 0.00 as float64 data type.
- We also identified distinct sellers in the closed_deals table who were not found in the seller's table 'seller_id', so those were also dropped from closed_deals.

Modelling

The original tables can be found under Appendix Section 1: Tables. This is the original diagram of the relationship between tables provided by Olist:



Based on our analysis, we have decided to proceed with the dimensional modelling approach using fact and dimension tables, which will make it easier to extract insights and perform payment-related analysis later on.

The fact tables will store measurable or any transactional data, such as items, payment values, etc., that can be analyzed. The dimension tables will only have descriptive attributes, such as geolocation data, to help provide more context for the fact tables.

Fact Tables (Quantifiable)

- order_payments
- order_items
- closed_deals

Dimension Tables (Qualitative)

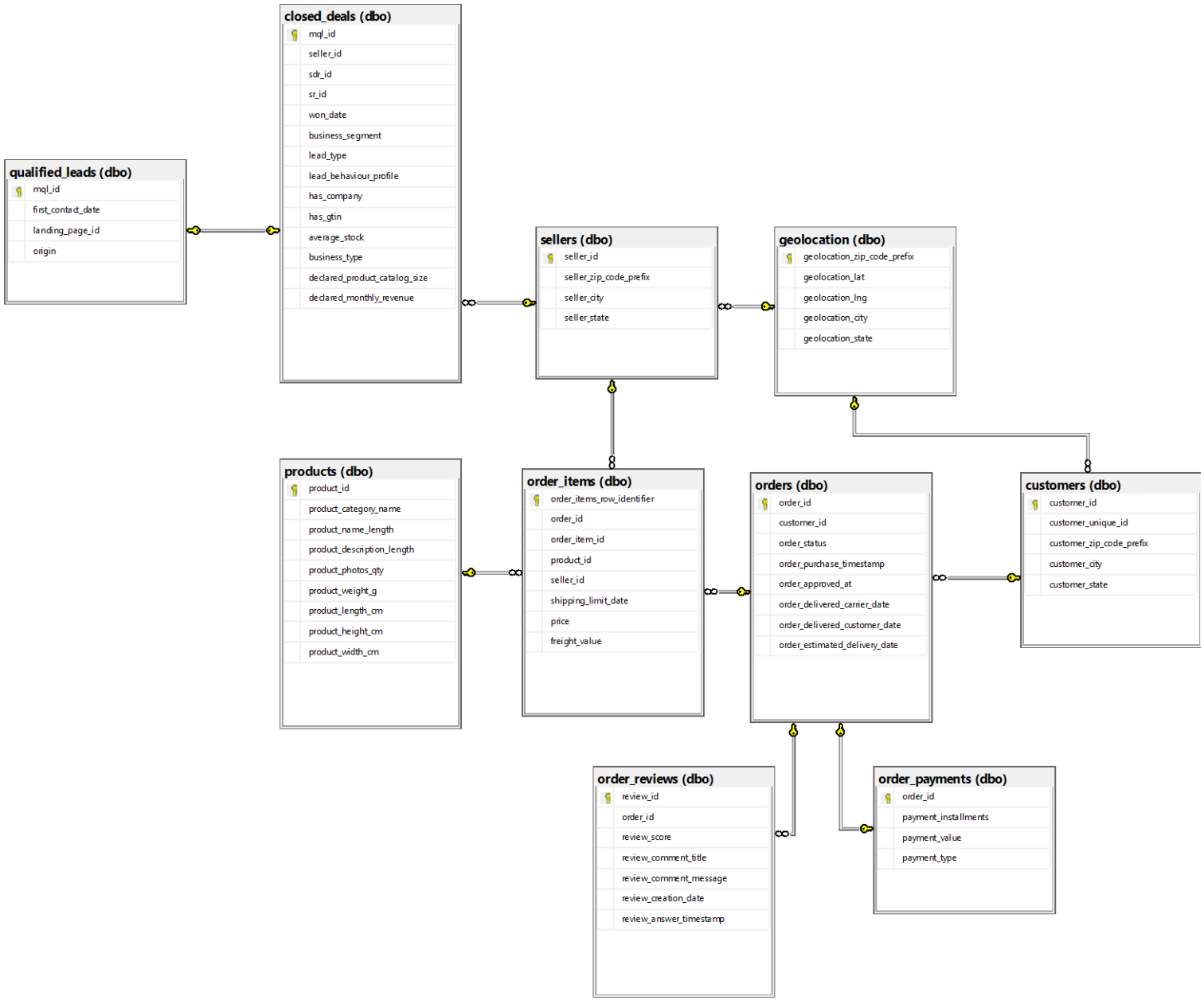
- customers
- sellers
- orders
- order_reviews
- products
- geolocation
- qualified_leads

Some examples of how we use it in our analysis:

- Sellers Lifetime Value analysis: Joining the customer dimension tables to fact tables such as order_items to find out which are the low, med, high and very high-value sellers.

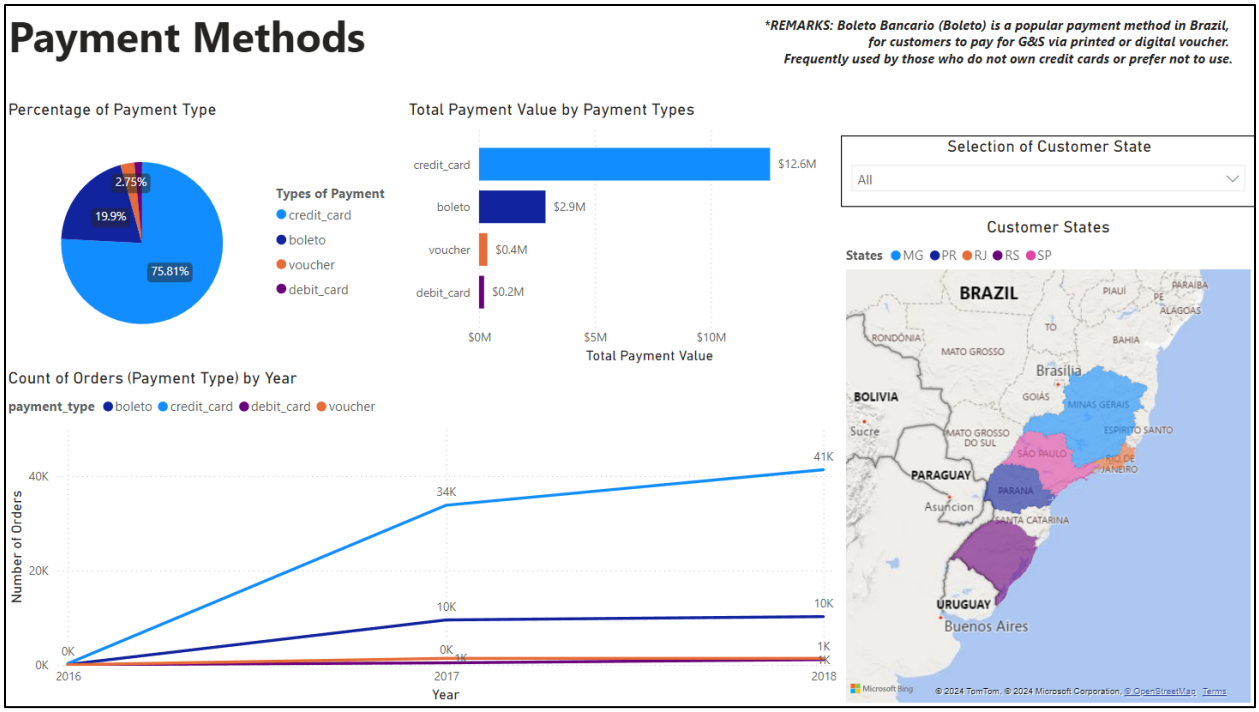
With those specifics in mind, we could draft out SQL scripts to initialize the tables and set the primary key and foreign key constraints, which can be found in Appendix: Section 2: SQL Scripts. Once SSMS populates it, the ERD will be automatically generated.

The result is an ERD with a star schema containing fact and dimension tables.


qualified_leads (dbo)
mql_id
first_contact_date
landing_page_id
origin
closed_deals (dbo)
mql_id
seller_id
sdr_id
sr_id
won_date
business_segment
lead_type
lead_behaviour_profile
has_company
has_gtin
average_stock
business_type
declared_product_catalog_size
declared_monthly_revenue
sellers (dbo)
seller_id
seller_zip_code_prefix
seller_city
seller_state
geolocation (dbo)
geolocation_zip_code_prefix
geolocation_lat
geolocation_lng
geolocation_city
geolocation_state
products (dbo)
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
order_items (dbo)
order_items_row_identifier
order_id
order_item_id
product_id
seller_id
shipping_limit_date
price
freight_value
orders (dbo)
order_id
customer_id
order_status
order_purchase_timestamp
order_approved_at
order_delivered_carrier_date
order_delivered_customer_date
order_estimated_delivery_date
customers (dbo)
customer_id
customer_unique_id
customer_zip_code_prefix
customer_city
customer_state
order_reviews (dbo)
review_id
order_id
review_score
review_comment_title
review_comment_message
review_creation_date
review_answer_timestamp
order_payments (dbo)
order_id
payment_installments
payment_value
payment_type

9. Analysis

Payment Methods

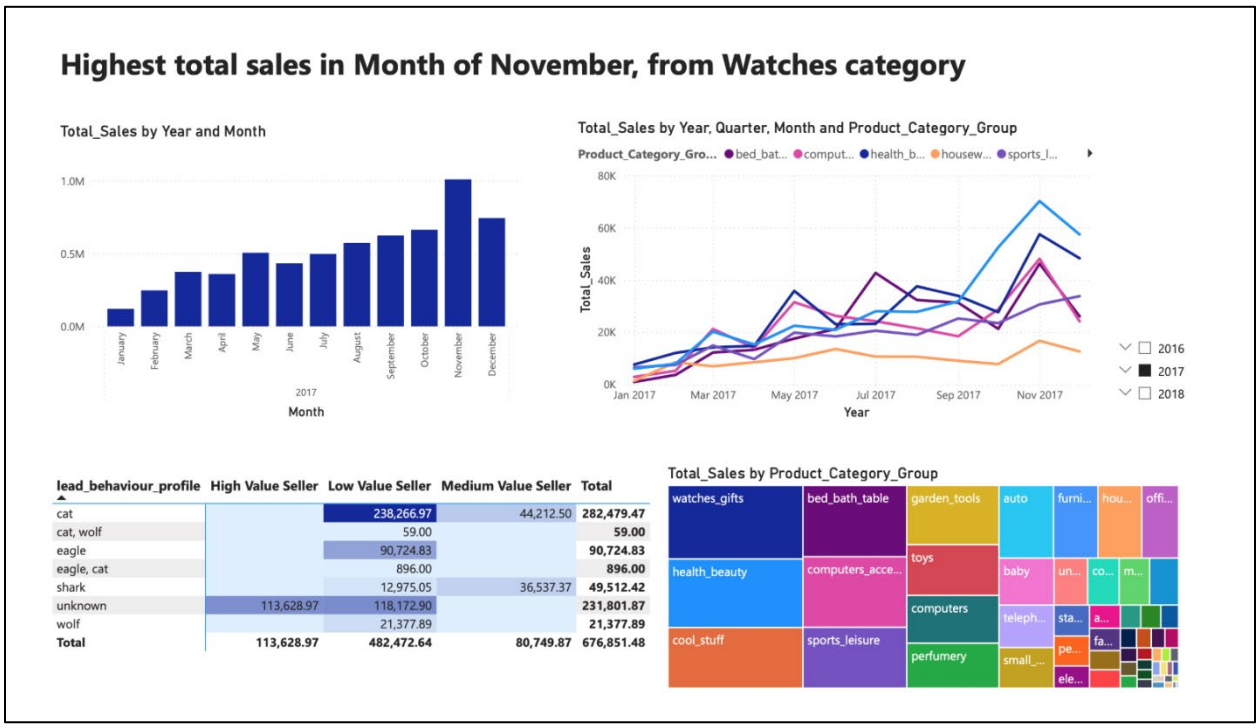


The analysis of payment methods on the Olist reveals significant insights into customer preferences and trends over time. For instance, the map of payment types by state highlights regional differences and helps Olist tailor its financial services accordingly to state needs.

The pie chart shows the relative share of different payment types, while the bar chart of total transacted amounts highlights which methods generate the most revenue. For example, credit cards may represent the highest share of the total transaction value, suggesting that customers prefer this payment type and are more willing to spend through it.

Additionally, a line chart depicting payment methods over time shows how customer preferences have evolved from 2016 to 2018. These trends provide valuable insights into shifting behaviours, which can help guide strategic decisions about expanding and promoting financial services. By leveraging these visual insights, Olist can improve its services, focusing on popular methods to enhance customer experience, expanding payment options where needed, and promoting the most effective payment solutions during key periods.

Transaction Periods



Note: The following analysis excludes 2016 and 2018 and is solely based on 2017, the only complete fiscal year in our dataset.

Bar chart

November is a key driver across multiple product categories, making Q4 Olist's best-performing quarter. This aligns with Brazil's consumer behaviour patterns, where the holiday season (Q4) has the highest spending across most market segments, particularly influenced by Black Friday. The overall upward trend suggests that some sellers are ready for **long-term business loans**.

Line chart, Tree map

In terms of product category, 'computer accessories' commands the highest average ticket throughout the year, and this, too, aligns with Brazil's upward trend of demand for electronic goods. Watches, health and beauty, bed bath tables, cool stuff, and computer accessories rank among the top five throughout the year as the highest total sales achieved.

- Olist can leverage these insights to forecast cash flow more accurately and ensure that sellers have the liquidity to stock inventory before peak periods, particularly around Black Friday. Secondly, by focusing on high-ticket items, Olist can identify segments with the greatest need for inventory financing.

- Olist can offer **short-term loans** or **seasonal credit** to a different segment: **the seasonal seller** and **niche market seller** who experience spikes in demand during the holiday season. These products can be tailored to meet the needs of sellers in specific categories like gifts and accessories.

Heat map

Olist profiles its sellers according to Segmenting and targeting leads based on their behaviour and characteristics, which helps Olist provide a customized set of tools for sellers to move the sales funnel. Additionally, aligning marketing and sales teams ensures a seamless lead transition between stages. Finally, following up and providing outstanding customer service (Olist to our sellers) provides feedback on the customization.

Olist's largest profile group is found in the intersection of the cat profile and low-value sellers segment, which accounts for the majority of Olist sales in 2016-2018. The 'Cat' profile for sellers points to the Communicator – or Conscientiousness (C) quadrant of DISC profiling, where individuals prioritize accuracy, are careful in decision-making, and are detail-oriented. The Lead Behavior implication is that sellers with a cat profile may be more cautious with their business decisions, lack foresight, and not like technical details.

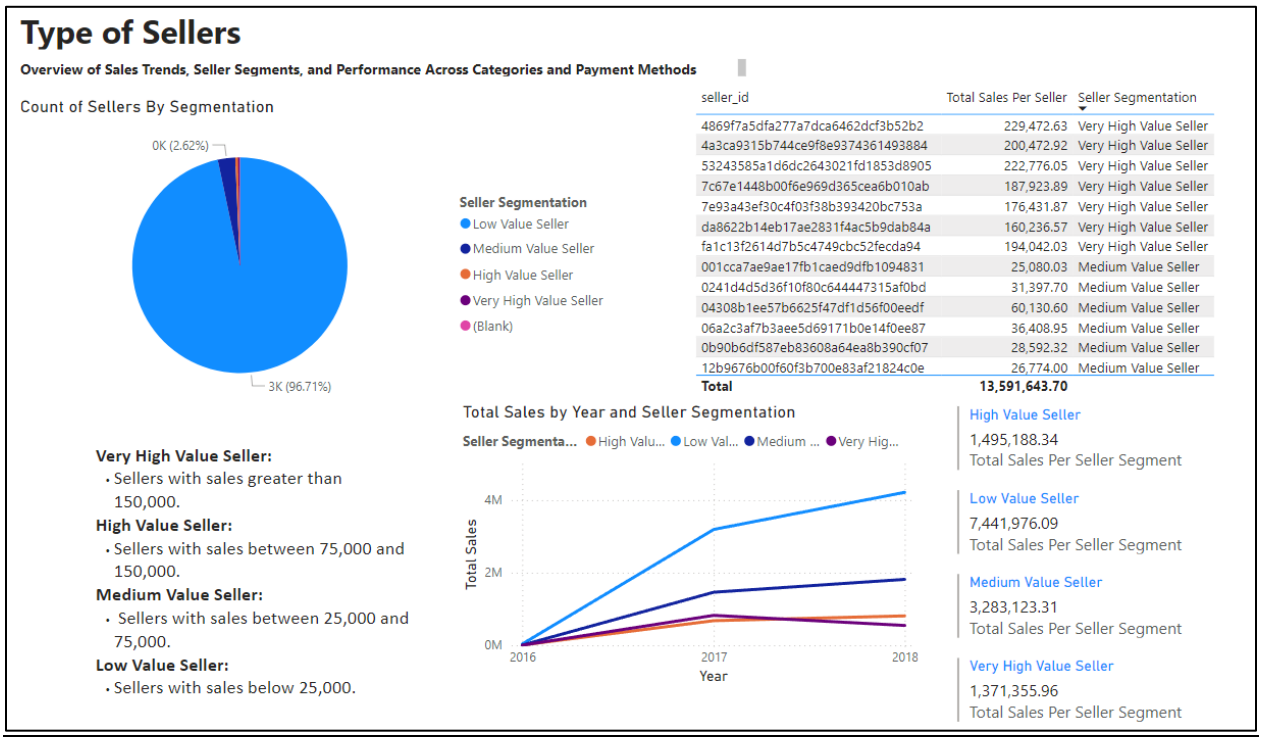
- This behaviour analysis can be used for marketing and how Olist may adapt its financial services to its biggest contributor, low-value sellers.

- 'Cat' may excel in categories that require precision or specialization (niche, high ticket)

- They might be risk-averse, leading them to **require low-risk, stable financial** options.

- A possibility to use predictive models to anticipate the future performance of sellers based on their behaviour profile, allowing Olist to offer more tailored financial products to intersectional segments.

Seller Value Analysis



1. Low-Value Sellers: The Backbone of Sales but Reliant on Traditional Payment Methods

Low-value sellers contribute significantly to overall sales, with over 7.44 million BRL in sales from 2016 to 2018. Despite being classified as "low value," they have the highest average order value (AOV) at 66.06 BRL, indicating a strong revenue potential within this segment.

However, these sellers are heavily reliant on the Boleto payment method, with over 1.31 million BRL in Boleto-based transactions. Their reliance on this method exposes them to security risks (such as Bolware fraud) and limits their ability to fully leverage modern payment technologies like PIX or credit cards.

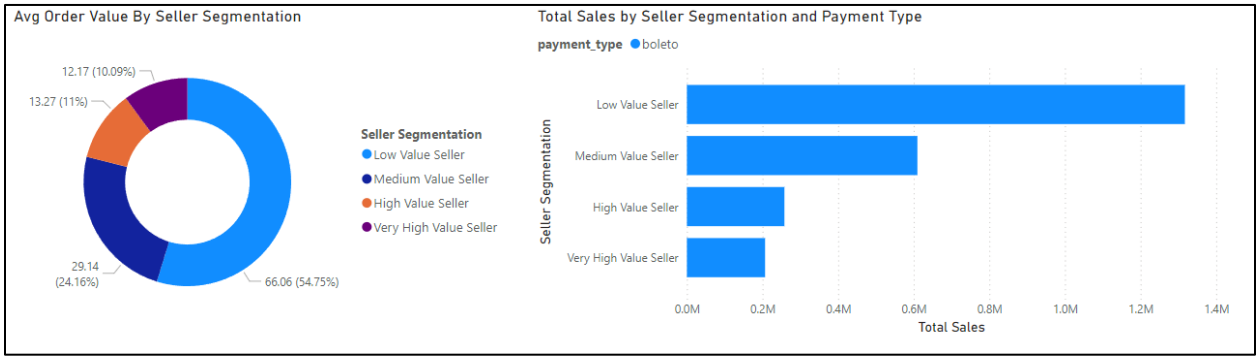
The over-reliance on Boleto may stem from the late adoption of modern payment methods due to financial barriers or lack of access to credit facilities. Gamification strategies and financial incentives (e.g., reduced fees and rewards for using PIX or credit cards) could encourage these sellers to transition away from Boleto, reducing their exposure to fraud and improving their financial flexibility.

2. Opportunities for High and Very High-Value Sellers to Increase AOV

While contributing significantly to total sales, high and very high-value sellers have the lowest AOV (13.27 BRL and 12.17 BRL, respectively). This suggests that these segments are generating large volumes of low-value transactions, indicating the potential to increase order size through upselling, cross-selling, or bundling strategies.

These sellers rely less on Boleto, suggesting they are better positioned to adopt modern payment systems and increase their operational efficiency.

High- and very-high-value sellers could focus on improving their AOV through targeted marketing strategies that promote larger orders. Encouraging these sellers to expand their product offerings or implement premium features could further enhance their revenue.



3. Payment Method Preferences: Transitioning to Modern Payment Solutions

The heavy reliance on Boleto among low—and medium-value sellers highlights the need for greater adoption of secure, modern payment methods like PIX and credit cards. These methods provide faster transaction processing, enhanced security, and better fraud protection.

PIX, with its growing popularity in Brazil, offers an opportunity to move sellers away from Boleto. It provides them with instant payments and reduces their exposure to fraud risks like Bolware.

A gradual shift to modern payment methods, driven by gamification strategies (e.g., cashback rewards, lower fees, loyalty points), will improve both the financial security and growth potential of low-value sellers. This shift can reduce their vulnerability to fraud and enhance overall sales performance.

product_category_name	High Value Seller	Low Value Seller	Medium Value Seller	Very High Value Seller	Total
health_beauty	197,208.46	588,048.33	460,216.31	13,208.24	**1,258,681.34**
watches_gifts	192,658.67	240,662.13	208,752.35	562,932.53	**1,205,005.68**
bed_bath_table	1,045.66	511,240.14	207,175.83	317,527.05	**1,036,988.68**
sports_leisure	31,369.33	657,839.70	293,816.10	5,023.84	**988,048.97**
computers_accessories	49,754.38	517,507.75	344,132.19	560.00	**911,954.32**
furniture_decor	150,399.35	410,759.77	151,843.89	16,759.48	**729,762.49**
cool_stuff	154,609.62	258,631.81	211,133.96	10,915.46	**635,290.85**
housewares	30,844.00	538,121.61	63,283.05		**632,248.66**
auto	1,259.99	413,819.92	177,030.45	609.75	**592,720.11**
garden_tools	167,922.69	248,967.37	68,366.40		**485,256.46**
toys	93,566.75	297,688.43	89,400.04	3,291.38	**483,946.60**
baby	97,205.15	275,882.69	37,782.73	894.32	**411,764.89**
perfumery	113,362.63	170,628.97	115,133.27		**399,124.87**
telephony	223.00	215,624.62	50,867.81	56,952.10	**323,667.53**
Total	**1,495,188.34**	**7,441,976.09**	**3,283,123.31**	**1,371,355.96**	**13,591,643.70**

Average Sales Per Seller Segment	Seller Segmentation
115,014.49	High Value Seller
2,485.63	Low Value Seller
40,532.39	Medium Value Seller
195,907.99	Very High Value Seller
4,390.07	

4. Best Performing Product Categories: Key Drivers of Revenue

The top-performing product categories—health and Beauty, Watches and Gifts, and Bed, Bath, and Table—have driven significant revenue across all seller segments, with sales exceeding 1 million BRL in each category.

These categories represent significant growth opportunities for all seller segments. Selling can further increase their sales potential by focusing on these high-revenue categories and expanding product offerings. Additionally, encouraging cross-selling and bundling strategies within these categories can enhance the AOV across all segments.

5. Strategic Focus for Seller Segmentation and Sales Trends

The overall analysis highlights that low-value sellers dominate in total sales, but their reliance on Boleto could be holding them back from achieving even greater growth. Transitioning them to modern payment systems will be crucial to unlocking their full potential.

Medium-value sellers show promise and could be strategically supported to transition into the high-value segment with the right combination of marketing tools, product diversification, and access to financial incentives.

High and Very High-Value Sellers have room to improve by increasing their AOV and leveraging premium products and offers to maximize profitability.

Olist can further their offerings for their sellers through the following:

1. Upselling and Bundling for High and Very High-Value Sellers: These sellers have low AOVs despite their volume of transactions. By offering premium products, personalized

marketing strategies, and bundling, they can increase their average transaction value and contribute even more to total sales.

2. Support for Low and Medium Value Sellers: These segments, particularly low-value sellers, need financial and operational support to adopt modern payment methods. Gamification strategies, targeted education, and financial incentives will encourage a gradual shift away from Boleto, improving its efficiency and security.
3. Leverage Top Product Categories: The top-performing categories should remain a focus for growth. Expanding product offerings, improving customer experience, and leveraging cross-selling opportunities within these categories will continue driving revenue for all seller segments.
4. Payment Solutions: Expanding payment options beyond Boleto is critical to improving the performance and security of low and medium-value sellers. PIX and credit cards offer safer, faster, and more reliable payment solutions, and providing sellers with incentives to adopt these methods will enhance their overall performance.

10. Challenges/Limitations

There were many unique challenges when it came to the overall completeness of the datasets provided by Olist. Besides the common data quality problems such as handling NULL, NaN or empty fields, there are many other issues that our project has to tackle. Some of the notable ones include:

Issue #1: Duplicated Values but Only Selected Columns

The issue with how Olist collected its data was that multiple instances of order_id or customer_id can be repeated in certain tables, such as customers or order_items. However, they are not exactly duplicates because a secondary column tracks or acts as a counter for some specific reasons.

For instance, in the orders table, there can be multiple similar order_ids; however, the secondary column customer_id can be unique, and vice versa. So when these two columns are combined, they are unique, which is akin to a surrogate key.

Issue #2: Duplicated Values Due to Accented Characters

Another issue with the data is pertaining to accented characters which stems from the cities and states of tables such as sellers, customers and geolocation. For example, **são paulo** and **sao paulo** are both considered unique values.

Issue #3: Duplicated Values of Geolocation Zip Codes

Currently, the default geolocation table has geolocation_zip_code_prefix intended to serve as a primary key. However, as seen from the initial data (e.g., index 1 and 2), there are multiple occurrences of the same zip_code_prefix value, such as 1046.

	geolocation_zip_code_prefix	geolocation_lat	geolocation_lng	geolocation_city	geolocation_state
0	1037	-23.545621	-46.639292	sao paulo	SP
1	1046	-23.546081	-46.644820	sao paulo	SP
2	1046	-23.546129	-46.642951	sao paulo	SP
3	1041	-23.544392	-46.639499	sao paulo	SP
4	1035	-23.541578	-46.641607	sao paulo	SP

These repeated values indicate that geolocation_zip_code_prefix is not truly unique by itself. In fact, the column only becomes unique when combined with geolocation_lat and geolocation_lng (latitude and longitude).

Issue #4: Duplicated Order IDs within Order Items Dataframe

The current issue with the order_items table is related to the presence of 13,984 rows with duplicated order_id values.

	order_id	order_item_id	product_id	seller_id	shipping_limit_date	price	freig
0	00010242fe8c5a6d1ba2dd792cb16214	1	4244733e06e7ecb4970a6e2683c13e61	48436dade18ac8b2bce089ec2a041202	2017-09-19 09:45:35	58.90	
1	00018f77f2f0320c557190d7a144bdd3	1	e5f2d52b802189ee658865ca93d83a8f	dd7ddc04e1b6c2c614352b383efe2d36	2017-05-03 11:05:13	239.90	
2	000229ec398224ef6ca0657da4fc703e	1	c777355d18b72b67abbeef9df44fd0fd	5b51032eddd242adc84c38acab88f23d	2018-01-18 14:48:30	199.00	
3	00024acbcdf0a6daa1e931b038114c75	1	7634da152a4610f1595efa32f14722fc	9d7a1d34a5052409006425275ba1c2b4	2018-08-15 10:10:18	12.99	
4	00042b26cf59d7ce69dfabb4e55b4fd9	1	ac6c3623068f30de03045865e4e10089	df560393f3a51e74553ab94004ba5c87	2017-02-13 13:57:51	199.90	

```
# Count duplicates in specific columns
duplicate_counter = order_items.duplicated(subset=['order_id']).sum()
print(f"Number of duplicate rows: {duplicate_counter}")
```

```
Number of duplicate rows: 13984
```

However, these rows are not true duplicates. Instead, the order_item_id increments by 1 each time an order_id repeats, which indicates that Olist uses this pattern to track the individual items within a single order.

Issue #5: Duplicated Order IDs within Order Payments Dataframe

The current issue with the order_payments table is similar to the one discussed previously. Multiple instances of the same order_id exist, but these rows are not duplicates. Instead, Olist uses this method to record how each order is paid. For example, if a credit card pays an order in two installments, the payment_sequential column will have values 1 and 2 for the same order_id.

```
order_payments.head()
```

	order_id	payment_sequential	payment_type	payment_installments	payment_value
0	b81ef226f3fe1789b1e8b2acac839d17	1	credit_card	8	99.33
1	a9810da82917af2d9aefd1278f1dcfa0	1	credit_card	1	24.39
2	25e8ea4e93396b6fa0d3dd708e76c1bd	1	credit_card	1	65.71
3	ba78997921bbcdc1373bb41e913ab953	1	credit_card	8	107.78
4	42fdf880ba16b47b59251dd489d4441a	1	credit_card	2	128.45

```
# Count duplicates in specific columns
duplicate_counter = order_payments.duplicated(subset=['order_id']).sum()
print(f"Number of duplicate rows: {duplicate_counter}")
```

```
Number of duplicate rows: 4446
```

Issue #6: Duplicated Review IDs within Order Reviews Dataframe

The current issue with the order_reviews table is similar to the previous sections. Multiple instances of the same order_id exist, but these rows are not true duplicates. This is how Olist records reviews for orders, resulting in apparent duplicates.

```
order_reviews.head()
```

	review_id	order_id	review_score	review_comment_title	review_comment_message	review_creation_date
0	7bc2406110b926393aa56f80a40eba40	73fc7af87114b39712e6da79b0a377eb	4	No Title	No Review Provided	2018-01-18 00:00:00
1	80e641a11e56f04c1ad469d5645fdfde	a548910a1c6147796b98fdf73dbeba33	5	No Title	No Review Provided	2018-03-10 00:00:00
2	228ce5500dc1d8e020d8d1322874b6f0	f9e4b658b201a9f2ecdecbb34bed034b	5	No Title	No Review Provided	2018-02-17 00:00:00
3	e64fb393e7b32834bb789ff8bb30750e	658677c97b385a9be170737859d3511b	5	No Title	Recebi bem antes do prazo estipulado.	2017-04-21 00:00:00
4	f7c4243c7fe1938f181bec41a392bdeb	8e6bfb81e283fa7e4f11123a3fb894f1	5	No Title	Parabéns lojas lannister adorei comprar pela I...	2018-03-01 00:00:00

```
# Count duplicates in specific columns
duplicate_counter = order_reviews.duplicated(subset=['review_id']).sum()
print(f"Number of duplicate rows: {duplicate_counter}")
```

```
Number of duplicate rows: 814
```

```
# Count duplicates in specific columns
duplicate_counter = order_reviews.duplicated(subset=['order_id']).sum()
print(f"Number of duplicate rows: {duplicate_counter}")
```

```
Number of duplicate rows: 551
```

For example, an order_id might have multiple review_id values if there are several items within the order, each with its own review. Alternatively, a single review_id might be associated with multiple order_id values if one review is given for multiple orders.

Issue #7: Geolocation Dataframe Does Not Match Customers/Sellers for Referential Integrity

The issue is that customer and seller zip codes are not in the geolocation table. This leads to referencing errors when attempting to set up foreign keys pointing to the geolocation table. To fix this, we need to address the missing zip codes in order to ensure referential integrity between the geolocation table and the customers and sellers tables; additional steps are required to match records.

11. Solutions

Solution #1: Handling Duplicated Values for Selected Columns

The solution is to write a custom user-defined function that checks for the desired combinations.

```
def check_drop_duplicates(df, columns):
    """
    Check for duplicate combinations based on specific columns,
    count them, and drop duplicates if found.

    Parameters:
    df (pd.DataFrame): The dataframe to check for duplicates.
    columns (list): List of columns to check for duplicate combinations.

    Returns:
    pd.DataFrame: Dataframe with duplicates dropped, if found.
    """
    # Check for duplicated combinations
    duplicates = df[df.duplicated(subset=columns, keep=False)]

    if not duplicates.empty:
        # Count the number of duplicated combinations
        duplicate_count = duplicates.shape[0]
        print(f"{duplicate_count} duplicated combinations found in columns: {columns}")

        # Drop duplicates and keep the first occurrence
        df_deduped = df.drop_duplicates(subset=columns)
        print(f"{duplicate_count} duplicated combinations dropped.")
        return df_deduped
    else:
        print("No duplicated combinations were found.")
        return df
```

The Python code above takes in the table that we are checking and the columns we want to specifically check for. So in the case of the issue we highlighted earlier, we can pass the orders dataframe and columns order_id and customer_id. And see if the resulting output reflects any true duplicated values or not. This user-defined function is applied across other dataframes such as: customers, sellers, order_items, order_payments, order_reviews, products, closed_deals and qualified_leads.

Solution #2: Handling Accented Characters

The solution is to use the unicodedata Python module and a customer user-defined function that holds different functions inside to first check for specific columns of a dataframe and then check within the column itself for accented characters.

```
import unicodedata

def normalize_accented_characters(text):
    """
    Normalize accented characters in a string to their non-accented form.

    Parameters:
    text (str): The text to normalize.

    Returns:
    str: The normalized text.
    """
    # Normalize to decomposed form and then remove accents
    text = unicodedata.normalize('NFD', text)
    text = ''.join(char for char in text if not unicodedata.combining(char))
    return unicodedata.normalize('NFC', text)

def find_remaining_accented_characters(text):
    """
    Identify any remaining accented characters in a string.

    Parameters:
    text (str): The text to check.

    Returns:
    str: The string containing any remaining accented characters.
    """
    return ''.join(char for char in text if unicodedata.combining(char))

def normalize_column(df, column_name):
    """
    Apply normalization to a specified column in a dataframe and ensure no accented characters remain.

    Parameters:
    df (pd.DataFrame): The dataframe containing the column.
    column_name (str): The name of the column to normalize.

    Returns:
    pd.DataFrame: The dataframe with the specified column normalized.
    """
    if column_name in df.columns:
        # Normalize the column
        df[column_name] = df[column_name].astype(str).apply(normalize_accented_characters)

        # Check for any remaining accented characters
        remaining_accented = df[column_name].apply(find_remaining_accented_characters)

        # Print remaining accented characters if any
        if remaining_accented.str.len().sum() > 0:
            print("Remaining accented characters found:")
            print(remaining_accented[remaining_accented.str.len() > 0].drop_duplicates())
        else:
            print("No remaining accented characters found.")

        return df
    else:
        raise ValueError(f"Column '{column_name}' does not exist in the dataframe.")
```

Each character has two types of normal forms: a normal form C and a normal form D (reference). By applying NFD, we decompose the accented characters to their normal characters, e.g., accented a to a normal a. A simple one line of code to iterate through each character of the decomposed string and leave only the base character – unicodedata.combining(char) helps to return a non-zero value for combining characters such as an accented a, so it filters them out. Finally, applying NFC helps to recompose the text into its canonical form.

Solution #3: Fix Duplicated Values of Geolocation Zip Codes

To resolve this issue, one possible approach is to group the data by zip_code_prefix and then calculate each group's average latitude and longitude values. This solution works well because the exact coordinates are not crucial for our analysis.

```
def group_by_zipcode_and_average_lat_lng(df):
    """
    Groups the geolocation dataframe by zip_code_prefix and calculates the average latitude and longitude.
    Assumes that city and state are the same for a given zip_code_prefix and retains the first occurrence of each.

    Parameters:
    df (DataFrame): The input pandas dataframe containing geolocation data.

    Returns:
    DataFrame: A new dataframe with unique zip_code_prefix and the averaged lat/lng.
    """
    grouped_df = df.groupby('geolocation_zip_code_prefix').agg({
        'geolocation_lat': 'mean',
        'geolocation_lng': 'mean',
        'geolocation_city': 'first',  # Retain the first city for each zip_code_prefix
        'geolocation_state': 'first' # Both assume state & city are the same for the same prefix
    }).reset_index() # Reset index to flatten the dataframe

    return grouped_df
```

Averaging the coordinates simplifies the data while preserving a unique zip_code_prefix that can still identify a general location. This method maintains enough location information to be useful while ensuring that each zip_code_prefix remains distinct.

Solution #4: Fixing Duplicated Order IDs with Unique Role Identifier

To address this, a potential solution is to create a new column that serves as a unique identifier for each row.

```
def add_unique_row_identifier(df, pk_name='order_items_row_identifier'):
    """
    Adds a unique row identifier acting as a primary key column to a DataFrame.

    Parameters:
    df (pd.DataFrame): The DataFrame to which the row identifier will be added.
    pk_name (str): The name of the unique row idnetifier column (default is 'order_items_row_identifier').

    Returns:
    pd.DataFrame: The DataFrame with the new unique row identifier column.
    """
    # Create the surrogate key by assigning a unique auto-incrementing integer to each row
    df[pk_name] = range(1, len(df) + 1)

    # Reorder the columns to put the unique row identifier at the beginning
    cols = [pk_name] + [col for col in df.columns if col != pk_name]
    df = df[cols]

    df[pk_name] = df[pk_name].astype('int64')

    return df
```

```
order_items = add_unique_row_identifier(order_items)
order_items.head()
```

	order_items_row_identifier	order_id	order_item_id	product_id	seller_id	shippi
0	1	00010242fe8c5a6d1ba2dd792cb16214	1	4244733e06e7ecb4970a6e2683c13e61	48436dade18ac8b2bce089ec2a041202	2017-0
1	2	00018f77f2f0320c557190d7a144bdd3	1	e5f2d52b802189ee658865ca93d83a8f	dd7ddc04e1b6c2c614352b383efe2d36	2017-0
2	3	000229ec398224ef6ca0657da4fc703e	1	c777355d18b72b67abbeef9df44fd0fd	5b51032eddd242adc84c38acab88f23d	2018-0
3	4	00024acbcdf0a6daa1e931b038114c75	1	7634da152a4610f1595efa32f14722fc	9d7a1d34a5052409006425275ba1c2b4	2018-0
4	5	00042b26cf59d7ce69dfabb4e55b4fd9	1	ac6c3623068f30de03045865e4e10089	df560393f3a51e74553ab94004ba5c87	2017-0

This way, we can uniquely identify every item in an order, ensuring that the data structure accurately reflects each product within a customer's purchase and, more importantly, still be able to use order_id as a foreign key referencing order_id in the orders dataframe/table.

Solution #5: Addressing Duplicated Order IDs within the Order Payments Dataframe

To address this, we can group the data by order_id to ensure each row is unique. Once grouped, we can aggregate the payment_value column to get the total payment amount for each order_id. Regarding payment_installments, we can simplify it by creating a binary indicator: if the number of installments is greater than 2, we set it to TRUE; otherwise, we set it to FALSE. This transformation allows us to keep track of whether the payment was made in installments without focusing on the exact number. Finally, since we are grouping by order_id, the payment_sequential column will be dropped, as it will no longer be needed.

```
# Function to clean and aggregate the 'order_payments' DataFrame
def clean_order_payments(df):
    # Group by 'order_id' and aggregate sum of 'payment_value'
    grouped_df = df.groupby('order_id', as_index=False).agg({
        'payment_value': 'sum',
        'payment_installments': lambda x: any(x >= 2),  # Check if any instalment is 2 or more
        'payment_type': 'first'  # Keep first payment_type
    })

    # Convert 'payment_instalments' to boolean (TRUE for >=2, FALSE otherwise)
    grouped_df['payment_installments'] = grouped_df['payment_installments'].astype(bool)

    # Drop 'payment_sequential' since it's no longer needed
    # This step is unnecessary because 'payment_sequential' was dropped during the grouping

    return grouped_df
```

```
# store the order_payments in another dataframe for another analysis in this notebook
original_order_payments = order_payments
original_order_payments.head(2)
```

	order_id	payment_sequential	payment_type	payment_installments	payment_value
0	b81ef226f3fe1789b1e8b2acac839d17	1	credit_card	8	99.33
1	a9810da82917af2d9aefd1278f1dcfa0	1	credit_card	1	24.39

To further elaborate on the line 'payment_type': 'first': This means that, during aggregation, we keep the first payment_type encountered for each order_id. Essentially, we assume that the first payment_type recorded for an order represents the entire order's payment method. This simplification assumes consistency, where the initial payment method is used for all subsequent payments for that order.

Solution #6: Fixing Duplicated Review IDs within the Order Reviews Dataframe

```
def clean_order_reviews(df):
    # Group by 'review_id'
    cleaned_df = df.groupby('review_id').agg({
        'order_id': 'last',  # Keep the last order_id associated with each review_id
        'review_score': 'mean',  # Average the review scores
        'review_comment_title': 'last',  # Take the latest entry for titles
        'review_comment_message': 'last',  # Take the latest entry for messages
        'review_creation_date': 'last',  # Take the latest creation date
        'review_answer_timestamp': 'last'  # Take the latest answer timestamp
    }).reset_index()

    # and then 'order_id'
    cleaned_df = cleaned_df.groupby('order_id').agg({
        'review_id': 'first',  # Take the first review_id for the unique order_id
        'review_score': 'mean',  # Average review_score (could be adjusted based on your needs)
        'review_comment_title': 'last',  # Take the latest review_comment_title
        'review_comment_message': 'last',  # Take the latest review_comment_message
        'review_creation_date': 'last',  # Take the latest review_creation_date
        'review_answer_timestamp': 'last'  # Take the latest review_answer_timestamp
    }).reset_index()

    # Revert review_score back to int64
    cleaned_df['review_score'] = cleaned_df['review_score'].astype('int64')

    return cleaned_df
```

```
order_reviews = clean_order_reviews(order_reviews)
```

```
# Count duplicates in specific columns
duplicate_counter = order_reviews.duplicated(subset=['review_id']).sum()
print(f"Number of duplicate rows: {duplicate_counter}")
```

```
Number of duplicate rows: 0
```

```
# Count duplicates in specific columns
duplicate_counter = order_reviews.duplicated(subset=['order_id']).sum()
print(f"Number of duplicate rows: {duplicate_counter}")
```

```
Number of duplicate rows: 0
```

To resolve this, we can start by grouping the data by review_id, ensuring each row is unique by review. After that, we group by order_id to further consolidate the data. When aggregating, we take the mean value for review_score to provide an average rating for each order. For text-based columns like review_comment_title and review_comment_message, we retain the latest entry, assuming that the most recent review is the most relevant. Similarly, for columns related to dates (review_creation_date and review_answer_timestamp), we keep the latest entry, which should capture the most updated information regarding the review process.

Solution #8: Create Additional Records in the Geolocation Dataframe to Match Tables

```
# Select the distinct customer_zip_code_prefix values that are not in geolocation_zip_code_prefix
distinct_customer_zip_prefix = customers[
    ~customers['customer_zip_code_prefix'].isin(geolocation['geolocation_zip_code_prefix'])
]['customer_zip_code_prefix'].drop_duplicates()

# Display the result
print(distinct_customer_zip_prefix)
```

First, we can detect distinct zip codes that exist in the customers and sellers tables but do not exist in the geolocation table. Using a custom function, we create a new dataframe to handle these missing entries. We append the new zip codes as the first column while setting the latitude and longitude values to NaN and their float values to 0.00. We also set the city and state columns to "Unknown".

```
def insert_and_populate_geolocation(customers, geolocation):

    # Step 1: Find distinct zip codes not in geolocation
    new_zipcodes = customers[
        ~customers['customer_zip_code_prefix'].isin(geolocation['geolocation_zip_code_prefix'])
    ]['customer_zip_code_prefix'].drop_duplicates()

    # Step 2: Create a DataFrame with new zip codes
    new_entries = pd.DataFrame({
        'geolocation_zip_code_prefix': new_zipcodes,
        'geolocation_lat': np.nan,
        'geolocation_lng': np.nan,
        'geolocation_city': 'Unknown',
        'geolocation_state': 'Unknown'
    })

    # Step 3: Append new entries
    geolocation = pd.concat([geolocation, new_entries], ignore_index=True)

    # Step 4: Set missing lat and lng to 0.00 and ensure float64 format
    geolocation['geolocation_lat'] = geolocation['geolocation_lat'].fillna(0.00).astype('float64')
    geolocation['geolocation_lng'] = geolocation['geolocation_lng'].fillna(0.00).astype('float64')

    return geolocation
```

Adding these placeholder records to the geolocation table ensures that every unique customer or seller zip code can be found in the geolocation table's primary key column. This allows us to establish referential integrity within the database without errors. The same is applied to the seller's dataframe.

Furthermore, even though the added entries use placeholder values (0.00 for latitude/longitude and "Unknown" for city/state), we can still easily filter them out during analysis. For example, when visualizing data in Power BI, we can use page filters or similar techniques to exclude these placeholder entries, ensuring accurate and meaningful insights.

12. Conclusion

The data indicates a strong customer preference for credit card payments, with approximately 12 million Brazilian Reals transacted, compared to 2.9 million Brazilian Reals via boleto, reflecting a significant segment of customers who prefer cash-based alternatives or lack access to credit. To leverage these insights, Olist can implement several strategies, such as **optimising the infrastructure** by working with local banks to reduce processing times for credit cards, **reducing credit card fees** to enhance appeal for both sellers and customers and increasing transaction volume and profit margins. And offering **in-house banking services** such as cash deposit options at designated locations can provide immediate fund availability on the Olist platform, catering to underbanked customers and reducing reliance on boleto. This approach also ensures quicker seller access to funds and simplifies payment reversals for returns or disputes.

Olist can leverage these insights to forecast cash flow more accurately and ensure that sellers have the liquidity to stock inventory ahead of peak periods, particularly around Black Friday and Easter. Offering interest-free periods or flexible repayment plans during high-sales months would give these sellers a financial boost to capture increased holiday demand.

Additionally, Olist can offer short-term loans or seasonal credit to niche and seasonal sellers who experience spikes in demand during the holiday season. These products can be tailored to meet the needs of sellers in specific categories, such as gifts and accessories.

The Cat behaviour profile will likely be risk-averse and will benefit from low-risk financial products focusing on stable growth. These products would give them access to financial resources that match their conservative business approach. These tailored offerings would enhance Olist's value proposition, fostering long-term partnerships with its diverse seller base. Olist can also develop predictive models to anticipate seller performance based on their behaviour profile, which will allow the platform to offer more tailored financial solutions.

The Seller Segmentation analysis underscores the importance of strategic initiatives aimed at boosting the performance of sellers across all segments. Low-value sellers, in particular, stand to benefit greatly from adopting modern payment methods like PIX and credit cards, which will reduce their exposure to fraud and improve their financial flexibility. High and very high-value sellers can increase their profitability by encouraging upselling, bundling, and expanding into top-

performing product categories. With the right support and incentives, all seller segments can achieve sustainable growth, driving overall sales and operational efficiency improvements.

13. Key Takeaways

One significant takeaway from this project is learning to balance cost and functionality when working on an ETL project using Azure Cloud. Azure Cloud provides powerful scalability, accessibility, and integration with various services.

However, these advantages come with costs that can quickly add up, especially if resources are not efficiently managed or workflows run longer than necessary. In contrast, a local ETL deployment might involve upfront hardware costs but typically offers better cost predictability in the long run. Weighing the flexibility and scaling capabilities of the cloud against the potentially high costs is an ongoing challenge for developers, especially for personal projects or smaller-scale applications such as in the case of working with Olist data.

Another challenge is debugging ETL workflows in a cloud environment. Azure Cloud provides tools for debugging, but the cost implications of running tests or troubleshooting issues in the cloud can be a limiting factor. Unlike local ETL development, where one can debug freely without incurring extra charges, debugging in the cloud can lead to higher compute costs, particularly if multiple iterations are needed to fix an issue.

A potential improvement for our final project could be deploying Apache Airflow locally. This change could lead to significant cost savings by avoiding the variable costs associated with cloud resources, such as storage, data transfer, and computing charges. A local Airflow deployment offers a predictable cost structure, which is especially beneficial during frequent debugging or development phases. While a local deployment may require initial hardware and setup investments, it can be easily overseen for this project as Olist is not a huge dataset.

14. References

[1] Pymnts, “Brazil’s Olist Expands Banking Services for Retailer Clients,” *PYMNTS.com*, May 29, 2024. https://www.pymnts.com/news/banking/2024/brazils-olist-expands-banking-services-for-retailer-clients/

15. Appendix

Section 1. Tables

- PK = Primary Key
- FK = Foreign Key

The tables are loaded in Microsoft Azure SQL Database along with appropriate data types (dtype).

Table 1: geolocation

Column name	dtype	Description
geolocation_zip_code_prefix	INTEGER	PK
geolocation_lat	FLOAT	Latitude
geolocation_lng	FLOAT	Longitude
geolocation_city	VARCHAR(8000)	City
geolocation_state	VARCHAR(8000)	State

Table 2: customers

Column name	dtype	Description
customer_id	NVARCHAR(450)	PK
customer_unique_id	VARCHAR(8000)	Unique string identifying customer
customer_zip_code_prefix	INTEGER	FK – to geolocation table's PK
customer_city	VARCHAR(8000)	City
customer_state	VARCHAR(8000)	State

Table 3: sellers

Column name	dtype	Description
seller_id	NVARCHAR(450)	PK
seller_zip_code_prefix	INTEGER	FK – to geolocation table's PK
seller _city	VARCHAR(8000)	City
seller _state	VARCHAR(8000)	State

Table 4: orders

Column name	dtype	Description
order_id	NVARCHAR(450)	PK
customer_id	NVARCHAR(450)	FK – to customers's PK
order_status	VARCHAR(8000)	Status of the order
order_purchase_timestamp	DATETIME2(3)	Timestamp for purchase (YYYY-MM-DD HH:MM:SS.fff)
order_approved_at	DATETIME2(3)	Timestamp for order approval (YYYY-MM-DD HH:MM:SS.fff)
order_delivered_carrier_date	DATETIME2(3)	Timestamp for courier delivered date (YYYY-MM-DD HH:MM:SS.fff)
order_delivered_customer_date	DATETIME2(3)	Timestamp for customer received date (YYYY-MM-DD HH:MM:SS.fff)
order_estimated_delivery_date	DATE	Estimated delivery date (YYYY-MM-DD)

Table 5: products

Column name	dtype	Description
product_id	NVARCHAR(450)	PK
product_category_name	VARCHAR(8000)	Product category names
product_name_length	FLOAT	Name of the products
product_description_length	FLOAT	Description of the products
product_photos_qty	FLOAT	Number of photos for the product
product_weight_g	FLOAT	Weight in grams
product_length_cm	FLOAT	Length of product in centre metres
product_height_cm	FLOAT	Height of product in centre metres
product_width_cm	FLOAT	Width of product in centre metres

Table 6: order_items

Column name	dtype	Description
order_items_row_identifier	INTEGER	PRIMARY KEY
order_id	NVARCHAR(450)	FK referencing customer's PK
order_item_id	INTEGER	Sequential counter for items in an order
product_id	NVARCHAR(450)	FK referencing products table's PK
seller_id	NVARCHAR(450)	FK referencing sellers table's PK
shipping_limit_date	DATETIME(2)	Timestamp for shipping date (YYYY-MM-DD HH:MM:SS.fff)
price	FLOAT	The price of each item within an order
freight_value	FLOAT	Freight value

Table 7: order_payments

Column name	dtype	Description
order_id	NVARCHAR(450)	PK/FK referencing orders table's PK
payment_installments	BIT	TRUE/FALSE if an order has instalments
payment_value	FLOAT	Total payment value of the order
payment_type	VARCHAR(8000)	Type of the payment

Table 8: order_reviews

Column name	dtype	Description
review_id	NVARCHAR(450)	PK
order_id	NVARCHAR(450)	FK referencing orders table PK
review_score	INTEGER	The score for the review
review_comment_title	VARCHAR(8000)	Title of the review
review_comment_message	VARCHAR(8000)	Message within the review
review_creation_date	DATETIME(2)	Timestamp for shipping date (YYYY-MM-DD HH:MM:SS.fff)
review_answer_timestamp	DATETIME(2)	Timestamp for shipping date (YYYY-MM-DD HH:MM:SS.fff)

Table 9: qualified_leads

Column name	dtype	Description
mql_id	NVARCHAR(450)	PK
first_contact_date	DATE	The first date from contacting the lead
landing_page_id	VARCHAR(8000)	The ID of the landing page for the lead
origin	VARCHAR(8000)	Origin of the lead under which campaign

Table 10: closed_deals

Column name	dtype	Description
mql_id	NVARCHAR(450)	PK
seller_id	NVARCHAR(450)	FK referencing sellers table PK
sdr_id	VARCHAR(8000)	Sales development representative ID
sr_id	VARCHAR(8000)	Sales representative ID
won_date	DATETIME2(3)	Timestamp for date lead was acquired (YYYY-MM-DD HH:MM:SS.fff)
business_segment	VARCHAR(8000)	Segment of business
lead_type	VARCHAR(8000)	Type of lead

lead_behaviour_profile	VARCHAR(8000)	The behaviour profile of lead (personality)
has_company	BIT	Whether the lead has a company
has_gtin	BIT	Whether the lead has a GTIN
average_stock	VARCHAR(8000)	The stock size
business_type	VARCHAR(8000)	Business type of the lead
declared_product_catalog_size	FLOAT	Product catalogue size of the lead
declared_monthly_revenue	FLOAT	Monthly revenue earned by lead

Section 2. SQL Scripts

For initialising OLIST Tables:

```
CREATE TABLE geolocation (
    geolocation_zip_code_prefix INTEGER PRIMARY KEY, -- Primary Key
    geolocation_lat FLOAT, -- Latitude as double equivalent
    geolocation_lng FLOAT, -- Longitude as double equivalent
    geolocation_city VARCHAR(8000), -- City
    geolocation_state VARCHAR(8000) -- State
);

CREATE TABLE customers (
    customer_id NVARCHAR(450) PRIMARY KEY,
    customer_unique_id VARCHAR(8000),
    customer_zip_code_prefix INTEGER,
    customer_city VARCHAR(8000),
    customer_state VARCHAR(8000),
);

CREATE TABLE sellers (
    seller_id NVARCHAR(450) PRIMARY KEY,
    seller_zip_code_prefix INTEGER,
    seller_city VARCHAR(8000),
    seller_state VARCHAR(8000),
);

CREATE TABLE orders (
    order_id NVARCHAR(450) PRIMARY KEY,
    customer_id NVARCHAR(450),
    order_status VARCHAR(8000), -- Order Status
    order_purchase_timestamp DATETIME2(3), -- Timestamp with fractional
seconds (YYYY-MM-DD HH:MM:SS.fff)
    order_approved_at DATETIME2(3), -- Timestamp with fractional seconds
(YYYY-MM-DD HH:MM:SS.fff)
    order_delivered_carrier_date DATETIME2(3), -- Timestamp with fractional
seconds (YYYY-MM-DD HH:MM:SS.fff)
    order_delivered_customer_date DATETIME2(3), -- Timestamp with fractional
seconds (YYYY-MM-DD HH:MM:SS.fff)
    order_estimated_delivery_date DATE, -- Date (YYYY-MM-DD)
);

CREATE TABLE products (
    product_id NVARCHAR(450) PRIMARY KEY,
    product_category_name VARCHAR(8000), -- Category Name
    product_name_length FLOAT, -- Double equivalent in SQL Server
    product_description_length FLOAT, -- Double equivalent in SQL Server
    product_photos_qty FLOAT, -- Double equivalent in SQL Server
    product_weight_g FLOAT, -- Weight in grams (double)
    product_length_cm FLOAT, -- Length in cm (double)
    product_height_cm FLOAT, -- Height in cm (double)
    product_width_cm FLOAT -- Width in cm (double)
);

CREATE TABLE order_items (
    order_items_row_identifier INTEGER PRIMARY KEY, -- Primary Key
```

```
    order_id NVARCHAR(450), -- Foreign key referencing orders
    order_item_id INTEGER,
    product_id NVARCHAR(450), -- Foreign key referencing products
    seller_id NVARCHAR(450), -- Foreign key referencing sellers
    shipping_limit_date DATETIME2(3), -- Timestamp (YYYY-MM-DD HH:MM:SS.fff)
    price FLOAT, -- Double equivalent in SQL Server
    freight_value FLOAT, -- Double equivalent in SQL Server
);

CREATE TABLE order_payments (
    order_id NVARCHAR(450) PRIMARY KEY, -- Primary Key and Foreign Key
referencing orders
    payment_installments BIT, -- Boolean type for whether has instalments
    payment_value FLOAT, -- Payment value (double type)
    payment_type VARCHAR(8000), -- Type of payment (e.g., credit card, debit
card)
);

CREATE TABLE order_reviews (
    review_id NVARCHAR(450) PRIMARY KEY,
    order_id NVARCHAR(450), -- Foreign key referencing orders
    review_score INTEGER, -- Long type equivalent in SQL Server
    review_comment_title VARCHAR(8000), -- Comment title
    review_comment_message VARCHAR(8000), -- Comment Message
    review_creation_date DATETIME2(3), -- Timestamp (YYYY-MM-DD
HH:MM:SS.fff)
    review_answer_timestamp DATETIME2(3), -- Timestamp (YYYY-MM-DD
HH:MM:SS.fff)
);

CREATE TABLE qualified_leads (
    mql_id NVARCHAR(450) PRIMARY KEY,
    first_contact_date DATE, -- First contact date (YYYY-MM-DD)
    landing_page_id VARCHAR(8000), -- Landing page identifier
    origin VARCHAR(8000) -- Origin of the lead
);

CREATE TABLE closed_deals (
    mql_id NVARCHAR(450) PRIMARY KEY, -- also foreign key
    seller_id NVARCHAR(450), --foreign key
    sdr_id VARCHAR(8000), -- Sales Development Representative ID
    sr_id VARCHAR(8000), -- Sales Representative ID
    won_date DATETIME2(3), -- Timestamp (YYYY-MM-DD HH:MM:SS.fff)
    business_segment VARCHAR(8000), -- Business segment
    lead_type VARCHAR(8000), -- Type of lead
    lead_behaviour_profile VARCHAR(8000), -- Behaviour profile of the lead
    has_company BIT, -- Boolean type for has company
    has_gtin BIT, -- Boolean type for has GTIN
    average_stock VARCHAR(8000), -- Average stock
    business_type VARCHAR(8000), -- Type of business
    declared_product_catalog_size FLOAT, -- Declared product catalogue size
(double type)
    declared_monthly_revenue FLOAT, -- Declared monthly revenue (double
type)
);
```

For declaring foreign keys:

```
--This script is to initialise the Foreign Key constraints for all the
tables.
--Only need to run it once, after the tables have been loaded from Azure
Data Factory's pipeline.
--You can run all at once, but make sure to run the delete NULLs from the
order_reviews table.

--Run this first:
DELETE FROM [dbo].[order_reviews]
WHERE [order_id] IS NULL OR
      [review_score] IS NULL OR
      [review_comment_title] IS NULL OR
      [review_comment_message] IS NULL OR
      [review_answer_timestamp] IS NULL;

--This is ok
ALTER TABLE dbo.customers
ADD CONSTRAINT fk_customer_zip
FOREIGN KEY (customer_zip_code_prefix) REFERENCES
dbo.geolocation(geolocation_zip_code_prefix);

--This is ok
ALTER TABLE dbo.sellers
ADD CONSTRAINT fk_seller_zip
FOREIGN KEY (seller_zip_code_prefix) REFERENCES
dbo.geolocation(geolocation_zip_code_prefix);

--This is ok
ALTER TABLE dbo.orders
ADD CONSTRAINT fk_customer_order
FOREIGN KEY (customer_id) REFERENCES dbo.customers(customer_id);

--This is ok
ALTER TABLE dbo.order_items
ADD CONSTRAINT fk_order
FOREIGN KEY (order_id) REFERENCES dbo.orders(order_id);

--This is ok
ALTER TABLE dbo.order_items
ADD CONSTRAINT fk_product
FOREIGN KEY (product_id) REFERENCES dbo.products(product_id);

--This is ok
ALTER TABLE dbo.order_items
ADD CONSTRAINT fk_seller
FOREIGN KEY (seller_id) REFERENCES dbo.sellers(seller_id);

--This is ok
ALTER TABLE dbo.order_payments
ADD CONSTRAINT fk_order_payment
FOREIGN KEY (order_id) REFERENCES dbo.orders(order_id);

--This is ok
ALTER TABLE dbo.order_reviews
ADD CONSTRAINT fk_order_review
```

```
FOREIGN KEY (order_id) REFERENCES dbo.orders(order_id);

--This is ok
ALTER TABLE dbo.closed_deals
ADD CONSTRAINT fk_mql
FOREIGN KEY (mql_id) REFERENCES dbo.qualified_leads(mql_id);

 --This is ok
ALTER TABLE dbo.closed_deals
ADD CONSTRAINT fk_seller_closed_deal
FOREIGN KEY (seller_id) REFERENCES dbo.sellers(seller_id);
```