# ORDERS & CUSTOMERS
# ANALYSIS

INSIGHTS FROM BRAZILLIAN E-CCOMMERCE
**MERCADO LIVRE**

BY MANAN SHAH
DATE: 2 JUNE, 2025

# PROBLEM STATEMENT

E-commerce platform Mercado Livre generate massive datasets involving orders, customers, products, and sellers. Making sense of this data is crucial to understand sales trends, product performance, customer behavior, and seller contributions. The goal of this analysis is to uncover actionable insights such as top product categories, repeat customer behavior, and yearly growth trends to help drive better decision-making.

# KEY INSIGHTS

# TOTAL ORDERS IN 2017

Count the number of orders placed in 2017

```python
orders['order_purchase_timestamp']=pd.to_datetime(orders['order_purchase_timestamp'])

order_year=orders[orders['order_purchase_timestamp'].dt.year==2017]

order_count=len(order_year)
order_count
```

```
45101
```

```sql
# Count the number of orders placed in 2017

select count(order_id) from orders
where year(order_purchase_timestamp)=2017;
```

Result Grid | Filter Rows: | Export:

| count(order_id) |
|---|
| 45101 |

➢This is used to find out how many total orders were placed by customers during the year 2017.

➢It selects and counts all the order IDs from the orders table where the year part of the order date is 2017.

➢It shows the total number of orders made in 2017, which helps us understand how many purchases happened that year.

# ORDER% PAID IN INSTALLMENTS

Calculate the percentage of orders that were paid in installments

```python
installment_payments = payments[payments['payment_installments'] > 1]
single_payments = payments[payments['payment_installments'] <= 1]

count_installments = len(installment_payments)
count_single = len(single_payments)
total = count_installments + count_single

installment_percentage = round((count_installments / total) * 100, 2)
installment_percentage
```

```
49.42
```

- This calculates the percentage of total orders that were paid using **installments** rather than in full.
- It divides the number of payments where installments were more than 1 by the total number of payments, multiplies by 100 to get a percentage, and rounds it to two decimal places.
- It tells us how common installment payments are among customers, giving insight into customer payment behavior and preferences.

```sql
# Calculate the percentage of orders that were paid in installments

select
  ROUND(
    (SELECT COUNT(*) FROM payments WHERE payment_installments > 1) * 100.0 /
    (SELECT COUNT(*) FROM payments),
  2) AS installment_percentage;
```

# CUSTOMER BY CITIES

➢This query lists all the unique cities where customers are located.

➢The query works by selecting only unique city names so that each city appears only once in the result.

➢As a result, we get a list of cities without any repetitions, showing where our customers are located.

➢The output will be a single-column list of city names like "sao paulo", "curitiba", "rio de janeiro", etc. — giving us insights into the geographic spread of customers.

List all cities where customers are located

```
cities=customers['customer_city'].unique()
city_df = pd.DataFrame(cities, columns=['customer_city'])
city_df
```

| | customer_city |
|---|---|
| 0 | franca |
| 1 | sao bernardo do campo |
| 2 | sao paulo |
| 3 | mogi das cruzes |
| 4 | campinas |
| ... | ... |
| 4114 | siriji |
| 4115 | natividade da serra |
| 4116 | monte bonito |
| 4117 | sao rafael |
| 4118 | eugenio de castro |

4119 rows × 1 columns

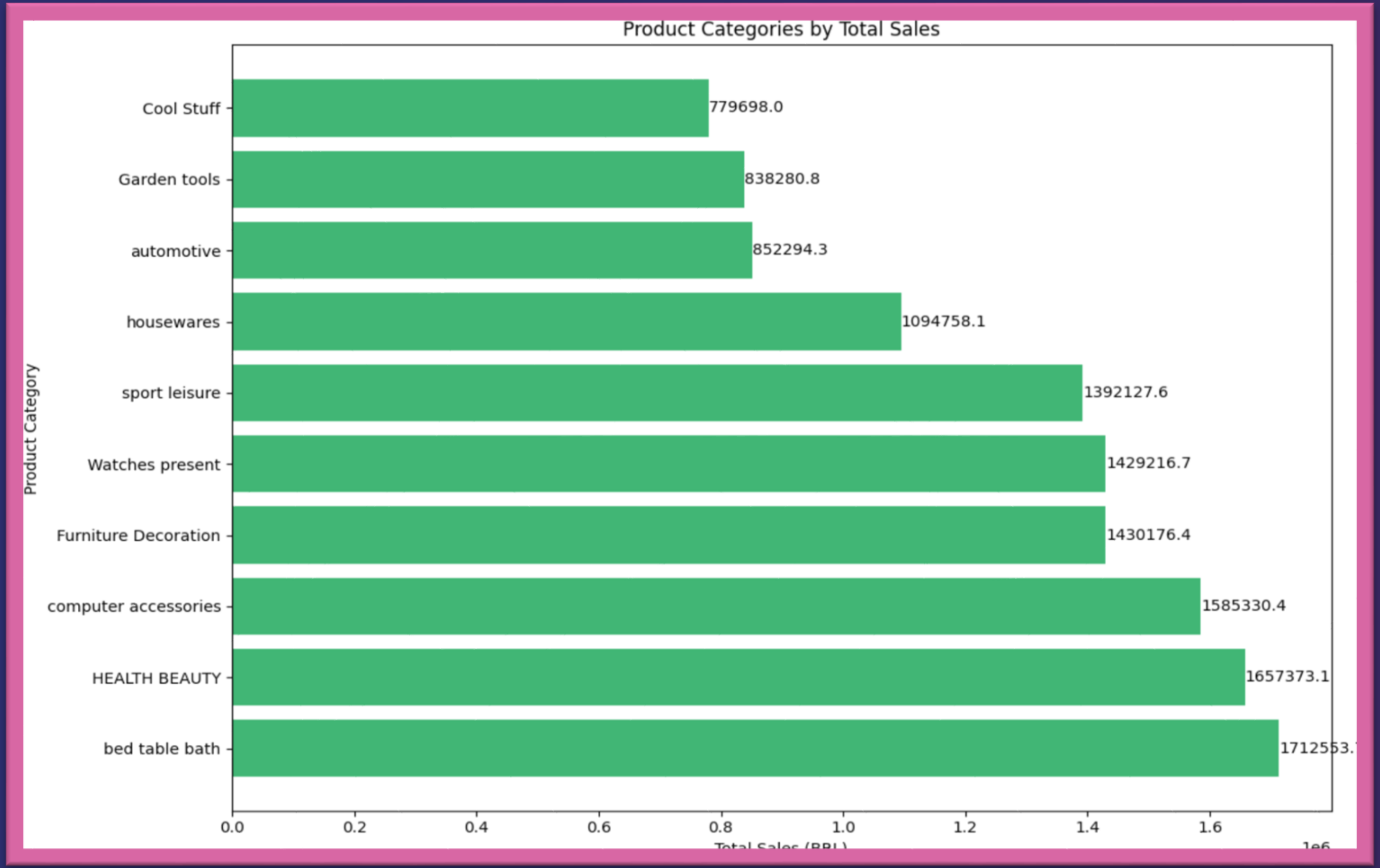
```
# List all cities where customers are located

select distinct(customer_city) from customers;
```

# PRODUCT CATEGORY SALES

Find total sales per category

```python
merged_df = products.merge(order_items, on='product_id') \
                    .merge(payments, on='order_id')

sales_by_category = merged_df.groupby('product category')['payment_value'] \
                             .sum() \
                             .round(1) \
                             .reset_index()

sales_by_category.sort_values('product category', inplace=True)

top_categories = sales_by_category.sort_values('payment_value', ascending=False).head(10)

plt.figure(figsize=(12, 8))
bars = plt.barh(top_categories['product category'], top_categories['payment_value'], color='mediumseagreen')

for bar in bars:
    plt.text(bar.get_width(), bar.get_y() + bar.get_height()/2,
             f'{bar.get_width():.1f}', va='center', ha='left')

plt.title('Product Categories by Total Sales')
plt.xlabel('Total Sales (BRL)')
plt.ylabel('Product Category')
plt.tight_layout()
plt.show()
```

```sql
# Find total sales per category

SELECT
    a.product_category,
    ROUND(SUM(c.payment_value), 1) AS Total_Sales
FROM
    products a
        JOIN
    order_items b ON a.product_id = b.product_id
        JOIN
    payments c ON b.order_id = c.order_id
GROUP BY a.product_category
ORDER BY a.product_category;
```




➢ This calculates the total sales amount (payment value) for each product category.

➢ It joins the products, order_items, and payments tables using product and order IDs, then groups the results by product category and sums the payment values.

➢ It shows how much revenue each product category has generated, helping us understand which categories contribute more or less to the total sales.

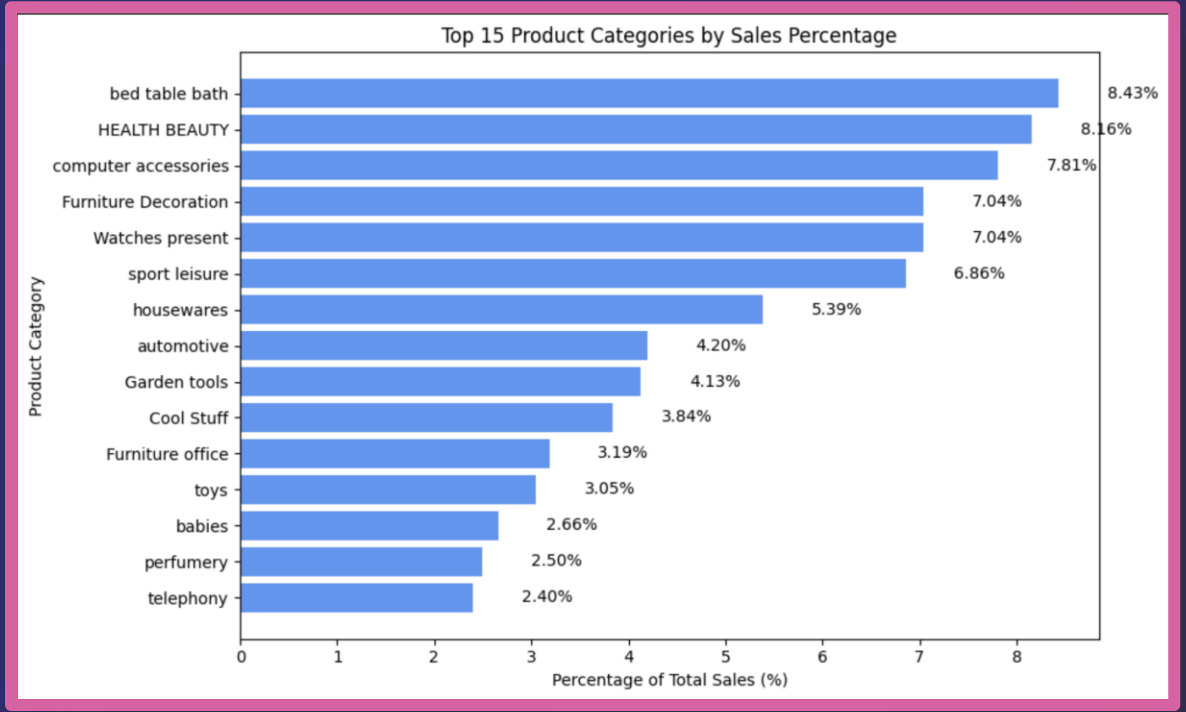
➢ Categories like **bed_table bath** perform well with high sales of around **12,992 BRL**, while categories like **la_cuisine** perform poorly with sales of just **12 BRL**, indicating uneven performance across product categories.

Calculate the percentage of total revenue contributed by each product category

+ Code + Text

```python
merged_df = products.merge(order_items, on='product_id') \
                    .merge(payments, on='order_id')
total_payment_value = merged_df['payment_value'].sum()
category_payment = merged_df.groupby('product category')['payment_value'] \
                            .sum() \
                            .reset_index()
category_payment['percentage'] = (category_payment['payment_value'] / total_payment_value) * 100
top_15 = category_payment.sort_values('percentage', ascending=False).head(15)

plt.figure(figsize=(10, 6))
bars = plt.barh(top_15['product category'], top_15['percentage'], color='cornflowerblue')

plt.gca().invert_yaxis()
for bar in bars:
    plt.text(bar.get_width() + 0.5, bar.get_y() + bar.get_height()/2,
             f'{bar.get_width():.2f}%', va='center')

plt.xlabel('Percentage of Total Sales (%)')
plt.ylabel('Product Category')
plt.title('Top 15 Product Categories by Sales Percentage')
plt.tight_layout()
plt.show()
```

```sql
# Calculate the percentage of total revenue contributed by each product category

SELECT
    a.product_category,
    ROUND(SUM(c.payment_value) * 100.0 / (SELECT
                SUM(payment_value)
            FROM
                payments),
        2) AS percentage
FROM
    products a
        JOIN
    order_items b ON a.product_id = b.product_id
        JOIN
    payments c ON b.order_id = c.order_id
GROUP BY a.product_category
ORDER BY a.product_category;
```




# PRODUCT CATEGORY
# SHARE OF TOTAL SALES

- This finds out what **percentage of total revenue** each product category contributes to the overall sales.
- It joins the products, order_items, and payments tables, sums the payment values per product category, then divides each category's total by the overall payment value sum and multiplies by 100 to get the percentage.
- It shows the **relative contribution** of each category to the company's entire revenue, not just the raw sales amount.
- The **bed_bath_table** category contributes the most with **8.93%** of the total revenue, while **security_and_services** contributes the least with just **0.02%**, highlighting which categories are driving business and which may need re-evaluation or promotion.

# CUSTOMER DISTRIBUTION

# BY STATE

➢ This counts how many customers are from each state.

➢ It groups the customers table by the customer_state column and counts the number of customers in each group.

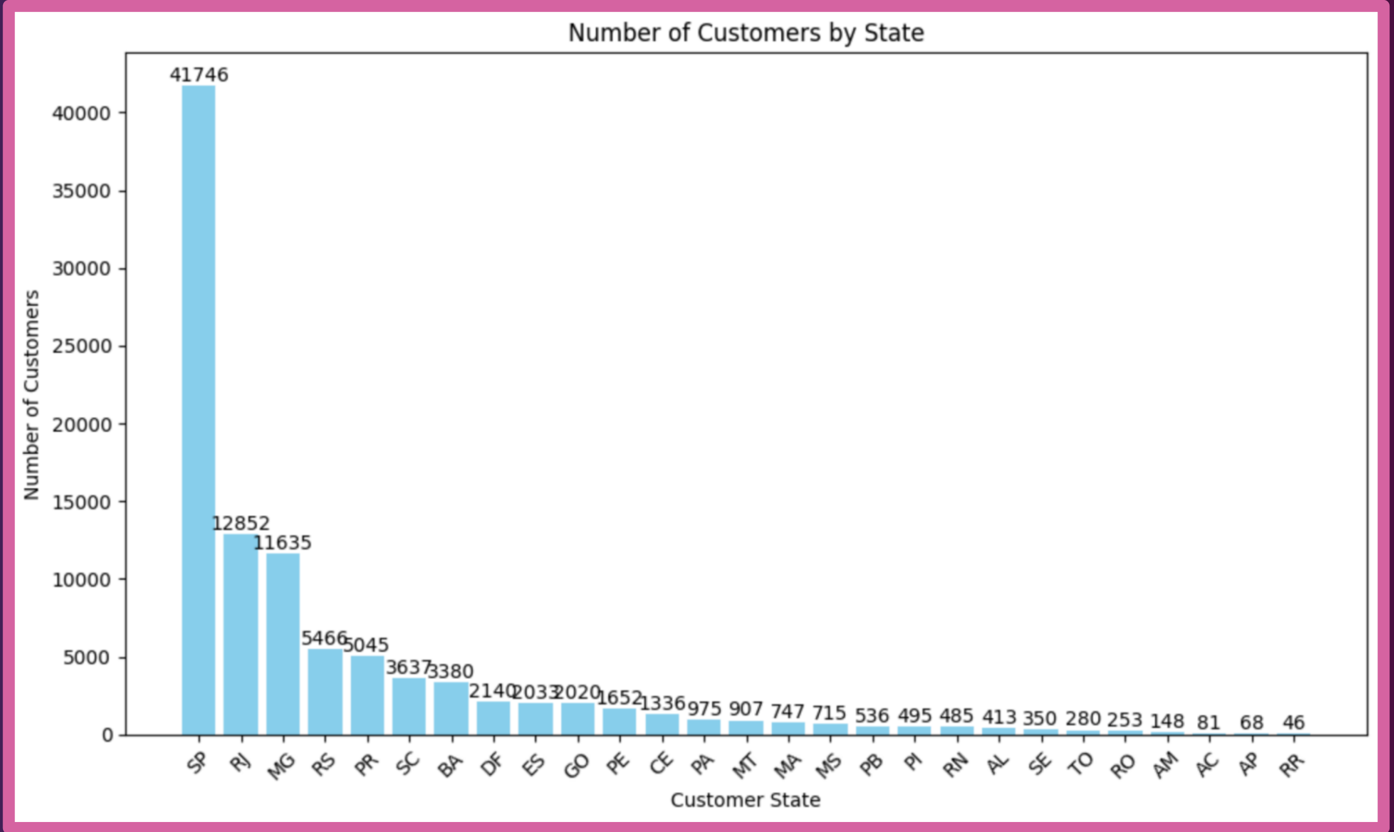
➢ It shows the number of customers living in each state, helping identify where most customers are located geographically.

➢ States like **SP (São Paulo)** have the highest number of customers, indicating a strong market presence, while states like **RR** with fewer customers may offer opportunities for expansion or targeted marketing.

Count the number of customers from each state

```
customer_counts = customers.groupby('customer_state')['customer_id'] \
                           .nunique() \
                           .reset_index(name='customer_count')
customer_counts = customer_counts.sort_values('customer_count', ascending=False)

plt.figure(figsize=(10, 6))
bars = plt.bar(customer_counts['customer_state'], customer_counts['customer_count'], color='skyblue')

for bar in bars:
    plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 10,
             str(bar.get_height()), ha='center', va='bottom')

plt.xlabel('Customer State')
plt.ylabel('Number of Customers')
plt.title('Number of Customers by State')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

```
# Count the number of customers from each state

SELECT
    customer_state, COUNT(customer_id) AS customer_count
FROM
    customers
GROUP BY customer_state
ORDER BY customer_state;
```



# ORDERS PER MONTH

# IN 2018

➢This calculates how many orders were made in each month of the year 2018. It helps you understand the monthly trend in customer orders during that year.

➢ To do this, we use the order date column from the orders table or dataset. We focus only on the year 2018 and then group the data by month.

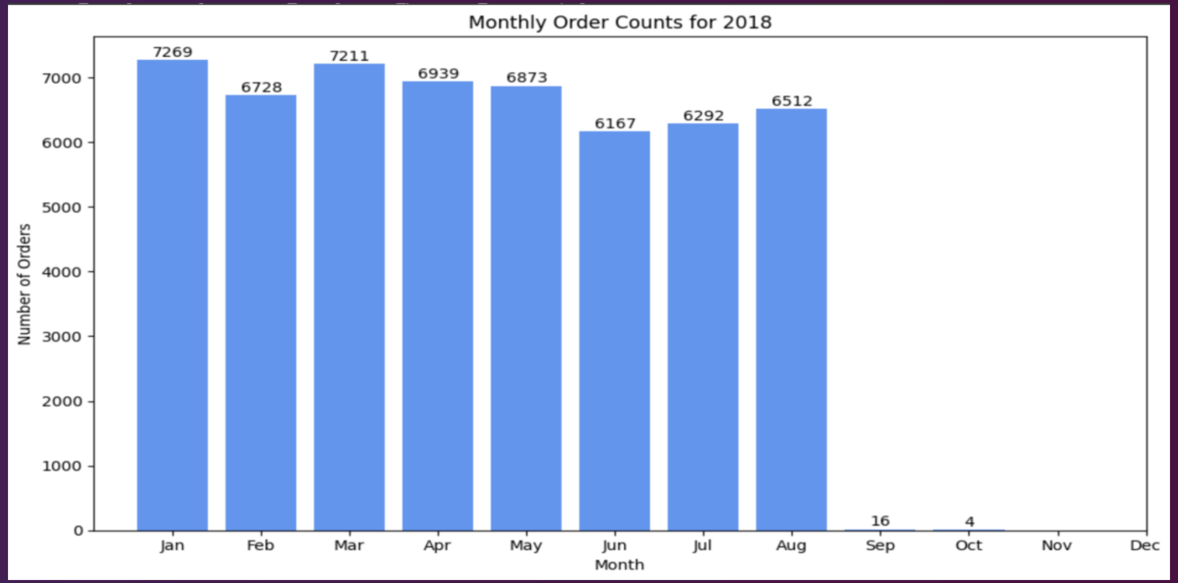
➢It helps us understand the monthly distribution of orders, showing which months had higher or lower order volumes.

➢The output shows that January had the highest number of orders, while December had the lowest.

Calculate the number of orders per month in 2018

```
orders['order_purchase_timestamp'] = pd.to_datetime(orders['order_purchase_timestamp'])
orders_2018 = orders[orders['order_purchase_timestamp'].dt.year == 2018]
orders_2018['month'] = orders_2018['order_purchase_timestamp'].dt.month
monthly_orders = orders_2018.groupby('month')['order_id'].count().reset_index(name='order_count')
monthly_orders = monthly_orders.sort_values('month')

plt.figure(figsize=(10,6))
bars = plt.bar(monthly_orders['month'], monthly_orders['order_count'], color='cornflowerblue')

for bar in bars:
    plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 5,
             str(bar.get_height()), ha='center', va='bottom')

plt.xlabel('Month')
plt.ylabel('Number of Orders')
plt.title('Monthly Order Counts for 2018')
plt.xticks(ticks=range(1,13), labels=['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
                                      'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
plt.tight_layout()
plt.show()
```

```sql
# Calculate the number of orders per month in 2018

SELECT
    MONTH(order_purchase_timestamp) AS months, COUNT(order_id)
FROM
    orders
WHERE
    YEAR(order_purchase_timestamp) = 2018
GROUP BY MONTH(order_purchase_timestamp)
ORDER BY MONTH(order_purchase_timestamp);
```

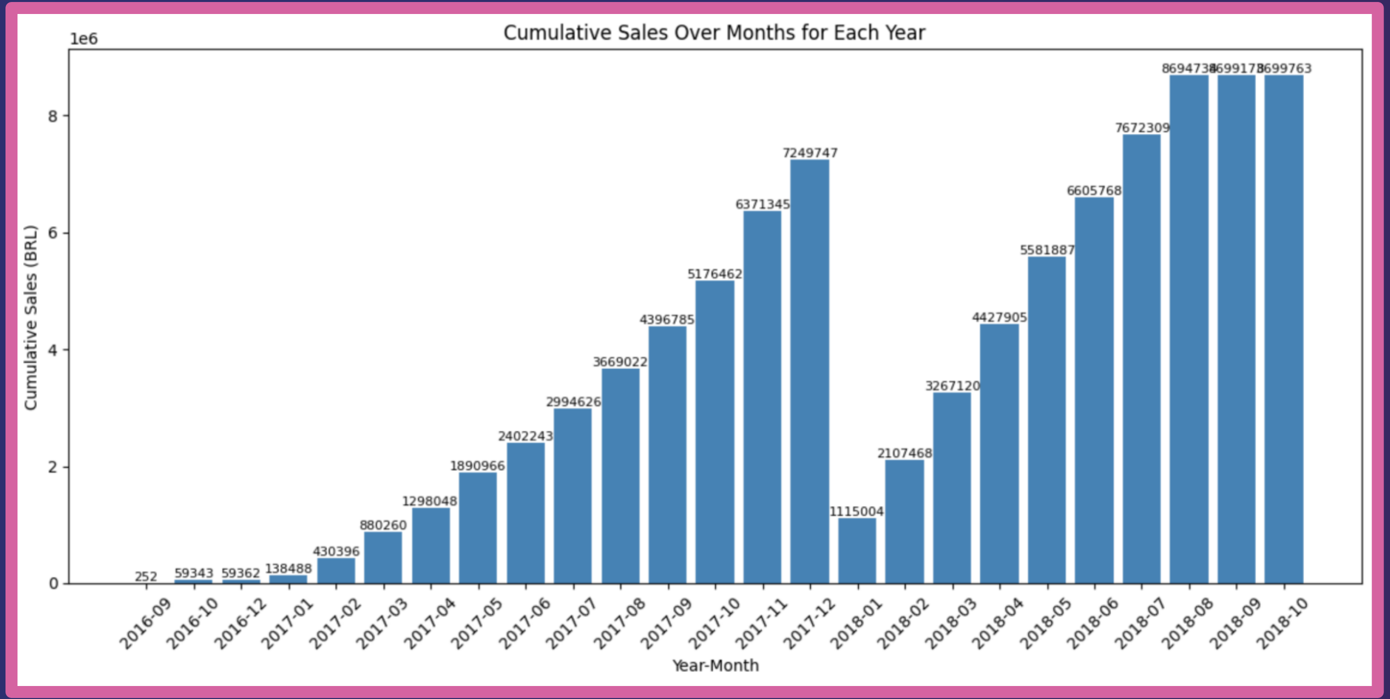


# CUMILATIVE SALES TREND

Calculate the cumilative sales per month for each year

```python
orders_payments = orders.merge(payments, on='order_id')
orders_payments['sales_year'] = pd.to_datetime(orders_payments['order_purchase_timestamp']).dt.year
orders_payments['sales_month'] = pd.to_datetime(orders_payments['order_purchase_timestamp']).dt.month
monthly_sales = orders_payments.groupby(['sales_year', 'sales_month'])['payment_value'] \
                               .sum() \
                               .round(2) \
                               .reset_index(name='monthly_sales')
monthly_sales['cumulative_sales'] = monthly_sales.groupby('sales_year')['monthly_sales'] \
                                                 .cumsum() \
                                                 .round(2)
monthly_sales['year_month'] = monthly_sales['sales_year'].astype(str) + '-' + monthly_sales['sales_month'].astype(str).str.zfill(2)

plt.figure(figsize=(12, 6))
bars = plt.bar(monthly_sales['year_month'], monthly_sales['cumulative_sales'], color='steelblue')

for bar in bars:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, height, f'{height:.0f}', ha='center', va='bottom', fontsize=8)

plt.xticks(rotation=45)
plt.xlabel('Year-Month')
plt.ylabel('Cumulative Sales (BRL)')
plt.title('Cumulative Sales Over Months for Each Year')
plt.tight_layout()
plt.show()
```

```sql
# Calculate the cumilative sales per month for each year

select year(order_purchase_timestamp) as sales_year,
month(order_purchase_timestamp) as sales_month,
round(sum(payment_value),2) as monthly_sales,
round(SUM(SUM(payment_value)) OVER (
        PARTITION BY YEAR(order_purchase_timestamp)
        ORDER BY MONTH(order_purchase_timestamp)
    ), 2) as cumilative_sales
from orders o join payments p on o.order_id=p.order_id
group by year(order_purchase_timestamp), month(order_purchase_timestamp)
order by sales_year, sales_month;
```



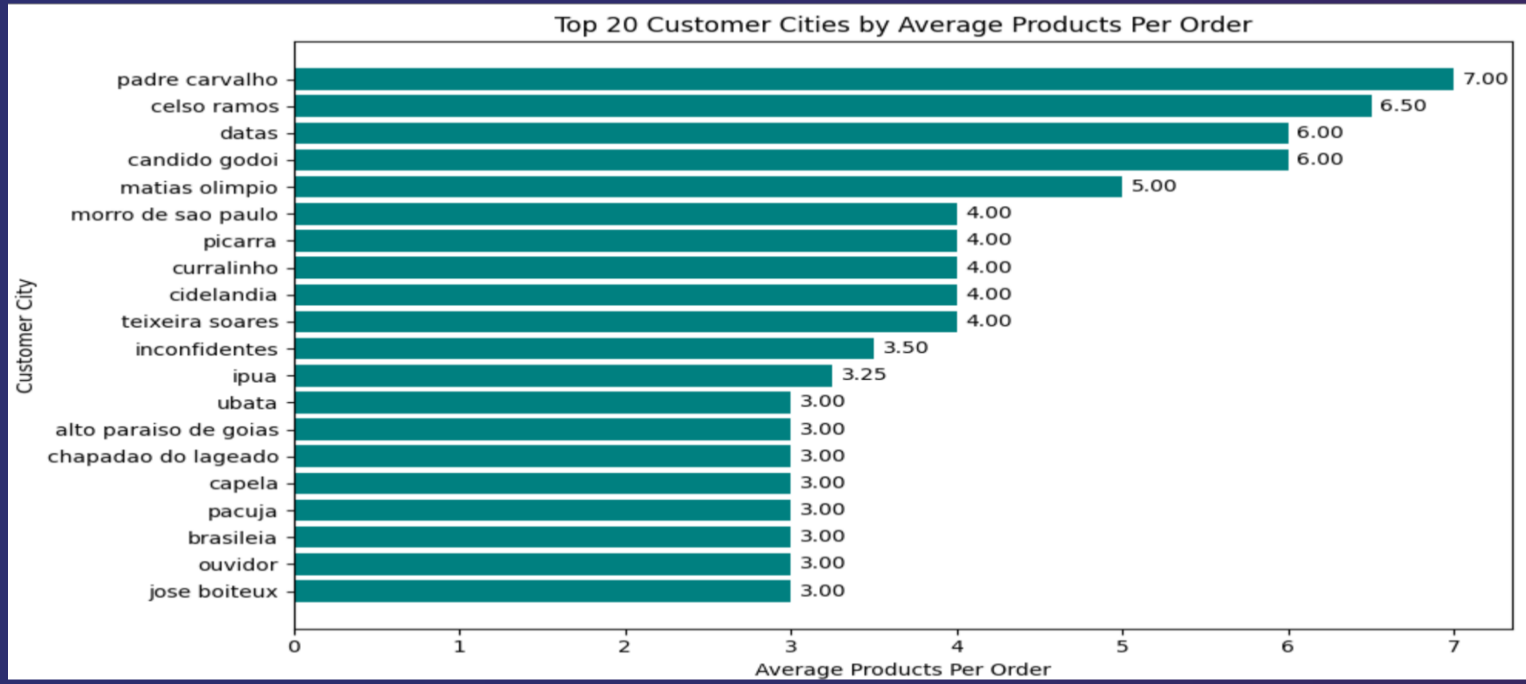

- ➢This calculates the total sales for each month and also the cumulative (running total) sales for each year over its months.
- ➢It groups sales data by year and month, sums payment values for each month, and then uses a window function to calculate the cumulative sum of sales within each year, ordered by month.
- ➢It shows not only monthly sales but also how sales accumulate throughout the year, giving insight into sales growth and trends over time.
- ➢The cumulative sales tend to rise steadily through the months of each year, indicating consistent growth, but certain months may show sharper increases, pointing to possible seasonal peaks or successful campaigns.

# AVERAGE PRODUCTS PER ORDER

Find the average number of products per order, grouped by customer city

```python
product_counts = order_items.groupby('order_id')['product_id'].count().reset_index(name='product_count')
order_product_counts = orders[['order_id', 'customer_id']].merge(product_counts, on='order_id')
order_customer_products = order_product_counts.merge(customers[['customer_id', 'customer_city']], on='customer_id')
avg_products_city = order_customer_products.groupby('customer_city')['product_count'] \
                                        .mean() \
                                        .round(2) \
                                        .reset_index(name='avg_products_per_order')

avg_products_city = avg_products_city.sort_values('avg_products_per_order', ascending=False).head(20)
plt.figure(figsize=(14, 8))
bars = plt.barh(avg_products_city['customer_city'], avg_products_city['avg_products_per_order'], color='teal')

plt.gca().invert_yaxis()
for bar in bars:
    plt.text(bar.get_width() + 0.05, bar.get_y() + bar.get_height()/2,
            f'{bar.get_width():.2f}', va='center')

plt.xlabel('Average Products Per Order')
plt.ylabel('Customer City')
plt.title('Top 20 Customer Cities by Average Products Per Order')
plt.tight_layout()
plt.show()
```

```sql
# Find the average number of products per order, grouped by customer city

SELECT
    c.customer_city,
    ROUND(AVG(product_count), 2) AS avg_products_per_order
FROM (
    SELECT
        o.order_id,
        COUNT(oi.product_id) AS product_count
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    GROUP BY o.order_id
) AS order_product_counts
JOIN orders o ON order_product_counts.order_id = o.order_id
JOIN customers c ON o.customer_id = c.customer_id
GROUP BY c.customer_city
ORDER BY avg_products_per_order DESC;
```




- This calculates the average number of products included in each order, grouped by customer city.
- It shows the average quantity of products per order for each city, helping us understand where customers tend to purchase more items in a single transaction.
- Cities with higher averages indicate more items bought per order — for example, cities like Nova Iguaçu or Campinas may show stronger buying behavior per transaction.
- These insights help identify cities with high-order volume potential, which can be useful for optimizing logistics, targeting upsell strategies, or offering location-specific bundles.

# CORRELATION BETWEEN PRODUCT PRICE & PURCHASED

Identify the correlation between product price and number of times a product has been purchased

```python
order_stats = order_items.groupby('order_id').agg(
    avg_value=('price', 'mean'),
    count_order=('order_id', 'count')
).reset_index()

correlation = order_stats['avg_value'].corr(order_stats['count_order'])

print(f"Correlation between average price and number of products per order: {correlation:.4f}")
```

```
Correlation between average price and number of products per order: -0.0585
```

```sql
# Identify the correlation between product price and number of times a product has been purchased

WITH order_stats AS (
    SELECT
        order_id,
        AVG(price) AS avg_value,
        COUNT(*) AS count_order
    FROM order_items
    GROUP BY order_id
),
summary AS (
    SELECT
        COUNT(*) AS n,
        SUM(avg_value) AS sum_x,
        SUM(count_order) AS sum_y,
        SUM(avg_value * count_order) AS sum_xy,
        SUM(avg_value * avg_value) AS sum_x2,
        SUM(count_order * count_order) AS sum_y2
    FROM order_stats
)
SELECT
    ROUND(
        (sum_xy - (sum_x * sum_y / n)) /
        SQRT((sum_x2 - (sum_x * sum_x / n)) * (sum_y2 - (sum_y * sum_y / n))),
        4
    ) AS correlation
FROM summary;
```

- This calculates the correlation between the average product price and the number of products in each order.
- It helps us understand whether orders that contain more items tend to have higher or lower average prices.
- A positive correlation value would suggest that orders with more items also have higher average prices, while a negative value would indicate that larger orders tend to contain cheaper products.
- The output is -0.05 which shows a negative correlation.

# RETENTION RATE

- This calculates the percentage of customers who made at least one repeat purchase within six months after their first order.
- It first finds each customer's first order date, then checks if they placed any additional orders within six months after that date, counts those customers, and calculates their percentage out of all customers.
- It shows how many customers stay loyal and keep buying from the store shortly after their first purchase, measuring customer retention.
- The retention rate is **0%**, meaning no customers made a repeat purchase within six months of their first order, which signals a significant challenge in customer retention and highlights a need to improve engagement and follow-up strategies.

```sql
# Calculate the retention rate of customers, defined as the percentage of customers
# who make another purchase within 6 months of their first purchase

WITH first_orders AS (
    SELECT customer_id, DATE(MIN(order_purchase_timestamp)) AS first_order_date
    FROM orders
    GROUP BY customer_id
),
repeat_orders AS (
    SELECT o.customer_id
    FROM orders o
    JOIN first_orders f ON o.customer_id = f.customer_id
    WHERE DATE(o.order_purchase_timestamp) > f.first_order_date
      AND DATE(o.order_purchase_timestamp) <= DATE_ADD(f.first_order_date, INTERVAL 6 MONTH)
    GROUP BY o.customer_id
)

SELECT
    ROUND(
        (SELECT COUNT(*) FROM repeat_orders) * 100.0 / COUNT(*),
        2
    ) AS retention_rate_percentage
FROM first_orders;
```

Calculate the retention rate of customers, defined as the percentage of customers who make another purchase within 6 months of their first purchase

```python
orders['order_purchase_timestamp'] = pd.to_datetime(orders['order_purchase_timestamp'])
first_orders = orders.groupby('customer_id')['order_purchase_timestamp'] \
                    .min() \
                    .reset_index() \
                    .rename(columns={'order_purchase_timestamp': 'first_order_date'})
orders_with_first = orders.merge(first_orders, on='customer_id')

orders_with_first['within_6_months'] = (
    (orders_with_first['order_purchase_timestamp'] > orders_with_first['first_order_date']) &
    (orders_with_first['order_purchase_timestamp'] <= orders_with_first['first_order_date'] + pd.DateOffset(months=6))
)
repeat_customers = orders_with_first[orders_with_first['within_6_months']].customer_id.unique()
retention_rate_percentage = round(len(repeat_customers) * 100.0 / len(first_orders), 2)

print(f"Retention Rate (6 months): {retention_rate_percentage}%")
```

```
Retention Rate (6 months): 0.0%
```

# YEAR-ON-YEAR (YOY)

# SALES GROWTH

- ➢ This compares total sales for each year with the previous year to calculate the percentage growth or decline in sales year over year.
- ➢ It first calculates total sales per year by summing payments, then joins the yearly sales data with itself offset by one year to find the previous year's sales, and finally calculates the percentage change between the two years.
- ➢ It shows how the business's sales performance is changing year to year, indicating growth trends or declines over time.
- ➢ The sales in 2017 grow over 12000% from 2016, showing a huge boost in sales and revenue, while 2018 saw a 20% jump from 2017.

```sql
# Calculate the year over year growth rate of total sales

WITH yearly_sales AS (
    SELECT
        YEAR(o.order_purchase_timestamp) AS sales_year,
        ROUND(SUM(p.payment_value), 2) AS total_sales
    FROM orders o
    JOIN payments p ON o.order_id = p.order_id
    GROUP BY YEAR(o.order_purchase_timestamp)
)
SELECT
    curr.sales_year,
    curr.total_sales,
    prev.total_sales AS prev_year_sales,
    ROUND(
        ((curr.total_sales - prev.total_sales) / prev.total_sales) * 100,
        2
    ) AS yoy_growth_percent
FROM yearly_sales curr
LEFT JOIN yearly_sales prev
    ON curr.sales_year = prev.sales_year + 1
ORDER BY curr.sales_year;
```

Calculate the year over year growth rate of total sales

```python
orders_payments = orders.merge(payments, on='order_id')
orders_payments['sales_year'] = pd.to_datetime(orders_payments['order_purchase_timestamp']).dt.year

yearly_sales = orders_payments.groupby('sales_year')['payment_value'] \
                              .sum() \
                              .round(2) \
                              .reset_index(name='total_sales')
yearly_sales['prev_year_sales'] = yearly_sales['total_sales'].shift(1)
yearly_sales['yoy_growth_percent'] = ((yearly_sales['total_sales'] - yearly_sales['prev_year_sales']) / yearly_sales['prev_year_sales']) * 100
yearly_sales['yoy_growth_percent'] = yearly_sales['yoy_growth_percent'].round(2)

yearly_sales
```

| SALES YEAR | TOTAL SALES | PREV YEAR SALES | GROWTH RATE |
|---|---|---|---|
| 2016 | 59362.34 | NAN | NAN |
| 2017 | 7249746.73 | 59362.34 | 12117.7 |
| 2018 | 8699763.05 | 7249746.73 | 20.0 |

# MOVING AVERAGE

# OF ORDER VALUE

- This calculates the moving (cumulative) average of order values for each customer across their entire purchase history.
- It uses the orders and payments tables, merging them to get the payment_value for each order, then groups the data by customer and order date to track the value of each purchase.
- This helps us understand how each customer's average order value changes over time — for example, whether a customer is spending more or less as their order history grows.
- The output shows the customer ID, order ID, order date, and the cumulative average order value revealing long-term purchase behavior and loyalty trends.

Calculate the moving average of order values for each customer over their order history

```python
orders_payments = orders.merge(payments, on='order_id')
order_payment_sum = orders_payments.groupby(['customer_id', 'order_id', 'order_purchase_timestamp'])['payment_value'] \
                                    .sum() \
                                    .reset_index()
order_payment_sum = order_payment_sum.sort_values(['customer_id', 'order_purchase_timestamp'])

order_payment_sum['cumulative_avg_order_value'] = order_payment_sum.groupby('customer_id')['payment_value'] \
                                                                    .expanding() \
                                                                    .mean() \
                                                                    .reset_index(level=0, drop=True) \
                                                                    .round(2)
result = order_payment_sum[['customer_id', 'order_id', 'order_purchase_timestamp', 'cumulative_avg_order_value']] \
          .sort_values(['customer_id', 'order_purchase_timestamp'])

result
```

```sql
# Calculate the moving average of order values for each customer over their order history

SELECT
    o.customer_id,
    o.order_id,
    o.order_purchase_timestamp,
    ROUND(
        AVG(SUM(p.payment_value)) OVER (
            PARTITION BY o.customer_id
            ORDER BY o.order_purchase_timestamp
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
        ), 2
    ) AS cumulative_avg_order_value
FROM orders o
JOIN payments p ON o.order_id = p.order_id
GROUP BY o.customer_id, o.order_id, o.order_purchase_timestamp
ORDER BY o.customer_id, o.order_purchase_timestamp;
```

| | customer_id | order_id | order_purchase_timestamp | cumulative_avg_order_value |
|---|---|---|---|---|
| **0** | 00012a2ce6f8dcda20d059ce98491703 | 5f79b5b0931d63f1a42989eb65b9da6e | 2017-11-14 16:08:26 | 114.74 |
| **1** | 000161a058600d5901f007fab4c27140 | a44895d095d7e0702b6a162fa2dbeced | 2017-07-16 09:40:32 | 67.41 |
| **2** | 0001fd6190edaaf884bcaf3d49edf079 | 316a104623542e4d75189bb372bc5f8d | 2017-02-28 11:06:43 | 195.42 |
| **3** | 0002414f95344307404f0ace7a26f1d5 | 5825ce2e88d5346438686b0bba99e5ee | 2017-08-16 13:09:20 | 179.35 |
| **4** | 000379cdec625522490c315e70c7a9fb | 0ab7fb08086d4af9141453c91878ed7a | 2018-04-02 13:42:17 | 107.01 |
| **...** | ... | ... | ... | ... |
| **99435** | fffecc9f79fd8c764f843e9951b11341 | 814d6a3a7c0b32b2ad929ac6328124e9 | 2018-03-29 16:59:26 | 81.36 |
| **99436** | fffeda5b6d849fbd39689bb92087f431 | 8c855550908247a7eff50281b92167a8 | 2018-05-22 13:36:02 | 63.13 |
| **99437** | ffff42319e9b2d713724ae527742af25 | 83b5fc912b2862c5046555ded1483ae9 | 2018-06-13 16:57:05 | 214.13 |
| **99438** | ffffa3172527f765de70084a7e53aae8 | d0e7be325a1c986babc4e1cdb91edc03 | 2017-09-02 11:53:32 | 45.50 |
| **99439** | ffffe8b65bbe3087b653a978c870db99 | 2e935fa1d39497aa0ec3f1107fbfb5b8 | 2017-09-29 14:07:03 | 18.37 |

99440 rows × 4 columns

# SELLER REVENUE

# RANKING

- This calculates how much revenue each seller earned and ranks them from highest to lowest based on total revenue generated.
- It adds up each seller's product price and freight value using the order_items table, groups the data by seller_id, and then uses the RANK() function to assign a rank based on descending revenue.
- It shows which sellers are generating the most revenue on the platform, helping identify top-performing sellers.
- Rank 1 seller generated sales of 2.5 BR, while seller of least rank generated sales of just 12 BR.

Calculate revenue generated by each seller and rank them by revenue

```python
order_items['total_price_freight'] = order_items['price'] + order_items['freight_value']

seller_revenue = order_items.groupby('seller_id')['total_price_freight'] \
                            .sum() \
                            .round(2) \
                            .reset_index(name='Total_Revenue')
seller_revenue['revenue_rank'] = seller_revenue['Total_Revenue'].rank(method='min', ascending=False).astype(int)

seller_revenue = seller_revenue.sort_values('revenue_rank')
seller_revenue
```

```sql
# Calculate revenue generated by each seller and rank them by revenue

select
  o.seller_id,
  ROUND(SUM(o.price + o.freight_value), 2) as Total_Revenue,
  RANK() OVER (ORDER BY ROUND(SUM(o.price + o.freight_value), 2) DESC) as revenue_rank
from order_items o
join sellers s on s.seller_id = o.seller_id
group by o.seller_id;
```

| | seller_id | Total_Revenue | revenue_rank |
|---|---|---|---|
| 857 | 4869f7a5dfa277a7dca6462dcf3b52b2 | 249640.70 | 1 |
| 1535 | 7c67e1448b00f6e969d365cea6b010ab | 239536.44 | 2 |
| 1013 | 53243585a1d6dc2643021fd1853d8905 | 235856.68 | 3 |
| 881 | 4a3ca9315b744ce9f8e9374361493884 | 235539.96 | 4 |
| 3024 | fa1c13f2614d7b5c4749cbc52fecda94 | 204084.73 | 5 |
| ... | ... | ... | ... |
| 1370 | 702835e4b785b67a084280efca355756 | 18.56 | 3091 |
| 869 | 4965a7002cca77301c82d3f91b82e1a9 | 16.36 | 3092 |
| 373 | 1fa2d3def6adfa70e58c276bb64fe5bb | 15.90 | 3093 |
| 1465 | 77128dec4bec4878c37ab7d6169d6f26 | 15.22 | 3094 |
| 2519 | cf6f6bc4df3999b9c6440f124fb2f687 | 12.22 | 3095 |

Identify top 3 customers who spent most money in each year

```python
cust_orders_payments = customers.merge(orders, on='customer_id') \
                                .merge(payments, on='order_id')
cust_orders_payments['order_year'] = pd.to_datetime(cust_orders_payments['order_purchase_timestamp']).dt.year
grouped = cust_orders_payments.groupby(['customer_id', 'order_year'])['payment_value'] \
                              .sum() \
                              .round(2) \
                              .reset_index(name='total_spent')

grouped['rank_order'] = grouped.groupby('order_year')['total_spent'] \
                              .rank(method='dense', ascending=False)

top_customers_per_year = grouped[grouped['rank_order'] <= 3] \
                         .sort_values(['order_year', 'rank_order'])

top_customers_per_year
```

| | customer_id | order_year | total_spent | rank_order |
|---|---|---|---|---|
| 66073 | a9dc96b027d1252bbac0a9b72d837fc6 | 2016 | 1423.55 | 1.0 |
| 11353 | 1d34ed25963d5aae4cf3d7f3a4cda173 | 2016 | 1400.74 | 2.0 |
| 28708 | 4a06381959b6670756de02e07b83815f | 2016 | 1227.78 | 3.0 |
| 8546 | 1617b1357756262bfa56ab541c47bc16 | 2017 | 13664.08 | 1.0 |
| 77521 | c6e2731c5b391845f6800c97401a43a9 | 2017 | 6929.31 | 2.0 |
| 24771 | 3fd6777bbce08a352fddd04e4a7cc8f6 | 2017 | 6726.66 | 3.0 |
| 91984 | ec5b2ba62e574342386871631fafd3fc | 2018 | 7274.88 | 1.0 |
| 95123 | f48d464a0baaea338cb25f816991ab1f | 2018 | 6922.21 | 2.0 |
| 87396 | e0a2412720e9ea4f26c1ac985f6a7358 | 2018 | 4809.44 | 3.0 |

```sql
# Identify top 3 customers who spent most money in each year

SELECT *
FROM (
    SELECT
        c.customer_id,
        YEAR(o.order_purchase_timestamp) AS order_year,
        ROUND(SUM(p.payment_value), 2) AS total_spent,
        RANK() OVER (
            PARTITION BY YEAR(o.order_purchase_timestamp)
            ORDER BY SUM(p.payment_value) DESC
        ) AS rank_order
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    JOIN payments p ON o.order_id = p.order_id
    GROUP BY c.customer_id, YEAR(o.order_purchase_timestamp)
) ranked_customers
WHERE rank_order <= 3
ORDER BY order_year, rank_order;
```

# TOP 3 CUSTOMERS PER YEAR

- This identifies the top 3 customers who spent the most money in each year based on their total payment value.
- It calculates the total amount spent by each customer in each year, and then ranks them within each year using the rank() function.
- This helps us highlight the most valuable customers annually — those who contribute the highest revenue which is useful for loyalty programs, personalized offers, or VIP targeting.
- The output includes the customer ID, year, total amount spent, and their rank showing the top 3 customers per year.

# CONCLUSION

- The platform recorded a significant number of orders in 2017, indicating good early traction and growth during that year.
- A good portion of customers chose to pay in multiple installments, showing that flexible payment options are important for user convenience.
- Categories such as watches_gifts and telephony contributed a large percentage to overall revenue, while several niche categories made up very little.
- Most customers are concentrated in states like São Paulo, while some states have very few customers, indicating regional usage patterns.
- The cumulative sales graphs show consistent monthly growth over each year, especially in 2018, which reflects increased customer activity and higher order volumes.
- Sales have increased each year, showing overall growth and success in attracting more customers or encouraging more purchases.
- A few sellers account for most of the revenue, with top sellers earning many times more than average ones. This shows a clear difference in seller performance.
- Every year, a few customers spend much more than others, and identifying these high-value customers can help with loyalty strategies.

# RECOMENDATIONS

- Since the 6-month repeat purchase rate is 0%, a retention strategy should be launched. This could include email follow-ups, offers, or loyalty programs.
- Invest more in stocking and promoting best-selling categories like health_beauty or telephony to maximize returns.

- Support and reward top sellers with better visibility or tools so they can keep contributing high revenue.
- Launch marketing campaigns in states with fewer customers to spread platform usage more evenly across the country.
- Promote installment payment options more clearly, as they seem to help increase customer spending and affordability.

# THANK YOU!

DATA ANALYSIS IS KEY TO BUSINESS GROWTH AND SUCCESS!