POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Systems and Methods for Big and Unstructured Data Project

Author(s): **Angelo Attivissimo 10667094**

Riccardo Domingo Compagnoni 10730930

Academic Year: 2023-2024

Contents

1 | Introduction

The aim of this project is to analyze a dataset of an ecommerce platform that contains information about 100k orders made at multiple marketplaces in Brazil in the period from 2016 and 2018. The dataset comes in eight separate files, whose elements are highly connected to each other in a graph-like way. This is the reason why we chose Neo4j, as well as for the fact that many queries require to take into consideration long paths between data points.

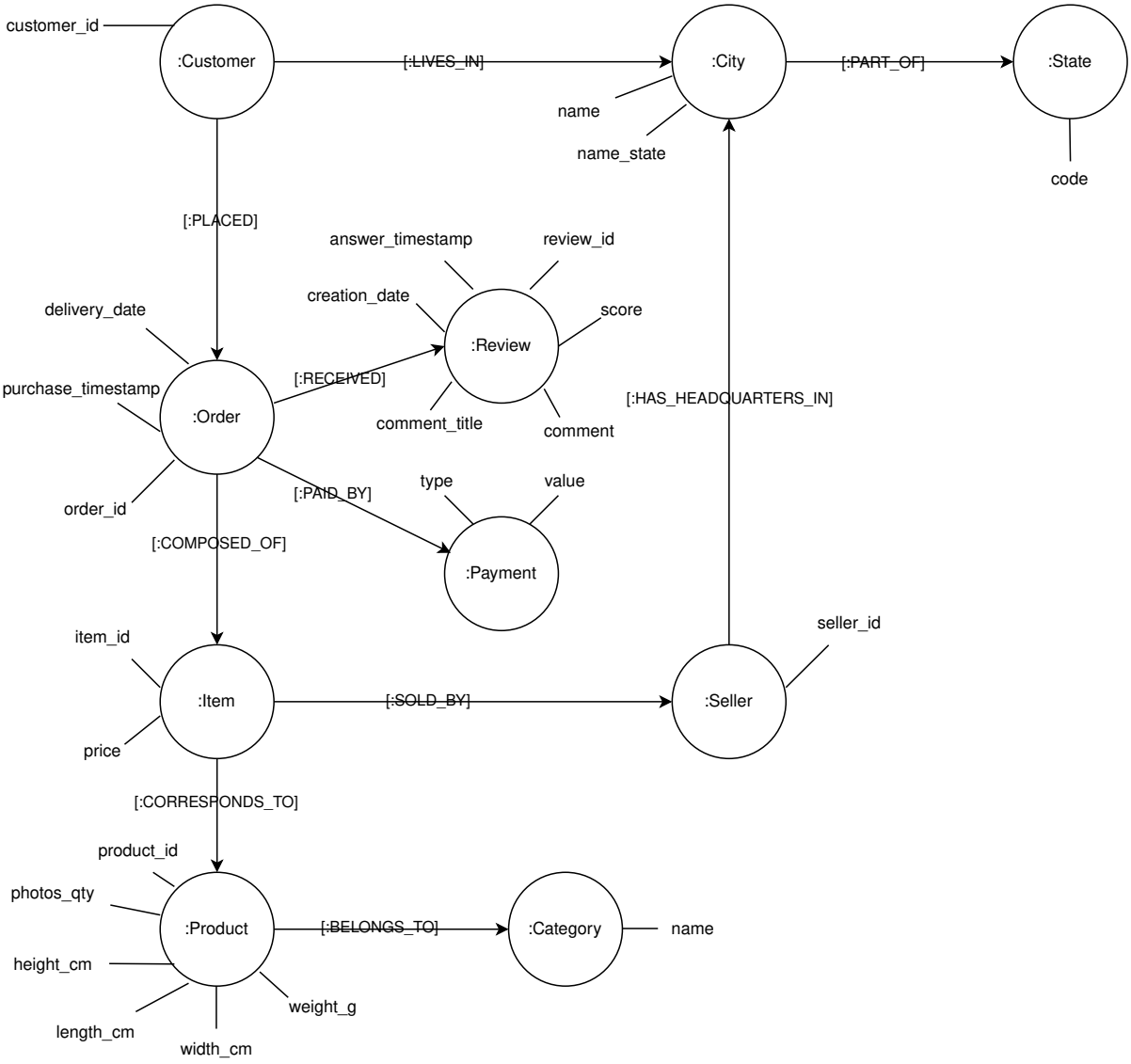
Here is the link: https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce

2 | Data Wrangling

The only data wrangling operation that we performed on the dataset was deleting the character '\' in a *review_comment_message* of the *olist_order_reviews_dataset.csv*. At line 77918 the original comment message is "foi escrito que o produto já foi entregue, mas não chegou nada :\" and we replaced it with "foi escrito que o produto já foi entregue, mas não chegou nada". We had to do this because Neo4J had some problems dividing the line into the corresponding attributes with an extra escape character '\' at the end. The csv provided in the delivery .zip has already been corrected (dataset folder).
Even though we haven't modified anything else, we can also make some other considerations about the format of the csv files and how we dealt with some other inconsistencies:

- Some *review_id* appear to be associated with two *order_id* in *olist_order_reviews_dataset.csv*. From the community forum on Kaggle we seemed to understand that this issue was due to a bug in the ecommerce system that associated the same review to two orders that were placed less than a second apart from each other. Since we couldn't know which of the two orders the review was supposed to be linked to, we just decided to leave the file as it was and connect multiple orders to the same review. We could have also skipped over the affected reviews, but they were too many, so we would have ended up with few nodes left. Even though this might not be the best solution in a real-world scenario, we felt this was the best choice for educational purposes, where having few nodes would have affected the results of the queries.
- Many columns have missing values, especially for *olist_products _dataset.csv*, but we adopted a schemaless approach, so we didn't do anything about this because it didn't affect neither the loading process nor the results of our queries.

3 | Dataset



The image above illustrates the schema we adopted for our database.

3.1. Nodes

Here follows a description of each node of our database:

- **:Customer**
  - *customer_id* corresponds to *customer_unique_id* in the csv file and it is unique.
- **:Seller**
  - *seller_id* corresponds to the *seller_id* in *olist_sellers_dataset.csv*.
- **:City**
  - *name* is the name of the city
  - *name_state* corresponds to its name with the code of the :State it is part of appended to it and it is unique. We created it because some names are not unique and can refer to multiple cities in different States.
- **:State**
  - *code* is the code of the State and it is unique.
- **:Order**
  - *order_id* corresponds to the id of an order and it is unique.
  - *purchase_timestamp* corresponds to the timestamp of the confirmation of the order.
  - *delivery_date* corresponds to the date in which the order was delivered
- **:Review**
  - *review_id* corresponds to the id of a review and it is unique.
  - *creation_date* corresponds to the date in which the review was created.
  - *answer_timestamp* corresponds to the timestamp in which the answer to the order's survey is submitted.
  - *score* corresponds to the score of the review
  - *comment_title* corresponds to the title of the review's comment message.
  - *comment* correspond to text message of the review
- **:Payment**
  - *type* corresponds to the method that was used to pay (credit card, boleto (bank ticket), etc.)

  - *value* corresponds to the amount in Brazilian reals.

- **:Item**
  - *item_id* corresponds to *order_id* in the csv file with *order_item_id* appended to it.
  - *price* corresponds to the amount in Brazilian reals. This attribute belongs to this node instead of :Product, because different sellers might charge a different sum for the same product.

- **:Product**
  - *product_id* corresponds to the id in *olist_products_dataset.csv*
  - *height_cm, length_cm* and *width_cm* correspond to the dimensions of the product in centimeters.
  - *weight_g* is the weight in grams.

- **:Category**
  - *name* identifies the category. We linked the products that didn't have any category associated to a new :Category with name "NA", so that queries could be easier.

3.2. Relations

Here follows a description of each relation of our database:

- **:PART_OF** links a :City to one and only one :State.
- **:HAS_HEADQUARTERS_IN:** links a :Seller to one and only one :City
- :**LIVES_IN** links a :Customer to one and only one :City
- **:PLACED** links a :Customer to an :Order they placed
- **:COMPOSED_OF** links an :Order to an :Item it contains
- **:SOLD_BY** links an :Item to its :Seller
- **:CORRESPONDS_TO** links an :Item to one and only one :Product
- **:BELONGS_TO** links a :Product to the corresponding :Category
- **:RECEIVED** links an :Order to a corresponding :Review node

- **:PAID_BY** links an :Order to a corresponding :Payment node. It is important to note that a customer can pay an order in multiple transactions: that is why :Order can be linked to more than one :Payment node

3.3. Constraints

Here is the complete cypher constraints script (the file is '/code/constraint.cypher'):

```
CREATE CONSTRAINT FOR (c:Customer) REQUIRE c.customer_id IS UNIQUE;
CREATE CONSTRAINT FOR (o:Order) REQUIRE o.order_id IS UNIQUE;
CREATE CONSTRAINT FOR (r:Review) REQUIRE r.review_id IS UNIQUE;
CREATE CONSTRAINT FOR (p:Product) REQUIRE p.product_id IS UNIQUE;
CREATE CONSTRAINT FOR (i:Item) REQUIRE i.item_id is UNIQUE;
CREATE CONSTRAINT FOR (s:Seller) REQUIRE s.seller_id is UNIQUE;
CREATE CONSTRAINT FOR (s:State) REQUIRE s.code is UNIQUE;
CREATE CONSTRAINT FOR (c:City) REQUIRE c.name_state is UNIQUE;
CREATE CONSTRAINT FOR (c:Category) REQUIRE c.name is UNIQUE;
```

3.4. Import

Here is the complete cypher import script (the file is '/code/import.cypher'):

```
CREATE INDEX customer_order_id_index FOR (o:Order) ON o.customer_id;

//Order
LOAD CSV WITH HEADERS FROM "file:///olist_orders_dataset.csv" AS orders
WITH orders WHERE orders.order_id IS NOT NULL
CREATE (o:Orderorder_id: orders.order_id, customer_id: orders.customer_id,
purchase_timestamp: datetime( epochMillis: apoc.date.parse(orders.order_purchase_timestamp,
'ms', 'yyyy-MM-dd HH:mm:ss') ),
delivery_date: datetime( epochMillis: apoc.date.parse(orders.order_estimated_delivery_date,
'ms', 'yyyy-MM-dd HH:mm:ss') ));

//Seller
LOAD CSV WITH HEADERS FROM "file:///olist_sellers_dataset.csv" AS sellers
WITH sellers WHERE sellers.seller_id IS NOT NULL
CREATE (s:Sellerseller_id: sellers.seller_id);
```

```
//Customer
LOAD CSV WITH HEADERS FROM "file:///olist_customers_dataset.csv" AS customers
WITH customers WHERE customers.customer_id IS NOT NULL
AND customers.customer_unique_id IS NOT NULL
MERGE (c:Customercustomer_id: customers.customer_unique_id);

//City and State
LOAD CSV WITH HEADERS FROM "file:///olist_geolocation_dataset.csv" AS locations
WITH locations WHERE locations.geolocation_city IS NOT NULL
AND locations.geolocation_state IS NOT NULL
MERGE (c:Cityname_state: locations.geolocation_city + "-" + locations.geolocation_state)
ON CREATE SET c.name = locations.geolocation_city
MERGE (s:State code: locations.geolocation_state)
MERGE (c)-[:PART_OF]->(s);

//Item
LOAD CSV WITH HEADERS FROM "file:///olist_order_items_dataset.csv" AS items
WITH items WHERE items.order_id IS NOT NULL AND items.order_item_id IS NOT
NULL AND items.seller_id IS NOT NULL
CREATE (i:Itemitem_id: items.order_id + "-" + items.order_item_id,
price: toFloat(items.price));

//Product
LOAD CSV WITH HEADERS FROM "file:///olist_products_dataset.csv" AS products
WITH products WHERE products.product_id IS NOT NULL
CREATE (p:Productproduct_id: products.product_id,
photos_qty: toInteger(products.product_photos_qty),
weight_g: toInteger(products.product_weight_g),
length_cm: toInteger(products.product_length_cm),
height_cm: toInteger(products.product_height_cm),
width_cm: toInteger(products.product_width_cm))
MERGE (c:Categoryname: coalesce(products.product_category_name,"NA"));
```

```
//Review
LOAD CSV WITH HEADERS FROM "file:///olist_order_reviews_dataset.csv" AS reviews
WITH reviews WHERE reviews.review_id IS NOT NULL
MERGE (r:Reviewreview_id: reviews.review_id)
ON CREATE SET
r.score = toInteger(reviews.review_score),
r.comment_title = CASE WHEN trim(reviews.comment_title) = "" THEN null ELSE reviews.comment_title END,
r.comment = CASE WHEN trim(reviews.review_comment_message) = "" THEN null ELSE reviews.review_comment_message END,
r.creation_date = datetime( epochMillis: apoc.date.parse(reviews.review_creation_date, 'ms', 'M/d/yyyy H:mm') ),
r.answer_timestamp = datetime( epochMillis: apoc.date.parse(reviews.review_answer_timestamp, 'ms', 'M/d/yyyy H:mm') );

//——Relations——
LOAD CSV WITH HEADERS FROM "file:///olist_customers_dataset.csv" AS customers_rel
WITH customers_rel WHERE customers_rel.customer_id IS NOT NULL AND customers_rel.customer_unique_id IS NOT NULL
MATCH (c:Customercustomer_id: customers_rel.customer_unique_id)
MATCH (ci:Cityname_state: customers_rel.customer_city + "-" + customers_rel.customer_state)
MERGE (c)-[:LIVES_IN]->(ci);

LOAD CSV WITH HEADERS FROM "file:///olist_customers_dataset.csv" AS customers_rel
WITH customers_rel WHERE customers_rel.customer_id IS NOT NULL AND customers_rel.customer_unique_id IS NOT NULL
MATCH (c:Customercustomer_id: customers_rel.customer_unique_id)
MATCH (o:Ordercustomer_id: customers_rel.customer_id)
CREATE (c)-[:PLACED]->(o);

MATCH (o:Order)
```

```
REMOVE o.customer_id; //remove the attribute customer_id


DROP INDEX customer_order_id_index;


LOAD CSV WITH HEADERS FROM "file:///olist_sellers_dataset.csv" AS sellers_rel
WITH sellers_rel WHERE sellers_rel.seller_id IS NOT NULL
MATCH (s:Sellerseller_id: sellers_rel.seller_id)
MATCH (c:Cityname_state: sellers_rel.seller_city + "-" + sellers_rel.seller_state)
MERGE (s)-[:HAS_HEADQUARTERS_IN]->(c);


:auto LOAD CSV WITH HEADERS FROM "file:///olist_order_items_dataset.csv" AS
items_rel
CALL{
    WITH items_rel
    WITH items_rel WHERE items_rel.order_id IS NOT NULL AND
    items_rel.order_item_id IS NOT NULL AND items_rel.seller_id IS NOT NULL
    MATCH (i:Itemitem_id: items_rel.order_id + "-" + items_rel.order_item_id)
    MATCH (o:Orderorder_id: items_rel.order_id)
    MATCH (s:Sellerseller_id: items_rel.seller_id)
    MATCH (p:Productproduct_id: items_rel.product_id)
    MERGE (o)-[:COMPOSED_OF]->(i)
    MERGE (i)-[:SOLD_BY]->(s)
    MERGE (i)-[:CORRESPONDS_TO]->(p)
}IN TRANSACTIONS OF 500 ROWS;


LOAD CSV WITH HEADERS FROM "file:///olist_products_dataset.csv" AS prod-
ucts_rel
WITH products_rel WHERE products_rel.product_id IS NOT NULL
MATCH (p:Productproduct_id: products_rel.product_id)
MATCH (c:Categoryname: products_rel.product_category_name)
MERGE (p)-[:BELONGS_TO]->(c);


LOAD CSV WITH HEADERS FROM "file:///olist_order_payments_dataset.csv" AS
payments_rel
WITH payments_rel WHERE payments_rel.order_id IS NOT NULL
```

MATCH (o:Orderorder_id: payments_rel.order_id)
CREATE (p:Paymentvalue: toFloat(payments_rel.payment_value),
type: payments_rel.payment_type)
MERGE (o)-[:PAID_BY]->(p);

LOAD CSV WITH HEADERS FROM "file:///olist_order_reviews_dataset.csv" AS reviews_rel
WITH reviews_rel
MATCH (o:Orderorder_id: reviews_rel.order_id)
MATCH (r:Reviewreview_id: reviews_rel.review_id)
MERGE (o)-[:RECEIVED]->(r);

The import process is quite straightforward: for each csv file we imported the corresponding nodes and then created the relationships between them. The only thing to note is that at the begging the *customer_order_id* index is created to speed up the loading of the :Order nodes, since there is no constraint on *customer_id*. We just temporarily created this attribute to link :Order to the corresponding :Customer, because the *olist_orders_dataset.csv* doesn't for some reason use *customer_unique_id* to identify a customer. We then dropped the index and removed the attribute when all the :PLACED relationships were created.

4 | Queries

Here follows the list of our twenty queries (the file is '/code/queries.cypher'). Most of them are analytical, since the data is historical.

4.1. Category nation percentage

For each category the percentage of customers of each origin in descending order. E.g.: for "agro_industria_e_comercio" 45.28% of the clients are from São Paulo (SP), while 13.21% from Rio De Janeiro (RJ) and 9.91% from Minas Gerais.

```
MATCH (c:Category)<-[:BELONGS_TO]-(:Product)<-[:CORRESPONDS_TO]-(:Item)<-
[:COMPOSED_OF]-(:Order)<-[:PLACED]-(cu:Customer)
WITH c, toFloat(count(cu)) AS customer_number
MATCH (c)<-[:BELONGS_TO]-(:Product)<-[:CORRESPONDS_TO]-(:Item)<-[:COMPOSED
_OF]-(:Order)<-[:PLACED]-(cust:Customer),
(cust)-[:LIVES_IN]->(:City)-[:PART_OF]->(st:State)
WITH c.name AS Category, st.code AS State, customer_number, round(100*count(cust)/customer
_number, 2) AS Percentage
RETURN Category, State, Percentage
ORDER BY Category ASC, Percentage DESC
```

JSON output(first three rows):

```
{ "Category": "agro_industria_e_comercio", "State": "SP", "Percentage": 45.28 },
{ "Category": "agro_industria_e_comercio", "State": "RJ", "Percentage": 13.21 },
{ "Category": "agro_industria_e_comercio", "State": "MG", "Percentage": 9.91 }
```

4.2. Profit's distribution between States

For each seller, the percentage of profit that each nation generates in decreasing order. For example, the seller "0015a82c2db000af6aaaf3ae2ecb0532" receives 66.67% of their profit

from Minas Gerais and 33.33% from Paraná.

```
MATCH (s:Seller)<-[:SOLD_BY]-(i:Item)<-[:COMPOSED_OF]-(:Order)<-[:PLACED]-
(cu:Customer), (cu)-[:LIVES_IN]->(:City)-[:PART_OF]->(:State)
WITH s, sum(i.price) AS revenue
MATCH (s:Seller)<-[:SOLD_BY]-(it:Item)<-[:COMPOSED_OF]-(:Order)<-[:PLACED]-
(cust:Customer), (cust)-[:LIVES_IN]->(:City)-[:PART_OF]->(st:State)
WITH s.seller_id as Seller, st.code as State, revenue, round(100*sum(it.price)/revenue,
2) AS Percentage
RETURN Seller, State, Percentage
ORDER BY Seller ASC, Percentage DESC;
```

JSON output(first two rows):


```
{"Seller": "0015a82c2db000af6aaaf3ae2ecb0532", "State": "MG", "Percentage": 66.67 },
{"Seller": "0015a82c2db000af6aaaf3ae2ecb0532", "State": "PR", "Percentage": 33.33 }
```


4.3. Top 10 companies of the year

For each year the ranking of the 10 most profitable companies. The query returns for each year the collection of the seller and the corresponding profit.

```
MATCH (s:Seller)<-[:SOLD_BY]-(i:Item)<-[:COMPOSED_OF]-(o:Order)
WITH o.purchase_timestamp.year AS Year, s.seller_id AS Seller, round(sum(i.price), 2)
AS Revenue
ORDER BY Revenue DESC
RETURN Year, COLLECT([Seller, Revenue])[0..10] AS Sellers
ORDER BY Year DESC;
```

JSON output (first row):


```
{ "Year": 2018, "Sellers": [
[ "4869f7a5dfa277a7dca6462dcf3b52b2", 138414.6 ],
[ "955fee9216a65b617aa5c0531780ce60", 117340.86 ],
[ "7d13fca15225358621be4086e1eb0964", 113628.97 ],
[ "1025f0e2d44d7041d6cf58b6550e0bfa", 105196.71 ],
```

["fa1c13f2614d7b5c4749cbc52fecda94", 95013.42],
["7c67e1448b00f6e969d365cea6b010ab", 92746.9],
["da8622b14eb17ae2831f4ac5b9dab84a", 89450.47],
["4a3ca9315b744ce9f8e9374361493884", 72888.15],
["6560211a19b47992c3666cc44a7e94c0", 67823.93],
["a1043bafd471dff536d0c462352beb48", 61187.05]] }

4.4. Most profitable products of each company

For each company the percentage of profit coming from each product in descending order. For example, the JSON output tells us that the profit of the seller "0015a82c2db000af6aaaf3ae2ecb0532" comes all from the product "a2ff5a97bf95719e38ea2e3b4105bce8".

```
MATCH (:Product)<-[:CORRESPONDS_TO]-(i:Item)-[:SOLD_BY]->(s:Seller)
WITH s, sum(i.price) AS revenue
MATCH (p:Product)<-[:CORRESPONDS_TO]-(it:Item)-[:SOLD_BY]->(s:Seller)
WITH s.seller_id as Seller, p.product_id as Product, revenue, round(100*sum(it.price)/revenue,
2) AS Profit_percentage
RETURN Seller, Product, Profit_percentage
ORDER BY Seller ASC, Profit_percentage DESC
```

JSON output (first three rows):

{ "Seller": "0015a82c2db000af6aaaf3ae2ecb0532",
"Product": "a2ff5a97bf95719e38ea2e3b4105bce8",
"Profit_percentage": 100.0 },
{ "Seller": "001cca7ae9ae17fb1caed9dfb1094831",
"Product": "08574b074924071f4e201e151b152b4e",
"Profit_percentage": 42.9 },
{ "Seller": "001cca7ae9ae17fb1caed9dfb1094831",
"Product": "e251ebd2858be1aa7d9b2087a6992580",
"Profit_percentage": 22.3 }

4.5. Top ten products by reviews

For each category, the ten products with the highest average review scores. The query returns the products and the corresponding average score in a collection.

```
MATCH (cat:Category)<-[:BELONGS_TO]-(p:Product)<-[:CORRESPONDS_TO]-(:Item)<-
[:COMPOSED_OF]-(:Order)-[:RECEIVED]->(r:Review)
WITH cat.name AS Category, round(avg(r.score),2) as Average, p.product_id as Id
ORDER BY Average DESC
RETURN Category, COLLECT([Id, Average])[0..10] AS Products
ORDER BY Category ASC;
```

JSON output (first row):


```
{ "Category": "agro_industria_e_comercio", "Products": [
[ "1df1a2df8ad2b9d3aa49fd851e3145ad", 5.0 ],
[ "5f0a49e6e539d4e186eeddfd69db9863", 5.0 ],
[ "613d093272cb8f74f25a01e430155a6a", 5.0 ],
[ "07f01b6fcacc1b187a71e5074199db2d", 5.0 ],
[ "ffd63ee42a5c8cc5a15a1c8e2aa50011", 5.0 ],
[ "cd2f5c10e4e8dbc701f0bb68a09fdfe8", 5.0 ],
[ "1ff03883acc92ad1f4eb8ed40ad25bf9", 5.0 ],
[ "c75a6578b6475cf760d40cb340b3971c", 5.0 ],
[ "f3c179e260e0eeffbe02340259404cb1", 5.0 ],
[ "24fee4a800146a47846fa0e345b6d6ed", 5.0 ] ] }
```


4.6. Most profitable month

For each seller the month when they earn the most and the corresponding profit.

```
MATCH (o:Order)-[:COMPOSED_OF]->(i:Item)-[:CORRESPONDS_TO]->(p:Product),
(i)-[:SOLD_BY]->(s:Seller)
WITH s.seller_id AS Seller, o.purchase_timestamp.month AS Month, round(sum(i.price),
2) AS Revenue
ORDER BY Revenue DESC
WITH Seller, COLLECT(Month)[0] AS Month, COLLECT(Revenue)[0] AS Revenue
```

```
RETURN Seller, ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
"Oct", "Nov", "Dec"][Month-1] AS Month, Revenue
ORDER BY Seller ASC;
```

JSON Output:


```
{ "Seller": "0015a82c2db000af6aaaf3ae2ecb0532", "Month": "Oct", "Revenue": 1790.0
},
{ "Seller": "001cca7ae9ae17fb1caed9dfb1094831", "Month": "Mar", "Revenue": 2862.6
},
{ "Seller": "001e6ad469a905060d959994f1b41e4f", "Month": "Aug", "Revenue": 250.0 }
```


4.7. Selling frequency

For each product, the average number of times it is bought per month. The frequency is calculated as the number of times a product was bought divided by the number of days between the first order of that product and the last update of the database. We assumed that this date corresponds to the day after the day the last order was placed.

```
MATCH (o:Order)
WITH datetime.truncate('day', max(o.purchase_timestamp)) AS Now
MATCH (p:Product)<-[:CORRESPONDS_TO]-(Item)<-[:COMPOSED_OF]-(o:Order)
WITH p.product_id AS Product, Now, COUNT(o) as Count, toFloat(Now.epochSeconds+
24*3600-datetime.truncate('day', min(o.purchase_timestamp)).epochSeconds) AS TimeS-
pan
RETURN Product, round((30*24*3600*Count)/TimeSpan, 2) AS Frequency
ORDER BY Frequency DESC;
```

JSON output (first three rows):


```
{ "Product": "aca2eb7d00ea1a7b8ebd4e68314663af", "Frequency": 35.13 },
{ "Product": "e7cc48a9daff5436f63d3aad9426f28b", "Frequency": 31.7 },
{ "Product": "3dd2a17168ec895c781a9191c1e95ad7", "Frequency": 29.25 }
```

4.8. Payments method percentage

For each company, the percentages of profit coming from payments made by credit card, cash (boleto) and other methods.

```
MATCH (p:Payment)<-[:PAID_BY]-(o:Order)-[:COMPOSED_OF]->(:Item)-[:SOLD_BY]-
>(s:Seller)
WITH DISTINCT o.purchase_timestamp.year AS Year, s.seller_id AS Seller, SUM(p.value)
AS Total
WITH Seller, Year, Total, COLLECT{
    MATCH (p:Payment)<-[:PAID_BY]-(o:Order)-[:COMPOSED_OF]->(:Item)-[:SOLD_BY]-
    >(s:Seller)
    WHERE p.type = "credit_card" AND s.seller_id = Seller AND o.purchase_timestamp.year=
    Year
    WITH s, Total, 100*SUM(p.value)/Total as Credit_Card_Percentage
    RETURN Credit_Card_Percentage
}[0] AS Credit_Card, COLLECT{
    MATCH (p:Payment)<-[:PAID_BY]-(o:Order)-[:COMPOSED_OF]->(:Item)-[:SOLD_BY]-
    >(s:Seller)
    WHERE p.type = "boleto" AND s.seller_id = Seller AND o.purchase_timestamp.year=Year
    WITH s, Total, 100*SUM(p.value)/Total as Boleto_Percentage
    RETURN Boleto_Percentage
}[0] AS Boleto
RETURN Seller, Year, round(Credit_Card, 2) AS Credit_Card_Percentage, round(Boleto,
2) AS Boleto_Percentage, round(100-(Credit_Card+Boleto), 2) AS Other_percentage
ORDER BY Seller ASC, Year DESC
```

JSON output (first three rows):


```
{ "Seller": "0015a82c2db000af6aaaf3ae2ecb0532", "Year": 2017, "Credit_Card_Percentage":
100.0, "Boleto_Percentage": null, "Other_percentage": null },
{ "Seller": "001cca7ae9ae17fb1caed9dfb1094831", "Year": 2018, "Credit_Card_Percentage":
51.74, "Boleto_Percentage": 45.89, "Other_percentage": 2.36 },
{ "Seller": "001cca7ae9ae17fb1caed9dfb1094831", "Year": 2017, "Credit_Card_Percentage":
78.41, "Boleto_Percentage": 20.1, "Other_percentage": 1.49 }
```

4.9. Relationship between number of photos and profit

The revenue, the number of sells and the average price of products based on their number of published photos. For example, the output tells us that only one product with twenty photos was sold and that it has generated 110.27 real of profit.

```
MATCH (i:Item)-[:CORRESPONDS_TO]->(pr:Product)
WITH DISTINCT pr.photos_qty AS Photos_Quantity, round(sum(i.price),2) AS Revenue, count(i) AS Sold_Products
RETURN Photos_Quantity, Revenue, Sold_Products, round(Revenue/Sold_Products,2) AS Average_Price
ORDER BY Photos_Quantity DESC;
```

JSON Output (first three rows):

{ "Photos_Quantity": null, "Revenue": 179535.28, "Sold_Products": 1603, "Average_Price": 112.0 },
{ "Photos_Quantity": 20, "Revenue": 110.27, "Sold_Products": 1, "Average_Price": 110.27 },
{ "Photos_Quantity": 19, "Revenue": 277.88, "Sold_Products": 2, "Average_Price": 138.94 }

4.10. Product of the year

For each year, the most purchased product and the number of purchases.

```
MATCH (p:Product)<-[:CORRESPONDS_TO]-(:Item)<-[:COMPOSED_OF]-(o:Order)
WITH DISTINCT o.purchase_timestamp.year AS Year, p.product_id AS Product, COUNT(p) AS Quantity_Sold
ORDER BY Quantity_Sold DESC
RETURN Year, COLLECT(Product)[0] AS Product, COLLECT(Quantity_Sold)[0] AS Quantity_Sold
ORDER BY Year DESC;
```

JSON output:

{ "Year": 2018, "Product": "aca2eb7d00ea1a7b8ebd4e68314663af", "Quantity_Sold": 413 },
{ "Year": 2017, "Product": "99a4788cb24856965c36a24e339b6058", "Quantity_Sold": 359 },
{ "Year": 2016, "Product": "eba7488e1c67729f045ab43fac426f2e", "Quantity_Sold": 11 }

4.11. Last ten products purchased

For each customer the last 10 products purchased. For example, the output below tells us that the three customers have only placed one order each, so the "Products" collections contain only one *product_id* each.

```
MATCH (c:Customer)-[:PLACED]->(o:Order)-[:COMPOSED_OF]->(:Item)-
[:CORRESPONDS_TO]->(p:Product)
WITH DISTINCT c.customer_id AS Customer, p.product_id AS Product, o.purchase_timestamp
AS TimeStamp
ORDER BY TimeStamp DESC
RETURN Customer, COLLECT(Product)[0..10] AS Products
ORDER BY Customer ASC;
```

JSON output (first three lines):

{ "Customer": "0000366f3b9a7992bf8c76cfdf3221e2",
"Products": ["372645c7439f9661fbbacfd129aa92ec"] },
{ "Customer": "0000b849f77a49e4a4ce2b2a4ca5be3f",
"Products": ["5099f7000472b634fea8304448d20825"] },
{ "Customer": "0000f46a3911fa3c0805444483337064",
"Products": ["64b488de448a5324c4134ea39c28a34b"] }

4.12. Last ten products sold

For each seller the last 10 products sold. To differentiate this query from the previous one, we returned the results in separate rows instead of gathering them in a collection. From the

output it appears that the last three orders of the seller "0015a82c2db000af6aaaf3ae2ecb0532" all contain the product "a2ff5a97bf95719e38ea2e3b4105bce8".

```
MATCH (o:Order)-[:COMPOSED_OF]->(i:Item)-[:CORRESPONDS_TO]->(p:Product),
(i)-[:SOLD_BY]->(s:Seller)
WITH s.seller_id AS Seller, p.product_id AS Products, o.purchase_timestamp AS TimeS-
tamp
ORDER BY TimeStamp DESC
WITH Seller, COLLECT(Products)[0..10] AS Collection
UNWIND Collection AS Product
RETURN Seller, Product
ORDER BY Seller ASC;
```

JSON output (first three rows):

```
{ "Seller": "0015a82c2db000af6aaaf3ae2ecb0532",
"Product": "a2ff5a97bf95719e38ea2e3b4105bce8" },
{ "Seller": "0015a82c2db000af6aaaf3ae2ecb0532",
"Product": "a2ff5a97bf95719e38ea2e3b4105bce8" },
{ "Seller": "0015a82c2db000af6aaaf3ae2ecb0532",
"Product": "a2ff5a97bf95719e38ea2e3b4105bce8" }
```

4.13. Average product's weight of each city

The average weight of the items shipped to each city. From the output it appears that the average weight of the products shipped to "itabera" is more than ten times the average weight of the items shipped to "santa clara d'oeste".

```
MATCH (p:Product)<-[:CORRESPONDS_TO]-(:Item)<-[:COMPOSED_OF]-(:Order)<-
[:PLACED]-(:Customer)-[:LIVES_IN]->(c:City)-[:PART_OF]->(s:State)
WITH c, s.code AS State, round(avg(p.weight_g),2) AS Target_Weight_g
RETURN c.name AS City, State, Target_Weight_g;
```

JSON output (first three rows):

{ "City": "palmares paulista", "State": "SP", "Target_Weight_g": 600.0 },
{ "City": "itabera", "State": "SP", "Target_Weight_g": 3881.25 },
{ "City": "santa clara d'oeste", "State": "SP", "Target_Weight_g": 200.0 }

4.14. Orders' provenience

For each customer the percentage of the State provenience of their orders. For example,the output tells us that the customer "0000366f3b9a7992bf8c76cfdf3221e2" has only bought from one seller that has headquarters in São Paulo.

```
MATCH (c:Customer)-[:PLACED]->(:Order)-[:COMPOSED_OF]->(:Item)-[:SOLD_BY]-
>(se:Seller)
WITH c, count(se) AS Sellers_Number
MATCH (c)-[:PLACED]->(:Order)-[:COMPOSED_OF]->(:Item)-[:SOLD_BY]->(se:Seller),
(se)-[:HAS_HEADQUARTERS_IN]->(:City)-[:PART_OF]->(st:State)
RETURN c.customer_id AS Customer, st.code AS State, Sellers_Number,
round(100*toFloat(count(se))/Sellers_Number,2) AS Percentage
ORDER BY Customer ASC, Percentage DESC;
```

JSON output (first three rows):

{ "Customer": "0000366f3b9a7992bf8c76cfdf3221e2", "State": "SP", "Sellers_Number":
1, "Percentage": 100.0 },
{ "Customer": "0000b849f77a49e4a4ce2b2a4ca5be3f", "State": "SP", "Sellers_Number":
1, "Percentage": 100.0 },
{ "Customer": "0000f46a3911fa3c0805444483337064", "State": "SP", "Sellers_Number":
1, "Percentage": 100.0 }

4.15. Worst product

For each seller the product with the lowest reviews' score.

```
MATCH (s:Seller)<-[:SOLD_BY]-(:Item)-[:CORRESPONDS_TO]->(p:Product),
(p)<-[:CORRESPONDS_TO]-(:Item)<-[:COMPOSED_OF]-(:Order)-[:RECEIVED]->(r:Review)
WITH s.seller_id AS Seller, p.product_id AS Products, round(avg(toFloat(r.score)), 2)
```

```
AS Average
ORDER BY Average ASC
RETURN Seller, COLLECT(Products)[0] AS Product, min(Average) AS Score
ORDER BY Seller ASC;
```

JSON output (first three rows):


```
{ "Seller": "0015a82c2db000af6aaaf3ae2ecb0532",
"Product": "a2ff5a97bf95719e38ea2e3b4105bce8",
"Score": 3.67 },
{ "Seller": "001cca7ae9ae17fb1caed9dfb1094831",
"Product": "86b22a03cb72239dd53996a67df35c63",
"Score": 2.67 },
{ "Seller": "002100f778ceb8431b7a1020ff7ab48f",
"Product": "2ceff1056fe5bd4ee443433402962fa4",
"Score": 3.0 }
```


4.16. Average size and weight

For each seller the average size and weight of the products they sold.

```
MATCH (p:Product)<-[:CORRESPONDS_TO]-(:Item)-[:SOLD_BY]->(s:Seller)
RETURN s.seller_id AS Seller, round(avg(p.width_cm),2) AS AVG_width_cm,
round(avg(p.height_cm),2) AS AVG_height_cm, round(avg(p.length_cm),2)
AS AVG_length_cm, round(avg(p.weight_g),2) AS AVG_weight_g;
```

JSON output (first three rows):


```
{ "Seller": "3442f8959a84dea7ee197c632cb2df15", "AVG_width_cm": 26.0,
"AVG_height_cm": 11.67, "AVG_length_cm": 31.33, "AVG_weight_g": 533.33 },
{ "Seller": "d1b65fc7debc3361ea86b5f14c68d2e2", "AVG_width_cm": 27.88,
"AVG_height_cm": 51.93, "AVG_length_cm": 39.61, "AVG_weight_g": 6189.02 },
{ "Seller": "ce3ad9de960102d0677a81f5d0bb7b2d", "AVG_width_cm": 26.0,
"AVG_height_cm": 6.0, "AVG_length_cm": 35.0, "AVG_weight_g": 400.0 }
```

4.17. Favorite company

For each customer, the company from which they bought the most.

```
MATCH (c:Customer)-[:PLACED]->(o:Order)-[:COMPOSED_OF]->(:Item)-[:SOLD_BY]-
>(s:Seller)
WITH DISTINCT c.customer_id AS Customer, s.seller_id AS Seller, COUNT(DISTINCT(o))
as Number_Orders
ORDER BY Number_Orders DESC
RETURN Customer, COLLECT(Seller)[0] AS Seller, COLLECT(Number_Orders)[0] AS
Number_Of_Orders
ORDER BY Customer ASC;
```

JSON output (first three rows):


```
{ "Customer": "0000366f3b9a7992bf8c76cfdf3221e2",
"Seller": "da8622b14eb17ae2831f4ac5b9dab84a", "Number_Of_Orders": 1 },
{ "Customer": "0000b849f77a49e4a4ce2b2a4ca5be3f",
"Seller": "138dbe45fc62f1e244378131a6801526", "Number_Of_Orders": 1 },
"Customer": "0000f46a3911fa3c0805444483337064",
"Seller": "3d871de0142ce09b7081e2b9d1733cb1", "Number_Of_Orders": 1 }
```


4.18. User's reviews

For each customer, the average score of the reviews that they wrote, the count, and the category they reviewed the most. For example, the customer "8d50f5eadf50201ccdcedfb9e2ac8455" has written seventeen reviews so far, with an average score of 4.76, and most of their reviews are for products in the category "esporte_lazer".

```
MATCH (c:Customer)-[:PLACED]->(:Order)-[:RECEIVED]->(re:Review)
WITH c.customer_id AS Customer, round(avg(toFloat(re.score)), 2) as AVG_Score,
count(re) AS Count
RETURN Customer, AVG_Score, Count, COLLECT{
    MATCH (c:Customer)-[:PLACED]->(o:Order)-[:RECEIVED]->(:Review)
    WHERE c.customer_id = Customer
    MATCH (cat:Category)<-[:BELONGS_TO]-(:Product)<-[:CORRESPONDS_TO]-
```

```
    (:Item)<- [:COMPOSED_OF]-(o:Order)
    WITH cat.name AS Category, count(o) AS Order_Count
    RETURN Category
    LIMIT 1
}[0] AS Most_Reviewed_Category
ORDER BY Count DESC;
```

JSON output (first three rows):

{ "Customer": "8d50f5eadf50201ccdcedfb9e2ac8455", "AVG_Score": 4.76, "Count": 17, "Most_Reviewed_Category": "esporte_lazer" },
{ "Customer": "3e43e6105506432c953e165fb2acf44c", "AVG_Score": 2.78, "Count": 9, "Most_Reviewed_Category": "cama_mesa_banho" },
"Customer": "47c1a3033b8b77b3ab6e109eb4d5fdf3", "AVG_Score": 4.86, "Count": 7, "Most_Reviewed_Category": "beleza_saude" }

4.19. Top reviewed company

For each year, the best company based on the average score of the reviews about its products .

```
MATCH (o:Order)-[:RECEIVED]->(r:Review),
(o)-[:COMPOSED_OF]->(:Item)-[:SOLD_BY]->(s:Seller)
WITH DISTINCT o.purchase_timestamp.year AS Year, s.seller_id AS Seller,
round(avg(toFloat(r.score)), 2) AS Score
ORDER BY Score DESC
RETURN Year, COLLECT(Seller)[0] AS Seller, COLLECT(Score)[0] AS Score
ORDER BY Year DESC;
```

JSON output:

{ "Year": 2018, "Seller": "d1b65fc7debc3361ea86b5f14c68d2e2", "Score": 5.0 },
{ "Year": 2017, "Seller": "166e8f1381e09651983c38b1f6f91c11", "Score": 5.0 },
{ "Year": 2016, "Seller": "f7496d659ca9fdaf323c0aae84176632", "Score": 5.0 }

4.20. Expenses for the year

For each customer, how much they spent every year. For example customer "0000366f3b9a7992bf8c76cfdf3221e2" spent 141.9 reals in 2018.

MATCH (c:Customer)-[:PLACED]->(o:Order)-[:PAID_BY]->(p:Payment)
RETURN DISTINCT o.purchase_timestamp.year AS Year, c.customer_id AS Customer, sum(p.value) AS Total_Spending
ORDER BY Customer ASC, Year DESC;

JSON output (first three rows):

{ "Year": 2018, "Customer": "0000366f3b9a7992bf8c76cfdf3221e2", "Total_Spending": 141.9 },
{ "Year": 2018, "Customer": "0000b849f77a49e4a4ce2b2a4ca5be3f", "Total_Spending": 27.19 },
{ "Year": 2017, "Customer": "0000f46a3911fa3c0805444483337064", "Total_Spending": 86.22 }