Amazone Database

Group Report

Submitted to: Dr Rotimi Ogunsakin

Student IDs:

11102696 | 10357571 | 11164061 | 10451254 | 11118569 | 11155827 | 11063737

Date: 13th Jan 2023

DATA70141 | Understanding Databases

MSc Data Science

Introduction

This report aims to outline the process and considerations for creating a NoSQL database for an online delivery system for Amazone. This report will cover the benefits of using NoSQL, data model, data storage and retrieval aspects of the database. This report covers task allocations between the team, a NoSQL schema for the database along with the rationale behind its design, an application for the delivery system and sample implementation results of the application.

Task Division

Task	Description	Responsibility
Schema Design	Schema Design Proposal	Whole Team
Customers	Data Population	Rayan
Current OrdersFresh		Soomin
Current Orders Other		Soomin
RecommendedProducts		Ali
Past Order		Soomin
Partners		Rayan
Partner Status		Yanfei
Warehouses		Hadhry
DailyInventoryLevel		Devesh
Average Product Ratings		Ahsan
Partner Ratings		Ahsan
Product		Ali
Shopping basket		Soomin
Order Status		Ali
GeoJson	Obtaining coordinates based on postcodes	Rayan
Shipping Cost	Calculating and assigning shipping cost	Rayan + Soomin
Recommendation	Recommendation calculation, data update	Ali
Find Drivers	Implement delivery driver selection algorithm and calculate ETA	Hadhry
Add to basket (cart)	Adding items to the cart	Soomin
Find Items Available at Morrisons	Check Morrissons stock	Hadhry + Devesh
Calculating shipping costs	Calculate shipping costs for shopping baskets, current orders, and past orders	Rayan + Soomin
Basket to Order	Making a purchase, moving items from basket to current order collection	Soomin

Stock Check	Check if there is stock before checkout and update the warehouse	Rayan
Managerial Queries	Four functions that are used for data analysis and data visualisation	Ahsan
Function Compilation	Compile all the functions and make one single application for Amazone	Hadhry + Devesh
Data Compilation	Compile everyone's notes on their specific functions	Yanfei + Devesh
Task 1 writeup	Write about the schema and design decisions	Rayan + Yanfei
Complete mandatory queries	Run the function to obtain query results	Hadhry
Pipeline queries	Create extra pipeline queries	Ali + Ahsan
Data Visualisation Queries	Prepare query results for data visualisation	Ahsan
Compile and write the report	Finish the report	Everyone

Table 1 YesQL Task Allocation

Assumptions

- The data that has been manually implemented has gone through all the procedures of the application at the point of entry, eg, DailyInventoryLevel has been adjusted for all current and past orders.
- All data has been entered randomly, including addresses of customers, and no analysis or predictions should be made on the existing data; it is strictly for demonstration purposes.
- All partners drive a vehicle with the capacity to take all their orders.

Schema

NoSQL databases provide flexibility regarding schema design, which means they can handle unstructured and semi-structured data. NoSQL databases are known for their simplicity compared to traditional relational databases, making them easier to manage (Sadalage and Fowler, 2013).

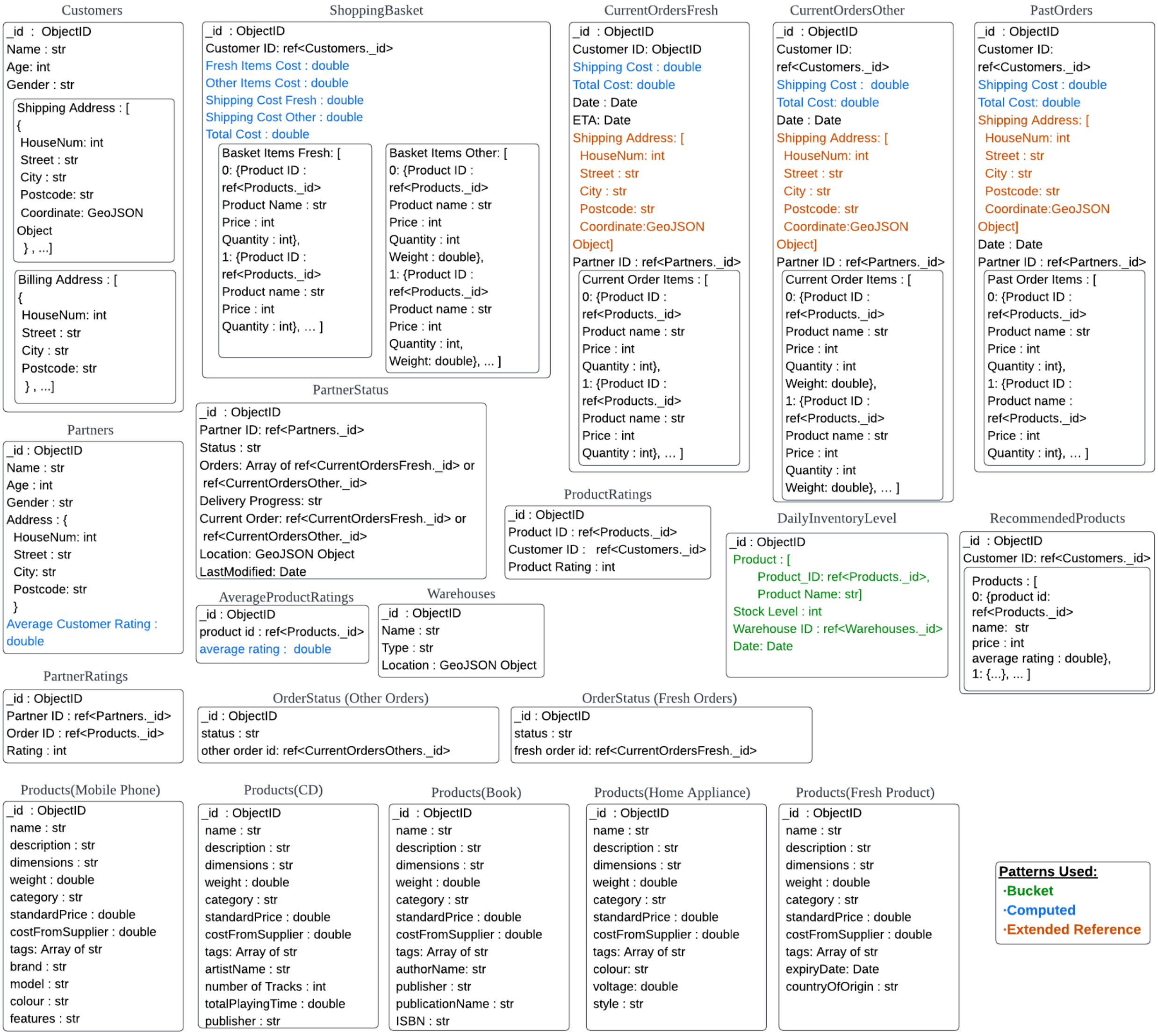
Taking advantage of this flexibility, a multiple schema design was implemented for this database. A multiple schema design allows for flexible data modelling as different data models can be used in different parts of the database. It increases data integrity by having multiple schemas it can enforce different constraints on the data. Furthermore, distinct schemas can be optimised for different queries, thus improving the performance of the database, and if required, can be partitioned within different databases efficiently (Sullivan, 2015).

The schema has 15 collections. The way the collections are divided is based on how frequently they will be queried or updated. Table 2 shows the frequently rewritten collections against the primarily read-only collections.

Frequently Read	Frequently Written
Products	Past Orders
Order Status	Order Status
Partner Status	Partner Status
Warehouses	Shopping Basket
Customers	Daily Inventory Level
Recommended Products	Current Orders Fresh
ProductRatings	Current Orders Others
Partner Ratings	
Current Orders Fresh	
Current Orders Others	

Table 2 Read/Write Collection Distribution

Frequently rewritten collections were separated into their collection to reduce querying complexity. This improves the performance of the database by skipping the need to traverse through multiple levels of nested data (Yoon et al., 2016). Collections like ‘Order Status’ and ‘Partner Status’ that require real-time collection tracking are refreshed every time a customer reloads their application and are therefore separated from the Current Orders collections. Partner status is also separated from ‘Partners’ as it is unnecessary to read through all partners’ details to see their current location. Product IDs are documented in Current Orders; however, the customer may want to query more product details when looking at their basket; therefore, an extended referencing pattern was used to avoid referencing the entire document every time a query is made.

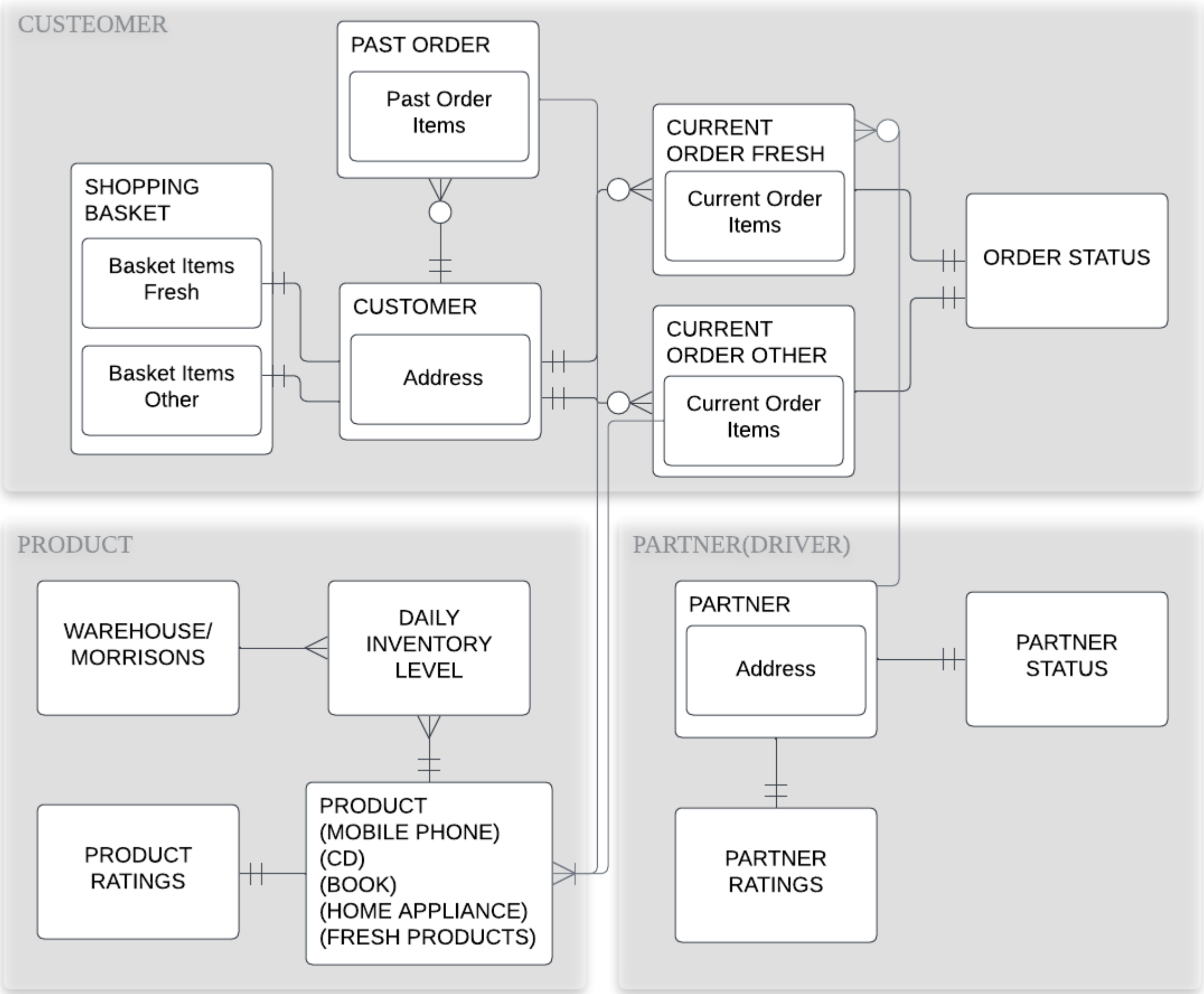


See JSON formatted Schema in the appendix.

Shipping address has been frequently queried, as it is shown in both current orders and past orders. 2DSPHERE indexes were applied to the location attributes of Warehouse and PartnerStatus locations to allow radial geospatial operations. A computed pattern is used for calculating average ratings and total costs to prevent the system from performing the calculation every time a query is performed.

The different product characteristics were not embedded into a single product document; instead, multiple schemas were adopted to accommodate the different attributes of each product. Embedding documents can make data modelling more complex and result in a less intuitive data model. NoSQL does not have a limitation on one fixed schema, and a multiple schema design has less complexity with fewer embedded documents (MongoDB, 2021).

ERD



Delivery Application

To make a working application that can be used for querying based on the needs of the customer and the company, several functions are created with the help of python and aggregation pipelines. All the functions used for querying are as follows:

1. **addToBasket**(CustomerID, ProductID, quantity): This function is designed to allow customers to add items to their basket. It uses the ShoppingBasket collection. Multiple conditions are used, like inserting a new basket, updating an old basket, and dividing the products into fresh & other categories.
2. **Shipping_cost_fresh** (CustomerID): This function is designed to compute the shipping cost for fresh products. For orders exceeding £40 in value, no shipping cost is applied.
3. **Shipping_cost_other** (CustomerID): The shipping cost of other products is calculated based on an index using weight, price & quantity and divided into four categories with a $5 increment from the second case with the increasing index value.

4. **FindMorrisonWithStock**(item_id, quantity): This function is created to find the nearest Morrison with listed products.
5. **FindPartner**(WarehouseCoord): This function finds the nearest active partner to the warehouse with the customer's ordered products in stock using geospatial queries.
6. **AssignOrder**(partnerID, orderID): This function assigns a partner with orders, updating partner status collection. The ETA increases by 30 minutes for each order.
7. **orderStatusUpdate**(orderID, orderType): This function updates the order status for the customer. Order status is divided into two categories based on orderType(0 for fresh, 1 for other products) attribute inside OrderStatusUpdate() function.
8. **basketToOrder**(customerID): Move the successfully purchased products into the CurrentOrder Collection.
9. **Total_Sales_Over_Time**(PastOrders,year): This function plots total sales over a particular year. It first calls the collection and converts it into a pandas data frame by normalising the JSON output. Then it selects the rows belonging to a particular year and sum the sales by month. Finally, the graph is plotted for total monthly sales, which helps a manager to see the seasonal trends and sales functions during a year.
10. **Top_10_Products_by_Revenue**(PastOrders): The function converts MongoDB collection into a pandas data frame and calculates the total revenue generated by each product over time. The function then sorts the products by revenue and picks the top 10 products for the plot.
11. **Total_Revenue_per_Customer**(PastOrders): It also calls customer collection and applies a left join on customer id to get customer names and then sum the total purchases of each customer.

12. **Revenue_by_tag**(db.Products, 'Top' or 'Lowest'): This function helps a manager to understand sales in a better way, e.g., if some products have a tag called ''Vegetarian'', then it is for the manager to separate all the veg products for data analysis. 'Top' for plotting the top 10 tags by revenue, and 'Lowest' for plotting the lowest 10 tags by revenue.

13. **Rating Averaging Function:** For new customers with no order history, the recommender function adds the two top-rated products to their recommendations. For the users with order history, tags are used to determine recommended products. For each customer, the tags are ordered according to how many products that customer has bought with that tag. The products matching the most frequent tag are then found, and two are added to the recommendations. If less than two products are found, then the next most frequent tag is used, until two products can be added to 'RecommendedProducts'. If two products are still not found after going through all tags, then the top-rated products are suggested.

14. **Recommender Function**: Using an aggregation pipeline on the ProductRatings collection, the ratings are grouped according to product id. For each product, the total sum of all ratings given to that product and the number of ratings are returned. From this, we can calculate the average rating for each product. All average ratings are rounded to 2 decimal places. A document is created if the product has not already got an average rating in the collection. Otherwise, the existing document is updated. The first step of the aggregation pipeline is to unwind the array field so that the recommended products are in separate documents. Then the documents are grouped by the Product ID, and the number of documents in which the Product ID appears is counted. Then the aggregated documents are sorted in descending order according to the count and limited, so only the first 5 documents are outputted.

Queries and sample results

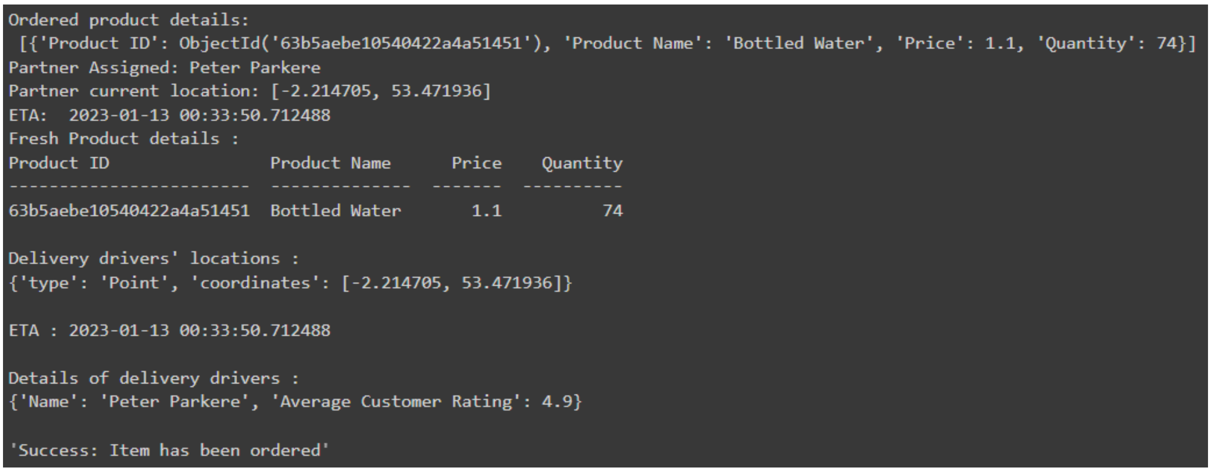
A query showing a customer ordering a fresh product, getting assigned a driver based on location and being given an order and order status.

Query 1: Customer "Markos Voss" (_id: ObjectId('63b8707d066488245e595987')) purchases his fresh product shopping basket containing 74 'Bottled Water’.

Input

basketToOrder(ObjectId('63b8707d066488245e595987'))

Output



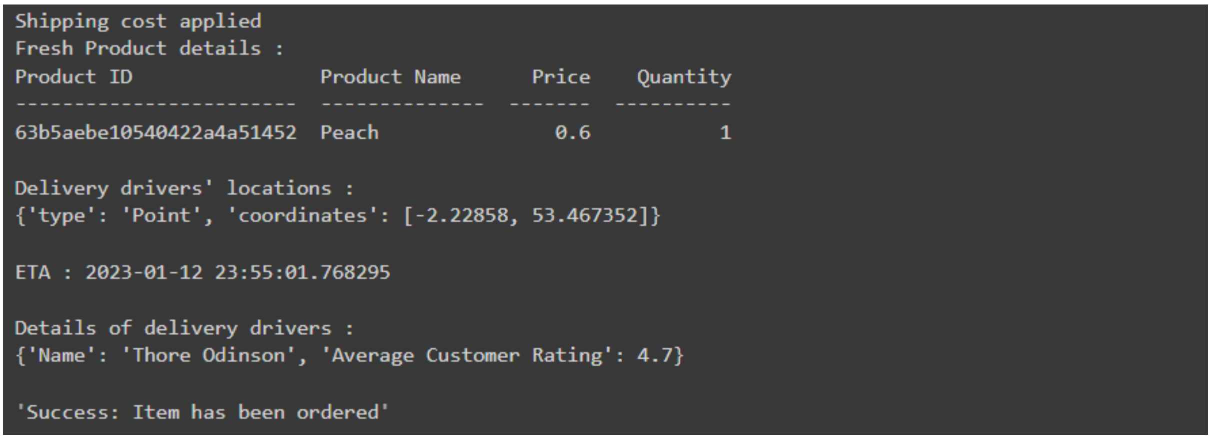
```
Ordered product details:
 [{'Product ID': ObjectId('63b5aebe10540422a4a51451'), 'Product Name': 'Bottled Water', 'Price': 1.1, 'Quantity': 74}]
Partner Assigned: Peter Parkere
Partner current location: [-2.214705, 53.471936]
ETA:  2023-01-13 00:33:50.712488
Fresh Product details :
Product ID                Product Name      Price    Quantity
------------------------  --------------  -------  ----------
63b5aebe10540422a4a51451  Bottled Water       1.1          74

Delivery drivers' locations :
{'type': 'Point', 'coordinates': [-2.214705, 53.471936]}

ETA : 2023-01-13 00:33:50.712488

Details of delivery drivers :
{'Name': 'Peter Parkere', 'Average Customer Rating': 4.9}

'Success: Item has been ordered'
```

Query 2: Customer "Plinius Dubicki" (_id: ObjectId("63b8707d066488245e595986")) adds 1 'Peach' to her basket and proceeds to purchase it.

Input

addToBasket(ObjectId("63b8707d066488245e595986"),ObjectId('63b5aebe10540422a4a51452'), 1)
basketToOrder(ObjectId("63b8707d066488245e595986"))

Output



```
Shipping cost applied
Fresh Product details :
Product ID                Product Name      Price    Quantity
------------------------  --------------  -------  ----------
63b5aebe10540422a4a51452  Peach               0.6           1

Delivery drivers' locations :
{'type': 'Point', 'coordinates': [-2.22858, 53.467352]}

ETA : 2023-01-12 23:55:01.768295

Details of delivery drivers :
{'Name': 'Thore Odinson', 'Average Customer Rating': 4.7}

'Success: Item has been ordered'
```

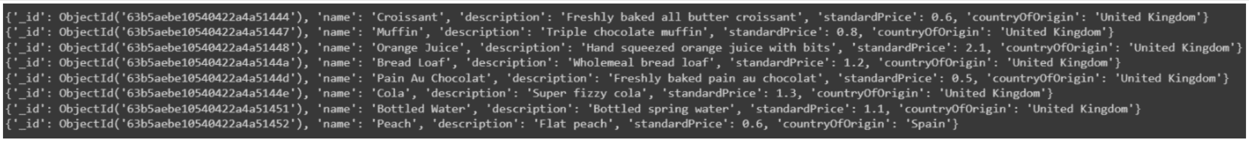
A customer querying for fresh products and getting availability based on their location.

Query 3: ‘John Doe’ searches for fresh products currently available to order to his address.

Input

AvailableFreshProducts(ObjectId("63b8707d066488245e595983"))

Output



```
{'_id': ObjectId('63b5aebe10540422a4a51444'), 'name': 'Croissant', 'description': 'Freshly baked all butter croissant', 'standardPrice': 0.6, 'countryOfOrigin': 'United Kingdom'}
{'_id': ObjectId('63b5aebe10540422a4a51447'), 'name': 'Muffin', 'description': 'Triple chocolate muffin', 'standardPrice': 0.8, 'countryOfOrigin': 'United Kingdom'}
{'_id': ObjectId('63b5aebe10540422a4a51448'), 'name': 'Orange Juice', 'description': 'Hand squeezed orange juice with bits', 'standardPrice': 2.1, 'countryOfOrigin': 'United Kingdom'}
{'_id': ObjectId('63b5aebe10540422a4a5144a'), 'name': 'Bread Loaf', 'description': 'Wholemeal bread loaf', 'standardPrice': 1.2, 'countryOfOrigin': 'United Kingdom'}
{'_id': ObjectId('63b5aebe10540422a4a5144d'), 'name': 'Pain Au Chocolat', 'description': 'Freshly baked pain au chocolat', 'standardPrice': 0.5, 'countryOfOrigin': 'United Kingdom'}
{'_id': ObjectId('63b5aebe10540422a4a5144e'), 'name': 'Cola', 'description': 'Super fizzy cola', 'standardPrice': 1.3, 'countryOfOrigin': 'United Kingdom'}
{'_id': ObjectId('63b5aebe10540422a4a51451'), 'name': 'Bottled Water', 'description': 'Bottled spring water', 'standardPrice': 1.1, 'countryOfOrigin': 'United Kingdom'}
{'_id': ObjectId('63b5aebe10540422a4a51452'), 'name': 'Peach', 'description': 'Flat peach', 'standardPrice': 0.6, 'countryOfOrigin': 'Spain'}
```

Customer adding a product to a cart and making payment.

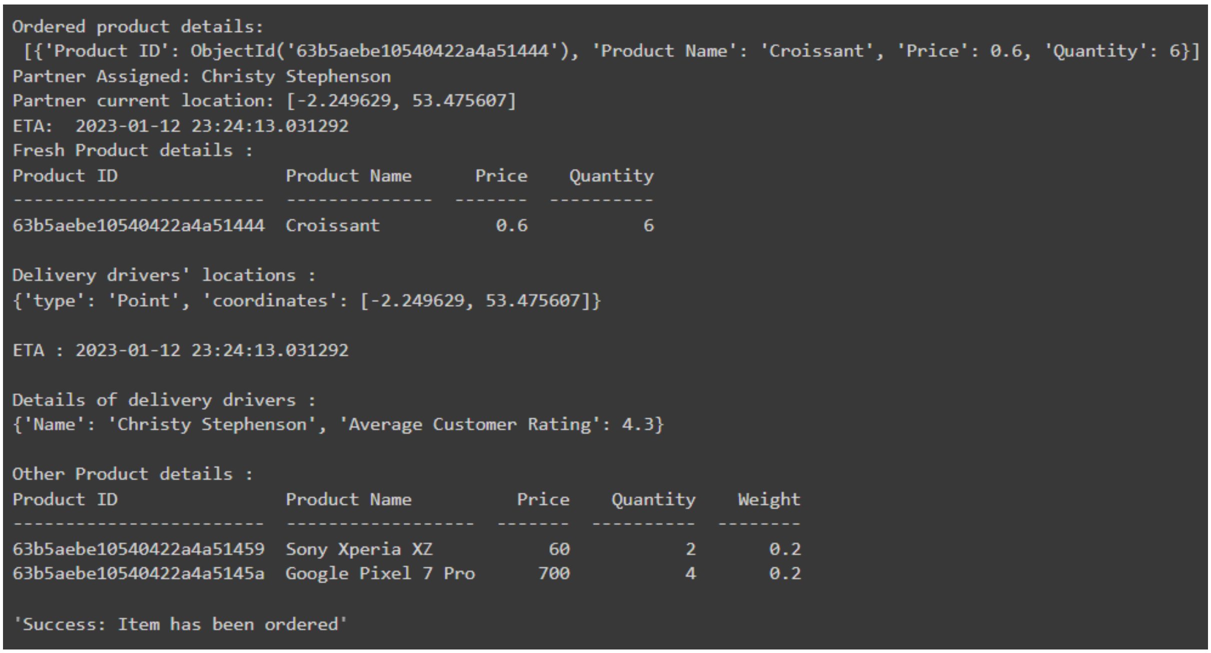
Query 4: Customer "Nikodemos Hero" (_id: ObjectId('63b8707d066488245e595983')), adds 1 ‘croissant’ to her basket, which already consists of 5 ‘croissants’ 2 'Sony Xperia XZ' and 4 'Google Pixel 7 Pro'. He proceeds to purchase all items in his basket

Input

addToBasket(ObjectId('63b8707d066488245e595983'),ObjectId('63b5aebe10540422a4a51444'), 1)

basketToOrder(ObjectId('63b8707d066488245e595983'))

Output



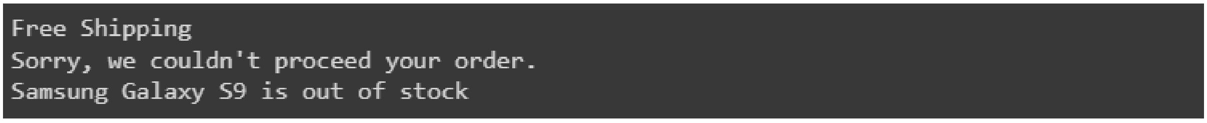
```
Ordered product details:
 [{'Product ID': ObjectId('63b5aebe10540422a4a51444'), 'Product Name': 'Croissant', 'Price': 0.6, 'Quantity': 6}]
Partner Assigned: Christy Stephenson
Partner current location: [-2.249629, 53.475607]
ETA:  2023-01-12 23:24:13.031292
Fresh Product details :
Product ID                Product Name      Price    Quantity
------------------------  --------------  -------  ----------
63b5aebe10540422a4a51444  Croissant           0.6           6

Delivery drivers' locations :
{'type': 'Point', 'coordinates': [-2.249629, 53.475607]}

ETA : 2023-01-12 23:24:13.031292

Details of delivery drivers :
{'Name': 'Christy Stephenson', 'Average Customer Rating': 4.3}

Other Product details :
Product ID                Product Name          Price    Quantity    Weight
------------------------  ------------------  -------  ----------  --------
63b5aebe10540422a4a51459  Sony Xperia XZ           60           2       0.2
63b5aebe10540422a4a5145a  Google Pixel 7 Pro      700           4       0.2

'Success: Item has been ordered'
```

Query 5: Customer 'Hadhry Haslimejuice' (_id: ObjectId("63b8707d066488245e595992")) adds 1 "Samsung Galaxy S9" to his basket and attempts to purchase. This order is, however unsuccessful as the product is out of stock.

Input

```
addToBasket(ObjectId("63b8707d066488245e595992"),ObjectId('63b5aebe10540422a4a51455'), 1)

basketToOrder(ObjectId("63b8707d066488245e595992"))
```

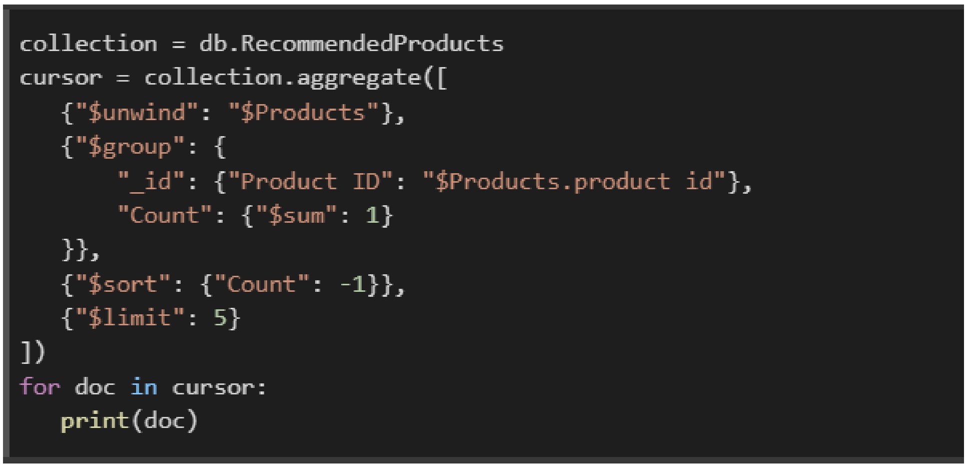
Output



```
Free Shipping
Sorry, we couldn't proceed your order.
Samsung Galaxy S9 is out of stock
```

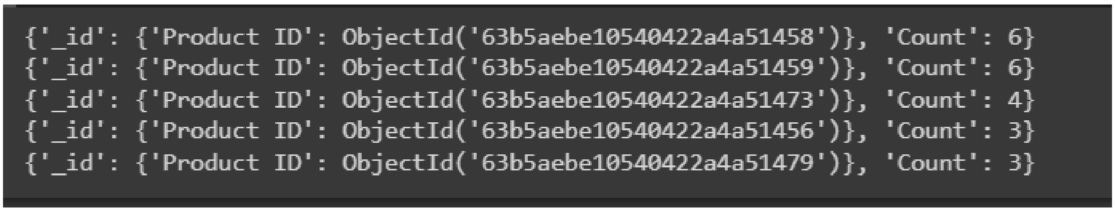
Query 6: Getting the top 5 most recommended items

Input:The first step of the aggregation pipeline is to unwind the array field so that the recommended products are in separate documents. Then the documents are grouped by the *Product ID* and the number of documents in which the *Product ID* appears is counted. Then the aggregated documents are sorted in descending order according to the count and limited so only the first 5 documents are outputted.



```
collection = db.RecommendedProducts
cursor = collection.aggregate([
   {"$unwind": "$Products"},
   {"$group": {
       "_id": {"Product ID": "$Products.product id"},
       "Count": {"$sum": 1}
   }},
   {"$sort": {"Count": -1}},
   {"$limit": 5}
])
for doc in cursor:
   print(doc)
```

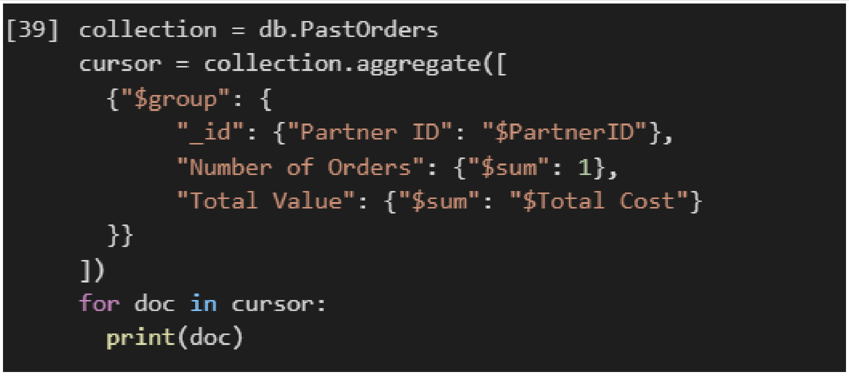
Output



```
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51458')}, 'Count': 6}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51459')}, 'Count': 6}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51473')}, 'Count': 4}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51456')}, 'Count': 3}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51479')}, 'Count': 3}
```

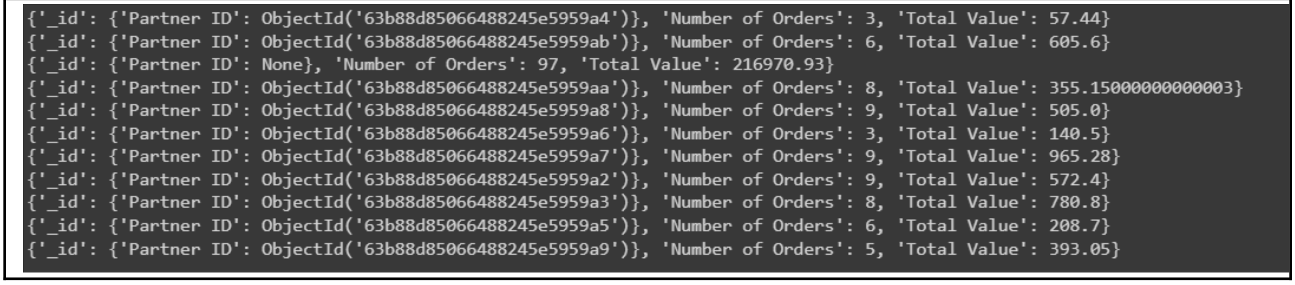
Query 7: Getting the number of deliveries and the total value of items each driver has completed:

Input: The *PastOrders* collection is grouped by *Partner ID* and the number of documents in which each id appears is counted to obtain the number of deliveries. The sum of the *Total Cost* for each document is also obtained to get the total value of items delivered for each driver.



```
[39] collection = db.PastOrders
     cursor = collection.aggregate([
       {"$group": {
            "_id": {"Partner ID": "$PartnerID"},
            "Number of Orders": {"$sum": 1},
            "Total Value": {"$sum": "$Total Cost"}
       }}
     ])
     for doc in cursor:
       print(doc)
```

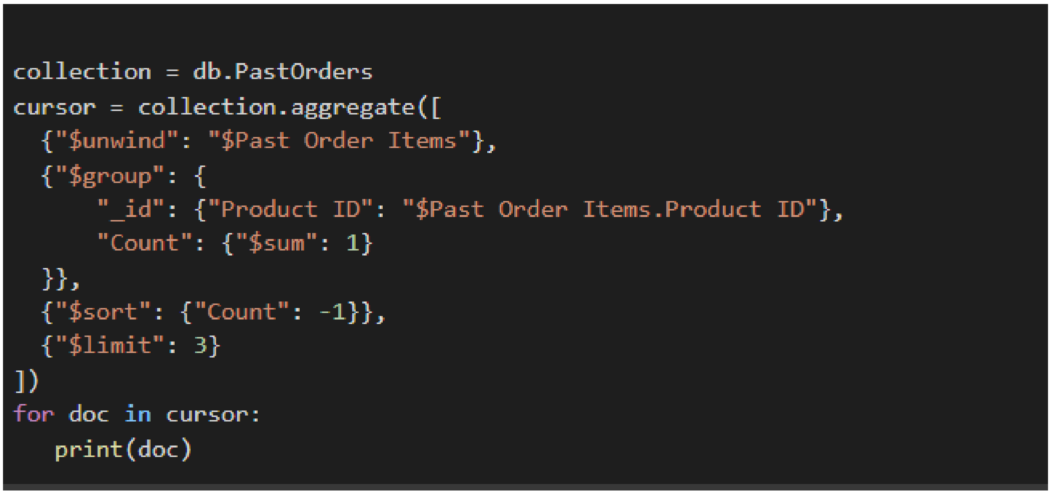
Output



```
{'_id': {'Partner ID': ObjectId('63b88d85066488245e5959a4')}, 'Number of Orders': 3, 'Total Value': 57.44}
{'_id': {'Partner ID': ObjectId('63b88d85066488245e5959ab')}, 'Number of Orders': 6, 'Total Value': 605.6}
{'_id': {'Partner ID': None}, 'Number of Orders': 97, 'Total Value': 216970.93}
{'_id': {'Partner ID': ObjectId('63b88d85066488245e5959aa')}, 'Number of Orders': 8, 'Total Value': 355.15000000000003}
{'_id': {'Partner ID': ObjectId('63b88d85066488245e5959a8')}, 'Number of Orders': 9, 'Total Value': 505.0}
{'_id': {'Partner ID': ObjectId('63b88d85066488245e5959a6')}, 'Number of Orders': 3, 'Total Value': 140.5}
{'_id': {'Partner ID': ObjectId('63b88d85066488245e5959a7')}, 'Number of Orders': 9, 'Total Value': 965.28}
{'_id': {'Partner ID': ObjectId('63b88d85066488245e5959a2')}, 'Number of Orders': 9, 'Total Value': 572.4}
{'_id': {'Partner ID': ObjectId('63b88d85066488245e5959a3')}, 'Number of Orders': 8, 'Total Value': 780.8}
{'_id': {'Partner ID': ObjectId('63b88d85066488245e5959a5')}, 'Number of Orders': 6, 'Total Value': 208.7}
{'_id': {'Partner ID': ObjectId('63b88d85066488245e5959a9')}, 'Number of Orders': 5, 'Total Value': 393.05}
```

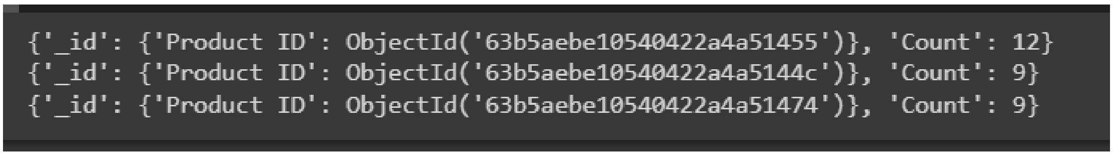
Query 8: Getting the top 3 most sold products

Input The unwind operator is used to separate each item in order out into separate documents. Then the documents are grouped according to *Product ID* the number of times each *Product ID* appears is counted. The aggregated documents are sorted according to the count in descending order, and the output is limited to 3 documents.



```
collection = db.PastOrders
cursor = collection.aggregate([
  {"$unwind": "$Past Order Items"},
  {"$group": {
      "_id": {"Product ID": "$Past Order Items.Product ID"},
      "Count": {"$sum": 1}
  }},
  {"$sort": {"Count": -1}},
  {"$limit": 3}
])
for doc in cursor:
   print(doc)
```

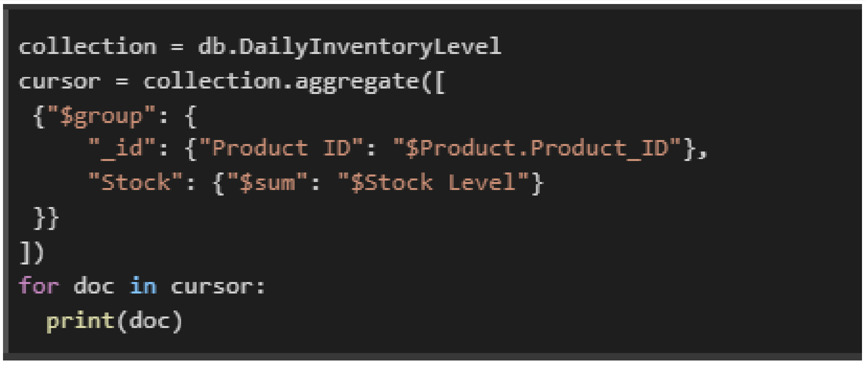
Output



```
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51455')}, 'Count': 12}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a5144c')}, 'Count': 9}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51474')}, 'Count': 9}
```

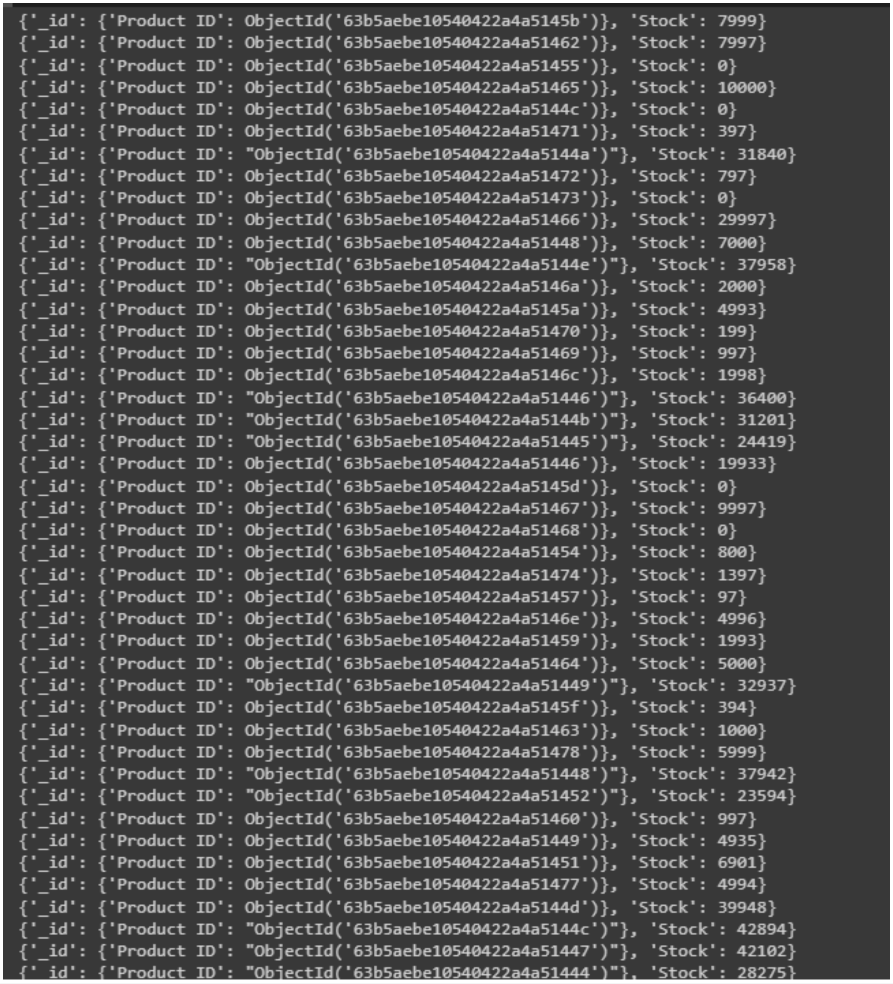
Query 9: Getting the total stock level across all warehouses for each product

Input: This query uses an aggregation pipeline over the *DailyInventoryLevel* collection. The pipeline group documents by *Product ID* and takes the sum of the stock levels for each product.



```
collection = db.DailyInventoryLevel
cursor = collection.aggregate([
 {"$group": {
     "_id": {"Product ID": "$Product.Product_ID"},
     "Stock": {"$sum": "$Stock Level"}
 }}
])
for doc in cursor:
  print(doc)
```

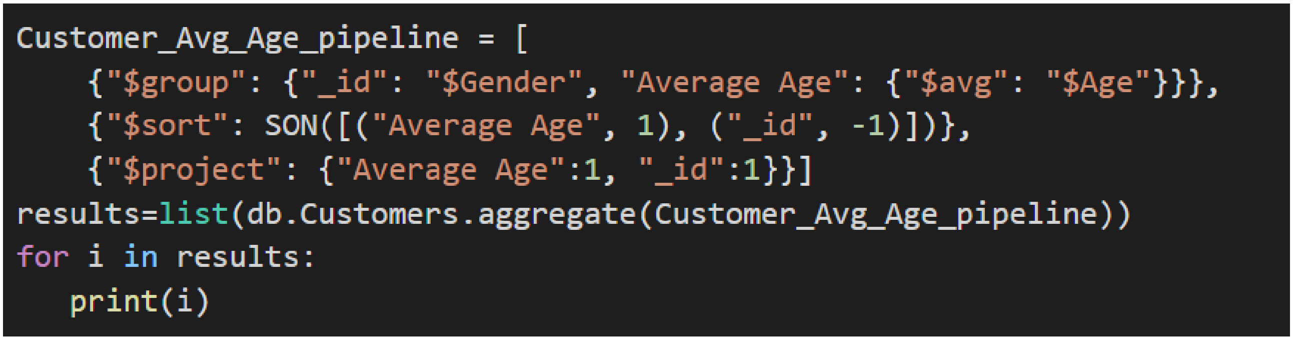
Output



```
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a5145b')}, 'Stock': 7999}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51462')}, 'Stock': 7997}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51455')}, 'Stock': 0}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51465')}, 'Stock': 10000}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a5144c')}, 'Stock': 0}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51471')}, 'Stock': 397}
{'_id': {'Product ID': "ObjectId('63b5aebe10540422a4a5144a')"}, 'Stock': 31840}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51472')}, 'Stock': 797}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51473')}, 'Stock': 0}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51466')}, 'Stock': 29997}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51448')}, 'Stock': 7000}
{'_id': {'Product ID': "ObjectId('63b5aebe10540422a4a5144e')"}, 'Stock': 37958}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a5146a')}, 'Stock': 2000}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a5145a')}, 'Stock': 4993}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51470')}, 'Stock': 199}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51469')}, 'Stock': 997}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a5146c')}, 'Stock': 1998}
{'_id': {'Product ID': "ObjectId('63b5aebe10540422a4a51446')"}, 'Stock': 36400}
{'_id': {'Product ID': "ObjectId('63b5aebe10540422a4a5144b')"}, 'Stock': 31201}
{'_id': {'Product ID': "ObjectId('63b5aebe10540422a4a51445')"}, 'Stock': 24419}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51446')}, 'Stock': 19933}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a5145d')}, 'Stock': 0}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51467')}, 'Stock': 9997}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51468')}, 'Stock': 0}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51454')}, 'Stock': 800}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51474')}, 'Stock': 1397}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51457')}, 'Stock': 97}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a5146e')}, 'Stock': 4996}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51459')}, 'Stock': 1993}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51464')}, 'Stock': 5000}
{'_id': {'Product ID': "ObjectId('63b5aebe10540422a4a51449')"}, 'Stock': 32937}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a5145f')}, 'Stock': 394}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51463')}, 'Stock': 1000}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51478')}, 'Stock': 5999}
{'_id': {'Product ID': "ObjectId('63b5aebe10540422a4a51448')"}, 'Stock': 37942}
{'_id': {'Product ID': "ObjectId('63b5aebe10540422a4a51452')"}, 'Stock': 23594}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51460')}, 'Stock': 997}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51449')}, 'Stock': 4935}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51451')}, 'Stock': 6901}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a51477')}, 'Stock': 4994}
{'_id': {'Product ID': ObjectId('63b5aebe10540422a4a5144d')}, 'Stock': 39948}
{'_id': {'Product ID': "ObjectId('63b5aebe10540422a4a5144c')"}, 'Stock': 42894}
{'_id': {'Product ID': "ObjectId('63b5aebe10540422a4a51447')"}, 'Stock': 42102}
{'_id': {'Product ID': "ObjectId('63b5aebe10540422a4a51444')"}, 'Stock': 28275}
```

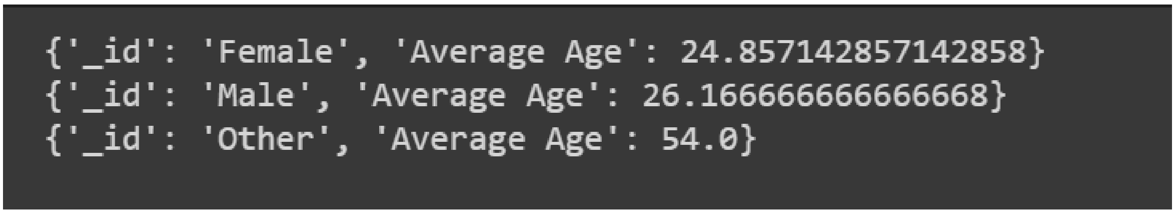
Query 10 : Finding the Customers' Average Age for Demographic Information

Input



```
Customer_Avg_Age_pipeline = [
    {"$group": {"_id": "$Gender", "Average Age": {"$avg": "$Age"}}},
    {"$sort": SON([("Average Age", 1), ("_id", -1)])},
    {"$project": {"Average Age":1, "_id":1}}]
results=list(db.Customers.aggregate(Customer_Avg_Age_pipeline))
for i in results:
   print(i)
```

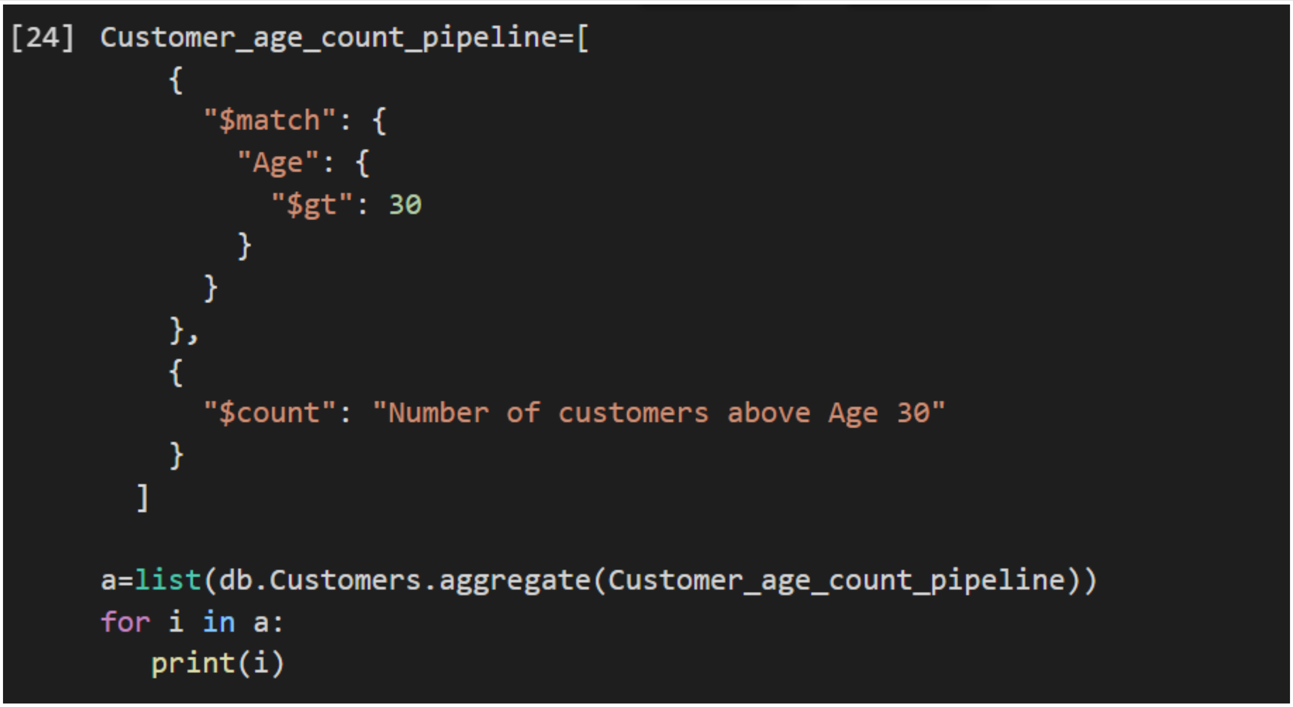
Output



```
{'_id': 'Female', 'Average Age': 24.857142857142858}
{'_id': 'Male', 'Average Age': 26.166666666666668}
{'_id': 'Other', 'Average Age': 54.0}
```

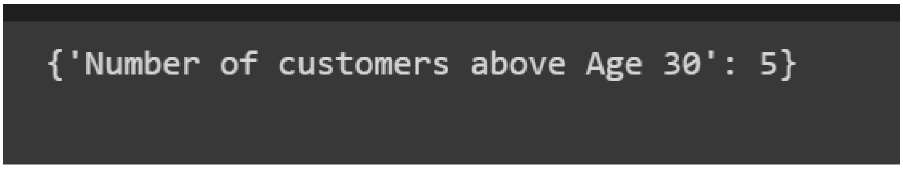
Query 11: Customer age count above a certain age

Input



```
[24] Customer_age_count_pipeline=[
         {
           "$match": {
             "Age": {
               "$gt": 30
             }
           }
         },
         {
           "$count": "Number of customers above Age 30"
         }
       ]

     a=list(db.Customers.aggregate(Customer_age_count_pipeline))
     for i in a:
        print(i)
```

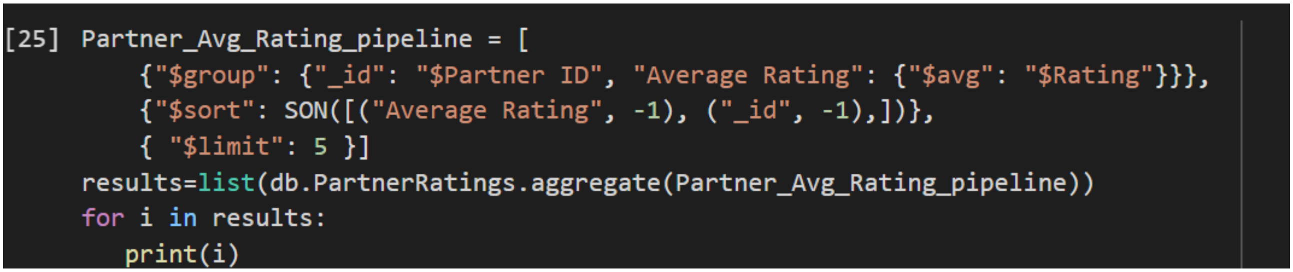
Output



```
{'Number of customers above Age 30': 5}
```

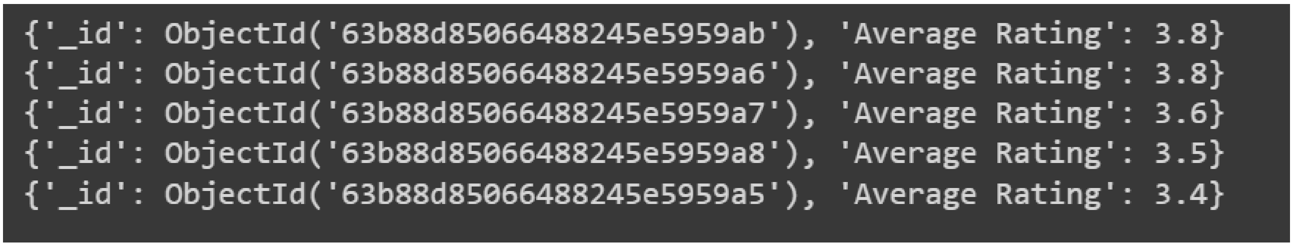
Query 12: Top 5 Partners by Rating

Input



```
[25] Partner_Avg_Rating_pipeline = [
        {"$group": {"_id": "$Partner ID", "Average Rating": {"$avg": "$Rating"}}},
        {"$sort": SON([("Average Rating", -1), ("_id", -1),])},
        { "$limit": 5 }]
     results=list(db.PartnerRatings.aggregate(Partner_Avg_Rating_pipeline))
     for i in results:
        print(i)
```

Output



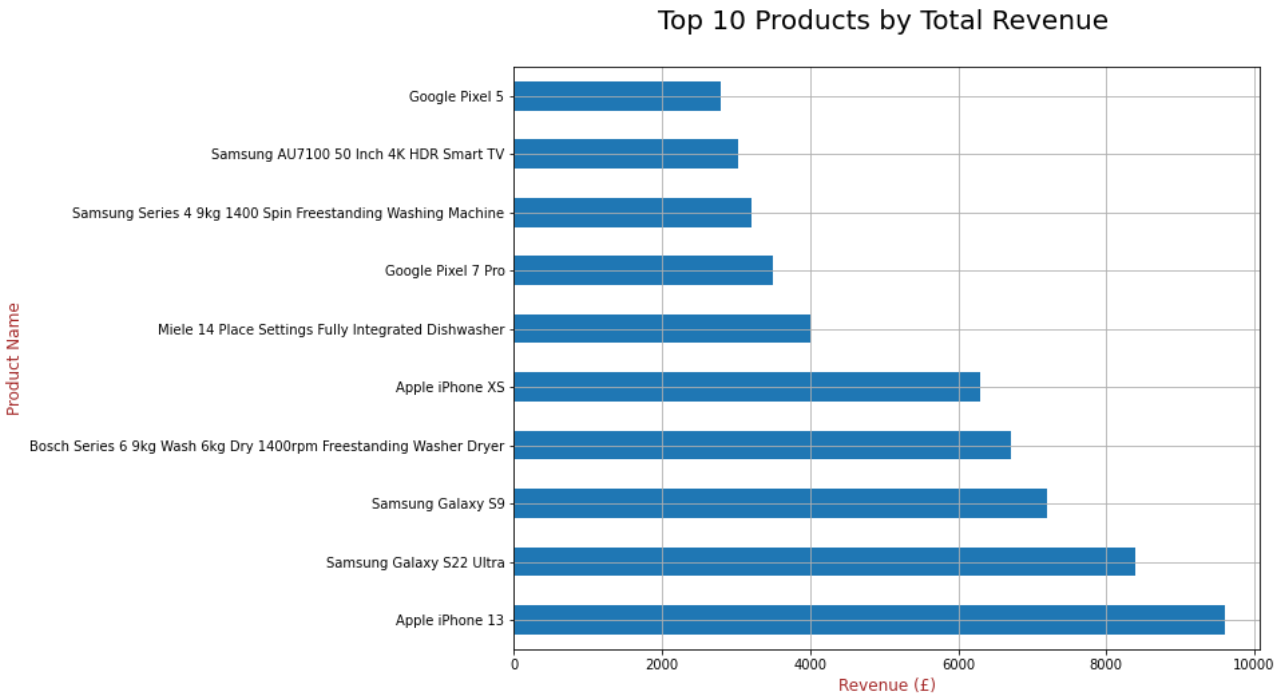
```
{'_id': ObjectId('63b88d85066488245e5959ab'), 'Average Rating': 3.8}
{'_id': ObjectId('63b88d85066488245e5959a6'), 'Average Rating': 3.8}
{'_id': ObjectId('63b88d85066488245e5959a7'), 'Average Rating': 3.6}
{'_id': ObjectId('63b88d85066488245e5959a8'), 'Average Rating': 3.5}
{'_id': ObjectId('63b88d85066488245e5959a5'), 'Average Rating': 3.4}
```

Query 13: Top 10 Products by Revenue

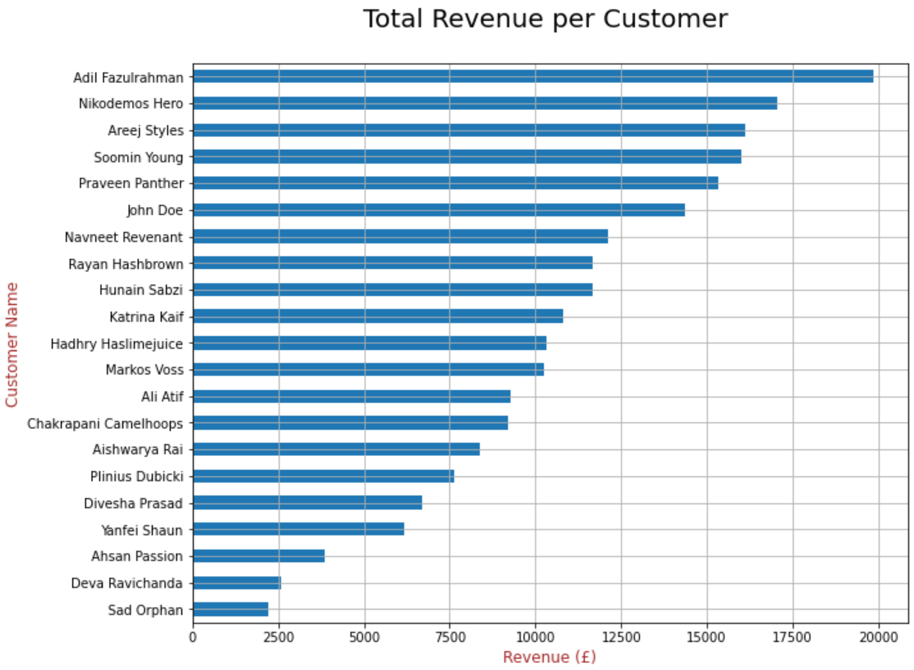
Input

Top_10_products_by_revenue(PastOrder)

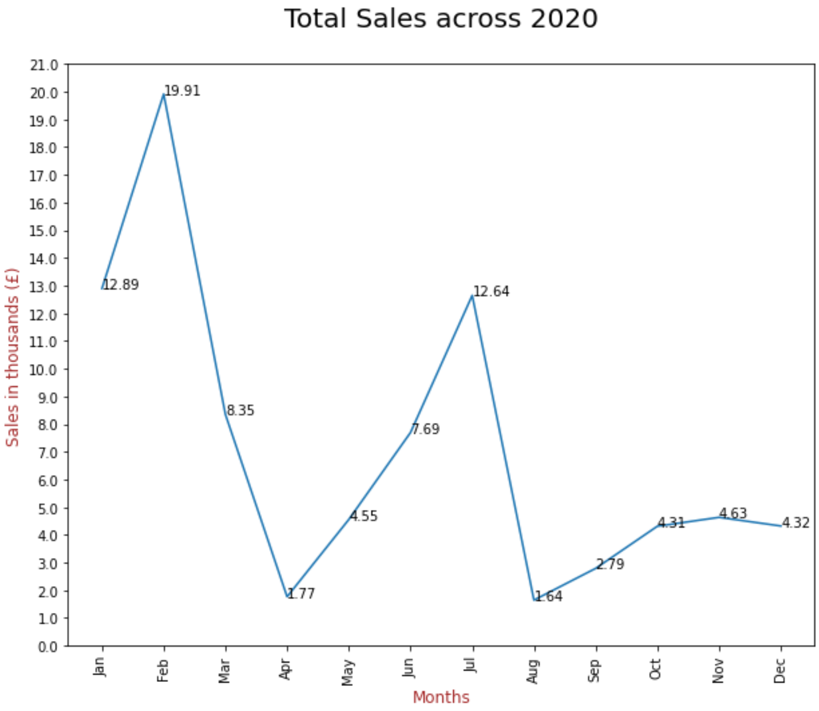
Output

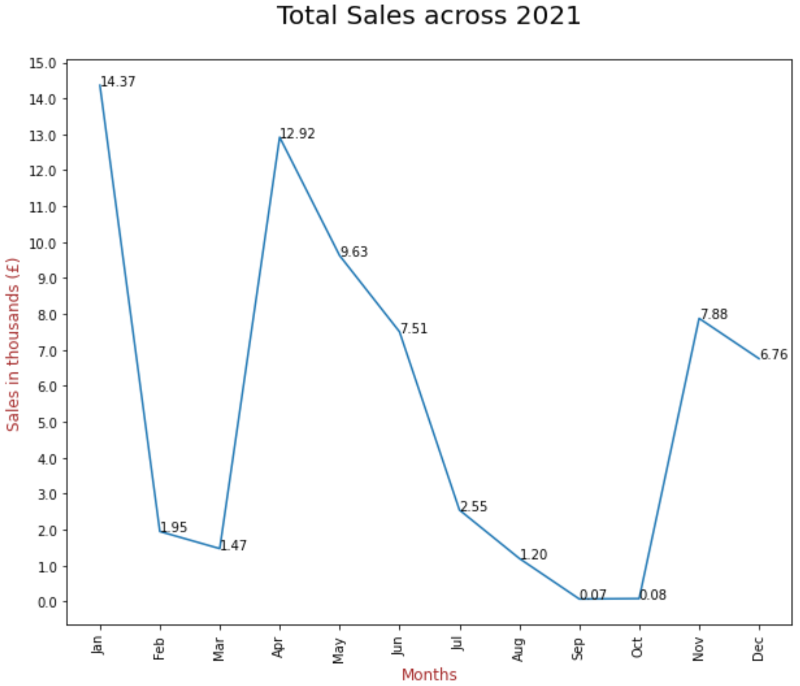


Query 14: Total Revenue by Customer

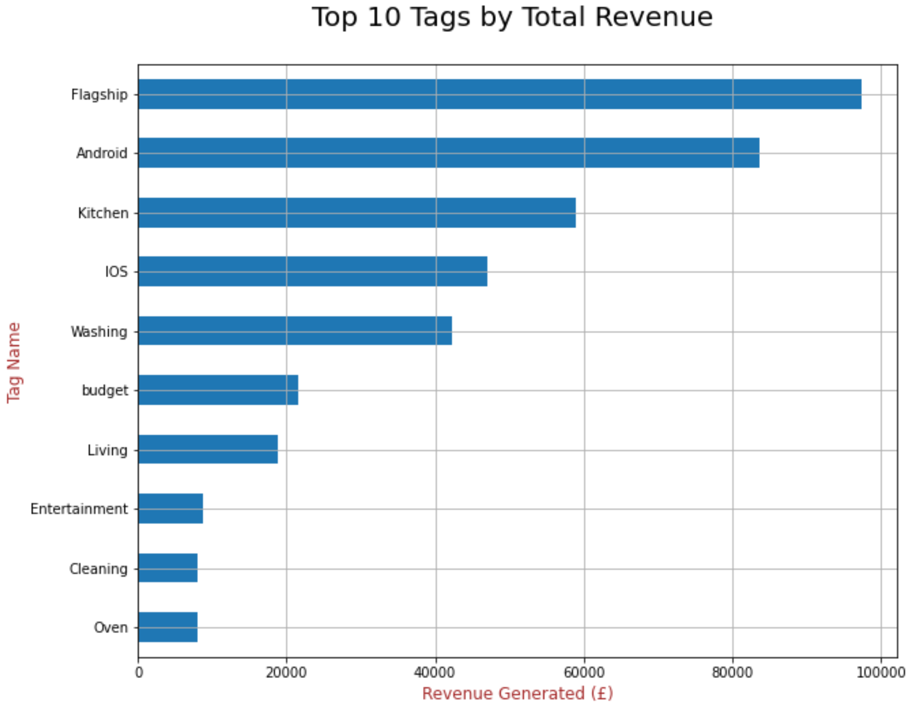
Input
Total_Revenue_per_Customer(PastOrders)
Output


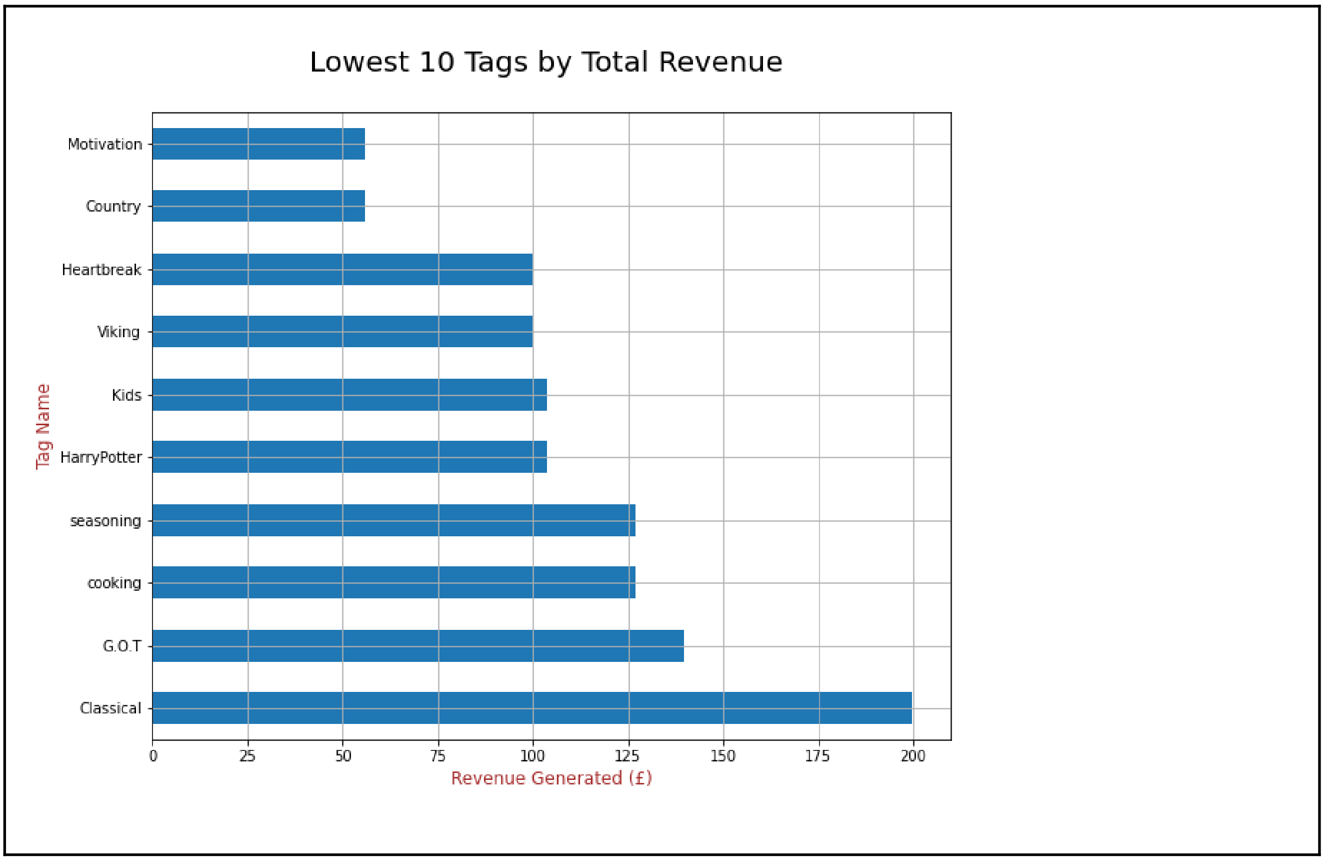
Query 15: Total Sales over time

Input
Total_Sales_Over_Time(PastOrders,year)
Output




Query 16: Total Revenue by tags

<u>Input</u>
Revenue_by_tag(Products, Type) - type : ‘Top’ or ‘Lowest’
<u>Output</u>



Lowest 10 Tags by Total Revenue
Motivation
Country
Heartbreak
Viking
Kids
HarryPotter
seasoning
cooking
G.O.T
Classical
Tag Name
0
25
50
75
100
125
150
175
200
Revenue Generated (£)

References

F. Shaikh, N. (2018) ‘Data Migration From SQL to MongoDB’, *HELIX*, 8(5), pp. 3701–3704. doi:10.29042/2018-3701-3704.

InfoQ (2020) *Data Modeling: Sample E-Commerce System with MongoDB*, *InfoQ*. Available at: https://www.infoq.com/articles/data-model-mongodb (Accessed: 13 January 2023).

MongoDB (2021) *How MongoDB makes custom e-commerce easy | MongoDB Blog*, *MongoDB*. Available at: https://www.mongodb.com/blog/post/how-mongodb-makes-custom-e-commerce-easy.

Sadalage, P.J. and Fowler, M. (2013) *NoSQL distilled : a brief guide to the emerging world of polyglot persistence*. Upper Saddle River, Nj: Addison-Wesley.

Sullivan, D. (2015) *NoSQL for mere mortals*. Hoboken, Nj: Addison-Wesley.

Yoon, J. *et al.* (2016) ‘Forensic investigation framework for the document store NoSQL DBMS: MongoDB as a case study’, *Digital Investigation*, 17, pp. 53–65. doi:10.1016/j.diin.2016.03.003.

Appendix

Section 1: Schema JSON

```
Customers {
      _id : ObjectID
      Name : str
      Age: int
      Gender : str
      Shipping Address : {
            HouseNum : int
            Street : str
            City : str
            Postcode: str
            Coordinate: GeoJSON Object
            }
      Billing Address : {
            House Num : int
            Street : str
            City : str
            Postcode: str
            }
}


ShoppingBasket {
      _id : ObjectID
      Customer ID: ref<Customers._id>
      Basket Items Fresh: [
            0: {Product ID : ref<Products._id>  //Embedded Document
            //product details: extended reference pattern
            Product name : str
            Price : int
            Quantity : int},
            1: {Product ID : ref<Products._id>
            Product name : str
            Price : int
            Quantity : int}, … ]
      ]

      Basket Items Other: [
            0: {Product ID : ref<Products._id>
            //product details: extended reference pattern
            Product name : str
            Price : int
            Quantity : int
            Weight : double},
            1: {Product ID : ref<Products._id>
            //product details: extended reference pattern
```

```
          Product name : str
          Price : int
          Quantity : int
          Weight : double}, … ]
     ]
     Fresh Items Cost : double
     Other Items Cost : double
     Shipping Cost Fresh : double
     Shipping Cost Other : double
     Total Cost : double
}


CurrentOrdersFresh {
     _id : ObjectID
     Customer ID: ref<Customers._id>
     Current Order Items : [
          0: {Product ID : ref<Products._id>  //Embedded Document
          // product details: extended reference pattern
          Product name : str
          Price : int
          Quantity : int},
          1: {Product ID : ref<Products._id>
          // product details: extended reference pattern
          Product name : str
          Price : int
          Quantity : int}, … ]
     Shipping Cost : double
     Total Cost : double
     Date : Date
     ETA : Date
     Partner ID : ref<Partners._id>
}

CurrentOrdersOther {
     _id : ObjectID
     Customer ID: ref<Customers._id>
     Current Order Items : [
          0: {Product ID : ref<products._id>  //Embedded Document
          // product details: extended reference pattern
          Product name : str
          Price : int
          Quantity : int
          Weight : double},
          1: {Product ID : ref<Products._id>
          // product details: extended reference pattern
          Product name : str
          Price : int
          Quantity : int
          Weight : double}, … ]
```

```
      Shipping Cost : double
      Total Cost : double
      Date : Date
      ETA : Date
      Partner ID : ref<Partners._ID>
}


PastOrders {
      _id : ObjectID
      Customer ID: ref<Customers._id>
      Shipping Cost : double
      Total Cost: double
      Shipping Address: [        // Extended reference pattern
            HouseNum: int
            Street : str
            City : str
            Postcode: str
            Coordinate:GeoJSON Object]
      Past Order Items : [
            0: {Product ID : ref<Products._id>
            // product details : Extended reference pattern
            Product name : str
            Price : int
            Quantity : int},
            1: {Product ID:ref<Products._id>
            // product details : extended reference pattern
            Product name : str
            Price : int
            Quantity : int}, … ]
      Shipping Cost : double   // Computed pattern
      Total Cost : double   // Computed pattern
      Date : Date
      Partner ID : ref<Partners._id>
}
```

```
OrderStatus {
      _id : ObjectID
      status : str
      fresh order id: ref<CurrentOrdersFresh._id>
            (Or other order id: ref<CurrentOrdersOthers._id>)
}
```

```
}


Partners {
     _id: ObjectID
     Name : str
     Age : int
     Gender : str
     Address : {
          HouseNum: int
          Street : str
          City: str
          Postcode: str
          }
     Average Customer Rating : double  // computed pattern
}

PartnerStatus {
     _id : ObjectID
     Partner ID: ref<Partners._id>
     Status : str
     Orders: Array of ref<CurrentOrdersFresh._id> or
                      ref<CurrentOrdersOther._id>
     Delivery Progress: str
     Current Order:
          ref<CurrentOrdersFresh._id> or ref<CurrentOrdersOther._id>
     Location: GeoJSON Object
     LastModified: Date
}

Warehouses {
     _id : ObjectID
     Name : str
     Type : str
     Location : GeoJSON Object

}

DailyInventoryLevel {
     _id : ObjectID
     Product : [  // Bucket pattern
          Product_ID: ref<Products._id>,
          Product Name: str]  // Extended reference pattern
          Stock Level : int
     Warehouse ID : ref<Warehouses._id>
     Date: Date
}

ProductRatings {
     _id : ObjectID
     Product ID : ref<Products._id>
     Customer ID : ref<Customers._id>
```

```
    Product Rating : int
}

PartnerRatings {
    _id : ObjectID
    Partner ID : ref<Partners._id>
    Order ID : ref<Products._id>
    Rating : int
}

Products {  // Mobile Phone
    _id : ObjectID
    Name : str
    Desc : str
    Dimensions : str
    Weight : double
    Category : str
    Standard Price : double
    Cost from Supplier : double
    Tags: array of str
    Brand : str
    Model : str
    Colour : str
    Features : str
    }

Products {  // CD
    _id : ObjectID
    Name : str
    Desc : str
    Dimensions : str
    Weight : double
    Category : str
    Standard Price : double
    Cost from Supplier : double
    Tags: array of str
    Artist Name : str
    Number of Tracks : int
    Total Playing time : double
    Publisher : str
    }

Products {  // BOOK
    _id : ObjectID
    Name : str
    Desc : str
    Dimensions : str
    Weight : double
    Category : str
    Standard Price : double
```

```
      Cost from Supplier : double
      Tags: array of str
      Author Name: str
      Publisher : str
      Publication Name : str
      ISBN : str
      }

Products {  // HOME APPLIANCES
      _id : ObjectID
      Name : str
      Desc : str
      Dimensions : str
      Weight : double
      Category : str
      Standard Price : double
      Cost from Supplier : double
      Tags: Array of str
      Colour: str
      Voltage: double
      Style : str
      }



Products {  // FRESH PRODUCTS
      _id : ObjectID
      Name : str
      Desc : str
      Dimensions : str
      Weight : double
      Category : str
      Standard Price : double
      Tags : Array of str
      Cost from Supplier : double
      ExpiryDate: Date
      Country of Origin : str
      }
```

Section 2: Python Functions

1. Find Customer Coords

```python
def SetCoords(CustomerID, CustomerPostcode):
    parameters = {
                        "key" : "sM7lCSW55GhFb541H9GBBHzrasK3J5B6",
                        "location": CustomerPostcode
                        }

    response = requests.get("http://www.mapquestapi.com/geocoding/v1/address",params = parameters)
    data = json.loads(response.text)['results']

        long = data[0]['locations'][0]['latLng']['lng']
        lat = data[0]['locations'][0]['latLng']['lat']

        db.Customers.update_one({"_id": CustomerID}, {"$set":{"Shipping Address.Coordinates": { "type": "Point", "coordinates": [ long, lat ] }}})
```

1. AvailableFreshProducts

```python
def AvailableFreshProducts(customerID):
  allMorrison = db.Warehouses.find({"Type":"Morrison"}).distinct("_id")

  customerCoord = db.Customers.find_one({"_id":customerID})['Shipping Address']['Coordinates']['coordinates']

```

```
    availableMorrison = warehouses.find(
        {
            "Location": {
                "$near": {
                    "$geometry": {
                        "type": "Point",
                        "coordinates": customerCoord
                        },
                    "$maxDistance": 3000,
                    }
                }, "_id": {"$in": allMorrison}
            }).distinct("_id")
    AvailableFreshProducts = db.DailyInventoryLevel.find({"Warehouse ID":{"$in":availableMorrison}, "Stock Level":{"$gte":1}}).distinct("Product.Product_ID")

    for i in AvailableFreshProducts:
        availableProduct = db.Products.find_one({"_id":i},{'name':1,'description':1,'standardPrice':1,'countryOfOrigin':1})
        print(availableProduct)
```

2. FindMorrison

```
def FindMorrison(customerCoord, availableWarehouses, productType = 0):
    if (productType == 0):
        try: # productType = 0 for fresh products, return the nearest Morrison within maxDistance
```

```
             morrison = warehouses.find_one(
                 {
                     "Location": {
                         "$near": {
                                "$geometry": {
                                    "type": "Point",
                                    "coordinates": customerCoord
                                    },
                                "$maxDistance": 3000,
                                }
                            }, "_id": {"$in": availableWarehouses}
                        })["_id"]
                return morrison
            except:
                return None

        else:  # other products, return the nearest warehouse regardless of the distance
            try:
                morrison = warehouses.find_one(
                    {
                        "Location": {
                            "$near": {
                                "$geometry": {
                                    "type": "Point",
                                    "coordinates": customerCoord
                                    }
                            }
                        }, "_id": {"$in": availableWarehouses}
                    })["_id"]
            return morrison
        except:
            return None
```

4. Shipping_cost_fresh

```
def shipping_cost_fresh(customerID):
    shoppingBasket = db.ShoppingBasket


    cursor = db.ShoppingBasket.find_one({"Customer ID": customerID})


    try:
        freshItemsCost = cursor["Fresh Items Cost"]
        oldShippingCost = cursor["Shipping Cost Fresh"]
    except Exception:
        print("Please check customer ID")
        return


    if (freshItemsCost < 40 and freshItemsCost > 0):
        if (oldShippingCost == 0):
            shoppingBasket.update_one({"Customer ID": customerID}, {
                "$inc": {"Total Cost": 4}})
            shoppingBasket.update_one({"Customer ID": customerID}, {
                "$inc": {"Shipping Cost Fresh": 4}})
    else:

        if (oldShippingCost > 0):

            shoppingBasket.update_one({"Customer ID": customerID}, {

                "$inc": {"Total Cost": -4}})

            shoppingBasket.update_one({"Customer ID": customerID}, {

                "$inc": {"Shipping Cost Fresh": -4}})

            print("Free Shipping!")
```

5. shipping_cost_other

```
def shipping_cost_other(customerID):


    # set all variables and assign cursor to shopping basket collection

    shoppingBasket = db.ShoppingBasket

    cursor = shoppingBasket.find_one({"Customer ID": customerID})


    try:

        basketProducts = cursor["Basket Items Other"]

        oldShippingCost = cursor["Shipping Cost Other"]


    except Exception:

        print("Please check customer ID")

        return


    if (oldShippingCost > 0):
        shoppingBasket.update_one({"Customer ID": customerID}, {"$inc":
{"Shipping Cost Other": -oldShippingCost,

"Total Cost": -oldShippingCost}})


    shippingCost = 0


    for doc in basketProducts:

        index = 0

        weight = doc["Weight"]

        price = doc["Price"]

        quantity = doc["Quantity"]

        index = weight*price*quantity


    # calculate shipping cost based on the index


    if index <= 50 and index > 0:  # free shipping for index less than 50

        print("Free Shipping")

    elif index > 50 and index <= 100:

        shippingCost += 5

        print("$5 Shipping cost applied")

    elif index <= 150 and index > 100:  # shipping cost is applied for
anything over

        shippingCost += 10

        print("$10 Shipping cost applied")

    elif index <= 250 and index > 150:

        shippingCost += 15

        print("$15 Shipping cost applied")
    elif index > 250:

        shippingCost += 20

        print("$20 Shipping cost applied")



    shoppingBasket.update_one({"Customer ID": customerID}, {"$inc": {"Total
Cost": round(shippingCost, 2),


"Shipping Cost Other": shippingCost}})
```

6. AddToBasket

```
def addToBasket(customerID, productID, quantity):



    # 2. insert document into shopping basket



    # 2.1. check the product category

    cursor = products.find_one({"_id": productID})



    try:

        category = cursor["category"]

        productName = cursor["name"]

        price = cursor["standardPrice"]

        weight = cursor["weight"]

    except Exception:

        print("Please check ProductID")

        return



    cost = round(price * quantity, 2)
    # 2.2. Fresh Categories to classify basket items
    freshList = ["bakery", "drinks", "fruits"]


    # 2.3. insert or update basket collection
    cursor = shoppingBasket.find({"Customer ID": customerID})


    if (len(cursor.distinct("_id")) != 0):  # if the basket for that
customer already exists, update
        basketID = cursor.distinct("_id")[0]


        freshItemsList = []
        otherItemsList = []


        freshItemsList = cursor.distinct("Basket Items Fresh.Product ID")
        otherItemsList = cursor.distinct("Basket Items Other.Product ID")


        if (category in freshList):  # if the product is fresh product




            # if the product already exists in basket, update quantity and
cost
            if (productID in freshItemsList):
                shoppingBasket.update_one({"_id": basketID, "Basket Items
Fresh.Product ID": productID},
                                          {"$inc": {"Basket Items
Fresh.$.Quantity": quantity, "Fresh Items Cost": cost,
                                                    "Total Cost": cost}})


            else:  # else, insert product into fresh items in the basket


                shoppingBasket.update_one({"_id": basketID},
                                          {"$push": {"Basket Items Fresh":
                                                     {"Product ID":
productID, "Product Name": productName, "Price": price,
                                                      "Quantity":
quantity}}, "$inc": {"Fresh Items Cost": cost,

"Total Cost": cost}}, upsert=True)


            # recalculate the shipping cost based on new fresh items cost
            shipping_cost_fresh(customerID)


        else:  # if the product is non-fresh product


            # if the product already exists in basket, update quantity and
cost
            if (productID in otherItemsList):
                shoppingBasket.update_one({"_id": basketID, "Basket Items
Other.Product ID": productID},
                                          {"$inc": {"Basket Items
Other.$.Quantity": quantity, "Other Items Cost": cost,
                                                    "Total Cost": cost}})
            else:  # else, insert product into fresh items in the basket
                shoppingBasket.update_one({"_id": basketID},
                                          {"$push": {"Basket Items Other":
```

```
                                                     {"Product ID":
productID, "Product Name": productName, "Price": price,
                                                      "Quantity": quantity,
"Weight": weight}}, "$inc": {"Other Items Cost": cost,

"Total Cost": cost}}, upsert=True)


            # calculate the shipping cost and update the basket
            shipping_cost_other(customerID)



    else:  # if the document doesnt exist, insert
        if (category in freshList):
            shoppingBasket.insert_one({"Customer ID": customerID,
                                       "Basket Items Fresh": [{"Product ID":
productID, "Product Name": productName, "Price": price,
                                                               "Quantity":
quantity}],
                                       "Basket Items Other": [],
                                       "Fresh Items Cost": round(cost, 2),
"Other Items Cost": 0, "Shipping Cost Fresh": 0,
                                       "Shipping Cost Other": 0, "Total
Cost": round(cost, 2)})

            shipping_cost_fresh(customerID)

        else:
            shoppingBasket.insert_one({"Customer ID": customerID,
                                       "Basket Items Fresh": [],
                                       "Basket Items Other": [{"Product ID":
productID, "Product Name": productName, "Price": price,

                                                                "Quantity":
quantity, "Weight": weight}], "Fresh Items Cost": 0,

                                       "Other Items Cost": round(cost, 2),
"Shipping Cost Fresh": 0,

                                       "Shipping Cost Other": 0, "Total
Cost": round(cost, 2)})


            shipping_cost_other(customerID)
```

7. FindPartner

```
def FindPartner(warehouseCoord):


    driver = db.PartnerStatus.find_one(

        {

            "Status": "Active",

            "Location": {

                "$near": {

                    "$geometry": {

                        "type": "Point",

                                "coordinates": warehouseCoord

                    },

                    "$maxDistance": 3000,

                }

            }, "Orders.4": {"$exists": False}

        })


    try:

        return driver['Partner ID']

    except:
        return None



# AssignOrder to assign orders to partners
def AssignOrder(partnerID, orderID):
    partnerStatus.update_one({"Partner ID": partnerID}, {
                             "$push": {"Orders": orderID}})
    partnerStatus.update_one({"Partner ID": partnerID}, [
                             {"$set": {"Delivery Progress": "On the way",
"Current Order": {"$arrayElemAt": ["$Orders", 0]}}}])

    PartnerInv = len(partnerStatus.find_one(
        {"Partner ID": partnerID})["Orders"])
    eta = datetime.now() + timedelta(minutes=30 * PartnerInv)



    return eta
```

8. UpdateOrderStatus

```
def update_order_status_despatched (fresh_order_id):
  db.OrderStatus.update_one({"fresh order
id":fresh_order_id},{"$set":{"status":"Despatched"}})
```

9. Complete Order Sub Functions

```
# to update OrderStatus
def orderStatusUpdate(orderID, orderType):
    if (orderType == 0): #orderType = 0 for fresh products
        orderStatus.insert_one({"fresh order id": orderID, "status":
"Processing"})
    else:
```

```python
        orderStatus.insert_one({"other order id": orderID, "status": "Dispatched"})



# basketToOrder to proceed items from Basket to Order
def basketToOrder(customerID):

    # Get basket item lists
    basketItemsFresh = []
    basketItemsOther = []
    cursor = shoppingBasket.find({"Customer ID": customerID})

    try:
        cur = shoppingBasket.find_one({"Customer ID": customerID})["Basket Items Fresh"]
    except:
        return "No Basket Found"

    freshPIDs = []
    freshQuantities = []
    freshNames = []
    for doc in cur: #Warning! Don't use distinct because it sorts the result automatically,
    #Resulting in any situation where product ids, names, and quantities don't match
        freshPIDs.append(doc["Product ID"])
        freshQuantities.append(doc["Quantity"])
        freshNames.append(doc["Product Name"])

    try:
        cur = shoppingBasket.find_one({"Customer ID": customerID})["Basket Items Other"]
    except:
        return "No Basket Found"
    otherPIDs = []
    otherQuantities = []
    otherNames = []
    for doc in cur:
        otherPIDs.append(doc["Product ID"])
        otherQuantities.append(doc["Quantity"])
        otherNames.append(doc["Product Name"])

    cursor2 = customers.find({"_id": customerID})
    customerCoord = cursor2.distinct("Shipping
Address.Coordinates.coordinates")
    shippingAddress = cursor2.distinct("Shipping Address")

    partnerIDs = []
    morrisonIDs = []
    warehouseIDs = []

    # Check inventory and find the nearest warehouses and available partners
    for i in range(len(freshPIDs)):

        availableWarehouses = FindMorrisonWithStock(
            freshPIDs[i], freshQuantities[i])
        morrisonID = FindMorrison(customerCoord, availableWarehouses, 0)

        if(morrisonID == None):
            print("Sorry, we couldn't proceed your order.")
            print(freshNames[i], "is out of stock")
            return
        warehouseCoord = warehouses.find(
            {"_id": morrisonID}).distinct("Location.coordinates")
        partnerID = FindPartner(warehouseCoord)

        if(partnerID == None):
            print("Sorry, we couldn't proceed your order.")
            print("All delivery drivers are busy now. Please try again
later")
            return

        if (partnerID not in partnerIDs):
            partnerIDs.append(partnerID)

        morrisonIDs.append(morrisonID)

    for j in range(len(otherPIDs)):
        availableWarehouses = FindMorrisonWithStock(
            otherPIDs[j], otherQuantities[j])

        warehouseID = FindMorrison(customerCoord, availableWarehouses, 1)
        if (warehouseID == None):
            print("Sorry, we couldn't proceed your order.")
            print(otherNames[j], "is out of stock")
            return

        warehouseIDs.append(warehouseID)

    basketItemsFresh = cursor.distinct("Basket Items Fresh")
    shippingCostFresh = cursor.distinct("Shipping Cost Fresh")[0]
    freshCost = cursor.distinct("Fresh Items Cost")[0]
    basketItemsOther = cursor.distinct("Basket Items Other")

    shippingCostOther = cursor.distinct("Shipping Cost Other")[0]

    otherCost = cursor.distinct("Other Items Cost")[0]


    if(not basketItemsFresh and not basketItemsOther):

        print("The basket is empty")

        return


    # Move basket items to current order

    if(len(basketItemsFresh) > 0):


        result = currentOrdersFresh.insert_one({"Customer ID": customerID,
"Shipping Address": shippingAddress,

                                                "Current Order Items":
basketItemsFresh,

                                                "Shipping Cost":
shippingCostFresh, "Total Cost": round(freshCost + shippingCostFresh, 2),

                                                "Date": datetime.now(),
"ETA": None, "PartnerID": None})


        # Assign drivers and update ETA with the maximum ETA between
products in the order

        orderID = result.inserted_id

        eta = datetime(1, 1, 1)


        for partnerID in partnerIDs:

            eta = max(eta, AssignOrder(partnerID, orderID))


        currentOrdersFresh.update_one(

            {"_id": orderID}, {"$set": {"ETA": eta, "PartnerID":
partnerIDs}})
```

```
        # update OrderStatus
        orderStatusUpdate(orderID, 0)
        update_order_status_despatched (orderID)

    if(len(basketItemsOther) > 0):

        result = currentOrdersOther.insert_one({"Customer ID": customerID,
"Shipping Address": shippingAddress,
                                       "Current Order Items":
basketItemsOther,
                                       "Shipping Cost": shippingCostOther,
"Total Cost": round(otherCost + shippingCostOther, 2),
                                       "Date": datetime.now()})

        orderID = result.inserted_id
        # update OrderStatus
        orderStatusUpdate(orderID, 1)

    # Print order details
    if(len(basketItemsFresh) > 0):
        print("Fresh Product details :")
        header = basketItemsFresh[0].keys()
        rows = [x.values() for x in basketItemsFresh]
        print(tabulate.tabulate(rows, header), "\n")
        print("Delivery drivers' locations :")
        for partnerID in partnerIDs:
            partnerLocation = partnerStatus.find_one(
                {"Partner ID": partnerID})["Location"]
            print(partnerLocation, "\n")
        print("ETA :", eta, "\n")
        partnerDetails = partners.find_one(
            {"_id": partnerID}, {"_id": 0, "Name": 1, "Average Customer Rating": 1})
        print("Details of delivery drivers :")
        print(partnerDetails, "\n")

    if(len(basketItemsOther) > 0):
        print("Other Product details :")
        header = basketItemsOther[0].keys()
        rows = [x.values() for x in basketItemsOther]
        print(tabulate.tabulate(rows, header), "\n")

    # update dailyInventoryLevel by deducting quantities of ordered products from chosen warehouses
    for i in range(len(freshPIDs)):
        result = inventory.update_one({"Product.Product_ID": freshPIDs[i], "Warehouse ID": morrisonIDs[i]}, {
            "$inc": {"Stock Level": -freshQuantities[i]}})
        if result.modified_count == 0:
            return "Error: Failed to update inventory(Fresh)"

    for j in range(len(otherPIDs)):
        result = inventory.update_one({"Product.Product_ID": otherPIDs[j], "Warehouse ID": warehouseIDs[j]}, {
            "$inc": {"Stock Level": -otherQuantities[j]}})
        if result.modified_count == 0:
            return "Error: Failed to update inventory(Other)"



    #return order information


    # remove items from basket
    shoppingBasket.delete_one({"Customer ID": customerID})



    return "Success: Item has been ordered"
```

10. CompleteOrder

```
def CompleteOrder (fresh_order_id):
  db.OrderStatus.update_one({"fresh order id":fresh_order_id},{"$set":{"status":"Delivered"}})


  order = db.CurrentOrdersFresh.find_one({"_id":fresh_order_id})
  db.PastOrders.insert_one(order)


  db.CurrentOrdersFresh.delete_one({"_id":fresh_order_id})


  db.PartnerStatus.update_one({"Current Order": fresh_order_id}, {"$pull":{"Orders":fresh_order_id}})


  cursor = db.PartnerStatus.find_one({"Current Order": fresh_order_id})



  if not cursor["Orders"]:
    db.PartnerStatus.update_one({"_id":cursor["_id"]}, {"$set":{"Delivery Progress": "Not on Errand"}})


  db.PartnerStatus.update_one({"Current Order": fresh_order_id}, [{"$set":{"Current Order":{"$arrayElemAt":["$Orders",0]}}}])
```

11. Top 10 Products Graph Function

```
def Top_10_Products_by_Revenue(PastOrders):
  plt.figure(figsize=(10, 8))
  data = list(PastOrders.find({}))
  df3 = pd.json_normalize(data, record_path=['Past Order Items'], meta=[
                        'Total Cost', 'Date', ])
  a=df3.groupby(by=['Product Name'])['Price'].sum().sort_values(ascending=False)
  g=a[:10].plot(kind='barh', grid=True)
  g.set_xlabel("Revenue (£)", fontdict={'fontsize': 12, 'fontweight' : 5, 'color' : 'Brown'})
  g.set_ylabel("Product Name", fontdict={'fontsize': 12, 'fontweight' : 5, 'color' : 'Brown'})
  plt.title('Top 10 Products by Total Revenue \n', fontdict={'fontsize': 20, 'fontweight' : 5, 'color' : 'Black'})
  plt.savefig('Top_10_Products_by_Revenue.png')
  return plt.show()
```

12. Total Revenue per Customer Graph Function

```
def Total_Revenue_per_Customer(PastOrders):
  plt.figure(figsize=(10, 8))
  datapoints = list(db.Customers.find({}))
  df1 = pd.json_normalize(datapoints)
  df1=df1.iloc[:,:2]
  datapoints = list(db.PastOrders.find({}))
  df2 = pd.json_normalize(datapoints)
  df_sales= df2[['Customer ID','Total Cost']]
  df_sales = df_sales.rename({'Customer ID': '_id', 'Total Cost': 'Sales'}, axis=1)  # new method
  df_sales = pd.merge(df_sales,df1, on='_id', how='left')
  df_sales = df_sales.rename({'Name': 'Customer'}, axis=1)  # new method

a=df_sales.groupby(by=['Customer'])['Sales'].sum().sort_values(ascending=True)
  rep_plot =a.plot(kind='barh', grid=True)
  rep_plot.set_ylabel("Customer Name", fontdict={'fontsize': 12,
'fontweight' : 5, 'color' : 'Brown'})

  rep_plot.set_xlabel("Revenue (£)", fontdict={'fontsize': 12, 'fontweight'
: 5, 'color' : 'Brown'})

  plt.title('Total Revenue per Customer \n', fontdict={'fontsize': 20,
'fontweight' : 5, 'color' : 'Black'})

  plt.savefig('Total_Revenue_per_Customer.png')

  return plt.show()
```

13. Total_Sales_Over_Time function

```
def Total_Sales_Over_Time(PastOrders,year):

  plt.figure(figsize=(10, 8))

  year=int(year)

  datapoints = list(PastOrders.find({}))

  df = pd.json_normalize(datapoints)

  df=df[["Date",'Total Cost']]

  df['year'] = df['Date'].dt.year

  df['month'] = df['Date'].dt.month

  import calendar

  df['Month'] = df['month'].apply(lambda x: calendar.month_abbr[x])

  months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",

          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

  df['Month'] = pd.Categorical(df['Month'], categories=months, ordered=True)

  df=df[["Total Cost",'year','Month']]

  df = df.rename({'Total Cost': 'Sales','year' : 'Year'}, axis=1)

  df =df[df['Year']== year]

  df=df.groupby('Month',
as_index=False)['Sales'].sum().rename(columns={'Month' : 'Month'})

  months=df['Month']

  sales=df['Sales']

  plt.plot(months, sales)
```

```
# Adding and formatting title
  r="Total Sales across " + str(year) + "\n"
  plt.title(r, fontdict={'fontsize': 20, 'fontweight' : 5, 'color' : 'Black'})

# Labeling Axes
  plt.xlabel("Months", fontdict={'fontsize': 12, 'fontweight' : 5, 'color' : 'Brown'})
  plt.ylabel("Sales in thousands (£)", fontdict={'fontsize': 12, 'fontweight' : 5, 'color' : 'Brown'} )

  ticks = np.arange(0, max(sales)+1500, 1000)
  labels = ["{}".format(i//1000) for i in ticks]
  plt.yticks(ticks, labels)

  plt.xticks(rotation=90)

  for xy in zip(months, sales):
    plt.annotate(s = "{0:.2f}".format(xy[1]/1000), xy = xy, textcoords='data')
  plt.savefig('Total_Sales_Over_Time.png')
  return plt.show()
```

14. Revenue by Tag Function

```
def Revenue_by_tag(Products, Type):
  global Graph_type
  global Graph_type2
  if Type == 'Top':
    Graph_type=False
    Graph_type2=True
  else:
    Graph_type= True
    Graph_type2=False

  datapoints = list(Products.find({}))
  df1 = pd.json_normalize(datapoints)

  pd.set_option('display.max_columns', None)

  df1.head(1)
  df1=df1[['_id','tags']]
  df1
  df5=pd.DataFrame(df1.tags.values.tolist()).add_prefix('tag_')
  df1 = df1.join(df5)
  df1 = df1.rename({'_id': 'Product ID'}, axis=1)  # new method

  datapoints = list(db.PastOrders.find({}))
  df8 = pd.json_normalize(datapoints, record_path=['Past Order Items'], meta=['Total Cost', 'Date', ])
  df8=df8.iloc[:,:4]
  df8['Total Revenue Contribution']=df8['Price']*df8['Quantity']
  df8
  df8 = pd.merge(df8,df1, on='Product ID', how='left')

  df15=df8.groupby('tag_0', as_index=False)['Total Revenue Contribution'].sum().rename(columns={'Total Revenue Contribution' : 'Month'})
  df16=df8.groupby('tag_1', as_index=False)['Total Revenue Contribution'].sum().rename(columns={'Total Revenue Contribution' : 'Month'})
  df17=df8.groupby('tag_2', as_index=False)['Total Revenue
Contribution'].sum().rename(columns={'Total Revenue Contribution' :
'Month'})

  df18=df8.groupby('tag_3', as_index=False)['Total Revenue
Contribution'].sum().rename(columns={'Total Revenue Contribution' :
'Month'})

  df19=df8.groupby('tag_4', as_index=False)['Total Revenue
Contribution'].sum().rename(columns={'Total Revenue Contribution' :
'Month'})



  df15 = df15.rename({'tag_0': 'tag'}, axis=1)  # new method


  df16 = df16.rename({'tag_1': 'tag'}, axis=1)  # new method


  df17 = df17.rename({'tag_2': 'tag'}, axis=1)  # new method


  df18 = df18.rename({'tag_3': 'tag'}, axis=1)  # new method



  df19 = df19.rename({'tag_4': 'tag'}, axis=1)  # new method


  df20 = pd.merge(df15, df16, on='tag', how='outer').merge(df17, on='tag',
how='outer')


  df20=df20.fillna(0)

  df20['Total']=df20['Month_x']+df20['Month_y']+df20['Month']

  df20=df20[['tag','Total']]

  df20 = pd.merge(df20,df18, on='tag', how='outer').merge(df19, on='tag',
how='outer')

  df20=df20.fillna(0)

  df20['Total1']=df20['Month_x']+df20['Month_y']+df20['Total']

  df20=df20[['tag','Total1']]
  df21=df20.sort_values(by='Total1', ascending=Graph_type)
  df21=df21[:10]
  df21 = df21.reset_index(drop=True)
  df20.to_csv('tags_by_revenue.csv')

  a=df21.groupby(by=['tag'])['Total1'].sum().sort_values(ascending=Graph_type2)
  plt.figure(figsize=(10, 8))
  rep_plot =a.plot(kind='barh', grid=True)
  rep_plot.set_ylabel("Tag Name", fontdict={'fontsize': 12, 'fontweight' : 5, 'color' : 'Brown'})
  rep_plot.set_xlabel("Revenue Generated (£)", fontdict={'fontsize': 12, 'fontweight' : 5, 'color' : 'Brown'})

  if Type == 'Top':
    plt.title('Top 10 Tags by Total Revenue \n', fontdict={'fontsize': 20, 'fontweight' : 5, 'color' : 'Black'})
  else:
    plt.title('Lowest 10 Tags by Total Revenue \n', fontdict={'fontsize': 20, 'fontweight' : 5, 'color' : 'Black'})
  if Type == 'Top':
    plt.savefig('Top 10 Total Revenue per Tag.png')
  else:
    plt.savefig('Lowest Performing Tags by Revenue.png')
  return plt.show()
```

15. Recommender System

```
def get_n_top_rated_products(n: int):
    id_list = []
    rating_list = []
    col = db.AverageProductRatings
    q = col.aggregate([{"$group":{
        "_id": {"Product ID": "$product id"},
        "rating": {"$sum": "$average rating"}
    }}, {"$sort": {"rating": -1, "_id": 1}}, {"$limit": n}])
    for doc in q:
        id_list.append(doc["_id"]["Product ID"])
        rating_list.append(doc["rating"])
    prod_list = []
    col = db.Products
    for id in id_list:
        q = col.find_one({"_id": id})
        prod_list.append(q)
    return prod_list, rating_list

def add_products_to_recommended(prod_list: list, ratings: list, customer_id: int):
    col = db.RecommendedProducts
    reduced_prod_list = []
    for prod in prod_list:
        i = 0
        reduced_prod = {
            "product id": prod["_id"],
            "name": prod["name"],
            "price": prod["standardPrice"],
            "average rating": ratings[i]
        }
        i += 1
        reduced_prod_list.append(reduced_prod)
    recommendation  = {
        "Customer ID": customer_id,
        "Products": reduced_prod_list
    }
    col.insert_one(recommendation)

def get_user_tags(customer_id):
    tags = []
    products_bought = []
    col = db.PastOrders
    q = col.find({"Customer ID": customer_id})
    for doc in q:
        for product in doc["Past Order Items"]:
            products_bought.append(product["Product ID"])
    col = db.Products
    q = col.find({"_id": {"$in": products_bought}})
    for doc in q:
        tags.extend(doc["tags"])
    counts = dict()
    for i in tags:
        counts[i] = counts.get(i, 0) + 1
    counts = dict(sorted(counts.items(), key=lambda item: item[1],
reverse=True))
    top_tags = list(counts.keys())
    count = 0
    tag_num = 0
    prod_list = []
    rating_list = []
    while count < 2 or tag_num == len(top_tags):
        q = col.find({"tags": {"$in": [top_tags[tag_num]]}})
        for doc in q:
            if count < 2:

                prod_list.append(doc)

                count += 1

        tag_num += 1

    col = db.AverageProductRatings

    ids = [prod["_id"] for prod in prod_list]

    for id in ids:

        q = col.find_one({"product id": id})

        if q is not None:

            rating_list.append(q["average rating"])

    if count < 2:

        prods, rats = get_n_top_rated_products(2-count)

        prod_list.extend(prods)

        rating_list.extend(rats)

    return prod_list, rating_list


collection = db.RecommendedProducts

collection.delete_many({})

collection = db.PastOrders

cursor = collection.aggregate([{"$group":{

    "_id": {"Customer ID": "$Customer ID"},

    "count": {"$sum": 1}

}}, {"$sort": {"_id": 1}}])

for doc in cursor:

    if doc["count"] == 0:

        top_rated, top_ratings = get_n_top_rated_products(2)

    else:

        top_rated, top_ratings = get_user_tags(doc["_id"]["Customer ID"])

    add_products_to_recommended(top_rated, top_ratings, doc["_id"]["Customer
ID"])
```

```
collection = db.ProductRatings

average_product_ratings = []

ids = []


cursor = collection.aggregate([{"$group":{

   "_id": {"Product ID": "$Product ID"},

   "totalRatings": {"$sum": "$Product Rating"},

   "numRatings": {"$sum": 1}

}}, {"$sort": {"_id": 1}}])

for doc in cursor:

   ids.append(doc["_id"]["Product ID"])


average_product_ratings.append(round(doc["totalRatings"]/doc["numRatings"],
2))


collection = db.AverageProductRatings


avr_list = []

for i in range(len(ids)):

   q = collection.find_one({"_id": ids[i]})

   if q is None:

       avr_dict = {

           "product id": ids[i],

           "average rating": average_product_ratings[i]

           }

       avr_list.append(avr_dict)

   else:

       collection.update_one(

           {"product id": ids[i]},

           {"$set": {"average rating": average_product_ratings[i]}}

       )
```

```
collection.insert_many(avr_list)
```

END