Team 1

Amazone Online Shopping Website Database

Team 1	IDs
Affan Abid Imam	7454267
Charlie George Unsworth	10458747
Menglei Guo	11624497
Poorva Shashikant Nimbalkar	11543286
Jiaqi Yao	11604233
Leming Liu	11517005
Silang Nimai	11562596

Content

MANCHESTER 1824

01

Members and Sub-tasks Assigned

02

ER Digrarm and NOSQL Schema

03

Database Design

04

Database Query Analysis

01

Members and Sub-tasks Assigned

1.1 Members and Sub-tasks Assigned

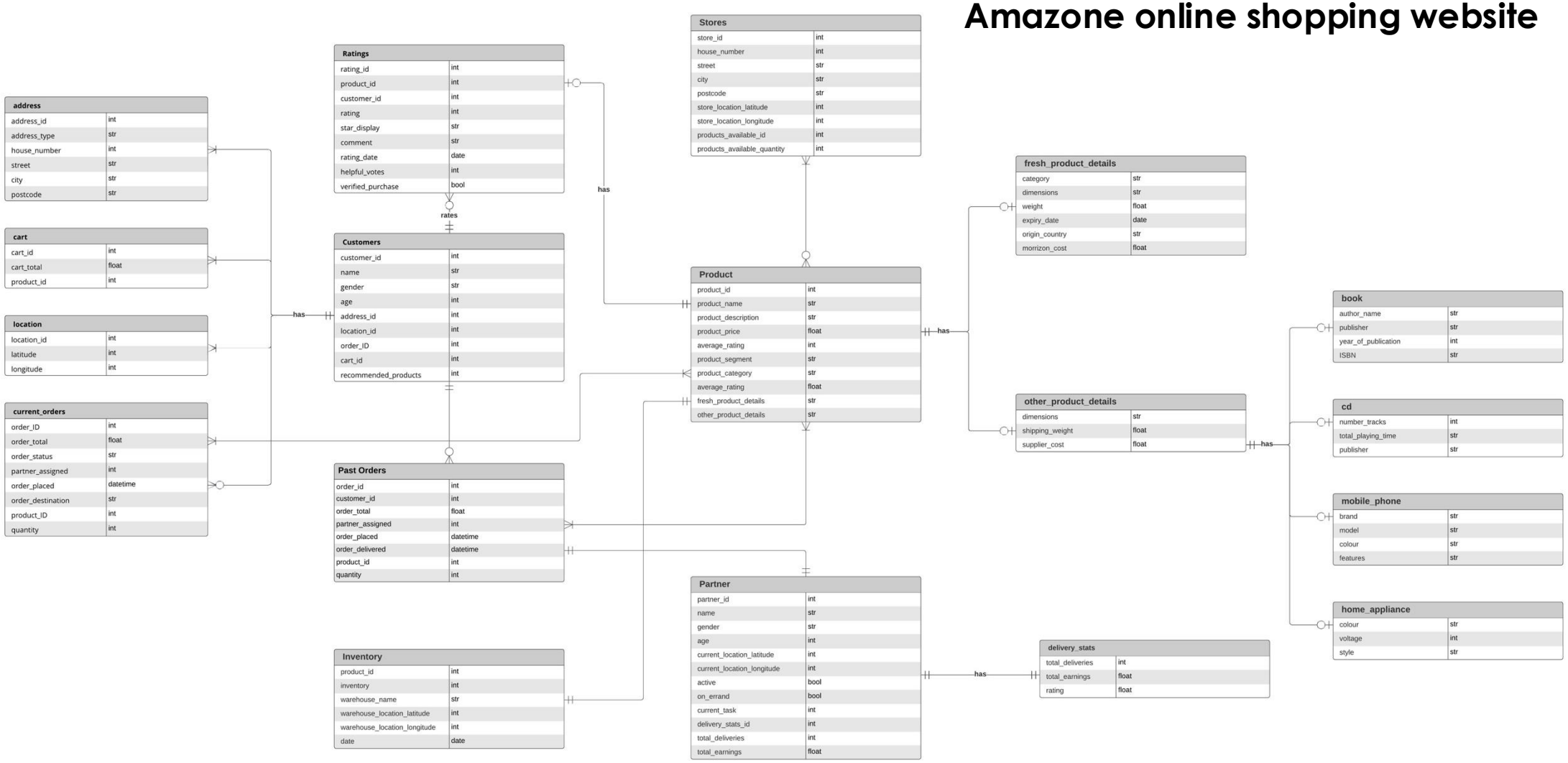
Name	Sub-tasks
Affan Abid Imam	Designing, itterating and implementing scripts to populate Products, Customers, Current and Past Orders
Charlie George Unsworth	Compiling data modelling and schema design, code to compute average ratings, Query: Find top earning partners
Menglei Guo	ER Diagram; 2 Customer Purchase Product Queries Design; 1 Customer Personalized Product Recommendations Query Design
Poorva Shashikant Nimbalkar	Designing script to populate collections: Stores, Partners and Inventory
Jiaqi Yao	Designing 4 queries about fresh product order
Leming Liu	Designing sales and inventory performance queries and visualizations.
Silang Nimai	Query: A user search for available fresh products Query: Find customers who haven't made any purchases in the last 60 days

02

ER Diagram and NOSQL Schema

2.1 ER Digrarm

Amazone online shopping website



2.2.1 NOSQL Schema

```
Products = [{
'product_ID': int,
'name': str,
'description': str,
'price': float,
'product_segment': str,
'product_category': str,
'average__rating': float,
'fresh_product_details': {
        'category': str,
        'dimensions': str,
        'weight': float,
        'expiry_date': date,
        'origin_country': str,
        'morrizon_cost': float
        },
```

```
'other_product_details':{
        'dimensions': str,
        'shipping_weight': float,
        'supplier_cost': float,
        'book':{
          'author_name': str,
          'publisher': str,
          'year_of_publication': int,
          'ISBN': str
        },
        'cd':{
          'artist_name': str,
          'number_tracks': int,
          'total_playing_time': str,
          'publisher': str
        },
        'mobile_phone':{
          'brand': str,
          'model': str,
          'colour': str,
          'features': str,
        },
        'home_appliance':{
          'colour': str,
          'voltage': int,
          'style': str
    }
}
}]
```

- Product details were embedded within the products collection to facilitate straightforward querying of this information
- Average rating for each product is stored in this collection but computed from the ratings collection
- product_ID is then referenced in other collections

2.2.2 NOSQL Schema

```
Customers = [{
'customer_ID': int (index),
'name': str,
'gender': str,
'age': int,
'addresses':[
    {
    'address_type': str,
    'house_number': int,
    'street': str,
    'city': str,
    'postcode': str
    }
],
'location':{
    'latitude': int,
    'longitude': int
},
'cart':{
    'cart_total': float,
    'products': [ref <Products.product_ID>]
},
'current_orders':[{
     'order_ID': int,
     'order_total': float,
     'order_status': str,
     'partner_assigned': int,
     'order_placed': datetime,
     'order_destination': str,
     'products':
        [{
        'product_ID': ref <Products.product_ID>,
        'quantity': int
        }]
    }],
'recommended_products': [ref <Products.product_ID>]
```

```
PastOrders = [{
'order_ID': int,
'customer_ID': ref <Customers.customer_ID>,
'order_total': float,
'partner_assigned': ref <Partners.partner_ID,
'order_placed': datetime,
'order_delivered': datetime,
'products':
    [{
    'product_ID': ref <Products.product_ID>,
    'quantity': int
    }]
}]
```

- Current orders have been embedded within the customers collection to facilitate fast reads
- However past orders are stored in a separate collection since we might expect them to grow by a large amount and relatively low frequency of read operations
- customer_id has been indexed to allow for even quicker retrieval of important information, this was a tradeoff against the slower creation of a new customer

2.2.3 NOSQL Schema

```
Stores = [{
    'store_ID': int,
    'address':{
        'house_number': int,
        'street': str,
        'city': str,
        'postcode': str
    },
    'location':{
        'latitude': int,
        'longitude': int
    },
    'products_available':[{
        'product_ID': ref <Products.product_ID>,
        'quantity': int
    }]
}]
```

```
Partners = [{
'partner_ID': int,
'name':str,
'gender':str,
'age':int,
'current_location': {
    'latitude': int,
    'longitude': int
},
'active': bool,
'on_errand': bool,
'current_task': ref <Customers.current_orders.order_ID>,
'delivery_stats':{
        'total_deliveries': int,
        'total_earnings': float,
        'rating': float
        }
}]
```

```
Ratings = [{
'rating_ID': int,
'customer_ID': ref <Customers.customer_ID>,
'product_ID': ref <Products.product_ID>,
'rating': int,
'star_display': str,
'comment': str,
'rating_date': date,
'helpful_votes': int,
'verified_purchase': bool
}]
```

```
Inventory = [{
'product_ID': ref <Products.product_ID>,
'inventory': int,
'warehouse_name': str,
'location': {
    'latitude': int,
    'longitude': int
    },
'date': date
}]
```

- Store location as well as address is stored to allow for queries relating to a customer's nearest store
- Similarly, location is stored for partners to allow for calculation of ETA and other metrics
- Store inventories are assumed to be a stable size, and so are embedded here for straightforward queries with references to the Products collection
- Ratings are stored as a collection since their number is large and likely to grow rapidly, the computed average rating is updated periodically and stored in the Products collection

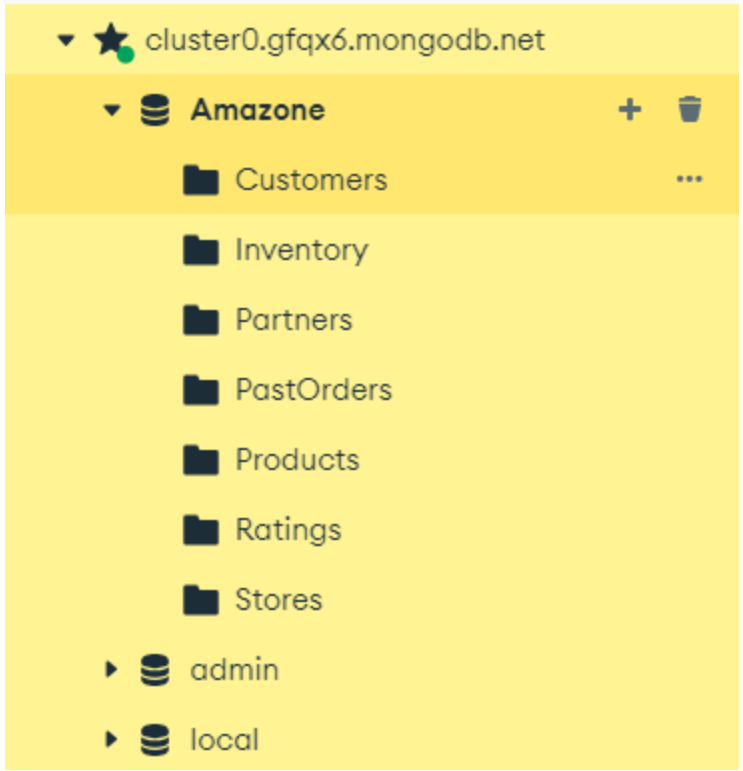
03

Database Design

3.1 Connect to MongoDB

- **MongoClient**: Establishes a connection to the MongoDB instance running on localhost.
- **DatabaseName**: Amazone
- **Collections**: References the Stores, Partners, and Inventory collections in the database.

```
# Connect to MongoDB
client = MongoClient('mongodb+srv://example:example@cluster0.gfqx6.mongodb.net/')
db = client['Amazone']
stores_collection = db['Stores']
partners_collection = db['Partners']
inventory_collection = db['Inventory']
```



3.2 Populate Stores Collection

- **Loops**: Creates 5 store entries with unique `store_ID`.
- **Address**: Randomly generates realistic UK addresses.
- **Location**: Uses the helper function to assign a location.
- **Products Available**: Each store has a list of products, randomly selecting `product_IDs` and quantities from the **Products** collection.

```python
# Part 1: Populate Stores Collection
stores = []
for i in range(1, 6):
    latitude, longitude = generate_uk_location()
    stores.append({
        "store_ID": i,
        "address": {
            "house_number": random.randint(1, 999),
            "street": random.choice(["High Street", "Station Road", "Church Lane", "Victoria Road", "Park Avenue"]),
            "city": random.choice(["London", "Manchester", "Birmingham", "Leeds", "Glasgow"]),
            "postcode": f"{random.choice(string.ascii_uppercase)}{random.randint(1, 99)} {random.randint(0, 9)}{''.join(random.choices(string.ascii_uppercase, k=2))}"
        },
        "location": {
            "latitude": latitude,
            "longitude": longitude
        },
        "products_available": [
            {"product_ID": random.choice(product_ids), "quantity": random.randint(10, 100)} for _ in range(5)
        ]
    })

stores_collection.insert_many(stores)
```

```python
# Fetch product IDs from Products collection
product_ids = [product["product_ID"] for product in db.Products.find({}, {"product_ID": 1})]
```

```python
# Helper function to generate random location within the UK
def generate_uk_location():
    latitude = random.uniform(49.3, 58.6)  # UK latitude bounds
    longitude = random.uniform(-8.1, 1.8)  # UK longitude bounds
    return latitude, longitude
```

3.3 Populate Partners Collection

```python
# Part 2: Populate Partners Collection
for i in range(1, 11):
    latitude, longitude = generate_uk_location()
    total_deliveries = random.randint(50, 200)
    base_rate_per_delivery = 8
    performance_bonus = 0
    if total_deliveries > 150:
        performance_bonus = 500  # Bonus for high performance

    total_earnings = round(total_deliveries * base_rate_per_delivery + performance_bonus, 2)

    partners.append({
        "partner_ID": i,
        "name": random.choice(["Alice", "Bob", "Chase", "Diana", "Ethan", "Fiona", "George", "Hannah", "Isaac", "Julia"]),
        "gender": random.choice(["Male", "Female"]),
        "age": random.randint(18, 60),
        "current_location": {
            "latitude": latitude,
            "longitude": longitude
        },
        "active": random.choice([True, False]),
        "on_errand": random.choice([True, False]),
        "current_task": random.randint(1001, 1010) if random.choice([True, False]) else None,
        "delivery_stats": {
            "total_deliveries": total_deliveries,
            "total_earnings": total_earnings,
            "rating": round(random.uniform(3.5, 5.0), 1)
        }
    })
partners_collection.insert_many(partners)
```

- **Partner Generation:** A loop creates 10 partners
- **Location and Status:** Each partner is assigned a random location within the UK and statuses such as active or on_errand.

Delivery Stats:

- **Total Deliveries:** Randomly generated between 50 and 200.
- **Base Earnings:** Calculated by multiplying total deliveries by a fixed base rate of 8.
- **Performance Bonus**: If total deliveries exceed 150, a bonus of 500 is added.
- **Total Earnings:** The sum of base earnings and any applicable bonus, rounded to two decimal places.
- **Rating:** Each partner receives a random rating between 3.5 and 5.0.

3.4 Populate Inventory Collection

- **Loops**: Generates inventory records for each product for two consecutive days.
- **Details**: Includes warehouse details, inventory counts, and daily statistics such as units sold, received, opening, and closing stock.

```python
# Part 3: Populate Inventory Collection
inventory = []
for product_id in product_ids:
    # Generate two days of inventory records for each product
    for day in range(2):
        current_date = datetime.now() - timedelta(days=day)
        latitude, longitude = generate_uk_location()
        inventory.append({
            "product_ID": product_id,
            "inventory": random.randint(50, 500),
            "warehouse_name": f"Warehouse_{random.randint(1, 5)}",
            "location": {
                "latitude": latitude,
                "longitude": longitude
            },
            "date": current_date.strftime("%Y-%m-%d"),
            # Add some additional daily inventory related information
            "daily_stats": {
                "units_sold": random.randint(0, 30),
                "units_received": random.randint(0, 50),
                "opening_stock": random.randint(50, 200),
                "closing_stock": random.randint(20, 150)
            }
        })

inventory_collection.insert_many(inventory)
```

3.5 Ratings_collection

- **Loops:**
  - **Purpose:** Uses nested loops to iterate through each customer and generate random ratings for a selection of products.
  - **Logic:** Iterates through each customer from the **Customers collection**. Processes one customer at a time. For each customer, **randomly selects 3 to 5 products** using random.sample. Iterates through these selected products to generate individual rating records.

- **Details: Includes customer, product, rating details, and metadata**
  - **Customer Details**: Each rating is linked to a specific customer using the customer_ID field.
  - The customer_ID ensures the ratings can be traced back to the corresponding customer.
  - **Product Details:** Each rating is associated with a specific product using the product_ID field. The product_ID is fetched from the Products collection to maintain consistency.
  - **Rating Details:**
    - **rating:** A randomly generated score between **1 and 5**.
    - **star_display:** A star-based visual representation of the rating (e.g., ★★★★☆ for 4 stars).
    - **comment:** A text-based review generated based on the rating:
    - **High ratings (≥ 4):** Uses templates from the positive comments.
    - **Neutral ratings (= 3):** Uses templates from the neutral comments.
    - **Low ratings (< 3):** Uses templates from the negative comments.

```python
def get_star_display(rating):
    """Convert numeric rating to star display"""
    return '★' * rating + '☆' * (5 - rating)

Explain | Doc | Test | ×
def generate_random_comment(rating):
    """Generate appropriate comment based on rating"""
    if rating >= 4:
        return random.choice(COMMENTS['positive'])
    elif rating == 3:
        return random.choice(COMMENTS['neutral'])
    else:
        return random.choice(COMMENTS['negative'])

Explain | Doc | Test | ×
def create_rating(rating_id, customer_id, product_id):
    """Create a single rating record"""
    rating = random.choice(POSSIBLE_RATINGS)
    random_days = random.randint(1, 365)
    rating_date = datetime.now() - timedelta(days=random_days)

    return {
        'rating_ID': rating_id,
        'customer_ID': customer_id,
        'product_ID': product_id,  # Ensure product_ID is an integer
        'rating': rating,
        'star_display': get_star_display(rating),
        'comment': generate_random_comment(rating),
        'rating_date': rating_date
    }

Explain | Doc | Test | ×
def generate_random_ratings():
    """Generate random ratings"""
    try:
        customers = list(db.Customers.find({}, {'customer_ID': 1, '_id': 0}))
        products = list(db.Products.find({}, {'product_ID': 1, '_id': 0}))

        if not customers or not products:
            raise ValueError("No customer or product data found")

        ratings_collection.drop()  # Clear previous ratings data
        rating_id = 1
        all_ratings = []

        for customer in customers:
            num_ratings = random.randint(3, 5)  # Number of products each customer rates
            selected_products = random.sample(products, num_ratings)

            for product in selected_products:
                product_id = product['product_ID']
                rating_doc = create_rating(
                    rating_id,
                    customer['customer_ID'],
                    int(product_id)  # Ensure product_ID is an integer
                )
                all_ratings.append(rating_doc)
                rating_id += 1

        if all_ratings:
            ratings_collection.insert_many(all_ratings)

        create_indexes()
        calculate_and_display_rating_counts()
        return f"Successfully generated {len(all_ratings)} ratings from {len(customers)} customers"
```

3.6 Calculating average ratings for each product

Query:

From the Ratings collection, grouping the ratings by product_ID and computing the mean average of those ratings to be stored in the Products collection, to be performed at regular intervals (eg once a day)

Design steps

- **Define a helper function to compute rating:** Taking the cursor output from the server as an input, then converting it to a list of ratings and finding the mean of that list
- **Finding a list of product_IDs**
- **Iterating over the list to apply the function**
- **Committing this rating to the Products collection**

```
def calculate_avg_rating(product_id):
    ratings_result = ratings_collection.find({'product_ID': product_id}, {'rating': 1, '_id': 0})
    ratings = [rating['rating'] for rating in ratings_result]
    return sum(ratings) / len(ratings)

# Computing ratings for all products
products = list(products_collection.find({}, {'product_ID':1, '_id':0}))

for product in products:
    product_ID = product['product_ID']
    avg_rating = calculate_avg_rating(product_ID)
    products_collection.update_one({'product_ID':product_ID}, {"$set":{"average_rating": avg_rating}})
```

3.7 Populate Products Collection

- **Product Categories**: Each product has a product category (fresh, mobile phone, CD, home appliance
- **Helper Functions**: Randomly generates variables for documents dependent on product category
- **Initialisation:** An empty products array initialised to store data generated and id initialised as 1.
- **Loops:** Uses helper functions to generate product data (10 per category) and append into the empty products array and then inserted into the products collection

```
# Generate 10 products for each category
categories = ['fresh_product', 'book', 'cd', 'mobile_phone', 'home_appliance']
products = []

product_id = 1
```

```
# Function to get a random name based on category
def get_random_name(category):
    bakery_random = ['Bun', 'Bread', 'Cake', 'Eclair', 'Croissant']
    drinks_random = ['Cola', 'Professor Pupper', 'Water', 'Coffee', 'Tea']
    fruit_veg_random = ['Tomato', 'Zucchini', 'Apple', 'Orange', 'Mango']

    if category == "bakery":
        return random.choice(bakery_random)
    elif category == "drinks":
        return random.choice(drinks_random)
    elif category == "fruit_veg":
        return random.choice(fruit_veg_random)
    else:
        return None
```

```
product_id = 1
for category in categories:
    for _ in range(10):  # Generate 10 products per category
        if category == 'fresh_product':
            product_details = generate_fresh_product_details()
        else:
            product_details = generate_other_product_details(category)

        product = {
            "product_ID": product_id,
            "name": f"{category.capitalize()} {product_id}",
            "description": f"A {category} product description.",
            "price": round(random.uniform(5, 500), 2),
            "product_segment": random.choice(["Consumer", "Luxury", "Essential"]),
            "product_category": category,
            "fresh_product_details" if category == 'fresh_product'
                else "other_product_details": product_details
        }

        products.append(product)
        product_id += 1

# Insert products into the collection
products_collection.insert_many(products)
```

3.8.1 Populate Customers Collection

- **Example Data**: Define arrays of data to help generate names and addresses via randomisation
- **Helper Functions**: Randomly generates location and address data for customers – limited to UK addresses, postcodes and longitudes and latitudes

```python
# Example data for generating customer records
male_names = ["Adam", "Charlie", "Edward", "George", "Jack",
              "Liam", "Nathan", "Oscar", "Paul", "Sam"]
female_names = ["Beth", "Diana", "Fiona", "Hannah", "Ivy",
                "Karen", "Mona", "Olivia", "Rachel", "Tina"]
last_initials = [chr(i) for i in range(65, 91)]  # A-Z
uk_cities = ["London", "Manchester", "Birmingham", "Leeds", "Glasgow",
             "Liverpool", "Bristol", "Sheffield", "Edinburgh", "Cardiff"]
uk_streets = ["High Street", "Station Road", "Church Lane", "Victoria Road",
              "Park Avenue", "Main Street", "Mill Lane", "The Crescent",
              "Queensway", "King Street"]
address_types = ["Home", "Work"]

# Helper function to generate UK postcodes
def generate_uk_postcode():
    area = random.choice(string.ascii_uppercase)  # One letter for the area
    district = str(random.randint(1, 99))  # One or two digits for the district
    outward_code = f"{area}{district}"
    sector = str(random.randint(0, 9))  # Single digit
    unit = ''.join(random.choices(string.ascii_uppercase, k=2))  # Two random letters
    inward_code = f"{sector}{unit}"
    return f"{outward_code} {inward_code}"

# Generate a random latitude and longitude for location within the UK
def generate_uk_location():
    latitude = random.uniform(49.3, 58.6)  # UK latitude bounds
    longitude = random.uniform(-8.1, 1.8)  # UK longitude bounds
    return latitude, longitude
```

3.8.2 Populate Customers Collection

- **Customers Array**: Empty array initialised to store generated customer data
- **Order ID**: Order ID set to 1 initially
- **Loop**: Outer Loop to generate customer data as per schema. Uses randomisation and helper functions

```
# Generate 20 customers
customers = []
order_id = 1
for i in range(20):
    if i < 10:
        first_name = male_names[i]
        gender = "Male"
    else:
        first_name = female_names[i - 10]
        gender = "Female"

    last_initial = random.choice(last_initials)
    name = f"{first_name} {last_initial}"
    age = random.randint(18, 70)

    # Generate 1 address for each customer
    addresses = [{
        "address_type": random.choice(address_types),
        "house_number": random.randint(1, 999),
        "street": random.choice(uk_streets),
        "city": random.choice(uk_cities),
        "postcode": generate_uk_postcode()  # UK postcode generator
    }]

    # Generate latitude and longitude within the UK
    latitude, longitude = generate_uk_location()
```

3.8.3 Populate Customers Collection

- **Current Orders**: Inner Loop to create 2-3 current orders for each customer, empty array is initialised which is then appended with randomly selected product IDs and other details
- **Cart**: Cart is generated with random choice from product IDs as well as total price of products in cart

```
# Fetch all product IDs and their prices from the Products collection
product_data = {product["product_ID"]: product["price"] for product
                in products_collection.find({}, {"product_ID": 1, "price": 1})}
product_ids = list(product_data.keys())

# Generate the cart with product details
cart_products = [
    {
        "product_ID": random.choice(product_ids),
        "quantity": random.randint(1, 5)
    }
    for _ in range(random.randint(1, 5))
]
cart_total = sum(product_data[product["product_ID"]] * product["quantity"] for product in cart_products)
```

```
# Generate 2-3 current orders for each customer
current_orders = []
for _ in range(2):
    products_in_order = [
        {
            "product_ID": random.choice(product_ids),
            "quantity": random.randint(1, 10)
        }
        for _ in range(random.randint(1, 5))  # Each order has 1-5 products
    ]

    # Calculate the order total based on product prices and quantities
    order_total = sum(product_data[product["product_ID"]] * product["quantity"]
    for product in products_in_order)

    # Select customer addresses to be the order destination
    order_destination = f"{addresses[0]['house_number']} {addresses[0]['street']},
    {addresses[0]['city']}, {addresses[0]['postcode']}"

    current_orders.append({
        "order_ID": order_id,
        "order_total": round(order_total, 2),
        "order_status": random.choice(["Pending", "Shipped", "Delivered"]),
        "partner_assigned": random.randint(1, 10),  # Random partner ID
        "order_placed": datetime.now(),  # Current datetime for order placement
        "order_destination": order_destination,
        "products": products_in_order
    })
    order_id += 1
```

3.8.4 Populate Customers Collection

- **Customers Append**: The details of the customer document is then appended into the intially empty customer array until multiple customers with 2-3 orders are generated. The customer_ID is itteraated upon.
- **Insert into Collection**: The customers array is then inserted into the customers collection with the use of insert_many

```
    # Create the customer document
    customers.append({
        "customer_ID": i + 1,
        "name": name,
        "gender": gender,
        "age": age,
        "addresses": addresses,
        "location": {"latitude": latitude,
                     "longitude": longitude},
        "cart": {"cart_total": round(cart_total, 2),
                 "products": cart_products},
        "current_orders": current_orders,
        "recommended_products": random.sample(
            product_ids, k=min(len(product_ids), 3))
    })

# Insert customers into the collection
customers_collection.insert_many(customers)
```

3.9 Populate Past Orders Collection

- **Finding details**: Customer, product and partner IDs are fetched
- **Initialise empty array and order_id**: Empty past_orders array to hold past orders data is initialised and order ID is set to initial value
- **Loop**: Loop is created to generate details per past order as per the schema and appended into empty array
- **Insert Into Colletion**: The past_orders array is then inserted into the past orders collection with the use of insert_many

```
# Fetch customer and product IDs
customer_ids = [customer['customer_ID'] for customer in customers_collection.find({}, {"customer_ID": 1})]
product_ids = [product['product_ID'] for product in products_collection.find({}, {"product_ID": 1})]
partner_ids = [partner['partner_ID'] for partner in partners_collection.find({}, {"partner_ID": 1})]
```

```
# Generate past orders for each customer
past_orders = []
order_id = 70  # Unique order ID
```

```
for customer_id in customer_ids:
    # Each customer has between 5 and 7 past orders
    num_orders = random.randint(5, 7)
    for _ in range(num_orders):
        # Random partner assignment
        partner_id = random.choice(partner_ids)

        # Order details
        num_products = random.randint(1, 5)
        products = []
        order_total = 0

        for _ in range(num_products):
            product_id = random.choice(product_ids)
            quantity = random.randint(1, 10)

            # Fetch product price from the Products collection
            product = products_collection.find_one({
                "product_ID": product_id}, {"price": 1})
            price = product['price'] if product else 0

            products.append({
                "product_ID": product_id,
                "quantity": quantity
            })
            order_total += price * quantity
```

```
        # Random timestamps for order placement and delivery
        # Delivered 1-5 days later
        order_placed = datetime.now() -
                       timedelta(days=random.randint(1, 365))
        order_delivered = order_placed +
                          timedelta(days=random.randint(1, 5))


        # Append the past order to the list
        past_orders.append({
            "order_ID": order_id,
            "customer_ID": customer_id,
            "order_total": round(order_total, 2),
            "partner_assigned": partner_id,
            "order_placed": order_placed,
            "order_delivered": order_delivered,
            "products": products
        })
        order_id += 1

# Insert into PastOrders collection
past_orders_collection.insert_many(past_orders)
```

3.10.1 Samples of Documents generated

An Example of a product document

```
_id: ObjectId('677b7b817b7cd86a9eb6028d')
product_ID : 17
name : "Intermediate Geography"
description : "A book product description.
price : 338.73
product_segment : "Consumer"
product_category : "book"
▾ other_product_details : Object
    dimensions : "12x18x4 cm"
    shipping_weight : 1.11
    supplier_cost : 8.47
  ▾ book : Object
      author_name : "Beth B"
      publisher : "Booksters"
      year_of_publication : 2001
      ISBN : "2BEA6sZM45GCv"
average_rating : 3.8
```

An Example of a customer document

```
_id: ObjectId('677b86537b7cd86a9eb60343')
customer_ID : 16
name : "Karen V"
gender : "Female"
age : 43
▾ addresses : Array (1)
  ▾ 0: Object
      address_type : "Work"
      house_number : 154
      street : "Victoria Road"
      city : "Bristol"
      postcode : "Y86 5WN"
▾ location : Object
    latitude : 56.50452275240466
    longitude : -7.578722303857445
▾ cart : Object
    cart_total : 146
  ▾ products : Array (2)
    ▾ 0: Object
        product_ID : 43
        quantity : 4
    ▾ 1: Object
        product_ID : 18
        quantity : 1
▾ current_orders : Array (2)
  ▾ 0: Object
      order_ID : 31
      order_total : 2045.04
      order_status : "Delivered"
      partner_assigned : 8
      order_placed : 2024-05-24T23:35:54.697+00:00
      order_destination : "154 Victoria Road, Bristol,
    ▾ products : Array (2)
      ▾ 0: Object
          product_ID : 27
          quantity : 4
      ▾ 1: Object
          product_ID : 11
          quantity : 4
email : "karen_v@example.com"
```

An Example of a past order document

```
_id: ObjectId('677b491e3e5ebb4159ecd917')
order_ID : 81
customer_ID : 2
order_total : 1015.86
partner_assigned : 5
order_placed : 2024-01-24T03:08:07.462+00:00
order_delivered : 2024-01-27T03:08:07.462+00:00
▾ products : Array (2)
  ▾ 0: Object
      product_ID : 14
      quantity : 1
  ▾ 1: Object
      product_ID : 16
      quantity : 7
```

3.10.2 Samples of Documents generated

An Example of a store document


```
_id: ObjectId('677b7bad7b7cd86a9eb602c5')
store_ID : 1
address : Object
    house_number : 509
    street : "Park Avenue"
    city : "London"
    postcode : "H99 0XG"
location : Object
    latitude : 57.28534978660986
    longitude : 0.4242731534472419
products_available : Array (5)
    0: Object
        product_ID : 37
        quantity : 62
    1: Object
        product_ID : 50
        quantity : 88
    2: Object
        product_ID : 6
        quantity : 35
    3: Object
        product_ID : 20
        quantity : 13
    4: Object
        product_ID : 34
        quantity : 95
```


An Example of a partners document


```
_id: ObjectId('678002276400bb0b396129c9')
partner_ID : 1
name : "Julia"
gender : "Female"
age : 32
current_location : Object
    latitude : 49.53187931787325
    longitude : -5.537699810207275
active : true
on_errand : true
current_task : 1010
delivery_stats : Object
    total_deliveries : 119
    total_earnings : 952
    rating : 4.8
```


An Example of a rating document


```
_id: ObjectId('677d5f1cd5d4fab8d54d2e9a')
rating_ID : 5
customer_ID : 2
product_ID : 2
rating : 4
star_display : "★★★★☆"
comment : "Highly recommended to everyone!"
rating_date : 2024-11-12T17:06:36.162+00:00
```


An Example of an inventory document


```
_id: ObjectId('677b7bad7b7cd86a9eb602d2')
product_ID : 2
inventory : 158
warehouse_name : "Warehouse_3"
location : Object
    latitude : 52.34924549705344
    longitude : -0.39212468471681294
date : "2025-01-05"
daily_stats : Object
    units_sold : 5
    units_received : 44
    opening_stock : 128
    closing_stock : 147
```

04

Database Query Analysis

4.1 Fresh Product Order Queries Design 1

4.1.1 Query:

Assign the store when a customer orders a fresh product.

The conditions set are: order_ID = 40.

4.1.2 Design steps

- Matching customers and their orders
- Finding stores with ordered products in stock
- Associating product Information
- Finding the closest store
- Projecting final result
- Updating the store inventory

4.1.3 Results

```
[{'_id': ObjectId('677b86537b7cd86a9eb60347'),
  'current_orders': {'order_ID': 40, 'products': {'quantity': 2}},
  'product_details': {'description': 'A fresh_product product description.',
                      'name': 'Mango',
                      'price': 91.92,
                      'product_ID': 2,
                      'product_category': 'fresh_product',
                      'product_segment': 'Consumer'},
  'store_details': {'address': {'city': 'Birmingham',
                                'house_number': 664,
                                'postcode': 'Y85 9LY',
                                'street': 'Station Road'},
                    'products_available': {'quantity': 55},
                    'store_ID': 5},
  'store_distance': 1.9580593759407698}]
Updated stock for Store ID 5, Product ID 2. Updated Quantity: 53.
```

```
pipeline_query1 = [

# 1. Match customers and their orders based on order_ID
    {"$unwind": "$current_orders"},
    {"$match": {"current_orders.order_ID": 40}},
    {"$unwind": "$current_orders.products"},

# 2. Find stores with ordered products in stock
    {"$lookup": { "from": "Stores","let":
    {"productID": "$current_orders.products.product_ID",
     "orderQty": "$current_orders.products.quantity"},
    "pipeline": [{"$unwind": "$products_available"},
                 {"$match":
                    {"$expr":
                      {"$and": [{"$eq": ["$products_available.product_ID", "$$productID"]},
                          {"$gte": ["$products_available.quantity", "$$orderQty"]}]}
                    }}],
            "as": "store_details"}},

# 3. Associate product Information
    {"$unwind": "$store_details"},
    {"$lookup": {"from": "Products",
                 "localField": "current_orders.products.product_ID",
                 "foreignField": "product_ID",
                 "as": "product_details"}},
    {"$unwind": "$product_details"},
# 4. Find the closest store
    {"$addFields": {"store_distance":
        {"$sqrt": {"$add": [{"$pow": [{"$subtract": ["$location.latitude", "$store_details.location.latitude"]}, 2]},
                            {"$pow": [{"$subtract": ["$location.longitude", "$store_details.location.longitude"]}, 2]}]
                   }
        }}},
    {"$sort": {"store_distance": 1}},
    {"$limit": 1},

# 5. Project final result
    {"$project": {"current_orders.order_ID": 1,
                  "current_orders.products.quantity": 1,
                  "product_details.name": 1,
                  "product_details.product_ID": 1,
                  "product_details.description": 1,
                  "product_details.product_segment": 1,
                  "product_details.product_category": 1,
                  "product_details.price": 1,
                  "store_details.address": 1,
                  "store_details.products_available.quantity": 1,
                  "store_details.store_ID": 1,
                  "store_distance": 1}}
                  ]

result_query1 = list(db.Customers.aggregate(pipeline_query1))

import pprint
if result_query1:
    pprint.pprint(result_query1)
# 6. Update the store inventory
if result_query1:
    for doc in result_query1:
        store_id = doc["store_details"]["store_ID"]
        product_id = doc["product_details"]["product_ID"]
        order_qty = doc["current_orders"]["products"]["quantity"]

        db.Stores.update_one({"store_ID": store_id, "products_available.product_ID": product_id},
            {"$inc": {"products_available.$.quantity": -order_qty}} )

        updated_store = db.Stores.find_one(
            {"store_ID": store_id, "products_available.product_ID": product_id},
            {"products_available.$": 1} )

        updated_qty = updated_store["products_available"][0]["quantity"] if updated_store else "N/A"

        print(f"Updated stock for Store ID {store_id}, Product ID {product_id}. Updated Quantity: {updated_qty}.")
```

4.2 Fresh Product Order Queries Design 2

4.2.2 Query:

Assign the nearest available delivery partner after assigning the store.

The conditions set are: order_ID = 40; assigned_store_ID=5.

4.2.2 Design steps

- Matching customers and their orders
- Adding Location Fields of the Customer and the Assigned Store
- Associating Partners and Finding the Nearest Partner
- Projecting final result
- Updating the delivery partner status

4.2.3 Results

```
[{'_id': ObjectId('677b86537b7cd86a9eb60347'),
  'assigned_store_ID': 5,
  'current_orders': {'order_ID': 40},
  'partner_details': {'age': 45,
                      'delivery_stats': {'rating': 3.7},
                      'gender': 'Male',
                      'name': 'Ethan',
                      'partner_ID': 4}}]
Updated partner ID 4 with current_task as 40 and on_errand set to True.
```

```
pipeline_query2 = [
# 1. Match customers and their orders based on order_ID
    {"$unwind": "$current_orders"},
    {"$match": {"current_orders.order_ID": 40}},

# 2. Add location fields of the customer and the assigned store
    {"$addFields": {"customer_location": "$location"}},
    {"$addFields": {"assigned_store_ID": 5}},
    { "$lookup": { "from": "Stores",
                   "localField": "assigned_store_ID",
                   "foreignField": "store_ID",
                   "as": "store_details"}},
    {"$unwind": "$store_details"},
    {"$addFields": {"store_location": "$store_details.location"}},

# 3. Associate partners and find the nearest partner
    {"$lookup": {"from": "Partners",
            "pipeline": [{"$addFields": {"partner_distance":
                {"$sqrt": {"$add": [{"$pow": [{"$subtract": ["$current_location.latitude", "$$store_location.latitude"]}, 2]},
                                    {"$pow": [{"$subtract": ["$current_location.longitude", "$$store_location.longitude"]}, 2]}]}}}},
                {"$sort": {"partner_distance": 1}},
                {"$limit": 1}],
            "let": {"store_location": "$store_location"},
            "as": "partner_details"}},
    {"$unwind": "$partner_details"},

 # 4. Project the final result
     {"$project": {
         "current_orders.order_ID": 1,
         "assigned_store_ID": 1,
         "partner_details.partner_ID": 1,
         "partner_details.name": 1,
         "partner_details.gender": 1,
         "partner_details.age": 1,
         "partner_details.phone": 1,
         "partner_details.delivery_stats.rating": 1,
         "partner_distance": 1
     }}
]

 result_query2 = list(db.Customers.aggregate(pipeline_query2))

 if result_query2:
     pprint.pprint(result_query2)

# 5. Update the delivery partner status
if result_query2:
    partner_id = result_query2[0]["partner_details"]["partner_ID"]
    order_id = result_query2[0]["current_orders"]["order_ID"]

    db.Partners.update_one(
        {"partner_ID": partner_id},
        {"$set": {"on_errand": True,
                  "current_task": order_id }})

    print(f"Updated partner ID {partner_id} with current_task as {order_id} and on_errand set to True.")
```

4.3 Fresh Product Order Queries Design 3

4.3.1 Query:

A customer can use the query to check the partner the partner's current location and ETA.

The conditions set are: order_ID = 40; assigned_store_ID=5; assigned_partner_ID: 4.

4.3.2 Design steps

- Matching customers and their orders
- Adding partner ID and lookup partner details
- Projecting the partner location etc.
- Calculating ETA

4.3.3 Results

```
{'ETA': 9.52,
 '_id': ObjectId('677b86537b7cd86a9eb60347'),
 'current_orders': {'order_ID': 40},
 'location': {'latitude': 53.32918311022124, 'longitude': -3.401784148761587},
 'partner_details': {'current_location': {'latitude': 54.51459008938555,
                                          'longitude': -8.011043744363022},
                     'delivery_stats': {'rating': 3.7},
                     'name': 'Ethan',
                     'partner_ID': 4}}
```

```python
from math import sqrt
pipeline_query3 = [

# 1. Match customers and their orders based on order_ID
    {"$unwind": "$current_orders"},
    {"$match": {"current_orders.order_ID": 40}},

# 2. Add partner ID and lookup partner details
    {"$addFields": {"assigned_partner_ID": 4}},
    {"$lookup": {"from": "Partners",
                 "localField": "assigned_partner_ID",
                 "foreignField": "partner_ID",
                 "as": "partner_details"}},
    {"$unwind": "$partner_details"},

# 3. Project the partner location etc.
    {"$project": {"location": 1,
                  "current_orders.order_ID": 1,
                  "partner_details.partner_ID": 1,
                  "partner_details.name": 1,
                  "partner_details.current_location": 1,
                  "partner_details.delivery_stats.rating": 1}}]

result_query3 = list(db.Customers.aggregate(pipeline_query3))

# 4. Calculate ETA
if result_query3:
    for doc in result_query3:
        customer_loc = doc["location"]
        partner_loc = doc["partner_details"]["current_location"]
        distance = sqrt((customer_loc["latitude"] - partner_loc["latitude"]) ** 2 +
            (customer_loc["longitude"] - partner_loc["longitude"]) ** 2)
        average_speed = 30
        eta = round((distance / average_speed) * 60, 2)
        doc["ETA"] = eta
        pprint.pprint(doc)
else:
    print("No partner could be assigned.")
```

4.4 Fresh Product Order Queries Design 4

4.4.1 Query:

A customer want to cancel the order before the products delivered.

The conditions set are: order_ID = 40; assigned_store_ID=5; product_id=2 ; order_qty= 2.

4.4.2 Design steps

- Restoring the store inventory
- Restoring the delivery partner status

4.4.3 Results

```
Restored stock for Store ID 5, Product ID 2. Updated Quantity: 55.
Updated partner ID 4:current_task set to None, on_errand set to False.
```

```
# Query 4: when a customer cancelled the order of current fresh product after the assignment

# 1. Restore the store inventory
order_id = 40
store_id = 5
product_id = 2
order_qty = 2

store_inventory = db.Stores.find_one(
    {"store_ID": store_id, "products_available.product_ID": product_id},
    {"products_available.$": 1}
)

if store_inventory:
    available_quantity = store_inventory["products_available"][0]["quantity"]

    updated_inventory = available_quantity + order_qty

    db.Stores.update_one({"store_ID": store_id, "products_available.product_ID": product_id},
                         {"$inc": {"products_available.$.quantity": order_qty}})

    updated_store = db.Stores.find_one({"store_ID": store_id, "products_available.product_ID": product_id},
                                       {"products_available.$": 1})

    updated_qty = updated_store["products_available"][0]["quantity"] if updated_store else "N/A"

    print(f"Restored stock for Store ID {store_id}, Product ID {product_id}. Updated Quantity: {updated_qty}.")

 # 2. restore the delivery partner status
 partner_id = 4

 db.Partners.update_one(
     {"partner_ID": partner_id},
     {"$set": {"on_errand": False,
               "current_task": None }})

 print(f"Updated partner ID {partner_id}:current_task set to None, on_errand set to False.")
```

4.5 Customer Purchase Product Queries Design 1

MANCHESTER
1824

4.5.1 Query:

Customers search for the most expensive product within a specific price range among a fixed category of goods, displaying its current inventory status.

The conditions set are： customer_id = 13， product_type = “book”, price_range = (10, 200)， quantity_to_buy = 5

4.5.2 Design steps

- Finding the Product
- Checking Current Inventory
- Adding to Cart
- Marking Purchase History
- Updating Inventory

4.5.3 Results

```
=== Purchase History ===
Previously Purchased: No

=== Inventory Check ===
+----+--------------+---------------------+---------------------+-----------+----------------------------+
|    |   Product ID |   Available Quantity |   Requested Quantity | Success   | Message                    |
+====+==============+=====================+=====================+===========+============================+
|  0 |           19 |                  127 |                    5 | Yes       | Sufficient stock available. |
+----+--------------+---------------------+---------------------+-----------+----------------------------+

=== Updated Cart ===
+----+--------------+------------+--------------+
|    |   product_ID |   quantity |   Total Price |
+====+==============+============+==============+
|  0 |            5 |          2 |        376.2 |
+----+--------------+------------+--------------+
|  1 |           19 |         10 |       1881   |
+----+--------------+------------+--------------+

=== Updated Inventory ===
Reduced inventory for Product ID 19 by 5

=== Purchase Summary ===
+----+---------------------------+-----------------+----------------------+--------------+---------------------+
|    | Book Name                 |   Price per Unit |   Quantity Purchased |   Total Cost |   Updated Cart Total |
+====+===========================+=================+======================+==============+=====================+
|  0 | Introduction to Statistics |           188.1 |                    5 |        940.5 |              2082.34 |
+----+---------------------------+-----------------+----------------------+--------------+---------------------+

=== Most Expensive Book Found ===
|    | _id                      |   product_ID | name                       | description                 |   price | product_segment   | product_category   | other_product_details
                                                                                              |   average_rating |
|  0 | 677b7b817b7cd86a9eb6028f |           19 | Introduction to Statistics | A book product description. |   188.1 | Essential         | book               | {'dimensions': '26x28x3 c
lisher': 'Oxnard Press', 'year_of_publication': 2020, 'ISBN': 'CcYTezMzE6cQu'}} |          2.71429 |
```

```python
# Query parameters
customer_id = 13
product_type = "book"
price_range = (10, 200)
quantity_to_buy = 5

def find_most_expensive_book() -> Optional[Dict]:
    """Find the most expensive book within the price range."""
    try:
        book = products_collection.find_one(
            {'product_category': product_type, 'price': {'$gte': price_range[0], '$lte': price_range[1]}},
            sort=[('price', -1)]
        )
        if not book:
            print("\nNo books found in the specified category and price range.")
            return None

        print("\n=== Most Expensive Book Found ===")
        print(tabulate(pd.DataFrame([book]), headers="keys", tablefmt="grid"))
        return book
    except Exception as e:
        print(f"Error finding book: {str(e)}")
        return None

def check_purchase_history(customer_id: int, product_id: int) -> bool:
    """Check if the customer has purchased the book before."""
    try:
        purchased = past_orders_collection.find_one({
            'customer_ID': customer_id,
            'products.product_ID': product_id
        }) is not None
        print(f"\n=== Purchase History ===\nPreviously Purchased: {'Yes' if purchased else 'No'}")
        return purchased
    except Exception as e:
        print(f"Error checking purchase history: {str(e)}")
        return False

def check_inventory(product_id: int, quantity: int) -> Dict:
    """Check if there is enough stock."""
    inventory_item = inventory_collection.find_one({'product_ID': product_id})
    if not inventory_item:
        return {'success': False, 'message': 'Product does not exist'}

    available_quantity = inventory_item.get('inventory', 0)
    success = available_quantity >= quantity
    message = "Sufficient stock available." if success else f"Only {available_quantity} available, cannot fulfill order."

    print("\n=== Inventory Check ===")
    inventory_df = pd.DataFrame([{
        "Product ID": product_id,
        "Available Quantity": available_quantity,
        "Requested Quantity": quantity,
        "Success": "Yes" if success else "No",
        "Message": message
    }])
    print(tabulate(inventory_df, headers="keys", tablefmt="grid"))
    return {'success': success, 'message': message}
```

4.5 Customer Purchase Product Queries Design 1

4.5.1 Query:

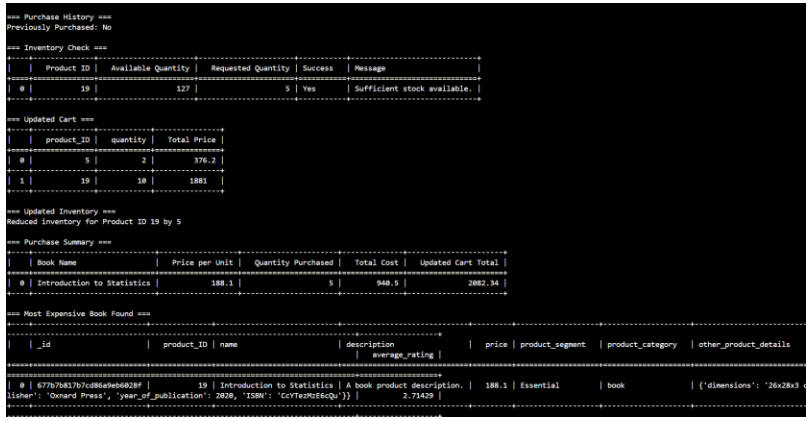
Customers search for the most expensive product within a specific price range among a fixed category of goods, displaying its current inventory status.

The conditions set are： customer_id = 13，product_type = “book”，price_range = (10, 200)， quantity_to_buy = 5

4.5.2 Design steps

- Finding the Product
- Checking Current Inventory
- Adding to Cart
- Marking Purchase History
- Updating Inventory

4.5.3 Results




```python
def update_cart(customer_id: int, book: Dict, quantity: int) -> Dict:
    """Update the customer's cart with the selected book."""
    try:
        item_total = book['price'] * quantity
        customer = customers_collection.find_one({'customer_ID': customer_id})
        if not customer:
            return {'success': False, 'message': f'Customer {customer_id} does not exist.'}

        cart = customer.get('cart', {'products': [], 'cart_total': 0})
        for product in cart['products']:
            if product['product_ID'] == book['product_ID']:
                product['quantity'] += quantity
                break
        else:
            cart['products'].append({'product_ID': book['product_ID'], 'quantity': quantity})

        cart['cart_total'] += item_total
        customers_collection.update_one({'customer_ID': customer_id}, {'$set': {'cart': cart}})

        print("\n=== Updated Cart ===")
        cart_df = pd.DataFrame(cart['products'])
        cart_df['Total Price'] = cart_df['quantity'] * book['price']
        print(tabulate(cart_df, headers="keys", tablefmt="grid"))
        return {'success': True, 'cart_details': cart}
    except Exception as e:
        return {'success': False, 'message': f'Failed to update cart: {str(e)}'}
```

```python
def update_inventory(product_id: int, quantity: int) -> None:
    """Reduce the inventory of the selected product."""
    inventory_collection.update_one({'product_ID': product_id}, {'$inc': {'inventory': -quantity}})
    print(f"\n=== Updated Inventory ===\nReduced inventory for Product ID {product_id} by {quantity}")
```

```python
def purchase_book(customer_id: int, quantity: int) -> None:
    """Main function to purchase a book."""
    try:
        book = find_most_expensive_book()
        if not book:
            return

        check_purchase_history(customer_id, book['product_ID'])

        inventory_check = check_inventory(book['product_ID'], quantity)
        if not inventory_check['success']:
            print(f"\nOperation failed: {inventory_check['message']}")
            return

        cart_update = update_cart(customer_id, book, quantity)
        if not cart_update['success']:
            print(f"\nOperation failed: {cart_update['message']}")
            return

        update_inventory(book['product_ID'], quantity)

        print("\n=== Purchase Summary ===")
```

4.6 Customer Purchase Product Queries Design 2

4.6.1 Query:

Customer checks the current shopping cart contents and remove the lowest-priced item. A new order is then created based on the remaining items in the shopping cart, and payment is processed.

The conditions set are: customer_id = 13, remove lowest-priced item

4.6.2 Design steps

- Retrieve Shopping Cart Contents
- Remove the Lowest-Priced Item
- Update Shopping Cart Total
- Create a New Order
- Record the Order in the Database
- Update Customer Document
- Adjust Inventory Levels

4.6.3 Results

=== Cart Contents ===

	product_ID	quantity
0	34	4
1	23	3

=== Cart After Removing Lowest-Priced Item ===

	product_ID	quantity	price
0	34	4	428.49

Removed product: 23.

=== Order Details ===

	Order ID	Customer ID	Order Total	Order Status	Partner Assigned	Order Destination
0	127	13	1713.96	Paid	3	{'address_type': 'Home', 'house_number': 967, 'street': 'Queensway', 'city': 'Manchester', 'postcode': 'M17 2OY'}

=== Inventory Updated ===

	product_ID	quantity	price	Status
0	34	4	428.49	Reduced Inventory

```python
def get_cart_contents(customer_id: int):
    """Retrieve the cart contents for a given customer ID."""
    customer = customers_collection.find_one({'customer_ID': customer_id})
    if not customer:
        return None, "Customer not found."

    cart = customer.get('cart', {})
    if cart.get("products"):
        print("\n=== Cart Contents ===")
        print(tabulate(pd.DataFrame(cart["products"]), headers="keys", tablefmt="grid"))
    else:
        print("\nCart is empty.")
    return cart, "Cart retrieved successfully."

Explain | Doc | Test | X
def remove_lowest_priced_item(cart: dict):
    if not cart.get('products'):
        return None, "Cart is empty."

    # Fetch product prices from the Products collection
    for product in cart['products']:
        product_data = products_collection.find_one({'product_ID': product['product_ID']}, {'price': 1})
        product['price'] = product_data['price'] if product_data else 0  # Set price to 0 if product not found

    # Find the lowest-priced product
    lowest_priced_product = min(
        cart['products'],
        key=lambda x: x['quantity'] * x['price']  # Sort by total price
    )

    # Remove the lowest-priced product
    cart['products'].remove(lowest_priced_product)

    # Update the cart total
    cart['cart_total'] = sum(
        product['quantity'] * product['price'] for product in cart['products']
    )

    print("\n=== Cart After Removing Lowest-Priced Item ===")
    print(tabulate(pd.DataFrame(cart['products']), headers="keys", tablefmt="grid"))
    return cart, f"Removed product: {lowest_priced_product['product_ID']}."

Explain | Doc | Test | X
def create_order(customer_id: int, cart: dict):
    """Create a new order for the customer."""
    if not cart.get('products'):
        return None, "Cannot create order: Cart is empty."

    # Randomly assign a delivery person from the Partners set
    partner = partners_collection.aggregate([{"$sample": {"size": 1}}])
    partner_id = list(partner)[0]['partner_ID'] if partner else None

    # Generate random destination from customer addresses
    customer = customers_collection.find_one({'customer_ID': customer_id})
    addresses = customer.get("addresses", [])
    order_destination = random.choice(addresses) if addresses else "Default Address"

    # Create the order document
    order = {
        "order_ID": db['PastOrders'].count_documents({}) + 1,  # Generate new order ID
        "customer_ID": customer_id,
        "order_total": cart['cart_total'],
        "order_status": "Paid",
```

```python
    # Insert the order into the PastOrders collection
    past_orders_collection.insert_one(order)

    print("\n=== Order Details ===")
    order_df = pd.DataFrame([{
        "Order ID": order["order_ID"],
        "Customer ID": order["customer_ID"],
        "Order Total": order["order_total"],
        "Order Status": order["order_status"],
        "Partner Assigned": order["partner_assigned"],
        "Order Destination": order["order_destination"]
    }])
    print(tabulate(order_df, headers="keys", tablefmt="grid"))
    return order, "Order created successfully."

Explain | Doc | Test | X
def update_inventory(order):
    """Update inventory quantities based on the order."""
    for product in order['products']:
        inventory_collection.update_one(
            {'product_ID': product['product_ID']},
            {'$inc': {'inventory': -product['quantity']}}
        )
    print("\n=== Inventory Updated ===")
    inventory_df = pd.DataFrame(order['products'])
    inventory_df['Status'] = 'Reduced Inventory'
    print(tabulate(inventory_df, headers="keys", tablefmt="grid"))

Explain | Doc | Test | X
def update_customer_document(customer_id: int, order: dict):
    """Update the customer's document with the new order."""
    update_result = customers_collection.update_one(
        {'customer_ID': customer_id},
        {
            '$set': {'cart': {"products": [], "cart_total": 0}},  # Empty the cart
            '$push': {'current_orders': {
                "order_ID": order['order_ID'],
                "order_total": order['order_total'],
                "order_status": order['order_status'],
                "partner_assigned": order['partner_assigned'],
                "order_placed": order['order_placed'],
                "order_destination": order['order_destination'],
                "products": order['products']
            }}  # Add the order details to current orders
        }
    )
    success_message = "Customer document updated successfully." if update_result.modified_cou
    print(f"\n=== Customer Document Update ===\n{success_message}")
    return update_result.modified_count > 0, success_message

Explain | Doc | Test | X
def process_cart_and_create_order(customer_id: int):
    """Process the cart and create an order for the customer."""
    # Step 1: Retrieve cart contents
    cart, message = get_cart_contents(customer_id)
    if not cart:
        print(message)
        return
```

4.7 Mobile Phone Sales Performance Analysis

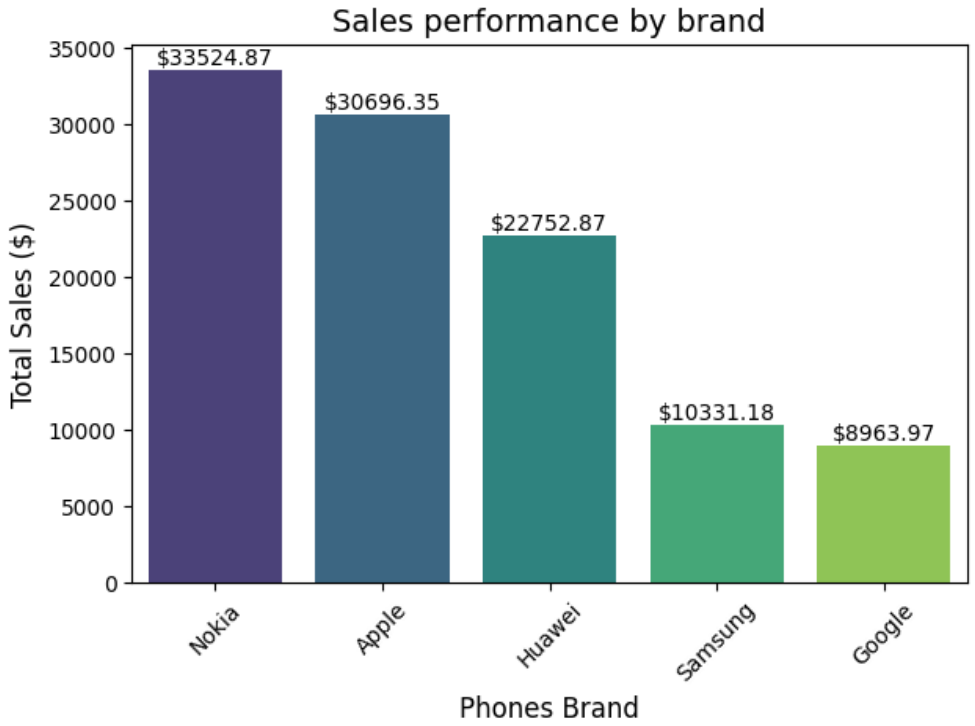
4.7.1 Query:

A manager check the sales performance of mobile phone, identify the top-performing brands to guide marketing or inventory strategies.
The conditions set are: "product_details.product_category“= "mobile_phone".

4.7.2 Design steps

- Match the mobile phones' product IDs in PastOrders .
- Group data by brand.
- Compute the total sales (quantity * price) for each brand.
- Sorting by total sales in descending order.
- Transform the results into a Pandas and use Seaborn to create a bar chart.

4.6.3 Results




```
[{'_id': 'Nokia', 'total_sales': 33524.87},
 {'_id': 'Apple', 'total_sales': 30696.35},
 {'_id': 'Huawei', 'total_sales': 22752.87},
 {'_id': 'Samsung', 'total_sales': 10331.179999999998},
 {'_id': 'Google', 'total_sales': 8963.97}]
```

4.8 Fresh Product Inventory Performance Analysis

4.8.1 Query:

A manager monitor inventory levels of fresh products to detect potential shortages or overstock situations, use query result to make marketing and inventory decision.
The conditions set are: "product_details.product_category“= "fresh_product".

4.8.2 Design steps

- Matching fresh product IDs in the Inventory collection
- Group data by category.
- Compute the total inventory for each category.
- Sorting by total sales in descending order.
- Transform the results into a Pandas and use Seaborn to create a bar chart.

4.7.3 Results


```
[{'_id': 'bakery', 'total_inventory': 2944},
 {'_id': 'drinks', 'total_inventory': 1292},
 {'_id': 'fruit_veg', 'total_inventory': 1054}]
```

4.9 Query: A user search for available fresh products

4.9.1 Query:

A user searching for available fresh products. The products should be displayed based on the user's location.

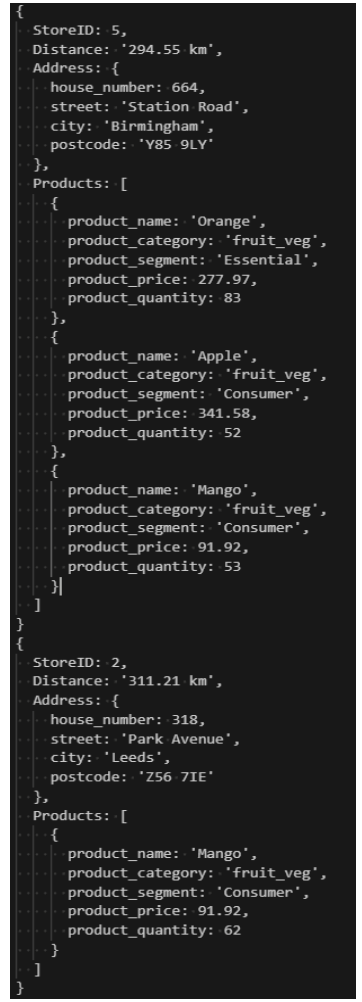
4.9.2 Design steps

- retrieve the document of a user from the Customers collection. Take customer_1 as an example.
- create a 2dsphere index on the location field in the Stores collection, enabling geospatial queries.
- use $geoNear to find stores near the customer_1's location and calculates the distance for each store, storing it in the distance field.
- use $unwind and $lookup to fetch details.
- filter fresh products and group the documents by store_ID.
- sort the stores in ascending order of their distance from the user.
- project the final output, showing the store ID, distance in kilometers, store_address and available fresh products details (name, category, price and quantity).

(MongoDB shell format)

Note: The excessively large calculated distances are due to the latitude and longitude coordinates of users and stores being randomly generated within a certain range.

4.8.3 Results



4.10 Query: Find customers who haven't made any purchases in the last 60 days

4.10.1 Query:

Find customers who haven't made any purchases in the last 60 days, along with additional operations such as sending promotional messages.

4.10.2 Design steps

- calculate the date 60 days ago from the current date.
- join the PastOrders collection with Customers based on customer_ID, adding a past_orders array to each customer document.
- add a new field last_order_date to each document, which is the latest order date (from both past_orders and current_orders).
- filters customers where last_order_date is earlier than 60 days ago.
- select only specific fields (customer_ID, name, email, last_order_date) to include in the output.
- sort the resulting customers by their last_order_date in ascending order.
- make a note to send promotional emails to customers who have not made a purchase in the last 60 days.

4.10.3 Results

```
Customers with no orders in the last 60 days:
   customer_ID      name                email         last_order_date
0           11    Beth P    beth_p@example.com 2024-09-24 04:44:36.616
1           17    Mona Q    mona_q@example.com 2024-10-13 03:08:13.092
2           14  Hannah K  hannah_k@example.com 2024-11-02 14:37:14.664
3            1    Adam M    adam_m@example.com 2024-11-02 16:50:41.327
4            6    Liam Y    liam_y@example.com 2024-11-03 03:08:08.882

note:
Sending promotional email to Beth P at beth_p@example.com.
Sending promotional email to Mona Q at mona_q@example.com.
Sending promotional email to Hannah K at hannah_k@example.com.
Sending promotional email to Adam M at adam_m@example.com.
Sending promotional email to Liam Y at liam_y@example.com.
```

```python
from pymongo import MongoClient
from datetime import datetime, timedelta
import pandas as pd

# note: Find customers who haven't made any purchases in the last 60 days,
# along with additional operations such as sending promotional messages.

# Connect to MongoDB
conn_str = "mongodb+srv://example:example@cluster0.gfqx6.mongodb.net/"
client = MongoClient(conn_str)
db = client["Amazone"]
customers_collection = db["Customers"]

# Step 1: Update all customers by adding an email field
customers = customers_collection.find({})  # Retrieve all customers
for customer in customers:
    customer_name = customer["name"]
    # Generate an email address using the customer's name
    email = f"{customer_name.lower().replace(' ', '_')}@example.com"
    # Update the document by adding the email field
    customers_collection.update_one(
        {"_id": customer["_id"]},
        {"$set": {"email": email}}
    )


# Step 1: Calculate the 60-day threshold date
threshold = datetime.now() - timedelta(days=60)
```

(Python format)

```python
# Step 2: Aggregation pipeline
pipeline = [
    {
        "$lookup": {
            "from": "PastOrders",
            "localField": "customer_ID",
            "foreignField": "customer_ID",
            "as": "past_orders"
        }
    },
    {
        "$addFields": {  # Find the latest order date from both PastOrders and Customers
            "last_order_date": {
                "$max": [
                    {"$max": "$past_orders.order_placed"},  # Maximum date from PastOrders
                    {"$max": "$current_orders.order_placed"}  # Maximum date from current_orders in Customers
                ]
            }
        }
    },
    {
        "$match": {  # Filter customers with no orders in the last 60 days
            "last_order_date": {"$lt": threshold}
        }
    },
    {
        "$project": {
            "_id": 0,
            "customer_ID": 1,
            "name": 1,
            "email": 1,
            "last_order_date": 1
        }
    },
    {
        "$sort": { "last_order_date": 1 }  # Sort by last order date (oldest first)
    }
]
# Step 3: Execute the query
results = list(customers_collection.aggregate(pipeline))

# Step 4: Convert to DataFrame for easy viewing
df = pd.DataFrame(results)
print("Customers with no orders in the last 60 days:")
print(df)

# Step 5: Make a note for sending promotional emails
print("\nnote:")
for customer in results:
    print(f"Sending promotional email to {customer['name']} at {customer['email']}.")
```

4.11 Query: Customer Personalized Product Recommendations

4.11.1 Query:

Through the customer's historical order records (PastOrders) and product information (Products), personalized product recommendations are generated based on the customer's consumption habits, and sorted by consumption category and score, and finally the recommendation results and consumption summary are displayed.

4.11.2 Design steps

- **Filter historical orders by customer ID**: filter the order records of a specific customer.
- **Expand product information in orders:** break down the array of products in each order into individual product records.
- **Calculate the following metrics for each product category:**
  - Total Spending (total_spending): quantity * price
  - Total Purchases (total_purchases): quantity
- **Sort by total spending:** order categories by total_spending in descending order, prioritizing categories where the customer spends the most.
- **Select the top two categories by spending:** results to the two categories with the highest spending.
- **Identify the most frequently purchased categories:** From the aggregation results, determine the top two product categories where the customer has spent the most.
- **Recommend based on ratings:**
  - Filter products in the Products collection that belong to these categories.
  - Sort the products by their ratings (average_rating) in descending order.
  - Select the **top 5 highest-rated** products in each category as recommendations.

4.11.3 Results

```python
class RecommendationSystem:
    Explain | Doc | Test | ×
    def __init__(self, uri, db_name):
        self.client = MongoClient(uri)
        self.db = self.client[db_name]

    Explain | Doc | Test | ×
    def assign_products_to_customers(self):
        """Assign 5 products to each customer and update Customers collection"""
        customers = list(self.db.Customers.find({}, {"customer_ID": 1, "_id": 0}))
        products = list(self.db.Products.find({}, {"product_ID": 1, "_id": 0}))

        if not customers or not products:
            print("No customer or product data found.")
            return

        table_data = []  # store the results for display

        for customer in customers:
            # Randomly select 5 products for the customer
            selected_products = random.sample(products, 5)
            product_ids = [product['product_ID'] for product in selected_products]

            # Update the Customers collection with the rated products
            self.db.Customers.update_one(
                {"customer_ID": customer["customer_ID"]},
                {"$set": {"rated_products": product_ids}}
            )

            # store the results for display
            table_data.append({"Customer ID": customer["customer_ID"], "Rated Products": product_ids})

        # show the results
        print("\n=== Assigned Products to Customers ===")
        print(tabulate(pd.DataFrame(table_data), headers="keys", tablefmt="grid"))

    Explain | Doc | Test | ×
    def get_customer_summary(self, customer_id):
        """Aggregate past orders to summarize spending by category"""
        pipeline = [
            {"$match": {"customer_ID": customer_id}},
            {"$unwind": "$products"},
            {
                "$lookup": {
                    "from": "Products",
                    "localField": "products.product_ID",
                    "foreignField": "product_ID",
                    "as": "product_details"
                }
            },
            {"$unwind": "$product_details"},
            {
                "$group": {
                    "_id": "$product_details.product_category",
                    "total_spending": {
                        "$sum": {"$multiply": ["$products.quantity", "$product_details.price"]}
                    },
                    "total_purchases": {"$sum": "$products.quantity"},
                }
            },
            {"$sort": {"total_spending": -1}},
```

4.12 Query: Customer Personalized Product Recommendations

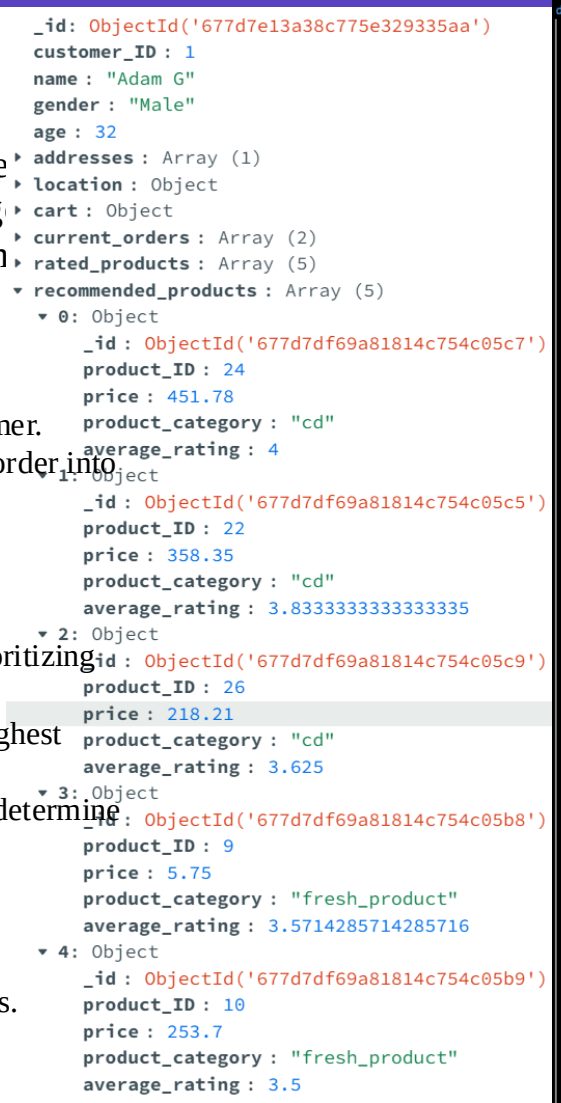
4.12.1 Query:

Through the customer's historical order records (PastOrders) and information (Products), personalized product recommendations are generate on the customer's consumption habits, and sorted by consumption categ score, and finally the recommendation results and consumption summ displayed.

4.12.2 Design steps

- **Filter historical orders by customer ID**: filter the order records of a specific customer.
- **Expand product information in orders:** break down the array of products in each order into individual product records.
- **Calculate the following metrics for each product category:**
  - Total Spending (total_spending): quantity * price
  - Total Purchases (total_purchases): quantity
- **Sort by total spending:** order categories by total_spending in descending order, prioritizing categories where the customer spends the most.
- **Select the top two categories by spending:** results to the two categories with the highest spending.
- **Identify the most frequently purchased categories:** From the aggregation results, determine the top two product categories where the customer has spent the most.
- **Recommend based on ratings:**
  - Filter products in the Products collection that belong to these categories.
  - Sort the products by their ratings (average_rating) in descending order.
  - Select the **top 5 highest-rated** products in each category as recommendations.

4.12.3 Results




```python
def generate_recommendations(self, customer_id):
    """Generate product recommendations based on inventory levels"""
    customer_summary = self.get_customer_summary(customer_id)

    if not customer_summary:
        print(f"No spending data for customer {customer_id}")
        return []

    recommendations = []
    for category_data in customer_summary:
        category = category_data["_id"]

        # Fetch products in the category from Products and Inventory collections
        pipeline = [
            {"$match": {"product_category": category}},
            {
                "$lookup": {
                    "from": "Inventory",
                    "localField": "product_ID",
                    "foreignField": "product_ID",
                    "as": "inventory_data"
                }
            },
            {"$unwind": "$inventory_data"},
            {
                "$project": {
                    "product_ID": 1,
                    "price": 1,
                    "product_category": 1,
                    "inventory": "$inventory_data.inventory"
                }
            },
            {"$sort": {"inventory": -1}},
            {"$limit": 5}
        ]

        category_recommendations = list(self.db.Products.aggregate(pipeline))
        recommendations.extend(category_recommendations)

    # Select the top 5 products across both categories
    recommendations = recommendations[:5]

    # Update recommended products in the Customers collection
    self.db.Customers.update_one(
        {"customer_ID": customer_id},
        {"$set": {"recommended2_products": recommendations}}
    )

    # show the results
    print("\n=== Recommended Products ===")
    recommendations_df = pd.DataFrame(recommendations)
    recommendations_df = recommendations_df[["product_ID", "price", "product_category", "inventory"]]
    recommendations_df.rename(columns={
        "product_ID": "Product ID",
        "price": "Price",
        "product_category": "Category",
        "inventory": "Inventory"
    }, inplace=True)
    print(tabulate(recommendations_df, headers="keys", tablefmt="grid"))
```

4.13 Query: Comparing earnings by partner

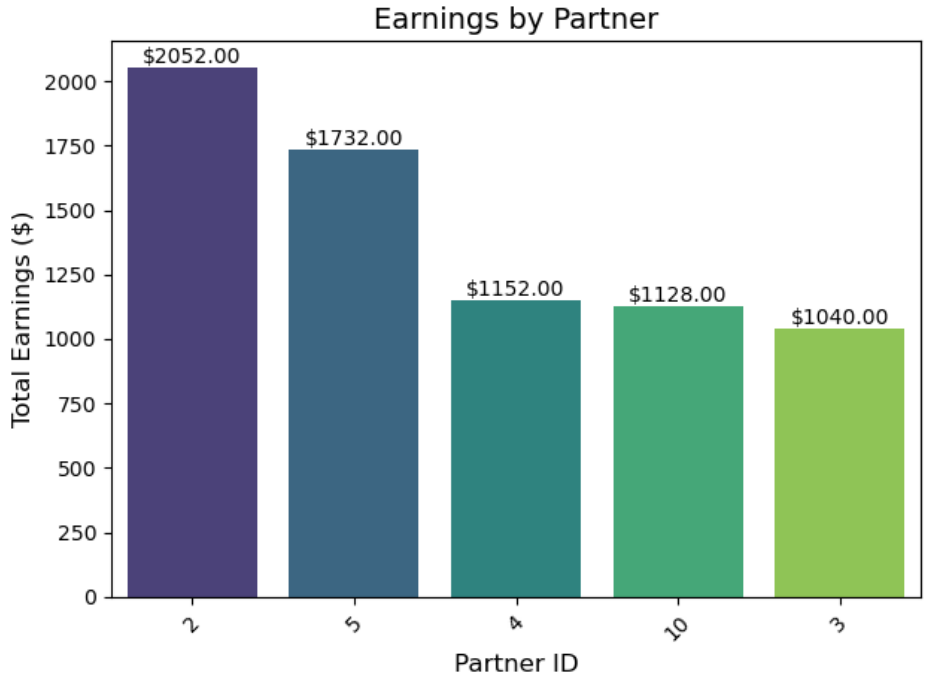
4.13.1 Query:

Computing and visualizing the total earnings by partner and finding the top 5 partners for total earnings

4.13.2 Design steps

- **Project partner_IDs and earnings**
- **Sort by earnings descending:** Limit to top 5 partners
- **Convert to a Pandas dataframe**
- **Use seaborn to create a barplot**

4.13.3 Results



	partner_ID	earnings
0	2	2052
1	5	1732
2	4	1152
3	10	1128
4	3	1040

```
pipeline = [
# Firstly getting the fields we need
    {"$project":{"partner_ID":1,
                 "earnings":"$delivery_stats.total_earnings",
                 "_id":0}},
# Sorting the output in decreasing order of earnings
    {"$sort":{"earnings":-1}},
# Finding the top 5 partners for earnings
    {"$limit":5}
]

# Query: Partner earnings
partner_stats = list(partners_collection.aggregate(pipeline))
```

Thanks for Watching