ST207 Databases Group Project

Group 10

Candidate /LSE ID Numbers:

23007

3809

22318

99981

Topic Description:

This report examines the creation of a General Hospital Management System database using a NOSQL application, MongoDB. It was chosen because it offers a flexible and scalable approach to data storage and retrieval compared to traditional relational databases. We found it to be most suitable to handle diverse and evolving data structures. Our hospital setting explores various essential entities for our chosen environment i.e. patients, doctors, appointments, procedures, diagnostic tests, insurance, prescriptions.

We aimed to model several complex relationships in each one of our queries, all considering a specific use case where specific data was required to answer necessary questions. Some relationships included: patient-doctor relations, appointment setting, revenue breakdown, diagnostic results, prescription dosages and health evolution. We aimed to model our data after diverse roles and departments in the medical field as well as myriad medicines and health conditions.

Data Description:

The data within a hospital environment is typically sensitive and not available publicly. Therefore, to uphold data privacy and confidentiality, we merged authentic data related to procedure types and billing with artificially generated data related to personnel. Our synthetic data was generated using AI technology, and the fusion of real and synthetic data forms a dynamic dataset ideal for simulating real-world scenarios.

The original dataset is sourced from Data.World and contains machine readable hospital pricing information about the Doylestown Hospital in Pennsylvania. The data was collected programmatically, using custom script written in Node.js and utilizing Microsoft Playwright. We chose corresponding data from outpatient fees to integrate into our procedures and billing csv files. In procedures, the Doylestown 'description' attribute is represented as 'type' and in billing, 'hospital_charge' is represented as 'procedurePrice' and 'self_pay' is modeled as 'coPayment.' Therefore, our database closely resembles an American hospital management system, drawing inspiration from the structure and practices found in Doylestown Hospital.

Beyond billing information, our dataset encompasses a diverse array of hospital data. Specifically, we have generated 100 personnel records, including 60 patients, 20 doctors, and 20 nurses. There are 100 appointment records, representing the interactions between patients and doctors. Furthermore, the dataset comprises 60 insurance records, 80 prescription records, 60 procedure records, 100 diagnostic test records, and 60 billing records. Our dataset serves as a condensed representation of the types of records found within our management system. To ensure data consistency and integrity, the relationships between these data entities are established by linking them through their unique IDs.

Our dataset is stored in various CSV files, and they are provided alongside this report. To process and manage this data, we have utilized PyMongo, a Python library for MongoDB, to read the CSV files and build relationships among the collections, so that the data is linked appropriately using object IDs. This process allows for efficient querying, analysis, and retrieval of hospital data.

Source of the original dataset:
https://data.world/ushealthcarepricing/doylestown-hospital-standard-charges

Data Modelling:

Below we will provide the collections and attributes that we used for our database. The relationships have been provided in our **ER diagram** which we attached in our Github file.

1. Personnel Collection:
    - `personnelID`: Original ID used to establish relationships.
    - `name`: User name.
    - `dateOfBirth`: Date of birth.
    - `address`: User address.
    - `contactInfo`: Contact information.
    - `timestamp`: Timestamp.

2. Patients:
    - `personnelID`: Related personnel object ID.
    - `patientID`: Patient's unique number.
    - `insuranceID`: Related patient insurance object ID.
    - `prescriptionID`: Related prescription object ID.
    - `diagnosticTestID`: Related diagnostic test object ID.
    - `appointmentID`: Related appointment object ID.
    - `weight`: Patient weight in pounds.
    - `height`: Patient height in inches.
    - `allergies`: Allergies description.
    - `familyHistory`: Disease history.
    - `severity`: Severity level.

3. Doctors:
    - `personnelID`: Related personnel object ID.
    - `doctorID`: Doctor's unique identifier.
    - `department`: Doctor's department.
    - `medicalLicenseNumber`: Doctor's medical license number.
    - `boardCertification`: Doctor's board certification.
    - `insuranceAccepted`: Types of insurance accepted by the doctor.
    - `prescriptions`: List of prescription object IDs associated with the doctor.
    - `diagnosticTests`: List of diagnostic test object IDs associated with the doctor.

4. Nurses:

- `personnelID`: Related personnel object ID.
- `nurseID`: Nurse's unique identifier.
- `department`: Nurse's department.
- `nursingLicenseNumber`: Nurse's nursing license number.
- `certifications`: Nurse's certifications.

5. Insurance:
    - `insuranceID`: Related insurance object ID.
    - `patientID`: Patient's unique number.
    - `insurancePlanName`: Name of the insurance plan.
    - `policyholder`: Relationship of the patient to the policyholder.
    - `policyNumber`: Policy number associated with the insurance.
    - `groupNumber`: Group number associated with the insurance.
    - `coverageStartDate`: Start date of coverage.
    - `coverageEndDate`: End date of coverage.

6. Appointments:
    - `appointmentID`: Appointment unique identifier.
    - `patientID`: Patient's unique number.
    - `doctorID`: Doctor's object ID.
    - `type`: Type of appointment or procedure.
    - `date`: Date and time of the appointment.

7. Prescriptions:
    - `prescriptionID`: Prescription unique identifier.
    - `patientID`: Patient's unique number.
    - `prescribingDoctorID`: Doctor's object ID.
    - `medication`: Name of the prescribed medication.
    - `dosage`: Dosage information for the medication.
    - `duration`: Duration for which the medication should be taken.

8. Procedures:
    - `procedureID`: Procedure unique identifier.
    - `patientID`: Patient's unique number.
    - `doctorID`: Doctor's unique identifier.
    - `nurseID`: Nurse's unique identifier.
    - `type`: Type of procedure.
    - `durationOfProcedure`: Duration of the procedure.
    - `dateOfProcedure`: Date and time of the procedure.

9. Diagnostic Tests:
    - `diagnosticTestID`: Diagnostic test unique identifier.
    - `patientID`: Patient's unique number.
    - `orderingDoctorID`: Ordering doctor's unique identifier.

- `testingDoctorID`: Testing doctor's unique identifier.
- `testType`: Type of diagnostic test.
- `testDate`: Date and time of the test.
- `testResult`: Result of the diagnostic test.
- `testLocation`: Location where the test was conducted.

10. Billing:
- `billingID`: Billing unique identifier.
- `patientID`: Patient's unique number.
- `procedureID`: Procedure unique identifier.
- `procedurePrice`: Price of the procedure.
- `coPayment`: Co-payment amount.
- `billingDate`: Date of billing.
- `serviceDate`: Date of service.
- `paymentReceivedDate`: Date when payment was received.

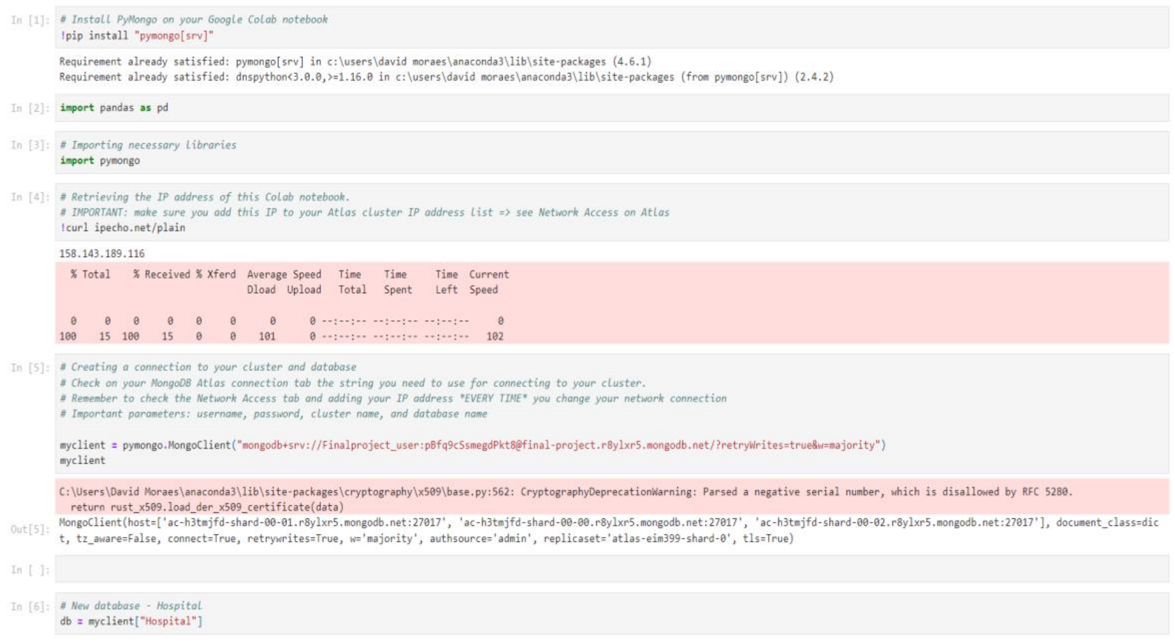
Database Creation and Design Steps:

The database are created by loading the CSV dataset and are processed for their relationships using pymongo library. The following describes some of the important parts of creating the database.

1. Connect to MongoDB cluster
2. We installed the pymongo and pandas library
3. We obtained the ip address of our colab notebook and pasted that into Mongo DB under Network access.
4. We created a Hospital database using the Mongo Client Object
5. We made a function to read the CSV data from the CSV's of all our data which we uploaded to Github.
6. We made relationships between collections by updating documents with the corresponding object ID's.

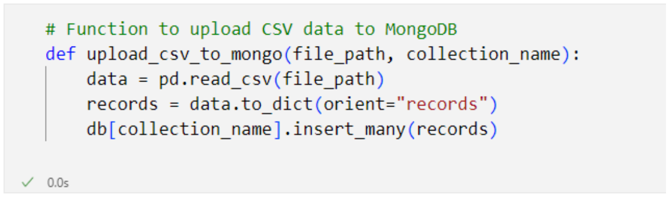
The report contains pictures of our code to demonstrate these steps.

Steps 1-4 (the 5th code block contains both the atlas connection key as well as password)

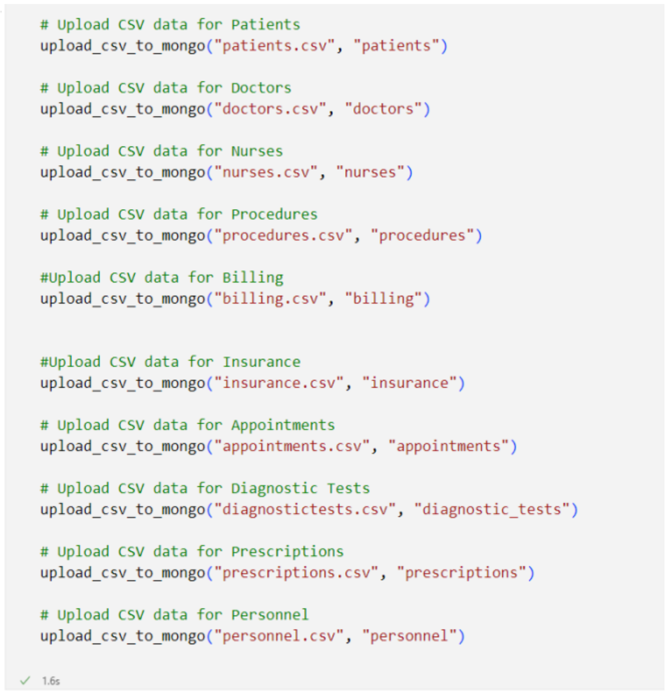


```
In [1]: # Install PyMongo on your Google Colab notebook
        !pip install "pymongo[srv]"

        Requirement already satisfied: pymongo[srv] in c:\users\david moraes\anaconda3\lib\site-packages (4.6.1)
        Requirement already satisfied: dnspython<3.0.0,>=1.16.0 in c:\users\david moraes\anaconda3\lib\site-packages (from pymongo[srv]) (2.4.2)

In [2]: import pandas as pd

In [3]: # Importing necessary libraries
        import pymongo

In [4]: # Retrieving the IP address of this Colab notebook.
        # IMPORTANT: make sure you add this IP to your Atlas cluster IP address list => see Network Access on Atlas
        !curl ipecho.net/plain

        158.143.189.116
          % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                         Dload  Upload   Total   Spent    Left  Speed

          0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
        100    15  100    15    0     0    101      0 --:--:-- --:--:-- --:--:--   102

In [5]: # Creating a connection to your cluster and database
        # Check on your MongoDB Atlas connection tab the string you need to use for connecting to your cluster.
        # Remember to check the Network Access tab and adding your IP address *EVERY TIME* you change your network connection
        # Important parameters: username, password, cluster name, and database name

        myclient = pymongo.MongoClient("mongodb+srv://Finalproject_user:pBfq9cSsmegdPkt8@final-project.r8ylxr5.mongodb.net/?retryWrites=true&w=majority")
        myclient

        C:\Users\David Moraes\anaconda3\lib\site-packages\cryptography\x509\base.py:562: CryptographyDeprecationWarning: Parsed a negative serial number, which is disallowed by RFC 5280.
          return rust_x509.load_der_x509_certificate(data)
Out[5]: MongoClient(host=['ac-h3tmjfd-shard-00-01.r8ylxr5.mongodb.net:27017', 'ac-h3tmjfd-shard-00-00.r8ylxr5.mongodb.net:27017', 'ac-h3tmjfd-shard-00-02.r8ylxr5.mongodb.net:27017'], document_class=dic
        t, tz_aware=False, connect=True, retrywrites=True, w='majority', authsource='admin', replicaset='atlas-eim399-shard-0', tls=True)

In [ ]:

In [6]: # New database - Hospital
        db = myclient["Hospital"]
```

Step 5: Define read CSV function and import each CSV file



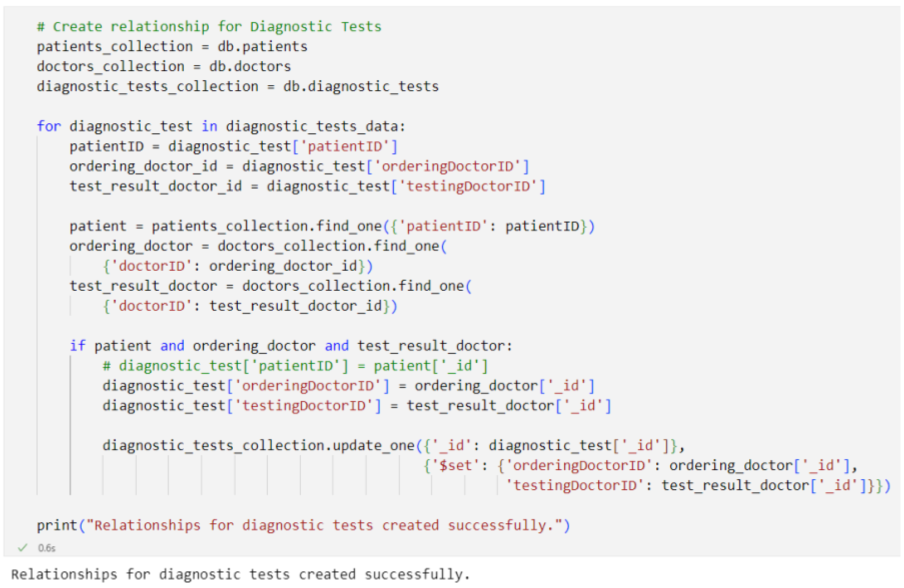
```
# Function to upload CSV data to MongoDB
def upload_csv_to_mongo(file_path, collection_name):
    data = pd.read_csv(file_path)
    records = data.to_dict(orient="records")
    db[collection_name].insert_many(records)

✓ 0.0s
```



```
# Upload CSV data for Patients
upload_csv_to_mongo("patients.csv", "patients")

# Upload CSV data for Doctors
upload_csv_to_mongo("doctors.csv", "doctors")

# Upload CSV data for Nurses
upload_csv_to_mongo("nurses.csv", "nurses")

# Upload CSV data for Procedures
upload_csv_to_mongo("procedures.csv", "procedures")

#Upload CSV data for Billing
upload_csv_to_mongo("billing.csv", "billing")


#Upload CSV data for Insurance
upload_csv_to_mongo("insurance.csv", "insurance")

# Upload CSV data for Appointments
upload_csv_to_mongo("appointments.csv", "appointments")

# Upload CSV data for Diagnostic Tests
upload_csv_to_mongo("diagnostictests.csv", "diagnostic_tests")

# Upload CSV data for Prescriptions
upload_csv_to_mongo("prescriptions.csv", "prescriptions")

# Upload CSV data for Personnel
upload_csv_to_mongo("personnel.csv", "personnel")

✓ 1.6s
```

Step 6: Process relationships between collections

The following screen shot is one of the procedure. The other relationships are built in a similar way.

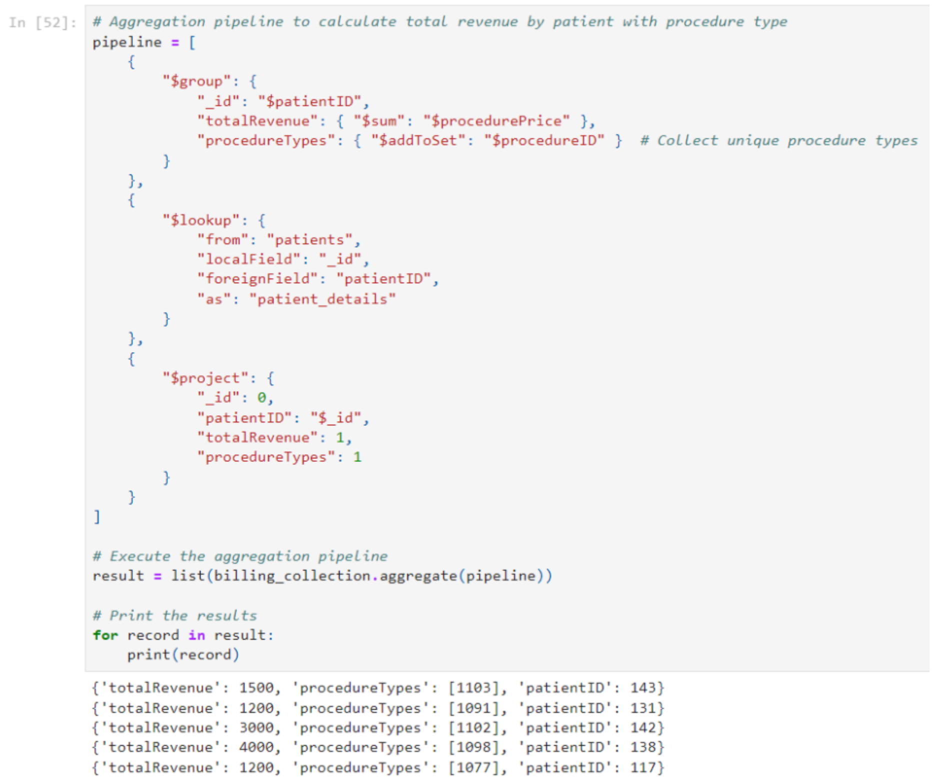


```
# Create relationship for Diagnostic Tests
patients_collection = db.patients
doctors_collection = db.doctors
diagnostic_tests_collection = db.diagnostic_tests

for diagnostic_test in diagnostic_tests_data:
    patientID = diagnostic_test['patientID']
    ordering_doctor_id = diagnostic_test['orderingDoctorID']
    test_result_doctor_id = diagnostic_test['testingDoctorID']

    patient = patients_collection.find_one({'patientID': patientID})
    ordering_doctor = doctors_collection.find_one(
        {'doctorID': ordering_doctor_id})
    test_result_doctor = doctors_collection.find_one(
        {'doctorID': test_result_doctor_id})

    if patient and ordering_doctor and test_result_doctor:
        # diagnostic_test['patientID'] = patient['_id']
        diagnostic_test['orderingDoctorID'] = ordering_doctor['_id']
        diagnostic_test['testingDoctorID'] = test_result_doctor['_id']

        diagnostic_tests_collection.update_one({'_id': diagnostic_test['_id']},
                                              {'$set': {'orderingDoctorID': ordering_doctor['_id'],
                                                        'testingDoctorID': test_result_doctor['_id']}})

print("Relationships for diagnostic tests created successfully.")
```

```
Relationships for diagnostic tests created successfully.
```

Database Usage:

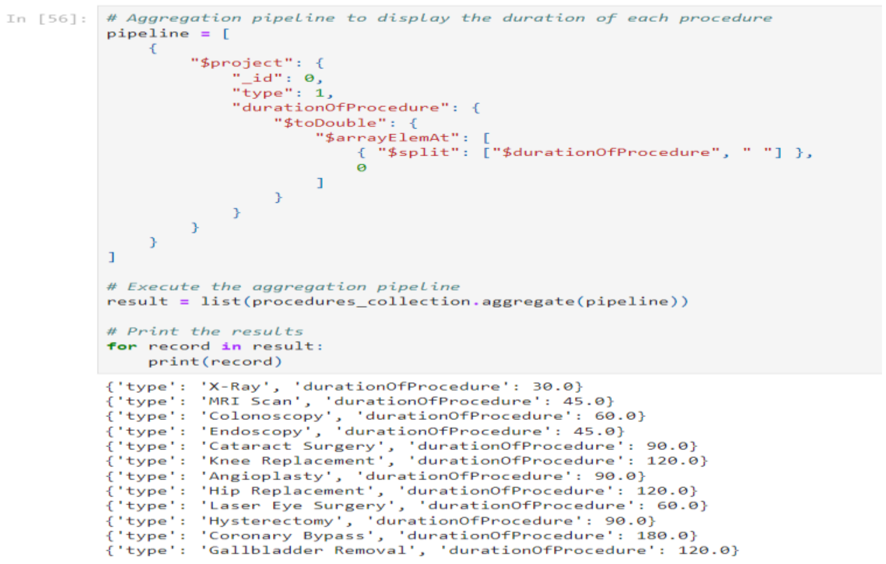
Query 1 rationale: Hospitals like any business, need a detailed breakdown of the effect all their procedures and clients (patients) have on their inflow of cash. This query uses an aggregation pipeline with $groupby, $lookup and $project for a detailed and concise view of the total revenue for each patient and for their respective procedure. The output continues for all other patients but its been cropped for conciseness, the same is true for all the other queries shown.



```
In [52]: # Aggregation pipeline to calculate total revenue by patient with procedure type
pipeline = [
    {
        "$group": {
            "_id": "$patientID",
            "totalRevenue": { "$sum": "$procedurePrice" },
            "procedureTypes": { "$addToSet": "$procedureID" }  # Collect unique procedure types
        }
    },
    {
        "$lookup": {
            "from": "patients",
            "localField": "_id",
            "foreignField": "patientID",
            "as": "patient_details"
        }
    },
    {
        "$project": {
            "_id": 0,
            "patientID": "$_id",
            "totalRevenue": 1,
            "procedureTypes": 1
        }
    }
]

# Execute the aggregation pipeline
result = list(billing_collection.aggregate(pipeline))

# Print the results
for record in result:
    print(record)
```

```
{'totalRevenue': 1500, 'procedureTypes': [1103], 'patientID': 143}
{'totalRevenue': 1200, 'procedureTypes': [1091], 'patientID': 131}
{'totalRevenue': 3000, 'procedureTypes': [1102], 'patientID': 142}
{'totalRevenue': 4000, 'procedureTypes': [1098], 'patientID': 138}
{'totalRevenue': 1200, 'procedureTypes': [1077], 'patientID': 117}
```

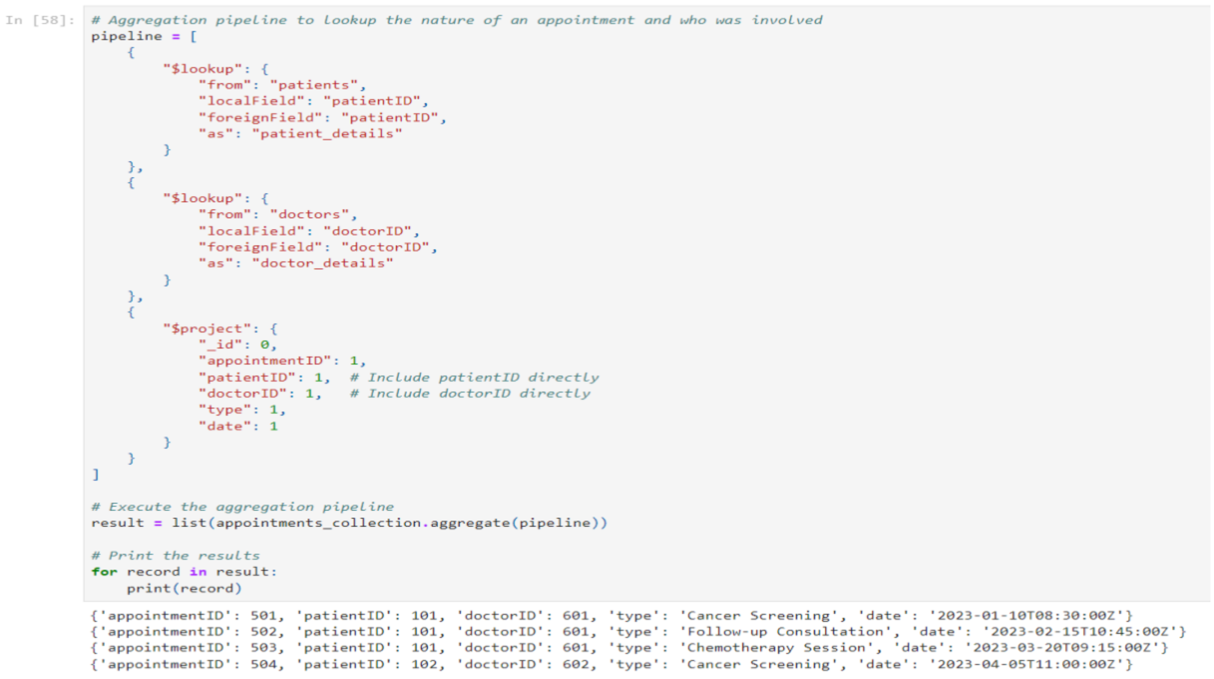
Query 2 rationale: It's important for a hospital to accurately understand the duration of its procedures for optimal scheduling and appointment setting. This query needed a $split function to separate the duration string into parts and a $arrayElemat to extract the numeric part of the string.



```
In [56]: # Aggregation pipeline to display the duration of each procedure
         pipeline = [
             {
                 "$project": {
                     "_id": 0,
                     "type": 1,
                     "durationOfProcedure": {
                         "$toDouble": {
                             "$arrayElemAt": [
                                 { "$split": ["$durationOfProcedure", " "] },
                                 0
                             ]
                         }
                     }
                 }
             }
         ]

         # Execute the aggregation pipeline
         result = list(procedures_collection.aggregate(pipeline))

         # Print the results
         for record in result:
             print(record)
```

```
{'type': 'X-Ray', 'durationOfProcedure': 30.0}
{'type': 'MRI Scan', 'durationOfProcedure': 45.0}
{'type': 'Colonoscopy', 'durationOfProcedure': 60.0}
{'type': 'Endoscopy', 'durationOfProcedure': 45.0}
{'type': 'Cataract Surgery', 'durationOfProcedure': 90.0}
{'type': 'Knee Replacement', 'durationOfProcedure': 120.0}
{'type': 'Angioplasty', 'durationOfProcedure': 90.0}
{'type': 'Hip Replacement', 'durationOfProcedure': 120.0}
{'type': 'Laser Eye Surgery', 'durationOfProcedure': 60.0}
{'type': 'Hysterectomy', 'durationOfProcedure': 90.0}
{'type': 'Coronary Bypass', 'durationOfProcedure': 180.0}
{'type': 'Gallbladder Removal', 'durationOfProcedure': 120.0}
```

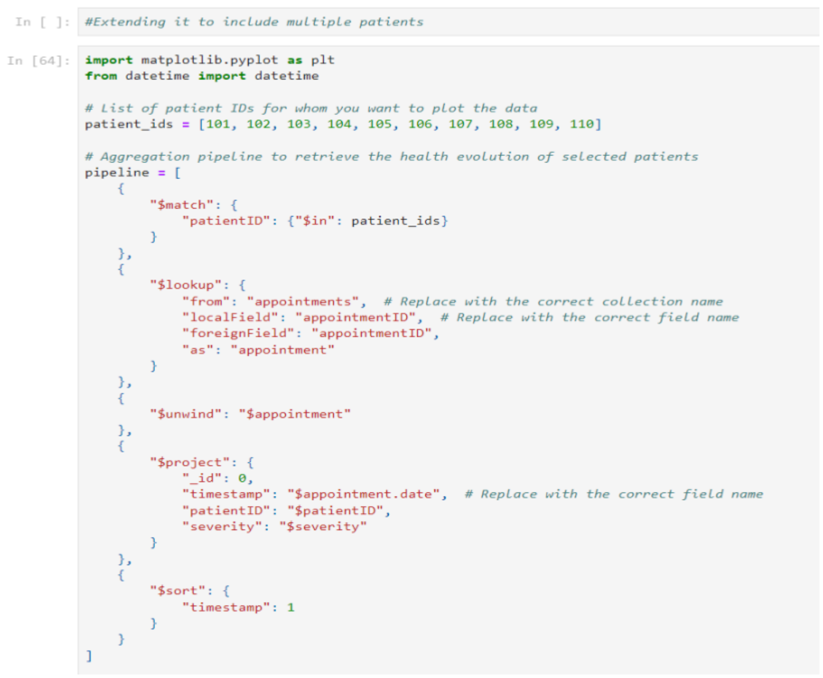
Query 3 rationale: This query focuses on creating a concise view for hospitals to track patient-doctor interactions as well as relevant procedures. It uses $lookup to get information from relevant collections and then $project to show the fields of interest.



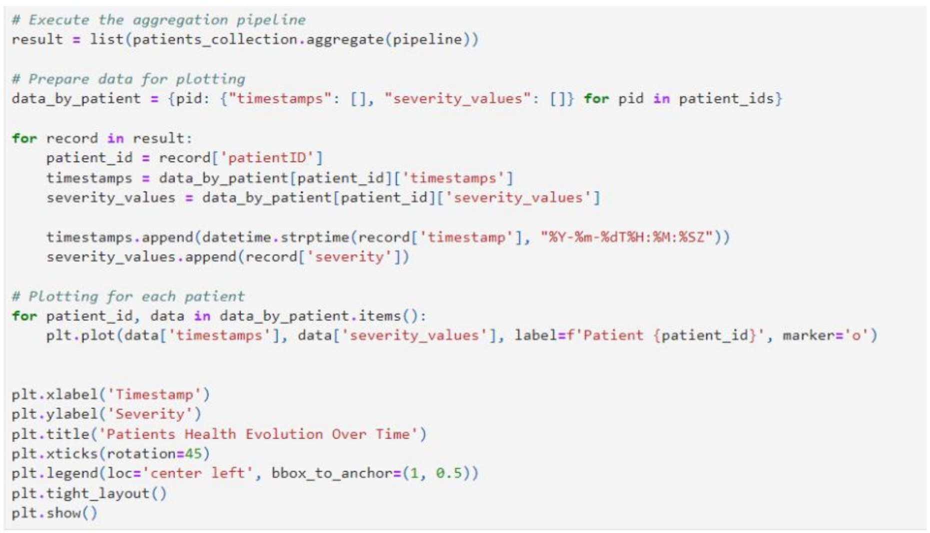
```
In [58]: # Aggregation pipeline to lookup the nature of an appointment and who was involved
         pipeline = [
             {
                 "$lookup": {
                     "from": "patients",
                     "localField": "patientID",
                     "foreignField": "patientID",
                     "as": "patient_details"
                 }
             },
             {
                 "$lookup": {
                     "from": "doctors",
                     "localField": "doctorID",
                     "foreignField": "doctorID",
                     "as": "doctor_details"
                 }
             },
             {
                 "$project": {
                     "_id": 0,
                     "appointmentID": 1,
                     "patientID": 1,  # Include patientID directly
                     "doctorID": 1,   # Include doctorID directly
                     "type": 1,
                     "date": 1
                 }
             }
         ]

         # Execute the aggregation pipeline
         result = list(appointments_collection.aggregate(pipeline))

         # Print the results
         for record in result:
             print(record)
```

```
{'appointmentID': 501, 'patientID': 101, 'doctorID': 601, 'type': 'Cancer Screening', 'date': '2023-01-10T08:30:00Z'}
{'appointmentID': 502, 'patientID': 101, 'doctorID': 601, 'type': 'Follow-up Consultation', 'date': '2023-02-15T10:45:00Z'}
{'appointmentID': 503, 'patientID': 101, 'doctorID': 601, 'type': 'Chemotherapy Session', 'date': '2023-03-20T09:15:00Z'}
{'appointmentID': 504, 'patientID': 102, 'doctorID': 602, 'type': 'Cancer Screening', 'date': '2023-04-05T11:00:00Z'}
```

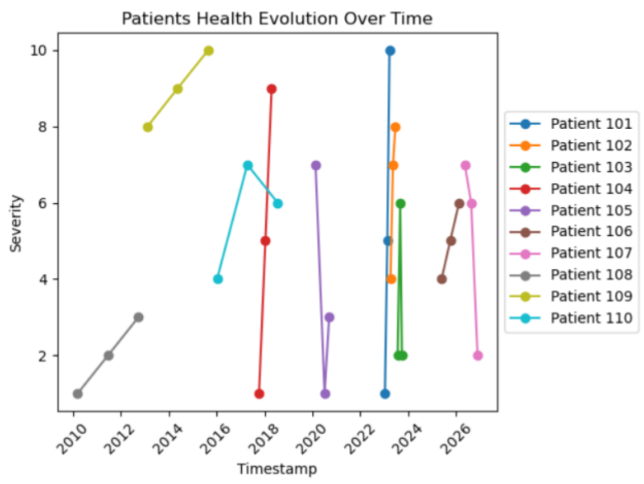
Query 4 rationale: According to our proposal feedback we realised the importance of temporal queries such as patients health evolution over time. Hospitals would employ these temporal tracking to appropriately respond to dynamic changes in a patients condition over time. This query needed $match to get specific patient ID's, $sort to sort by timestamp in ascending order and $unwind for further processing. It also used matplotlib to plot the changes in patients condition across several years. A similar but unincluded query was made to track patient weights across time.

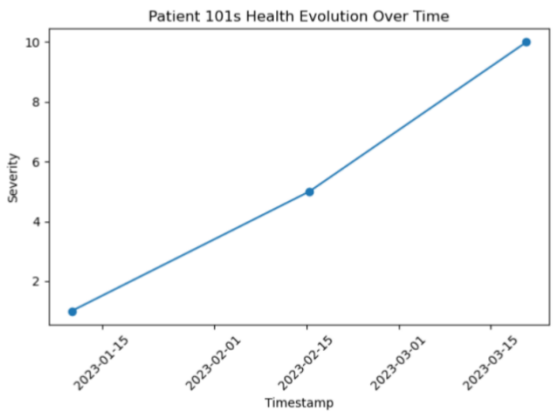


```
In [ ]: #Extending it to include multiple patients

In [64]: import matplotlib.pyplot as plt
         from datetime import datetime

         # List of patient IDs for whom you want to plot the data
         patient_ids = [101, 102, 103, 104, 105, 106, 107, 108, 109, 110]

         # Aggregation pipeline to retrieve the health evolution of selected patients
         pipeline = [
             {
                 "$match": {
                     "patientID": {"$in": patient_ids}
                 }
             },
             {
                 "$lookup": {
                     "from": "appointments",  # Replace with the correct collection name
                     "localField": "appointmentID",  # Replace with the correct field name
                     "foreignField": "appointmentID",
                     "as": "appointment"
                 }
             },
             {
                 "$unwind": "$appointment"
             },
             {
                 "$project": {
                     "_id": 0,
                     "timestamp": "$appointment.date",  # Replace with the correct field name
                     "patientID": "$patientID",
                     "severity": "$severity"
                 }
             },
             {
                 "$sort": {
                     "timestamp": 1
                 }
             }
         ]
```



```
# Execute the aggregation pipeline
result = list(patients_collection.aggregate(pipeline))

# Prepare data for plotting
data_by_patient = {pid: {"timestamps": [], "severity_values": []} for pid in patient_ids}

for record in result:
    patient_id = record['patientID']
    timestamps = data_by_patient[patient_id]['timestamps']
    severity_values = data_by_patient[patient_id]['severity_values']

    timestamps.append(datetime.strptime(record['timestamp'], "%Y-%m-%dT%H:%M:%SZ"))
    severity_values.append(record['severity'])

# Plotting for each patient
for patient_id, data in data_by_patient.items():
    plt.plot(data['timestamps'], data['severity_values'], label=f'Patient {patient_id}', marker='o')


plt.xlabel('Timestamp')
plt.ylabel('Severity')
plt.title('Patients Health Evolution Over Time')
plt.xticks(rotation=45)
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.tight_layout()
plt.show()
```



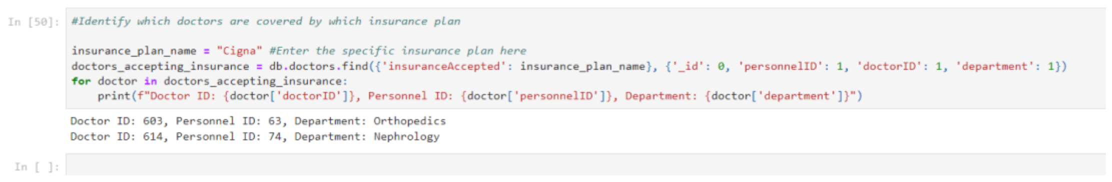


Query 5 rationale: Finally it's crucial to be able to update or change any information about patientss or anything related to the hospital. So this final query uses update_one to change a single document, $set to change a single field and importantly modified_count to make sure the update was actually carried through.

```
In [53]: # Patient ID to update - The result was shown in the database in MongoDB
         patient_id_to_update = 101

         # New weight value
         new_weight = 100  # Replace this with the desired new weight

         # Update the weight for the specified patient ID
         update_result = db.patients.update_one(
             {'patientID': patient_id_to_update},
             {'$set': {'weight': new_weight}}
         )

         # Check if the update was successful
         if update_result.modified_count > 0:
             print(f"Weight updated successfully for patient {patient_id_to_update}")
         else:
             print(f"No matching record found for patient {patient_id_to_update}")

         Weight updated successfully for patient 101

In [ ]:

In [54]: # Patient ID to retrieve the updated weight
         patient_id_to_retrieve = 101

         # Retrieve the updated weight for the specified patient ID
         patient = db.patients.find_one({'patientID': patient_id_to_retrieve}, {'_id': 0, 'weight': 1})

         # Check if the patient exists and print the updated weight
         if patient:
             print(f"The updated weight for patient {patient_id_to_retrieve} is: {patient['weight']}")
         else:
             print(f"No matching record found for patient {patient_id_to_retrieve}")

         The updated weight for patient 101 is: 100
```

We performed smaller queries that were not included in the report (in the html file) but which made use of more aspects of our data, such as the following and more:

Query 6 rationale: It's necessary for patients to know which doctors, departments or procedures are covered by their insurance plan. The following query requires us to input a specific insurance plan, in this example 'Cigna' and then it finds the relevant doctors and departments. From there we can dig further to find the procedures that are also covered by then plan.



```
In [50]: #Identify which doctors are covered by which insurance plan

         insurance_plan_name = "Cigna" #Enter the specific insurance plan here
         doctors_accepting_insurance = db.doctors.find({'insuranceAccepted': insurance_plan_name}, {'_id': 0, 'personnelID': 1, 'doctorID': 1, 'department': 1})
         for doctor in doctors_accepting_insurance:
             print(f"Doctor ID: {doctor['doctorID']}, Personnel ID: {doctor['personnelID']}, Department: {doctor['department']}")

         Doctor ID: 603, Personnel ID: 63, Department: Orthopedics
         Doctor ID: 614, Personnel ID: 74, Department: Nephrology

In [ ]:
```

```
In [50]: #Identify which doctors are covered by which insurance plan

         insurance_plan_name = "Cigna" #Enter the specific insurance plan here
         doctors_accepting_insurance = db.doctors.find({'insuranceAccepted': insurance_plan_name}, {'_id': 0, 'personnelID': 1, 'doctorID': 1, 'department': 1})
         for doctor in doctors_accepting_insurance:
             print(f"Doctor ID: {doctor['doctorID']}, Personnel ID: {doctor['personnelID']}, Department: {doctor['department']}")

         Doctor ID: 603, Personnel ID: 63, Department: Orthopedics
         Doctor ID: 614, Personnel ID: 74, Department: Nephrology

In [ ]:
```

In conclusion, our report showcases the complexities and capabilities of MongoDB, python and, and libraries to visualise, construct, and manage a Hospital Management database system. We selected MongoDB due to its meticulous document-oriented structure that is ideal for handling tedious medical records and its ability to accommodate data structures that grow horizontally. Its schema-less design brought a level of technical complexity that required detailed modelling once relationships between collections were established. Finally, we used matplotlib and pandas library to visualise the output of temporal queries.

We consider our efforts successful in showcasing a diverse array of queries aimed at modeling, realistic hospital scenarios that aligns with report's criteria for our Hospital Management database.