BIG DATA ASSIGNMENT 1

CSC4023Z

Group name: Data?IBarelyKnowHer
Bianca Acafrao – ACFBIA001
Stuart Heath – HTHSTU002
Ryan Lake – LKXRYA002
Erik Polzin – PLZERI001
Richard Taylor – TYLRIC007

Contents

(2) Designing the MongoDB database

The nature of the medical dataset we selected aligns well with the characteristics of a document-store database. Given the dataset's inherent variability in medical records, a JSON format suits it well, making it ideal for a document-store database. Given the diverse nature of medical records, where some patients may have more complex health histories than others such as more medical conditions or test results, a document-store database allows for creation of nested structures. This means we could use arrays within documents to efficiently store data such as medical conditions, medications, and test results within each patient's records.

Moreover, leveraging MongoDB's ability to organise data into collections, we were able to efficiently separate collections for "Cases" and "Patients". In our context, it was practical to do so, as each collection could contain their own relevant groups of data (refer to the design below).

Given the dynamic nature of healthcare advancements, we also considered the potential for the data to expand over time due to factors such as increased patient volume. As the dataset grows, there is potential for sharding of the collections across multiple servers or nodes since MongoDB supports horizontal scaling. This will maintain performance as the database scales.

Database design:

Our document design adheres to a straightforward structure, with each attribute name being self-explanatory. Here is the general structure of a document within each collection:

Case:
• _id: objectId
• Medical Condition": str
• Date of Admission": date
• Doctor: str
• Hospital: str
• Room Number: int
• Admission Type: str
• Discharge Date: date
• Medication: str
• Test Results: str
• Billing
  ○ Insurance Provider: str
  ○ Billing Amount: double
• PatientId: objectId
	Patient:
	• _id: objectId
	• Name: str
	• Age: int [>= 0]
	• Gender: str [Male|Female]
	• Blood Type: str [A+|B+|AB+|O+|A-|B-|AB-|O-]

(3) Creating and loading the database

The system includes a small Python script to facilitate loading data into the database. It does this in two steps:

1. Data is first converted from CSV to JSON, which more closely resembles the data in the document store. Given that the dataset is available as ‘healthcare_dataset.csv’ in the current directory, this can be done by running the command:

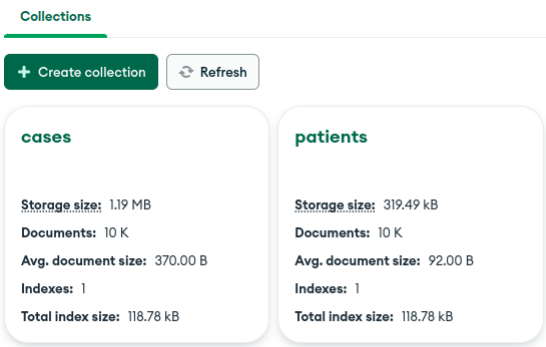
   ```
   python3 mongoshlongo.py convert healthcare_dataset.csv
   healthcare_dataset.json
   ```

   This should create a new file, ‘healthcare_dataset.json’ populated with JSON data

2. Data is then loaded into the database from the JSON file by running:

   ```
   python3 mongoshlongo.py import test_database healthcare_dataset.json
   ```

   If all goes well, the data should now be present in a new database called ‘test_database’. We can confirm the data has been loaded by inspecting the database in MongoDB Compass.



3. Once the database has been loaded, you can run quasi unit tests to make sure everything is working. Use:
   `python3 mongotest.py test_database`
   And inspect the output.

(4) Relative benefits and disadvantages of MongoDB

Graph-store

A graph-store database (such as Neo4j) is specifically designed for the storage and retrieval of graph-structured data. It uses a graph data model, consisting of nodes, edges, and properties, to represent and organise information. Seeing as our dataset includes relationships between patients, healthcare providers, and hospitals, a graph-based model could be considered, as it could be used for modelling and querying scenarios with interconnected healthcare entities. However, it might face challenges when the dataset becomes extremely large and there is a lack of flexibility.

MongoGB as a document-store database, on the other hand, provides the flexibility needed for accommodating variations in patient information - a more dynamic schema. Unlike graph-stores, where primarily flat and unconnected data is not easily dealt with, MongoDB can retrieve an entire patient record through a single query, which is very useful in the context of medical systems.

Although a graph-store database is not our primary choice for storing detailed patient records, and semi-structured data, it could be used in a polyglot persistence scenario to complement the value added by the document-store. MongoDB is not optimised for relationship-centric queries compared to graph-store databases, and so a graph database can be used to manage the relationships between patients, healthcare providers, and other entities. In conjunction with the graph database, MongoDB could be used as the primary database for storing collections of detailed patient and hospital-related information.

Key-value

MongoDB uses a document DBs which is different from a key-value store system and comes with its own benefits and drawbacks. Firstly, MongoDB is extremely flexible. Each document in a collection can have its own structure. This is extremely useful when dealing with data models that change over time. In contrast a key-value store system can be quite rigid with its schema as each record is a simple key-value pair and doesn’t easily allow evolving data models to be stored.

MongoDB has a very powerful query language and allows for complex queries including conditions and indexing. Key-Value stores do not allow for very powerful queries as one key value is linked to one record and therefore referencing multiple records is not feasible.

Due to MongoDB having more features and more powerful queries capabilities, it may require more resources. Key-Value stores is a very lightweight system that does not require many resources as it only does simple read and write operations.

If key-value stores were used in a system that uses polyglot persistence it could possibly serve as a session storage system. This is well suited for a key-value store as session data on web platforms is often key-based relying on the user's details which is always unique. On the same note, key-value stores are well suited for user profiles and preferences as each user can be represented with one key due to the uniqueness of user data. It is particularly beneficial when platforms require many reads and updates to user data due to how efficient key-value stores are regarding read and writes.

Column-family

Column-family is a NoSQL data storage model that organises data in a column-oriented way. As the name suggests, data is stored in columns, rather than in rows as a traditional relational database would be structured, this allows for more efficient reading and writing of specific columns and column families, as they can be updated simultaneously among multiple rows simultaneously. This gives column-family the advantage of performance at scale and also flexibility of data structure (and schema) as it supports a dynamic column model, where each row does not need to have the same columns. Unfortunately these advantages also come with some trade-offs, such as complexity, as the data model is less intuitive than other types of database storage and it is not as conducive to ad-hoc query capabilities and aggregations as MongoDB and relations databases due to more limited support.

This means that column-family is useful for applications such as Time-Series Data, which allows for efficient storing and querying of time-series data such as metrics, events, and logs - where each event is a row and attributes are stored in columns. Another popular application is Real-Time Data Processing scenarios that require real-time data processing and aggregation, where the ability to quickly access and update specific columns is crucial. Thus column-family stores are more often found in polyglot databases alongside MongoDB. Popular examples of column-family stores systems are Apache's Cassandra and HBase.

Relational

Document databases have flexible schemas which means that the database doesn't need to be finalised before development unlike relational database systems that have strict schemas and any changes to them during development are a lot harder and can run into possible issues

With regards to the CAP (Consistency, Availability, Partition Tolerance) Theorem document structure out performs the relational database system, with Partition Tolerance, by being able to only have issues when there is a complete network failure. This also has to do with MongoDB being easily horizontally and vertically scalable, whereas Relational Databases are vertically scalable which eventually leads to drops in performance as the volume of data increases.

Document database systems also store data in a JSON like structure which enables the data to be easily accessed, transferred , and converted for most programming languages to use, whereas for relation database systems queries would need to be converted into objects which is an extra added layer of complexity that document database systems remove.

However, one area where relational databases beat document databases is consistency, while MongoDB does have consistency, this is eventual consistency. So, there is a gap where possible errors and race conditions can occur that need to be handled, adding additional complexity, additionally with regards to consistency, given there is no forced consistency there can be missing data between different collections unless it is enforced locally.

Document structure databases also have weak atomicity where if a change is needing to be made to a record, you may need to go find all occurrences of the record and change it

whereas with relational databases it can be done once and because of the relationships it will be updated for all records that reference that record.

Relational databases have been around since the 1970s, and they can still be used today in a system that employs polyglot persistence. A relational database can store data in which the relationship between other data points is crucial, especially for complex data relationships. This includes inventory management for a company, as the dataset won't be large and can facilitate easy analysis using the relationship between inventory and sales, for example.

Hierarchal

A hierarchical database model stores records in a tree-like structure, where each child record is connected to the parent via links. Hierarchical databases have a rigorously defined schema, so they're relatively inflexible, and need to specify the field types in advance. It's a relatively old technology developed in the 60s, but still has fairly widespread use.

The most obvious difference to MongoDB is its inflexibility: the schema of a hierarchical database needs to be defined well in advance, and must have a nested character. In contrast, MongoDB documents have no predefined schema, so data can be inserted with any fields, and almost any types. This means that a hierarchical database may be less prone to bugs and type errors, but is less able to cope with changes to the data model.

Hierarchical databases may not scale as well as a MongoDB implementation, since the latter is specifically developed to cope with large volumes of data distributed across multiple servers, with minimal latency. MongoDB also has better indexing strategies, with support for single field and compound indexing. This may make it more performant when querying large datasets.

Overall, there's no real reason to incorporate a hierarchical database into a system like this. MongoDB will be more flexible, scalable and performant with large datasets, and the data doesn't have a deeply nested structure. Perhaps if the client wanted to incorporate some sort of booking system involving available rooms at a given hospital, a tree structure would be appropriate.

(5) Querying and updating the database

1.

Query	Find the earliest admitted case
Code	`db.cases.find().sort({"Date of Admission":1}).limit(1)`
Output	

```
[
  {
    _id: ObjectId('65df8a0c7fee017fa27df81e'),
    'Medical Condition': 'Asthma',
    'Date of Admission': ISODate('2018-10-30T00:00:00.000Z'),
    Doctor: 'Brandon Jackson',
    Hospital: 'Grant Ltd',
    'Room Number': 378,
    'Admission Type': 'Elective',
    'Discharge Date': ISODate('2018-11-10T00:00:00.000Z'),
    Medication: 'Penicillin',
    'Test Results': 'Inconclusive',
    Billing: {
      'Insurance Provider': 'Cigna',
      'Billing Amount': '46316.393646287564'
    },
    PatientId: ObjectId('65df8a0c7fee017fa27df81d')
  }
]
```

2.

Query	Find the hospital with the most patients
Code	`db.cases.aggregate({ $group: {_id: "$Hospital", count: {$count: {}}}}, {$sort: {"count":-1}}, {$limit: 1})`
Output	

```
test_database> db.cases.aggregate({$group: {_id: "$Hospital", count: {$count: {}
}}}, {$sort: {"count":-1}}, {$limit: 1})
[ { _id: 'Smith PLC', count: 19 } ]
```

3.

Query	Increase the age of all patients by one year
Code	`db.patients.updateMany({}, {$inc: {Age: 1}})`
Output	Initial record:

```
test_database> db.patients.aggregate([ { $group: { _id: null, averageAge: { $avg:
 "$Age"} } }])
[ { _id: null, averageAge: 51.4522 } ]
```

	Update code:
	
	`test_database> db.patients.updateMany({}, {$inc: {Age: 1}})`
	`{`
	`  acknowledged: true,`
	`  insertedId: null,`
	`  matchedCount: 10000,`
	`  modifiedCount: 10000,`
	`  upsertedCount: 0`
	`}`
	
	Updated record:
	
	`test_database> db.patients.aggregate([ { $group: { _id: null, averageAge: { $avg: "$Age"} } }])`
	`[ { _id: null, averageAge: 52.4522 } ]`

4.

Query	**Calculate average admission time in days**

Code	`db.cases.aggregate([{`
	`  $group: {_id: null, averageAdmission: {`
	`  $avg: {"$dateDiff": {`
	`  endDate:"$Discharge Date",`
	`  startDate:"$Date of Admission",`
	`  unit: "day"}}}}}])`

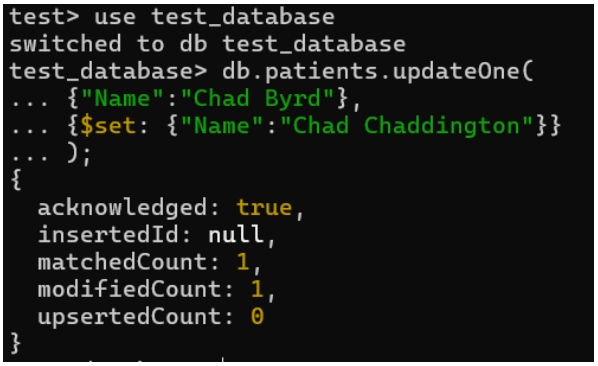
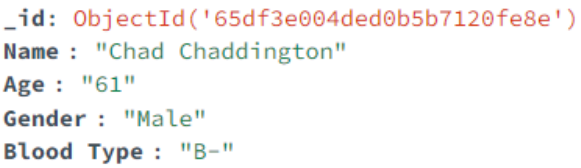
Output	`test_database> db.cases.aggregate([ { $group: { _id: null, averageAdmission: { $avg: {"$dateDiff": {endDate:"$Discharge Date", startDate:"$Date of Admission", unit: "day"}} } }])` `[ { _id: null, averageAdmission: 15.5618 } ]`

5.

Query	**Update existing record in database**

Code	`db.patients.updateOne(`
	`... {"Name":"Chad Byrd"},`
	`... {$set: {"Name":"Chad Chaddington"}}`
	`... );`

Output	Initial record:
	
	`_id: ObjectId('65df3e004ded0b5b7120fe8e')`
	`Name : "Chad Byrd"`
	`Age : "61"`
	`Gender : "Male"`
	`Blood Type : "B-"`

	Update Code:  Updated record: 

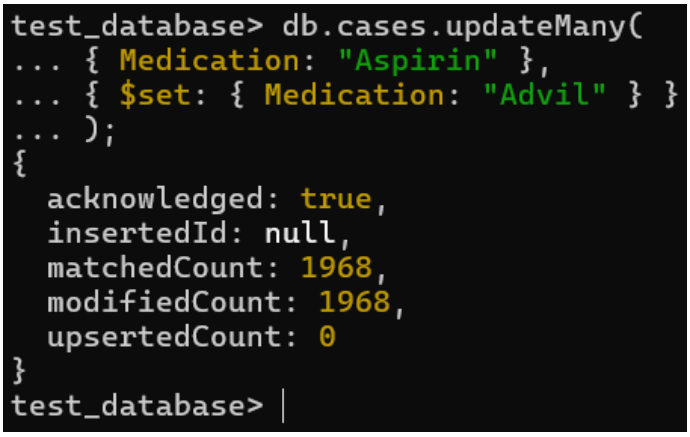
```
test> use test_database
switched to db test_database
test_database> db.patients.updateOne(
... {"Name":"Chad Byrd"},
... {$set: {"Name":"Chad Chaddington"}}
... );
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

```
_id: ObjectId('65df3e004ded0b5b7120fe8e')
Name : "Chad Chaddington"
Age : "61"
Gender : "Male"
Blood Type : "B-"
```

6.

Query	Replace all “Aspirin” medications with “Advil”

Code	`db.cases.updateMany([`
	`… { Medication: "Aspirin" },`
	`… { $set: { Medication: "Advil" } });`

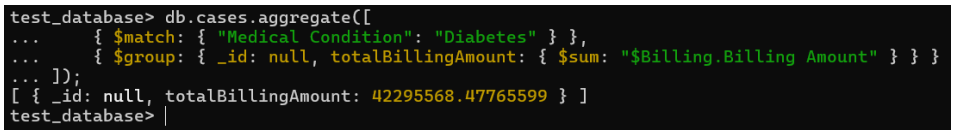
Output	

```
test_database> db.cases.updateMany(
... { Medication: "Aspirin" },
... { $set: { Medication: "Advil" } }
... );
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1968,
  modifiedCount: 1968,
  upsertedCount: 0
}
test_database>
```

7.

Query	Get the total billing amount for patients with diabetes

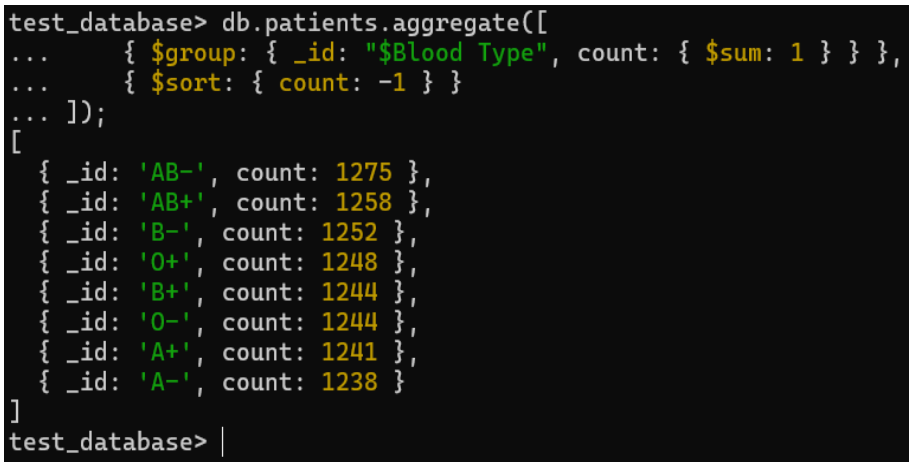
Code	`db.cases.aggregate([`
	`… { $match: { "Medical Condition": "Diabetes" } },`
	`… { $group: { _id: null, totalBillingAmount: {`
	`… $sum: "$Billing.Billing Amount" } } }`
	`… ]);`

Output	

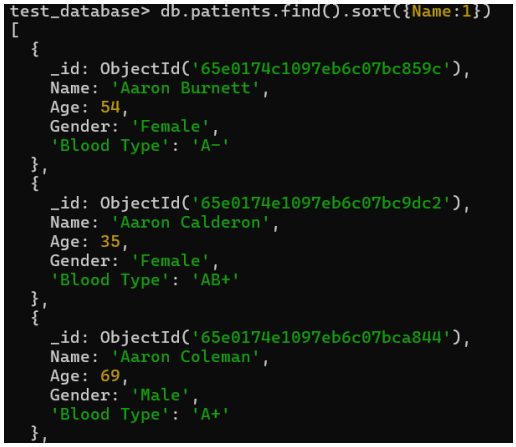
8.

Query	Count the number of documents for each unique blood type sorted in descending order

Code	`db.patients.aggregate([`
	`… { $group: { _id: "$Blood Type", count: { $sum: 1 } } },`
	`… { $sort: { count: -1 } }`
	`]);`

Output	

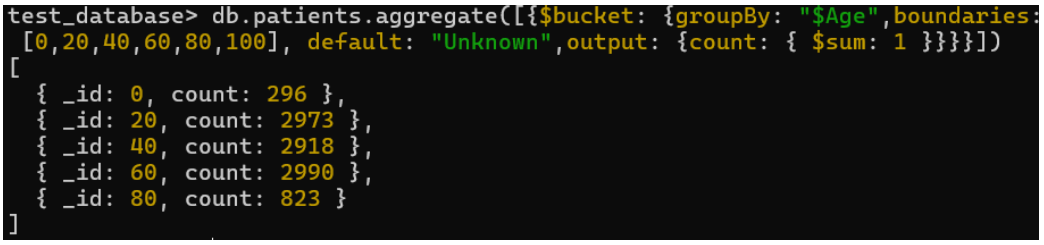
9.

Query	Displays of patients in alphabetical order:
Code	`db.patients.find().sort({Name:1})`
Output	(First 3 results) 

10.

Query	Number of patients in 5 different age groups

Code	`db.patients.aggregate([{$bucket: {`
	`… groupBy: "$Age",`
	`… boundaries: [0,20,40,60,80,100],`
	`… default: "Unknown",`
	`… output: {count: { $sum: 1 }}}}])`

Output	

11.

Query	Find all O, positive and negative, type patients named Chad
Code	`db.patients.find({Name: /^Chad/,'Blood Type': /^O/});`
Output	(First 3 outputs)

```
test_database> db.patients.find({Name: /^Chad/,'Blood Type': /^O/})
[
  {
    _id: ObjectId('65e0174c1097eb6c07bc8202'),
    Name: 'Chad Martin',
    Age: 48,
    Gender: 'Male',
    'Blood Type': 'O+'
  },
  {
    _id: ObjectId('65e0174c1097eb6c07bc87e4'),
    Name: 'Chad Anthony',
    Age: 26,
    Gender: 'Male',
    'Blood Type': 'O+'
  },
  {
    _id: ObjectId('65e0174c1097eb6c07bc8a36'),
    Name: 'Chad Garcia',
    Age: 74,
    Gender: 'Female',
    'Blood Type': 'O-'
  },
```

12.

Query	Most common medical conditions
Code	`db.cases.aggregate([{$group:{_id: "$Medical Condition", totalCases:{$sum:1}}}]);`
Output	

```
test_database> db.cases.aggregate([{$group:{_id: "$Medical Condition",totalC
ases:{$sum:1}}}]);
[
  { _id: 'Arthritis', totalCases: 1650 },
  { _id: 'Hypertension', totalCases: 1688 },
  { _id: 'Cancer', totalCases: 1703 },
  { _id: 'Asthma', totalCases: 1708 },
  { _id: 'Diabetes', totalCases: 1623 },
  { _id: 'Obesity', totalCases: 1628 }
]
```

13.

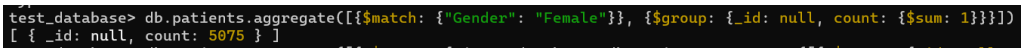
Query	Shows which Insurance Providers have the highest bills:

Code	`db.cases.aggregate([{$group: {`
	`… _id: "$Billing.Insurance Provider",`
	`… totalAmountBilled : {`
	`… $sum: {$toDouble: "$Billing.Billing Amount"}}}},`
	`… {$sort: {totalAmountBilled : -1 }}])`

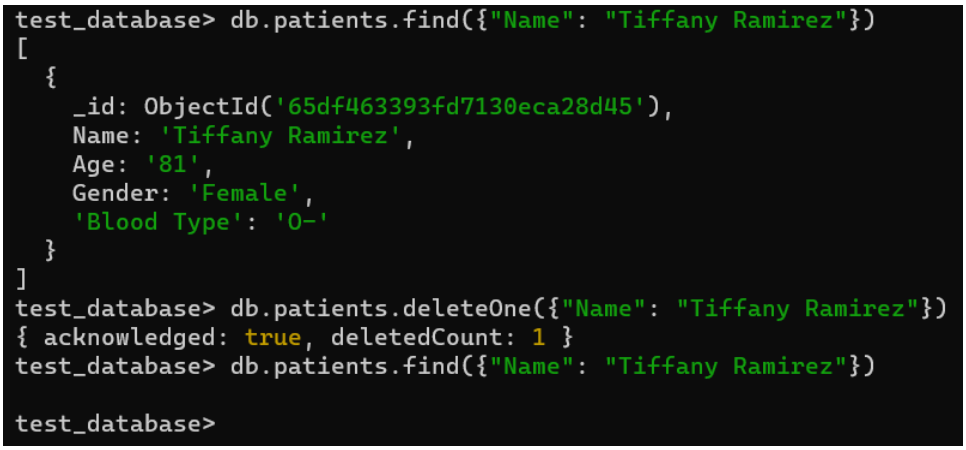
Output	

```
test_database> db.cases.aggregate([{$group: {_id: "$Billing.Insurance Provid
er",totalAmountBilled : {$sum: {$toDouble: "$Billing.Billing Amount"}}}},{$s
$sort: {totalAmountBilled : -1 }}])
[
  { _id: 'Cigna', totalAmountBilled: 52340171.592402816 },
  { _id: 'Aetna', totalAmountBilled: 52321794.759911746 },
  { _id: 'Blue Cross', totalAmountBilled: 52125858.901628375 },
  { _id: 'UnitedHealthcare', totalAmountBilled: 50250467.698130846 },
  { _id: 'Medicare', totalAmountBilled: 48129774.8253102 }
]
```

14.

Query	Counts number of females in database
Code	db.patients.aggregate([… { $match: { “Gender”: “Female” } }, … { $group: {_id: null, count : { $sum: 1} } } …])
Output	

15.

Query	Deletes one from database by name
Code	db.patients.deleteOne({ “Name”: “Tiffany Ramirez”})
Output	

16.

Query	Calculates the average age of patients
Code	db.patients.aggregate([… { $group: {_id: null, … averageAge: { … $avg: { $convert: {input: “$Age”, to: “double”}}}}} …])
Output	

17.

Query	Shows which doctor was given the most cases
Code	db.cases.aggregate([… {$group: {_id: "$Doctor", totalCases: {$sum:1}}}, … {$sort: {totalCases: -1}}, … {$limit:1} …])
Output	test_database> db.cases.aggregate([{$group: { _id: "$Doctor", totalCases: {$sum: 1}}}, {$sort: {totalCases: -1}}, {$limit: 1}]) [{ _id: 'Michael Johnson', totalCases: 7 }]

18.

Query	Add myself to the database
Code	db.patients.insertOne({ … Name:"Richard Taylor", … Age:23, … Gender:"Male", … "Blood Type":"A+"})

Output	healthcare_db> db.patients.insertOne({Name:"Richard Taylor", Age:23, Gender:"Male", "Blood Type":"A+"})
	{
	  acknowledged: true,
	  insertedId: ObjectId('65e107ab05d8b6dedc93c2db')
	}
	healthcare_db> db.patients.find({Name: "Richard Taylor"})
	[
	  {
	  _id: ObjectId('65e107ab05d8b6dedc93c2db'),
	  Name: 'Richard Taylor',
	  Age: 23,
	  Gender: 'Male',
	  'Blood Type': 'A+'
	  }
	]

19.

Query	Find all rooms that saw new patients in January 2023
Code	db.cases.find({"Date of Admission": {"$gt": ISODate('2023-01-01T00:00:00.000Z'), "$lt": ISODate('2023-01-31T00:00:00.000Z')}}, {_id: 0, "Room Number": 1})

Output	healthcare_db> db.cases.find({"Date of Admission": {"$gt": ISODate('2023-01-01T00:00:00.000Z'), "$lt": ISODate('2023-01-31T00:00:00.000Z')}}, {_id: 0, "Room Number": 1})
	[
	  { 'Room Number': 400 }, { 'Room Number': 225 },
	  { 'Room Number': 334 }, { 'Room Number': 213 },
	  { 'Room Number': 193 }, { 'Room Number': 161 },
	  { 'Room Number': 174 }, { 'Room Number': 123 },
	  { 'Room Number': 266 }, { 'Room Number': 395 },
	  { 'Room Number': 170 }, { 'Room Number': 368 },
	  { 'Room Number': 366 }, { 'Room Number': 383 },
	  { 'Room Number': 222 }, { 'Room Number': 345 },
	  { 'Room Number': 414 }, { 'Room Number': 456 },
	  { 'Room Number': 400 }, { 'Room Number': 499 }
	]

20.

Query	Find all doctors on cases that are not Emergencies or urgent
Code	`db.cases.find({"Admission Type": {$nin: ["Emergency", "Urgent"]}}, {_id: 0, Doctor: 1})`
Output	

```
healthcare_db> db.cases.find({"Admission Type": {$nin: ["Emergency", "Urgent"]}}, {_id:
 0, Doctor: 1})
[
  { Doctor: 'Patrick Parker' },
  { Doctor: 'Brian Kennedy' },
  { Doctor: 'Kristin Dunn' },
  { Doctor: 'Laura Roberts' },
  { Doctor: 'James Carney' },
  { Doctor: 'Clayton Mcknight' },
  { Doctor: 'Debra Meyers' },
  { Doctor: 'Megan Sanders' },
  { Doctor: 'Natalie Sullivan' },
  { Doctor: 'Nicole Mcclain' },
  { Doctor: 'Pamela Brown' },
  { Doctor: 'John Harvey' },
  { Doctor: 'Sylvia Johnson' },
  { Doctor: 'Melissa Kelley' },
  { Doctor: 'Leon Price' },
  { Doctor: 'Nicholas Aguirre' },
  { Doctor: 'Mr. John Short' },
  { Doctor: 'Gina Maddox' },
  { Doctor: 'Caleb Rasmussen' },
  { Doctor: 'Angela Mcdonald' }
]
Type "it" for more
```

21.

Query	Adds new column to the cases collection - the duration of the cases in days

Code	`db.cases.aggregate([{$project: {`
	`… 'Medical Condition': 1,`
	`… Doctor: 1,`
	`… Hospital: 1,`
	`… 'Room Number': 1,`
	`… 'Admission Type': 1,`
	`… Medication: 1,`
	`… 'Test Results': 1,`
	`… 'Billing.Insurance Provider': 1,`
	`… 'Billing.Billing Amount': 1,`
	`… PatientId: 1,`
	`… Duration: {$divide: [{$subtract: [`
	`…  "$Discharge Date", "$Date of Admission"]`
	`… }, 86400000]}}}])`

Output	

```
healthcare_db> db.cases.aggregate([{$project: {'Medical Condition': 1, Doctor: 1, Hospi
tal: 1, 'Room Number': 1, 'Admission Type': 1, Medication: 1, 'Test Results': 1, 'Billi
ng.Insurance Provider': 1, 'Billing.Billing Amount': 1, PatientId: 1, Duration: {$divid
e: [{$subtract: ["$Discharge Date", "$Date of Admission"]}, 86400000]}}}])
[
  {
    _id: ObjectId('65e106daabf080a17ee2d7e8'),
    'Medical Condition': 'Diabetes',
    Doctor: 'Patrick Parker',
    Hospital: 'Wallace-Hamilton',
    'Room Number': 146,
    'Admission Type': 'Elective',
    Medication: 'Aspirin',
    'Test Results': 'Inconclusive',
    Billing: {
      'Insurance Provider': 'Medicare',
      'Billing Amount': '37490.98336352819'
    },
    PatientId: ObjectId('65e106daabf080a17ee2d7e7'),
    Duration: 14
  },
```

(6) Linking the database to a program

Please refer to mongotest,py included in our submission file, as well as the README.md. This script automatically runs one query for every team member. The script uses Pymongo to connect to our MongoDB database, and the proceed to execute the following statements:

- Displaying patients sorted in alphabetical order (Stuart)
- Displaying most common medical conditions (Erik)
- Deleting a patient from the database (Ryan)
- Adding a new column to the database (Richard)
- Displaying the earliest admitted case (Bianca)

(7) Contribution statement

Member	**Contribution**
Erik Polzin	Erik performed the database operations 1-4 and focussed on the comparison between document and hierarchal databases. Erik also worked on formatting the data into JSON format and configured the python code for setting up our initial database and running scripts.
Bianca Acafrao	Bianca performed the database operations 5-8 and focused on comparing document-stores to graph-store databases (question 4). She organised the report structure and was the scribe for the meeting in which the group discussed our database design justification (question 2). Bianca also edited the type values for database entries for calculations to work.
Stuart Heath	Stuart performed operations 9-13 and wrote about the comparison between document and relational databases. He also contributed to the initial layout and planning of the document, overlooking correctness of the various sections.
Ryan Lake	Ryan performed operations 14-17 and compared MongoDB to key-value databases. He contributed to team collaboration by setting up our report on Google Docs, and organised our meeting structure.
Richard Taylor	Richard performed database operations 18-21 and wrote about the comparison between document databases and column-family databases. Additionally, he created Python scripts to automate our written queries, contributing significantly to linking the database to a program (question 6).