Chapter 22: NoSQL

Contents

1 What is NoSQL
2 Differences between relational databases and NoSQL databases 3 Advantages of NoSQL
4 Applications of relational databases and NoSQL databases
5 Introduction to MongoDB
    5.1 Terms used in MongoDB
    5.2 Document in MongoDB
    5.3 Starting the MongoDB server
6 PyMongo
    6.1 Create a MongoClient
    6.2 Create a Mongo database
    6.3 Create a collection
    6.4 Inserting documents
        6.4.1 Inserting one document
        6.4.2 Inserting multiple documents
        6.4.3 Importing from a text file
        6.4.4 Importing from a JSON file
    6.5 Searching for documents
        6.5.1 Find the first document
        6.5.2 Find all documents
        6.5.3 Find documents that satisfy certain criteria
        6.5.4 Find documents that satisfy multiple criteria
        6.5.5 Count the number of documents in a collection
    6.6 Update documents
    6.7 Remove documents
    6.8 Remove a collection
    6.9 Remove a database
Annex 1 – Using MongoDB shell

Syllabus Learning Outcomes

3.3 Databases and Data Management

Understand, create and use SQL and NoSQL databases, as well as understand techniques to protect the privacy and integrity of data.

3.3.6 Understand how NoSQL database management system addresses the shortcomings of relational database management system (SQL)

3.3.7 Explain the applications of SQL and NoSQL

3.3.8 Use a programming language to work with both SQL and NoSQL databases.

1 Introduction to NoSQL

NoSQL, stands for “non SQL” or “not only SQL”, is a type of database management system (DBMS) that is designed to handle and store large volumes of unstructured and semi-structured data. Unlike traditional relational databases that use tables with pre-defined schemas to store data, NoSQL databases use non-relational data structures which are flexible and allows dynamic schema for unstructured data. NoSQL databases can adapt to changes in data structures and are capable of scaling horizontally to handle growing amounts of data.

There are four main types of NoSQL databases: document databases, key-value databases, wide-column databases, and graph databases.

2 Differences between relational databases and NoSQL databases 2.1

Data Structure

Relational databases have a fixed, predefined schema where data must fit into tables with specific columns and data types. This rigid structure ensures consistency, and it works well for applications with stable, well-structured, and predictable data requirements.

In contrast, NoSQL databases adopt flexible data models, allowing for dynamic and non schematic data storage. This flexibility enables developers to insert data without a predefined schema. NoSQL databases are most useful in scenarios where data structures may be undefined, not fully known in advance, or are subject to frequent changes.

2.2 Scalability

Relational databases and NoSQL databases emphasize different scaling strengths due to their designs. Relational databases typically rely on vertical scaling, which involves improving and adding resources, such as faster processors and more memory, to the same server to handle increased load. Such high-performance components can be expensive, and upgrades are limited by the capacity of a single machine.

Horizontal scaling, typically seen in NoSQL systems, is achieved by adding more servers or nodes to a distributed system, which then helps increase capacity. The nodes communicate with each other and distribute the load, so adding more nodes helps increase the overall capacity of the system. This is a more scalable and cost-effective solution for managing a growing database and increasing database traffic.

2.3 Properties

Relational database and NoSQL database management systems take different approaches to ensuring reliability. Relational databases rely on **ACID properties**:

- **Atomicity:** A change in the database must either be performed completely or not performed at all.
- **Consistency:** A transaction (read/write operations) must take a whole database from one consistent state to another consistent state, for example in a bank transfer transaction the amount of money in the whole system must be the same at the end of the transaction as it was at the beginning.
- **Isolation:** A transaction must be performed in isolation so that other users or processes cannot have access to the data concerned until the new consistent state has been committed (saved). In practice, this means that while an operation is being performed in a record, the record is locked. This may involve making the record invisible to others or it may only lock the record for writing. After the transaction is committed, the record may be unlocked again.
- **Durability:** Once a change has been made to a database, the change must not be lost because of any subsequent system failure or operator error. Ideally, the transaction is written immediately to secondary storage.

ACID properties ensure immediate and strict consistency in the database. SQL queries guarantee that either all or none of the changes made during a transaction are committed to the database and have rules for how to handle concurrent transactions and unexpected events.

On the other hand, NoSQL databases, emphasize scalability and distributed architectures (a distributed system is a network that stores data on more than one node (physical or virtual machines) at the same time.), adopt the concept of eventual consistency. Eventual consistency acknowledges that, in a distributed system, it may take some time for all nodes to converge to a consistent state after an update. While NoSQL databases sacrifice immediate consistency for scalability and fault tolerance, they ensure that, given enough time, all replicas of the data will eventually converge to the same state.

This trade-off allows NoSQL systems to handle large-scale, distributed environments where real time consistency might be challenging to achieve efficiently.

In general, the differences between relational databases and NoSQL databases can be summarised in the table below.

	Relational databases	**NoSQL databases**
Language	Use structured query languages to perform operations	Use a dynamic schema to query data. Also, some NoSQL databases use SQL-like syntax for document manipulation.
Data Schema	Have a predefined and fixed format, which cannot be changed for new data.	NoSQL databases are more flexible. This flexibility means that records in the databases can be created without having a predefined structure, and each record has its own structure.
Scalability	Is vertically scalable, meaning that a single machine needs to increase CPU, RAM, SSD, at a certain level to meet the demand.	horizontally scalable, meaning that additional machines are added to the existing infrastructure to satisfy the storage demand.
Big Data Support	The vertical scaling makes it difficult for relational databases to store very big data	The horizontal scaling and dynamic data schema make NoSQL suitable for big data.
Properties	Use the ACID (Atomicity, Consistency, Isolation, Durability) property.	Settles for eventual consistency

3 Advantages of NoSQL

Modern applications face several challenges that can be solved by NoSQL databases. For instance, applications process a large data volume from disparate sources like social media, smart sensors, and third-party databases. All of this disparate data doesn't fit neatly into the relational model. Enforcing tabular structures can lead to redundancy, data duplication, and performance issues at scale.

NoSQL databases are purpose-built for non-relational data models and have flexible schemas for building modern applications. They are widely recognized for their ease of development, functionality, and performance at scale. The benefits of NoSQL databases are listed below.

Flexibility	Having a flexible data model also means NoSQL databases can address large volumes of rapidly changing data, making them great for agile development, quick iterations, and frequent code pushes.
Cost-effectiveness	NoSQL databases are typically designed to scale out horizontally by using distributed clusters of hardware, as opposed to scaling up by adding expensive and robust servers
Fast queries	Queries in NoSQL databases can be faster than SQL databases. Data in SQL databases is typically normalized, so queries for a single object or entity require you to join data from multiple tables. As the tables grow in size, the joins can become expensive. However, data in NoSQL databases is typically stored in a way that is optimized for queries. Queries typically do not require joins, so the queries are simpler and are very fast.
Replication	NoSQL replication functionality copies and stores data across multiple servers. This replication provides data reliability, ensuring access during downtime and protecting against data loss if servers go offline.

4 Applications of relational databases and NoSQL databases

SQL databases work best in scenarios when data is structured and predictable, complex relationships need to be accurately captured, and immediate data integrity is important.

Examples of industries that rely on relational databases:

- **Finance**: Many financial institutions manage transactional data and customer records. SQL's ACID properties ensure the data are accurate and that the transactions lead to an

immediately consistent database once processed.

- **Retail**: Many retail businesses leverage SQL databases, as they must manage complex relationships related to products, shipping, sales, customers, and supplier information. Their data are also typically well-structured and predictable.

- **Government and Public Sector**: Government agencies manage a lot of citizen records and public services which are subject to regulatory requirements. SQL's structured nature helps with regulatory adherence.

NoSQL databases work best when it's useful to have flexible data structures that can dynamically adapt to new information and schemas when scalability and performance are important and for unstructured data.

Examples of industries that make use of NoSQL's dynamic schemas and horizontal scaling:

- **Social Media**: Social media platforms handle large volumes of unstructured data, such as user profiles, posts, and interactions. The flexibility of NoSQL accommodates the dynamic nature of social media content and data.

- **Logistics and Supply Chain**: They use NoSQL databases for real-time tracking of shipments, inventory management, and other diverse and dynamic data sources across the supply chain. NoSQL's performance and scalability for real-time data make it well suited for this industry.

- **Gaming**: The gaming industry leverages NoSQL databases for managing player data, leaderboards, and in-game analytics. The ability to scale horizontally is essential for handling the massive amounts of data generated by online multiplayer games.

5 Introduction to MongoDB

For the syllabus, we focus on **MongoDB**, an open-source document-oriented NoSQL database. Document databases store data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values. The values can typically be a variety of types including things like strings, numbers, booleans, arrays, or objects.

5.1 Terms used in MongoDB

Here are the terms in MongoDB with the corresponding terms in SQL.

MongoDB Term	SQL Term
Database	Database
Collection	Table
Document	Record (Row)
Field	Field (Column)

5.2 Document in MongoDB

A document in MongoDB looks similar to a Python dictionary or JSON-like. We can operate on the document as though it was a dictionary in Python.

```
{
 "name": "Amanda",
 "age": "45"
}
```

5.3 Starting the MongoDB server

To start the MongoDB server, click on the MongoDB Server icon in the Start menu.

This will start the database server on your device. We can access the MongoDB server either using Mongo Shell or PyMongo. In the A-level environment, we will focus on using PyMongo. You can refer to Annex 1 to learn more about Mongo Shell.

6 PyMongo

MongoDB databases can be accessed using different programming languages like C, Java, and Python. To access MongoDB databases using Python, we use the Python driver for MongoDB, PyMongo.

Before running PyMongo, ensure that you have a MongoDB server running in the background. The MongoDB server window should remain open when you want to access the MongoDB database.

6.1 Create a MongoClient

To use PyMongo, create a MongoClient to connect to the MongoDB server. You may use Jupyter Notebook or Python IDLE to type the codes. Avoid using cloud IDEs such as Google Colab as your MongoDB server is running on your local host.

You can adopt 3 different ways to create a MongoClient.
#1 – connect on default host and port
#2 & #3 – specify host "127.0.0.1" or "Local_Host" and port no 27017 which can be seen when you start the MongoDB server.

Create a MongoClient

```
1  from pymongo import MongoClient
2
3  #1 client = MongoClient() #connect on default host and
   port #2 client = MongoClient("127.0.0.1",27017)
   #3 client = MongoClient("Local_Host",27017)

   client.close() #closes connection to the server
```

We can check the databases in the server using the method `database_names()`.

```
Retrieves names of databases
1  from pymongo import MongoClient
2  client = MongoClient()
3  #Retrieves the databases' names in Python list
4  dbs = client.database_names()
5  print("The databases in the Mongo DB server
6  are:") print(dbs)
```



```
7  client.close()
```

6.2 Create a Mongo database

Create Mongo database – "entertainment"

```
1   from pymongo import MongoClient
2   client = MongoClient()
3
4   db = client.get_database("entertainment")
5   OR
6   db = client["entertainment"]
7
8   dbs = client.database_names()
9   print("The databases in the Mongo DB server
10  are:") print(dbs)
11  client.close()
```

Important: In MongoDB, a database is not created until it gets content! MongoDB waits until you have created a collection (table), with at least one document (record) before it creates the database (and collection).

Therefore, the database "entertainment" will not show if you run the above

code. **6.3 Create a collection**

Create a collection – "movies"

```
1   from pymongo import MongoClient
2   client = MongoClient()
3   db = client["entertainment"]
4
5   coll = db.get_collection("movies")
6   OR
7   coll = db["movies"]
8
9   dbs = client.database_names()
10  print("The databases in the Mongo DB server
11  are:") print(dbs)
```



```
12  client.close()
```

MongoDB will create the collection if it does not exist.

6.4 Inserting documents

We will look at two ways of inserting documents into a collection.
1. inserting one or more documents,
2. inserting by importing from a file.

For large amounts of data, it is more efficient to import the data from a

file. **6.4.1 Inserting one document**

We can insert documents one by one, using the `insert_one()` method. Notice that not all fields are required for insertion, as shown.

```
1  from pymongo import MongoClient
2  client = MongoClient()
3  db = client["entertainment"]
4  coll = db["movies"]
5  #All fields are not null
6  coll.insert_one({"_id":1, "title":"Johnny
7  Maths",  "genre":"comedy"})
8  #No _id and genre fields
9  coll.insert_one({"title":"Detection"})
10 #Retrieves database names
11 dbs = client.database_names()
12 print("The databases in the Mongo DB server
13 are:") print(dbs)
14 #Retrieves collection names
15 cs = db.collection_names()
16 print("The collections in entertainment database are",
17 cs) client.close()
```

We should now see one entertainment database in the server after documents are added to the collection in the database. We can also see the collections (as a list) in the database.

MongoDB will assign a unique `_id` to each document. You can customize the `_id` by stating it during the insertion process, as shown on line 6. This will also mean we cannot run this code

again because a document with `_id = 1` already exists. However, if we comment out line 6, and run this code again, we will create a duplicate of `{"title":"Detection"}` with another `_id`.

6.4.2 Inserting multiple documents

To insert multiple documents, we can use the `insert_many()` method to insert a list of documents.

```
from pymongo import MongoClient
client = MongoClient()
db = client.get_database("entertainment")
coll = db.get_collection("movies")

#List of movies
movies = [
{"title":"Badman", "genre":"adventure", "year":2015},
{"title":"Averages", "genre":["science fiction","adventure"],
"year":2017}, {"title":"Octopus Man", "genre":"adventure", "year":2017},
{"title":"Fantastic Bees", "genre":"adventure",
"year":2018}, {"title":"Underground", "genre":"horror",
"year":2014}
]

coll.insert_many(movies)

client.close()
```

6.4.3 Importing from a text file

The program below reads from a delimited text file `input.txt` and inserts the documents into the database.

input.txt

```
Amanda,45
Bala,28
Charlie,33
Devi,29
```

```
1   import pymongo, csv
2
3   client = pymongo.MongoClient("127.0.0.1",
4   27017) db = client["entertainment"]
5   coll = db["users"]
6
7   with open('input.txt') as csv_file:
8    rows = csv.reader(csv_file, delimiter=',')
9    for row in rows:
10   coll.insert_one({"name":row[0], "age":row[1]})
11  client.close()
```

6.4.4 Importing from a JSON file

A JSON file stores simple data structures and objects in JavaScript Object Notation (JSON) format, which is a standard data interchange format. It is primarily used for transmitting data between a web application and a server. JSON files are lightweight, text-based, human-readable, and can be edited using a text editor.

data.json

```
[
      {
      "name": "Amanda",
      "age": "45"
      },
      {
      "name": "Bala",
      "age": "28"
      },
      {
      "name": "Charlie",
      "age": "33"
      },
      {
      "name": "Devi",
      "age": "29"
      }
]
```

To import from JSON file, use the `load()` function. The program below reads from a delimited text file `data.json` and inserts the documents into the database.

```
1   import pymongo, json
2   client = pymongo.MongoClient()
3   with open('data.json') as file:
4    data = json.load(file)
5
6   db = client.get_database("entertainment")
7   coll = db.get_collection("movies")
8   coll.insert_many(data)
9   OR
10  client['entertainment']['users'].insert_many(data)
11
12  client.close()
```

In Line 10, we are using syntax similar to array indexing to access the

collection. **6.5 Searching for documents**

Similar to `SELECT * FROM <table>` in SQL, we need a method to display the contents of a collection in NoSQL.

The following is a list of commonly used query operators. This is also listed in your exam's Insert document.

`$eq`	Equals to
`$gt`	Greater than
`$gte`	Greater than or equal to
`$lt`	Less than
`$lte`	Less than or equal to
`$ne`	Not equal to
`$in`	In a specified `list`
`$nin`	Not in a specified `list`
`$or`	Logical OR
`$and`	Logical AND
`$not`	Logical NOT

`$exists`	Matches documents which have the named field

6.5.1 Find the first document

The `find one()` method returns the first occurrence in the selection.

```
1  import pymongo
2  client = pymongo.MongoClient()
3  coll = client["entertainment"]["movies"]
4
5  result = coll.find_one()
6
7  print(result)
8
9  client.close()
```

6.5.2 Find all documents

The find() method returns all documents in a collection.

```
1  import pymongo
2  client = pymongo.MongoClient()
3  coll = client["entertainment"]["movies"]
4
5  result = coll.find()
6  for doc in result:
7   print(doc)
8
9  client.close()
```

6.5.3 Find documents that satisfy certain criteria

To find all the movies in the adventure genre, we can do the following.

```
1  import pymongo
2  client = pymongo.MongoClient()
3  coll = client["entertainment"]["movies"]
4
5  result = coll.find({"genre":"adventure"})
6  for doc in result:
7   print(doc)
8  client.close()
```


6.5.4 Find documents that satisfy multiple criteria

To find all the movies in the adventure genre that are released after 2016, we can do the following.

```
1  import pymongo
2  client = pymongo.MongoClient()
3  coll = client["entertainment"]["movies"]
4
5  query1 = {"genre":"adventure","year":{'$gt':2016}}
6
7  result = coll.find(query1)
8  for doc in result:
9   print(doc.get("title"))
10 client.close()
```

Line 5 of the code creates the query to find the documents with adventure genre and year greater than 2016. It can be rewritten using the `$and` operator as well.

```
query2 = {'$and':[{'genre': 'adventure'}, {'year': {'$gt': 2016}}]}
```

Each document is just a Python dictionary, so you can use the usual built-in methods for dictionary. For example, line 9 uses the get() method to retrieve the value of title. This allows you to extract the value for a particular field in the document.

6.5.5 Count the number of documents in a collection

To count the number of documents in a collection, use `count()`.

```
coll = client["entertainment"]["movies"]

print(coll.count())
```


6.6 Update documents

To modify the content in the database, use the `update_one()` method to modify the first document that matches the query, or the `update_many()` method to modify all documents that matches the query.

The program below illustrates these.

```
import pymongo
client = pymongo.MongoClient()
db = client.get_database("entertainment")
coll = db.get_collection("movies")
result = coll.find()
print("Before Update:")
for doc in result:
 print(doc)
print()

search = {'year':{'$gt':2016}}
update = {'$set':{'year':2015}}

#update one document
coll.update_one(search, update)
result = coll.find()
print("After update_one:")
for doc in result:
 print(doc)
print()

#update many documents
coll.update_many(search, update)
result = coll.find()
print("After update_many:")
for doc in result:
 print(doc)
```

In the code above, line 11 is used to search for year values greater than 2016, and line 12 uses `$set` to set all the year values to be 2015. The `update_one()` method at line 15 will search for the **first** document that has a year value greater than 2016 and update the year value to 2015 for that document.

The `update_many()` method at line 23 will search for **all** documents with a year value greater than 2016 and update the values to 2015 for all documents that meet the search criteria.

Note: The `$set` operator can also be used to add a new field with the specified value if the field does not exist. If we specify multiple field-value pairs, the operator will update or add each field as required.

The $unset operator can be used to delete a particular field. The value specified in the $unset expression does not make any impact on the operation. The $unset has no effect when the field does not exist in the document.

```
1  #remove a field for all documents
2  update = {'$unset':{'year':0}}
3  coll.update_many({}, update)
```


6.7 Remove documents

To delete documents, you can use the `delete_one()` method to delete the first document that matches the given condition or `delete_many()` method to delete all the documents that match the condition.

```
import pymongo
client = pymongo.MongoClient("127.0.0.1",
27017) coll =
client["entertainment"]["movies"]

result = coll.find()
print("All documents in movies collection:")
for document in result:
      print(document)

coll.delete_one({'year':2015}) #delete first relevant doc

result = coll.find()
print("All documents in movies collection after
deleting  one:")
for document in result:
      print(document)

coll.delete_many({'year':2015}) #delete many documents

result = coll.find()
print("All documents in movies collection after
deleting  all:")
for document in result:
      print(document)

coll.delete_many({}) #delete all documents in a collection

result = coll.find()
print("All documents in movies collection after
deleting  all:")
for document in result:
      print(document)
client.close()
```


6.8 Remove a collection

To delete a collection, you can use `drop_collection()` as shown below.

```
1   import pymongo
2   client = pymongo.MongoClient()
3   db = client.get_database("entertainment")
4
5   collections = db.collection_names()
6   print("All collections in entertainment
7   database:") print(collections)
8
9   db.drop_collection("movies") #delete collection
10  collections = db.collection_names()
11  print("After movies collection is dropped:")
12  print(collections)
13
14  client.close()
15
16
17
```

6.9 Remove a database

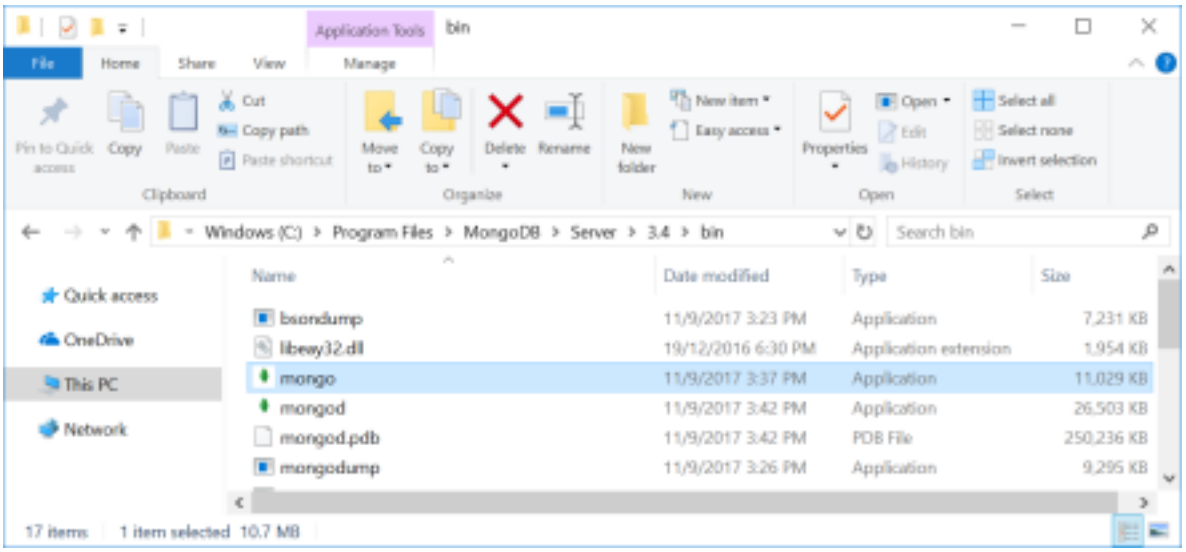
To delete the entire `entertainment` database, you can use the `drop_database()` method. That will remove all the collections and the documents within it.

```
client.drop_database("entertainment")
```

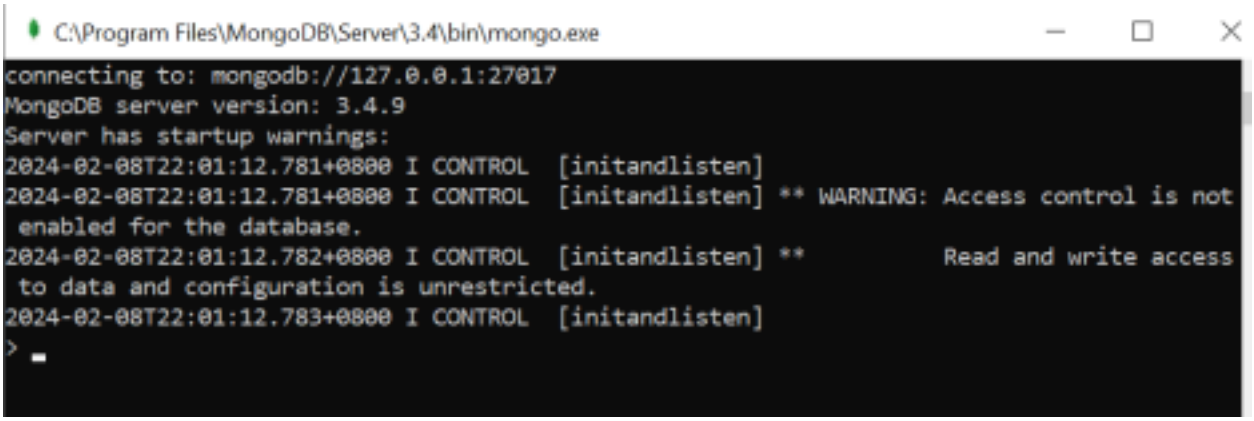

Annex 1 – Using MongoDB shell

One way to access the MongoDB server is through MongoDB shell. To open the MongoDB shell, use Windows Explorer and go to the MongoDB directory.

Double click on “mongo” application to open the MongoDB shell.



Starting the mongo shell program allows you to connect to the MongoDB server. You will then enter an interactive shell designed specifically for MongoDB.



You can use variables and mathematical operations similar to Python shell.

Look at the output on the MongoDB Server window as you type statements into the mongo shell and try to figure out what the server is doing.

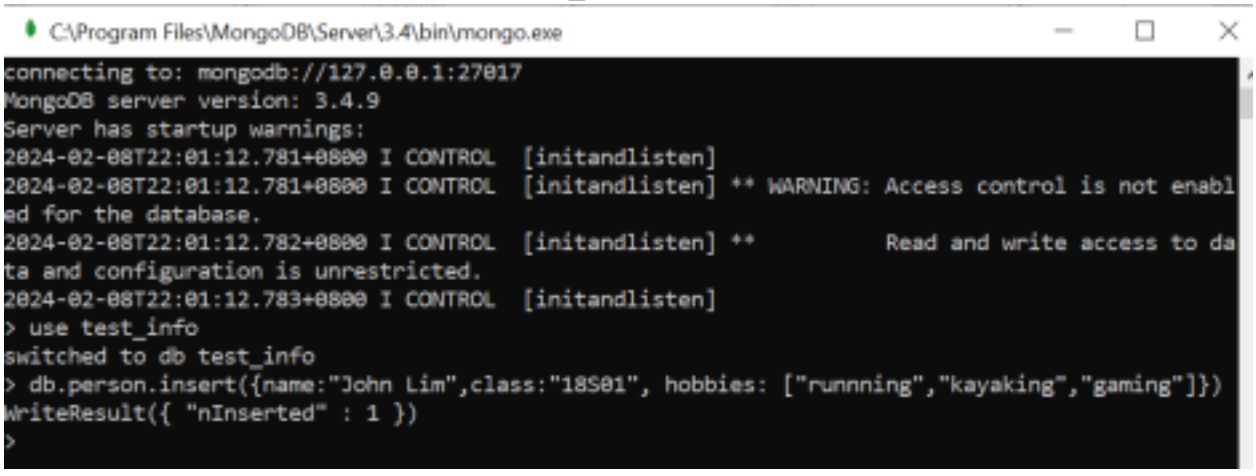
You can create a database with the `use` statement. You can insert documents into a collection using **`insert`** statement.

Try typing the following statements:

```
use test_info
db.person.insert({name:"John Lim", class:"18S01",
hobbies:["running","kayaking","gaming"] })
```

The first statement tells MongoDB to use the database `test_info`. The second statement will insert a document into the `person` collection. Your screen will look like below. This states that one document has been inserted.

Note! The database `test_info` is created only when a document is inserted.



Command	Description

`use <database_name>`	Creates a new database if none exists. Uses database as active database if exists.
`db`	Check for current active database
`show dbs`	Display all database(s)
`db.dropDatabase()`	Delete active database
`db.createCollection(<collection_name>)`	Creates collection
`show collections`	Display all collections in active database
`db.<collection_name>.drop()`	Delete collection in active database
`db.<collection_name>.insert()`	Insert document into collection
`db.<collection_name>.find()`	Display documents in collection

***Note*:**

- The database `test_info` is created only when a document is inserted.
- The collection is automatically created when a document is added. ● To exit the Mongo shell, type `quit()`.
- To shut down the MongoDB server, press `Ctrl-C`.

References

1. MOE CPDD NoSQL Starter Kit

2. https://www.thegeekstuff.com/2014/01/sql-vs-nosql-db/

3. https://www.3pillarglobal.com/insights/exploring-the-different-types-of-nosql databases

4. https://www.mongodb.com/scale/types-of-nosql-databases 5.

Dan Sullivan (2015). NoSQL for Mere Mortals. Person Education 6.

https://www.mongodb.com/what-is-mongodb

7. http://api.mongodb.com/python/current/tutorial.html

8. SQL vs NoSQL | What's the Difference?
https://www.youtube.com/watch?v=Pf-9pjJK1e0

9. MongoDB in 5 Minutes with Eliot Horowitz
https://www.youtube.com/watch?v=EE8ZTQxa0AM

10. MongoDB Explained in 10 Minutes | SQL vs NoSQL |
Jumpstart https://www.youtube.com/watch?v=RGfFpQF0NpE

11. Radar The analytics edition

12. https://aws.amazon.com/nosql/

13. https://www.ibm.com/topics/nosql-databases