Temasek Junior College

2023 JC2 H2 Computing

Writing MongoDB Queries

Syllabus Objectives

After completing this set of notes, you should be able to:

- Understand how NoSQL database management system addresses the shortcomings of relational database management system (SQL).
- Explain the applications of SQL and NoSQL databases.

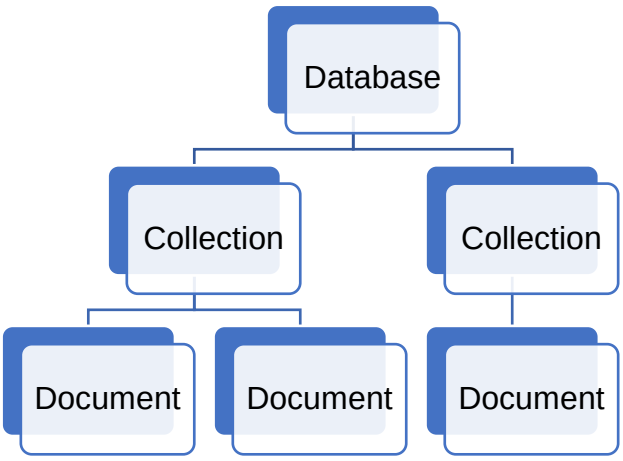
1 The MongoDB Document Database

Before proceeding further, let us look at some of the key concepts underlying MongoDB document databases

1.1 Terminologies

MongoDB Document Database Terminology	Equivalent SQL Database Terminology
Database	Database
Index	Index
Collection	Table
Document	Row
Field	Column

1.2 Database Structure



1.2.1 Collections

- A collection is a set of related documents.
- A MongoDB document database can have as many collections as it is required.
- In MongoDB, there are no join operations.
- Hence a MongoDB query is able to access and pull data from only one collection at a time.

1.2.2 Documents

- A document is a JSON (JavaScript Object Notation) object within a collection.
- Each document contains key-value pairs, similar to a Python dictionary.
- In each key-value pair, the key is the field and the value is the data associated with the field.
- The size limit for a document is 16 MB, which is generally sufficient in most cases.
- The syntax used in a document is as follows:

```
{
  field_1 : data_1
  field_2 : data_2
  …
}
```

- An example of a document is shown below:

```
{
  "_id" : ObjectID("5e9a568b12de6ebf44ce8ffe"),
  "ID" : 1,
  "Name" : "John",
  "Age" : 25
}
```

- Each document is identified using a unique key.
- The _id field is automatically indexed when the document is inserted into the database.
- IDs are 12-byte BSON (Binary JavaScript Object Notation, or Binary-JSON for short) objects.[1]
- As an ID is a BSON object and not a string, the ObjectID function is used to take in the ID as an argument.

[1] A BSON object is a binary-encoded serialisation of a JSON object. It is a textual object notation widely used to transmit and store data across web-based applications. JSON is easier to understand as it is human-readable, but compared to BSON, it supports fewer data types. BSON encodes type and length information, too, making it easier for machines to parse. Refer to Appendix A for a comparison between JSON and BSON.

2 CRUD Operations in MongoDB

2.1 Create Operations

2.1.1 Creating a new (and accessing an existing) database

The **use** command is used to create a new database or access an existing database.

Syntax
`use <database_name>`

Example 1
`use class_`

```
> use class_info
switched to db class_info
```

Entering `use class_info` in the MongoDB client shell will yield the output line `switched to db class_info`.

If the database `class_info` does not exist, it will be created.

To show the size of the database `class_info`, the **show** command can be used.

Syntax
`show dbs`

Example 2
`show dbs`

```
> show dbs
admin  0.000GB
local  0.000GB
```

At the moment, the size of the database `class_info` cannot be displayed as it has just been created and is empty.

The lines `admin 0.000GB` and `local 0.000GB` are displayed by default.
(You may not see `admin 0.000GB` if your account has no administrator rights conferred.)

2.1.2 Creating a new collection

Method 1: createCollection()

After selecting a database with the **use** command, the **createCollection()** method can be used to create a collection.

Syntax
```
db.createCollection(collection)
```

Example 3
```
db.createCollection('students')
```

```
> db.createCollection('students')
{ "ok" : 1 }
```

The line `{"ok" : 1}` will display when the collection `students` is successfully created.

Method 2: insert()

The insert() method which is used to insert documents into an existing collection can also be used to create a collection. This happens when the document is to be inserted into a collection that is yet to exist.

Syntax
```
db.<collection_name>.insert({document})
```

Example 4
```
db.teachers.insert({name : 'Miss Sasha', subject : 'General Paper'})
```

```
> db.teachers.insert({name : 'Miss Sasha', subject : 'General Paper'})
WriteResult({ "nInserted" : 1 })
```

The line `WriteResult({ "nInserted" : 1})` will display upon successful insertion of the document.

To show all the collections in the current database `class_info`, the command **show collections** can be used.

Syntax
```
show collections
```

Example 5
```
show collections
```

```
> show collections
students
teachers
```

Observe that the two collections added in **Examples 3** and **4** are now present in the database.

We shall now check the size of the `class_info` database.

Example 6

```
show dbs
```

```
> show dbs
admin       0.000GB
class_info  0.000GB
local       0.000GB
```

Although the `class_info` database is now displayed, indicating it is no longer empty, its size is still `0.000GB`.

The is due to the size of the database being too small for the number of significant figures displayed to illustrate its size precisely.

2.1.3 Inserting new documents

Method 1: insert()

As mentioned in the previous section, the **insert()** method can be used to insert a document to an existing collection.

Syntax

```
db.<collection_name>.insert({document})
```

Example 7

```
db.students.insert(
  {
    name : 'Lynn',
    gender : 'F',
    age : 17,
    hobbies : ['singing', 'dancing', 'gaming']
  }
)
```

```
> db.students.insert({name:'Lynn',gender:'F',age:17,hobbies:['singing','dancing','gaming']})
WriteResult({ "nInserted" : 1 })
```

The line `WriteResult({ "nInserted" : 1})` will display upon successful insertion of the document.

Method 2: insertOne()

The **insertOne()** method can be used to insert only one document.

Syntax

```
db.<collection_name>.insertOne({document})
```

Example 8

```
db.students.insertOne(
  {
    name : 'John',
    gender : 'M',
    age : 18,
    hobbies : ['soccer', 'gaming']
  }
)
```

```
> db.students.insertOne({name : 'John', gender : 'M', age : 18, hobbies : ['soccer', 'gaming']})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("62780ea0e7c31fec436a1df3")
}
```

The lines

```
{
  "acknowledged" : true
  "insertedId" : ObjectId(ID)
}
```

will display upon successful insertion of the document.

Method 3: insertMany()

The **insertMany()** method can be used to insert multiple documents at once.

Syntax

```
db.<collection_name>.insertMany([{document1}, {document2},…])
```

Example 9

```
db.teachers.insertMany(
  [
    {
      name : 'Mr Neo',
      subject : 'Phy'
    },
    {
      name : 'Mrs Kan',
      subject : 'Math'
    }
  ]
)
```

```
> db.teachers.insertMany([{name : 'Mr Neo', subject : 'Phy'}, {name : 'Mrs Kan', subject : 'Math'}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("62780eaae7c31fec436a1df4"),
                ObjectId("62780eaae7c31fec436a1df5")
        ]
}
```

The lines

```
{
  "acknowledged" : true
  "insertedIds" : [
          ObjectId(ID1),
          ObjectId(ID2)
  ]
}
```

will display upon successful insertion of the document.

Exercise 1

Create a NoSQL document database `petshop` on MongoDB containing the collections:

- `pets`
- `customers`

Insert the following details into the `pets` and `customers` collections respectively:

name	species
Mikey	Piranha
Davey	Goldfish
Suzy	Cat
Mikey	Dog
Terry	Dog
Mimi	Cat

name	email
John	john@gmail.com
Mary	mary@gmail.com

Run `show dbs` and `show collections` to verify that you have created the database correctly.

2.2 Read Operations

2.2.1 Reading all documents in a collection

The **find()** method can be used to display all documents in a collection. This is similar to the SQL statement SELECT * FROM <table>

Syntax
```
db.<collection_name>.find()
```

Example 10
```
db.students.find()
```

```
> db.students.find()
{ "_id" : ObjectId("62780e98e7c31fec436a1df2"), "name" : "Lynn", "gender" : "F", "age" : 17,
"hobbies" : [ "singing", "dancing", "gaming" ] }
{ "_id" : ObjectId("62780ea0e7c31fec436a1df3"), "name" : "John", "gender" : "M", "age" : 18,
"hobbies" : [ "soccer", "gaming" ] }
```

The documents of students that were added in **Examples 7** and **8** are displayed.

2.2.2 Reading document by ObjectID

If the ObjectID, which is a unique key referring to a particular document, is known, the **find(<ObjectID>)** method can be used to display that particular document.

Syntax
```
db.<collection_name>.find(ID)
```

Example 11
```
db.students.find("62780ea0e7c31fec436a1df3")
```

*Replace `"62780ea0e7c31fec436a1df3"` with one of the ObjectIDs generated for the documents that you have added to the `students` collection in the `class_info` database.

```
> db.students.find("62780ea0e7c31fec436a1df3")
{ "_id" : ObjectId("62780ea0e7c31fec436a1df3"), "name" : "John", "gender" : "M", "age" : 18,
"hobbies" : [ "soccer", "gaming" ] }
```

Depending on the build (version) of your MongoDB, you might need to use the following syntax instead.

Syntax
```
db.<collection_name>.find(ObjectID(ID))
```

Example 11 (Alternative Version)
```
db.students.find(ObjectID("62780ea0e7c31fec436a1df3"))
```

```
> db.students.find(ObjectId( "62780ea0e7c31fec436a1df3"))
{ "_id" : ObjectId("62780ea0e7c31fec436a1df3" ), "name" : "Lynn", "gender" : "F", "age" : 17,
"hobbies" : [ "singing", "dancing", "gaming" ] }
```

2.2.3 Reading documents that fulfil a given criterion

The **find(criteria)** method can be used to search for and display documents in a collection that fulfil a given criterion. This is similar to the SQL statement SELECT * FROM <table> WHERE <criteria>.

In MongoDB, the given criterion must be specified in JSON format.

Syntax

```
db.<collection_name>.find({criteria})
```

Example 12

Find all documents in the `students` collection of the `class_info` database where the student's name is Lynn.

```
db.students.find({name : 'Lynn'})
```

```
> db.students.find({name : 'Lynn'})
{ "_id" : ObjectId("62780e98e7c31fec436a1df2"), "name" : "Lynn", "gender" : "F", "age" : 17,
"hobbies" : [ "singing", "dancing", "gaming" ] }
```

2.2.4 Reading documents that fulfil a set of given criteria

The **find(criteria)** method can be extended to search for and display documents in a collection that fulfil a set of given criteria. This is done by specifying all given criterion into the argument in JSON format.

This is similar to the SQL statement SELECT * FROM <table> WHERE <criteria1> AND <criteria2> AND …

Syntax

```
db.<collection_name>.find({criteria1, criteria2, …})
```

Example 13

Find all documents in the `students` collection of the `class_info` database where the student's name is Lynn and the student's age is 17.

```
db.students.find({name : 'Lynn', age : 17})
```

```
> db.students.find({name : 'Lynn', age : 17})
{ "_id" : ObjectId("62780e98e7c31fec436a1df2"), "name" : "Lynn", "gender" : "F", "age" : 17,
"hobbies" : [ "singing", "dancing", "gaming" ] }
```

2.2.4 Reading the first document in a collection that fulfils a given criterion or a set of given criteria

The **findOne(criteria)** method can be used to search for and display the first document in a collection that fulfil a given criterion or a set of given criteria. This is similar to the SQL statement SELECT * FROM <table> WHERE <criteria1> AND <criteria2> AND … LIMIT 1.

Syntax

```
db.<collection_name>.findOne({criteria})
db.<collection_name>.findOne({criteria1, criteria2, …})
```

Example 14

Find the first document in the `students` collection of the `class_info` database where the student's name is Lynn.

```
db.students.findOne({name : 'Lynn'})
```

```
> db.students.findOne({name : 'Lynn'})
{
        "_id" : ObjectId("62780e98e7c31fec436a1df2"),
        "name" : "Lynn",
        "gender" : "F",
        "age" : 17,
        "hobbies" : [
                "singing",
                "dancing",
                "gaming"
        ]
}
```

Example 15

Find the first document in the `students` collection of the `class_info` database where the student's name is Lynn and the student's age is 17.

```
db.students.findOne({name : 'Lynn', age : 17})
```

```
> db.students.findOne({name : 'Lynn', age : 17})
{
        "_id" : ObjectId("62780e98e7c31fec436a1df2"),
        "name" : "Lynn",
        "gender" : "F",
        "age" : 17,
        "hobbies" : [
                "singing",
                "dancing",
                "gaming"
        ]
}
```

2.2.5 Attaining a more organised and systematic output

Observe that the output using **findOne(criteria)** is more systematic and organised compared to just using the **find()**, **find(<ObjectID>)** and **find(criteria)** methods.

We can use the **pretty()** method to achieve a more systematic and organised output.

Syntax

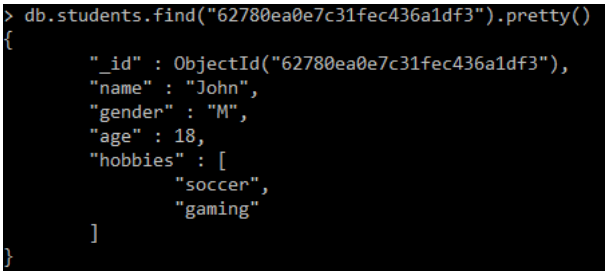
```
db.<collection_name>.find().pretty()
db.<collection_name>.find(ID).pretty()
db.<collection_name>.find({criteria}).pretty()
db.<collection_name>.find({criteria1, criteria2, …}).pretty()
```

Example 16

Repeat **Examples 10** to **11** by adding the pretty() method.

```
db.students.find().pretty()
```


```
db.students.find("62780ea0e7c31fec436a1df3").pretty()
```



Note that **Examples 12** and **13** will also yield outputs in the same format when the pretty() method is appended to the end of the statement i.e.

`db.students.find({name : 'Lynn'}).pretty()` in **Example 12** and
`db.students.find({name : 'Lynn', age : 17}).pretty()` in **Example 13**.

Exercise 2
Using the `petshop` database created in **Exercise 1**, perform the following:

- Add another piranha called Henry.
- List all the pets. Find the ID of Mikey the Dog.
- Use find to find Mikey by id.
- Use find to find all the cats (the 'species' is 'Cats').
- Find all the creatures named Mikey.
- Find all the creatures named Mikey who are piranha (the 'species' is 'Piranha').

2.2.6 Writing MongoDB Queries

Now that we know how to create and read documents in MongoDB, let us proceed to learn how MongoDB queries can be written. This is also part of the read operations in MongoDB.

Exercise 3
Add the following documents to the `students` collection in the `class_info` database.

1	`{name : 'Kate', gender : 'F', age : 16, cca : 'tennis'}`
2	`{name : 'Ernest', gender : 'M', age : 17, cca : 'choir', hobbies : ['singing']}`
3	`{name : 'Sam', gender : 'M', age : 16, hobbies : ['running', 'bowling']}`
4	`{name : 'Amy', gender : 'F', hobbies : ['drawing', 'painting']}`
5	`{name : 'Raul', gender : 'M'}`

From **Exercise 3**, we recall that documents in the same collection can have different schema from one another.

Comparison Query Operators

The table below gives the comparison query operators that can be used in MongoDB.

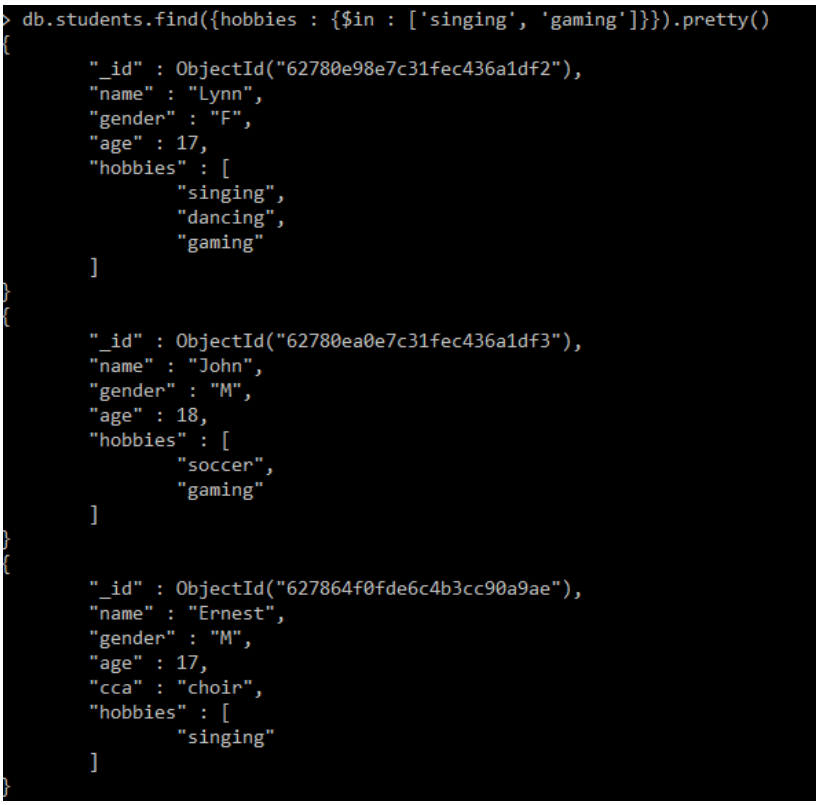
Operator	Description	Example
$eq	Matches values that are **equal** to a specified value.	`find({ age : {$eq : 17}})`
$gt	Matches values that are **greater than** a specified value.	`find({age : {$gt : 17}})`
$gte	Matches values that are **greater than or equal** to a specified value.	`find({age : {$gte : 17}})`
$in	Matches **any of the values** specified **in an array**.	`find({age : {$in : [15, 16, 17]}})`
$lt	Matches values that are **less than** a specified value.	`find({age : {$lt : 17}})`
$lte	Matches values that are **less than or equal** to a specified value.	`find({age : {$lte : 17}})`
$ne	Matches values that are **not equal** to a specified value.	`find({age : {$ne : 17}})`
$nin	Matches **none of the values** specified in an array,i.e. **not in** the array.	`find({age : { $nin : [18,17]}})`

Exercise 4

For **all** the comparison query operators, write a query for the `class_info` database.

An example using the `$in` operator has been done for you:

```
db.students.find({hobbies : {$in : ['singing', 'gaming']}})
```



```
> db.students.find({hobbies : {$in : ['singing', 'gaming']}}).pretty()
{
        "_id" : ObjectId("62780e98e7c31fec436a1df2"),
        "name" : "Lynn",
        "gender" : "F",
        "age" : 17,
        "hobbies" : [
                "singing",
                "dancing",
                "gaming"
        ]
}
{
        "_id" : ObjectId("62780ea0e7c31fec436a1df3"),
        "name" : "John",
        "gender" : "M",
        "age" : 18,
        "hobbies" : [
                "soccer",
                "gaming"
        ]
}
{
        "_id" : ObjectId("627864f0fde6c4b3cc90a9ae"),
        "name" : "Ernest",
        "gender" : "M",
        "age" : 17,
        "cca" : "choir",
        "hobbies" : [
                "singing"
        ]
}
```

Logical Query Operators

The table below gives the logical query operators that can be used in MongoDB.

Operator	Description	Example
$and	Joins query clauses with a logical **AND** returns all documents that match the conditions of both clauses.	`find({$and : [{age:{$gte: 17}}, {gender: "M"}]})`
$or	Joins query clauses with a logical **OR** returns all documents that match the conditions of either clause.	`find({$or : [{age : {$lt : 18}}, {gender : "M"}]})`
$not	Inverts the effect of a query expression and returns documents that **do not match the query expression**.	`find({name : {$not : {$eq : 'Lynn'}}})`

Exercise 5

For **all** the logical query operators, write a query for the `class_info` database. Combine the use of the logical query operators with the comparison query operators where appropriate.

An example using the `$and` operator has been done for you:

```
db.students.find({$and : [{age : {$gte : 17}}, {gender : 'M'}]})
```

```
> db.students.find({$and : [{age : {$gte : 17}}, {gender : 'M'}]}).pretty()
{
        "_id" : ObjectId("62780ea0e7c31fec436a1df3"),
        "name" : "John",
        "gender" : "M",
        "age" : 18,
        "hobbies" : [
                "soccer",
                "gaming"
        ]
}
{
        "_id" : ObjectId("627864f0fde6c4b3cc90a9ae"),
        "name" : "Ernest",
        "gender" : "M",
        "age" : 17,
        "cca" : "choir",
        "hobbies" : [
                "singing"
        ]
}
```

Element Query Operators

The table below gives the element query operators that can be used in MongoDB.

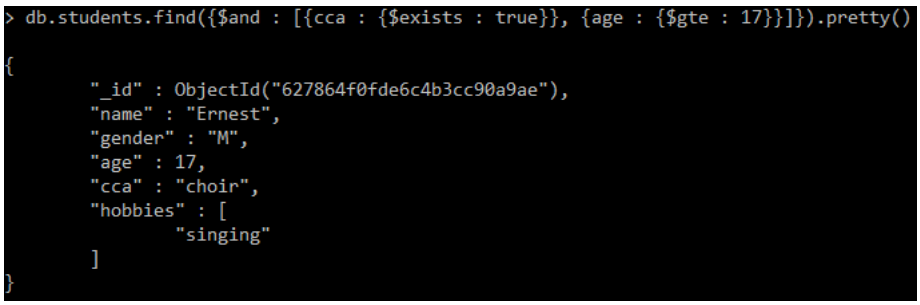
Operator	Description	Example
$exists	Matches documents that have the specified field.	`find({cca : {$exists : true}})`
	Finds all documents where the specified field does not exist.	`find({cca : {$exists : false}})`
$type	Selects documents if a field is of the specified type. $type is useful when querying highly unstructured data where data types are not predictable. The standard types in MongoDB are: “string”, “array”, “double” and “object”.	`find({hobbies : {$type : 'array'}})`

Exercise 6

For **all** the element query operators, write a query for the `class_info` database. Combine the use of the element query operators with other query operators where appropriate.

An example using the `$exists` operator has been done for you:

```
db.students.find({$and : [{cca : {$exists : true}}, {age : {$gte : 17}}]})
```



Exercise 7

Create a new collection `results` within the `class_info` database with the following documents:

1	`{name : 'Kate', scores : [{phy : 62}, {math : 54}, {chem : 71}]}`
2	`{name : 'Ernest', scores : [{bio : 58}, {math : 76}, {chem : 45}]}`
3	`{name : 'Sam', scores : [{comp : 67}, {math : 68}, {chem : 46}]}`
4	`{name : 'Amy', scores : [{math : 66}, {hist : 74}, {art : 75}]}`
5	`{name : 'Raul', scores : [{phy : 35}, {math : 44}, {chem : 49}]}`

Array Query Operators

The table below gives the array query operators that can be used in MongoDB.

Operator	Description	Example
$elemMatch	Matches documents that contain an array field with at least one element that matches all the specified query criteria.	`find({scores : {$elemMatch : {math : {$gte : 60}}}})`
Nested attribute syntax format		`find({"scores.math" : {$gte:60}})`

Exercise 7

For the array query operator $elemMatch and the nested attribute syntax format, write a query each for the `class_info` database. Combine the use of the array query operator with other query operators where appropriate. Do likewise for the nested attribute syntax format.

An example has been done for you:

```
db.results.find({scores : {$elemMatch: {math : {$gte : 60}}}})
```

```
> db.results.find({scores:{$elemMatch:{math:{$gte:60}}}})
{ "_id" : ObjectId("5e9bb1e5f82ddbb0af0bae7f"), "name" : "Ernest"
, "scores" : [ { "phy" : 58 }, { "math" : 76 }, { "chem" : 45 } ]
 }
{ "_id" : ObjectId("5e9bb1e5f82ddbb0af0bae80"), "name" : "Sam", "
scores" : [ { "comp" : 67 }, { "math" : 68 }, { "chem" : 46 } ] }

{ "_id" : ObjectId("5e9bb1e5f82ddbb0af0bae81"), "name" : "Amy", "
scores" : [ { "math" : 66 }, { "hist" : 74 }, { "art" : 75 } ] }
```

Exercise 8

Using the `class_info` database, write queries to find students who are:

(a) Female.
(b) Female or sing as a hobby.
(c) Male and without CCA.
(d) Having no CCAs and no hobbies.
(e) Males above the age of 16 and has a CCA
(f) At least 17 years old and does not like gaming.
(g) Scoring less than 50 marks in Chemistry.

Other Query Methods

Method	Description	Example	Outcome
limit()	Displays only the number of documents specified within the limit	find().limit(1)	Only the first document is displayed.
skip()	Skips the specified number of documents to be displayed	find().limit(3).skip(2)	First and second documents are skipped. 3rd, 4th and 5th documents are displayed.
sort()	Displays documents in sorted order as specified by the field(s) and sort order: 1 for ascending, -1 for descending	find().sort({age:1})	Documents sorted according to age, in ascending order.
count()	Count the number of documents	find().count()	Returns the number of documents.

Exercise 9

For the query methods limit(), skip(), sort() and count(), write a query each for the `class_info` database. Combine each method with the use of other query operators with and/or methods where appropriate.

An example has been done for you:

```
db.results.find({scores : {$elemMatch  :
                                {math : {$gte : 60}}}}).sort({name : 1})
```

```
> db.results.find({scores:{$elemMatch:{math:{$gte:60}}}}).sort({name:1})
{ "_id" : ObjectId("5e9bb1e5f82ddbb0af0bae81"), "name" : "Amy", "scores" : [ { "math" : 66 }, { "hist" : 74 }, { "art" : 75 } ] }
{ "_id" : ObjectId("5e9bb1e5f82ddbb0af0bae7f"), "name" : "Ernest", "scores" : [ { "phy" : 58 }, { "math" : 76 }, { "chem" : 45 } ] }
{ "_id" : ObjectId("5e9bb1e5f82ddbb0af0bae80"), "name" : "Sam", "scores" : [ { "comp" : 67 }, { "math" : 68 }, { "chem" : 46 } ] }
```

Exercise 10

Using the `class_info` database, write queries to find:

- Female students, sorted in ascending order of age, skipping the first female.
- Male studens, sorted in alphabetical order of name, limited to 2.
- Students scoring below 50 marks for chemistry, sorted in descending order of scores.
- Number of students who scored more than 50 for mathematics.
- The top 2 students in mathematics.

2.3 Update Operations

2.3.1 Updating an existing document

Method 1: update()

The **update()** method can be used to update an existing document.

Syntax

```
db.<collection_name>.update({criteria}, {update})
```

Example 17

```
db.teachers.update({name : 'Mr Lee'}, {name : 'Mr Lee', subject : 'Comp'})
```

```
> db.teachers.update({name:"Mr Lee"},{name:"Mr Lee",subject:"Comp"})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

The line `WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })` will display upon successful update of the document.

Method 2: update() with upsert

If **{upsert : True}** is included in the syntax, a new document will be created when no document matches the criteria. The default value of upsert is false, which does not insert a new document when no match is found.

Syntax

```
db.<collection_name>.update({criteria}, {update}, {upsert : true})
```

When multiple MongoDB clients shells issue the same update, including {upsert :true} prevents the same document from being inserted more than once.

Example 18

```
db.teachers.update({name : 'Mr Chua'}, {name : 'Mr Chua' , subject: 'Chem'
} , {upsert: true} )
```

```
> db.teachers.update({name:'Mr Chua'},{name:'Mr Chua',subject:"Chem"},{upsert:true})
WriteResult({
        "nMatched" : 0,
        "nUpserted" : 1,
        "nModified" : 0,
        "_id" : ObjectId("5e9b10dd687985a29f75eb50")
})
```

The lines

```
WriteResult({
        "nMatched" : 0,
        "nUpserted" : 1,
        "nModified" : 0,
        "_id" : ObjectID(ID)
})
```

will display upon successful upsert of the document.

2.3.2 Updating specified field of an existing document

The **$set** operator can be included in the update() method to update specified fields of an existing document.

Syntax
```
db.<collection_name>.update({criteria}, {$set : {specified field : update})
```

Example 19
```
db.teachers.update({name : 'Mr Chua'}, {$set : {subject: 'Bio'}})
```

```
> db.teachers.update({name:'Mr Chua'},{$set:{subject:'Bio'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

The line `WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1})` will display upon successful update of the specified fields in the document.

2.3.3 Updating first document found to match given criteria only

Using the **updateOne()** method, when more than one document fulfil the criteria specified in the method, only update the first occurrence is updated.

Syntax
```
db.<collection_name>.updateOne({criteria}, {update})
```

Example 20
```
db.students.updateOne({age: {$exists : false}}, {$set : {age : 17}})
```

```
> db.students.updateOne({age:{$exists:false}},{$set:{age:17}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

The line `{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1})` will display upon successful update of the first document found matching the criteria.

2.3.4 Updating all documents found to match given criteria

Using the **updateMany()** method, all documents that fulfil the criteria specified in the method will be updated.

Syntax
```
db.<collection_name>.updateMany({criteria}, {update})
```

Example 21
```
db.teachers.updateMany({name: {$exists : true}}, {$set : {positon : 'HOD'}})
```

```
> db.teachers.updateMany({name:{$exists:true}},{$set:{position:'HOD'}})
{ "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 4 }
```

The line `{ "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 4})` will display upon successful update of all documents found matching the criteria.

2.3.5 Removing a specified field

The **$unset** operator can be used to remove a particular field. As the values of that field no longer has any effect within the database, an empty string is used as replacement of the data for convenience.

Syntax

```
db.<collection_name>.update({criteria}, {$unset : {field: ''})
```

Example 22

```
db.teachers.updateMany({name: {$exists : true}}, {$set : {positon : 'HOD'}})
```

```
> db.teachers.update({name:'Mr Lee'},{$unset:{position:''}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

The line `WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1})` will display upon successful update of all documents found matching the criteria.

Exercise 11

Write statements to make the following changes to the `class_info` database:

- Add Raul's age as 19
- Add on to Amy's hobbies by including knitting
- For all males age 18 and above, include a new field enlist, and set it to false
- For those without ccafield, include it and set it to empty string
- Change position of Physics teacher to subject head
- Remove the position field for all teachers except for Miss Tan
- Change the chemistry score for Raul to 39

2.4 Delete Operations

2.4.1 Delete all documents fulfilling specified criteria

Method 1: remove()

The **remove()** method deletes all documents matching the specified criteria.

Syntax

```
db.<collection_name>.remove({criteria})
```

Note: To delete all documents using a pair of empty braces `{}` as the argument.

Example 23

```
db.teachers.remove({name: 'Mr Lim'})
```

```
> db.teachers.remove({name:'Mr Lim'})
WriteResult({ "nRemoved" : 1 })
```

The line `WriteResult({ "nRemoved" : x })` will display upon successful removal of all documents matching the criteria. Herein, `x` indicates the number of documents removed.

Method 2: deleteMany()

The **deleteMany()** method also deletes all documents matching the specified criteria.

Syntax

```
db.<collection_name>.deleteMany({criteria})
```

Note: To delete all documents using a pair of empty braces `{}` as the argument.

Example 24

```
db.students.deleteMany({age : 16})
```

```
> db.students.deleteMany({age:16})
{ "acknowledged" : true, "deletedCount" : 2 }
```

The line `{ "acknowledge" : true, "deletedCount" : x })` will display upon successful removal of all documents matching the criteria. Herein, `x` indicates the number of documents removed.

2.4.2 Delete first document found to fulfil specified criteria

The **deleteOne()** method deletes the first document matching the specified criteria.

Syntax
```
db.<collection_name>.deleteOne({criteria})
```

Note: To delete all documents using a pair of empty braces `{}` as the argument.

Example 25
```
db.students.deleteOne({gender: 'F'})
```

```
> db.students.deleteOne({gender:'F'})
{ "acknowledged" : true, "deletedCount" : 1 }
```

The line `{ "acknowledge" : true, "deletedCount" : 1 })` will display upon successful removal of the first document matching the criteria.

2.4.2 Delete an entire collection

The **drop()** method can be used to delete an entire collection

Syntax
```
db.<collection_name>.drop()
```

Example 26
```
db.teachers.drop()
```

```
> db.teachers.drop()
true
```

The return value of true is obtained when the collection exists and is successfully deleted.

Exercise 12

Write statements to make the following changes to the `class_info` database:

- Remove all female students.
- Remove the youngest male student.
- Remove the result with the worst math score.
- Drop results collection.

2.4.2 Delete an entire database

The **dropDatabase()** method is used to delete an entire database.

Syntax

```
db.dropDatabase()
```

The steps to drop a database are as follows:

1) show dbs #show all databases
2) use class_info #select the database which you want to drop
3) db #ensure that the current dbis the one you want to drop
4) db.dropDatabase() #drop the database
5) show dbs #show all databases and the dropped database will not show

```
> db.dropDatabase()
{ "dropped" : "class_info", "ok" : 1 }
```