WORKSHEET 8

Student Name: Vivek Kumar **UID:** 21BCS8129
DOMAIN CAMP: 16-01-2023 to 28-01-2023 **Section/Group:** DWWC-77
Subject Name: Database Management System

Question 1:- Query documents that belong to a specific customer.

Answer:- We use the find method to query documents from a MongoDB database. If used without any arguments or collections, find method retrieves all documents.
We want to see the document belongs to customer John so the name field needs to be specified in the find method.
Query:-

```
> db.customer.find( {name: "John"} )

{ "_id" : ObjectId("600c1806289947de938c68ea"), "name" : "John", "age" : 32,  "gender" : "male", "amount" : 32 }
```

We can attach pretty() method to make the document seem more appealing.

```
> db.customer.find( {name: "John"} ).pretty()

{
"_id" : ObjectId("600c1806289947de938c68ea"),
"name" : "John",
"age" : 32,
"gender" : "male",
"amount" : 32
}
```

It is easier to read now.

Question 2:- Query documents that belong to customers older than 40.

Answer:- The condition is applied to age field using a logical operator. The "$gt" stands for "greater than" and is used as follows.
Query:-

```
> db.customer.find( {age: {$gt:40}} ).pretty()

{
"_id" : ObjectId("600c19d2289947de938c68ee"),
"name" : "Jenny",
"age" : 42,
"gender" : "female",
```

"amount" : 36
}

Question 3:- Query documents that belong to female customers who are younger than 25.

Answer:- This example is like a combination of the previous two examples. Both conditions must be met so we use "and" logic to combine the conditions. It can be done by writing both conditions separated by comma.

Query:-

```
> db.customer.find( {gender: "female", age: {$lt:25}} ).pretty()
```

{
"_id" : ObjectId("600c19d2289947de938c68f0"),
"name" : "Samantha",
"age" : 21,
"gender" : "female",
"amount" : 41
}
{
"_id" : ObjectId("600c19d2289947de938c68f1"),
"name" : "Laura",
"age" : 24,
"gender" : "female",
"amount" : 51
}

The "$lt" stands for "less than".

Question 4:- In this example, we will repeat the previous example in a different way. Multiple conditions can also be combined with "and" logic as below.

Answer:- Query:-

```
> db.customer.find( {$and :[ {gender: "female", age: {$lt:25}} ]} ).pretty()
```

The logic used for combining the conditions is indicated at the beginning. The remaining part is same as the previous example but we need to put the conditions in a list ([ ]).

Question 5:- Query customers who are either male or younger than 25.

Answer:- This example requires a compound query with “or” logic. We just need to change “$and” to “$or”.

Query:-

```
> db.customer.find( { $or: [ {gender: "male"}, {age: {$lt: 22}} ] })

{ "_id" : ObjectId("600c1806289947de938c68ea"), "name" : "John", "age" : 32,  "gender" : "male", "amount" : 32 }

{ "_id" : ObjectId("600c19d2289947de938c68ed"), "name" : "Martin", "age" : 28,          "gender" : "male", "amount" : 49 }

{ "_id" : ObjectId("600c19d2289947de938c68ef"), "name" : "Mike", "age" : 29,  "gender" : "male", "amount" : 22 }

{ "_id" : ObjectId("600c19d2289947de938c68f0"), "name" : "Samantha", "age" : 21,       "gender" : "female", "amount" : 41 }
```

Question 6:- MongoDB allows for aggregating values while retrieving from the database. For instance, we can calculate the total purchase amount for males and females. The aggregate method is used instead of the find method.

Answer:- Query:-

```
> db.customer.aggregate([
{ $group: {_id: "$gender", total: {$sum: "$amount"} } }
])

{ "_id" : "female", "total" : 198 }
{ "_id" : "male", "total" : 103 }
```

Question 7:- Let’s take the previous example one step further and add a condition. Thus, we first select documents that “match” a condition and apply aggregation.

Answer:- The following query is an aggregation pipeline which first selects the customers who are older than 25 and calculates the average purchase amount for males and females.

Query:-

```
> db.customer.aggregate([
... { $match: { age: {$gt:25} } },
... { $group: { _id: "$gender", avg: {$avg: "$amount"} } }
... ])
{ "_id" : "female", "avg" : 35.33 }
{ "_id" : "male", "avg" : 34.33 }
```

Question 8:- The query in the previous example contains only two groups so it is not necessary to sort the results. However, we might have queries that return several values. In such cases, sorting the results is a good practice.

Answer:- We can sort the results of the previous query by the average amount in ascending order.

Query:-

```
> db.customer.aggregate([
... { $match: { age: {$gt:25} } },
... { $group: { _id: "$gender", avg: {$avg: "$amount"} } },
... { $sort: {avg: 1} }
... ])

{ "_id" : "male", "avg" : 34.33 }
{ "_id" : "female", "avg" : 35.33 }
```

We have just added "$sort" in the aggregation pipeline. The field used for sorting is specified along with the sorting behavior. 1 means in ascending order and -1 means in descending order.

Question 9:- How NoSQL queries are constructed into a SQL query and using JSON objects.

Answer:- A query consists of these parts:

1. fields to be extracted.
2. table to extract the records from.
3. expression for filtering the table rows.
4. groupby - fields to group the data under.
5. aggregate functions to be applied to columns in fields.
6. orderby - fields to order the return data by.
7. limit - an integer number of records to return.

Only the table and expression parameters are mandatory. The NoSQL queries are then constructed into a SQL query of the following form:

Syntax:-

```
SELECT fields with aggregation
FROM table
WHERE expression
GROUP BY groupby
ORDER BY orderby
LIMIT limit
```

NoSQL queries are constructed using JSON objects. Below is an example:

Syntax:-

```
{
```

```
    object: String,
    q: Expression,
    fields: Array of String,
    groupBy: Array of String,
    aggregation: Object mapping fields to aggregate functions
}
```

For example, the shortest query you can write would be:

```
{ "object": "String", "q": "Expression" }
```

This NoSQL object is converted into SQL Query:

```
SELECT *
FROM table
WHERE query
```

Examples:- This simple query retrieves the name and salary of all employees in position of "Sales Manager":

```
{
    "object": "employees",
    "q": {
      "position" : "Sales Manager"
    },
    "fields": ["name", "salary"]
}
```

Queries can also be used to compare an object's fields to constant values using common comparison operators. For example, to retrieve all fields for all employees under the age of 25, you can use the following query:

```
{
    "object": "employees",
    "q": {
    "age": { "$lt" : 25 }
    }
}
```

Question 10: How does MongoDB handle indexing?

Indexing is done using the ensureIndex() method. The syntax is:

```
db.COLLECTION_NAME.ensureIndex()
```