Fundamentals of Document-Oriented Database

A. CRUD Operation (Creating, Reading (Querying), Updating, and Deleting Documents)

Inserting Documents: Inserts are the basic method for adding data to MongoDB. To insert a single document, use the collection's *insertOne* method. *insertOne* will automatically add an *"_id"* key to the document if you do not provide one.

```
> db.movies.insertOne({"title" : "Stand by Me"})
```

If you need to insert **multiple documents** into a collection, you can use *insertMany*.

```
> db.movies.insertMany([{"title" : "Ghostbusters"},
...                     {"title" : "E.T."},
...                     {"title" : "Blade Runner"}]);
```

When performing a bulk insert using *insertMany*, if a document halfway through the array produces an error of some type, what happens depends on whether you have opted for *ordered* or *unordered* operations.

The *ordered* insert is the default if no order is specified. For *ordered* inserts, the array passed to *insertMany* defines the insertion order. If a document produces an insertion error, **no documents beyond that point in the array will be inserted**.
For *unordered* inserts, MongoDB **will attempt to insert all documents**, regardless of whether some insertions produce errors.

```
db.movies.insertMany([
... {"_id" : 3, "title" : "Sixteen Candles"},
... {"_id" : 4, "title" : "The Terminator"},
... {"_id" : 4, "title" : "The Princess Bride"},
... {"_id" : 5, "title" : "Scarface"}],
... {"ordered" : false})
```

Removing Documents: MongoDB provides *deleteOne* and *deleteMany* for this purpose. To delete the document with the "*_id*" value of 4, we use *deleteOne* in the mongo shell.

```
> db.movies.deleteOne({"_id" : 4})
```

```
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun", "year" : 1985 }
{ "_id" : 1, "title" : "Back to the Future", "year" : 1985 }
{ "_id" : 3, "title" : "Sixteen Candles", "year" : 1984 }
```

To delete all the documents that **match a filter**, use *deleteMany*:

```
> db.movies.deleteMany({"year" : 1985})
{ "acknowledged" : true, "deletedCount" : 2 }
```

Updating Documents: Once a document is stored in the database, it can be changed using one of several update methods: *updateOne*, *updateMany*, and *replaceOne*. *updateOne* and *updateMany* each take a filter document as their **first parameter** and a modifier document, which describes changes to make, as the **second parameter**.

```
{
"_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
"name" : "Joe",
"friends" : 32,
"enemies" : 2
}
```

replaceOne fully replaces a matching document with a new one. This can be useful to do a dramatic schema migration. Example: suppose we are making major changes to a user document, which we want to move the "*friends*" and "*enemies*" fields to a "*relationships*" subdocument. We can change the structure of the document in the shell and then replace the database's version with a *replaceOne*.

```
> var Joe = db.users.findOne({"name" : "Joe"});
> Joe.relationships = {"friends" : Joe.friends, "enemies" : Joe.enemies};
{
        "friends" : 32,
        "enemies" : 2
}
> Joe.username = Joe.name;
"joe"
> delete Joe.friends;
```

```
true
> delete Joe.enemies;
true
> delete Joe.name;
true
> db.users.replaceOne({"name" : "Joe"}, Joe);
```

The updated document will show the following structure and values:

```
{
"_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
"username" : "Joe",
"relationships" :
        {
                "friends" : 32,
                "enemies" : 2
        }
}
```

Updating documents using $set operator
"*$set*" operator sets the value of a field. If the field does not yet exist, it will be created. This can be handy for updating schemas or adding user-defined keys. For example, suppose you have a simple *user* profile stored as a document having values of the following:

```
> db.users.findOne()
{
        "_id" : ObjectId("4b253b067525f35f94b60a31"),
        "name" : "John",
        "age" : 24,
        "sex" : "male",
        "location" : "Taytay"
}
```

Now, we want to update the document by adding a key, which is his favorite book.

```
> db.users.updateOne({"name" : "John"}, {"$set" : {"favorite book" :
"Green Eggs and Ham"}})
```

"*$set*" can even change the type of the key it modifies. For example, we want to add another value of books that the user likes; this will change the type of "*favorite book*" key into an array.

```
> db.users.updateOne({"name" : "John"},
... {"$set" : {"favorite book" :
... ["Cat's Cradle", "Foundation Trilogy", "Ender's Game"]}})
```

If we want to remove the "*favorite book*" key altogether, we can use the "*$unset*" operator.

```
> db.users.updateOne({"name" : "John"}, {"$unset" : {"favorite book" :
1}})
```

We can also use "*$set*" to reach in and change **embedded documents**. Assume that our document has the following values:

```
> db.books.findOne()
{
"_id" : ObjectId("4b253b067525f35f94b60a31"),
"title" : "MongoDB-2019",
"author" :
  {
        "name" : "Nicole",
        "email" : "Nic@example.com"
  }
}
```

If we want to update the author name where it is inside the author subdocument. Use this query:

```
>db.books.updateOne({"author.name" : "Nicole"},
... {"$set" : {"author.name" : "Nicole Tagayon"}})
> db.books.findOne()
  {
        "_id" : ObjectId("4b253b067525f35f94b60a31"),
        "title" : "MongoDB-2019",
        "author" : {
        "name" : "Nicole Tagayon",
        "email" : "Nic@example.com"
  }
}
```

To **modify all of the documents** matching a filter, use *updateMany*. *updateMany* follows the same semantics as *updateOne* and takes the same parameters. The key difference is in the number of documents that might be changed.
Assume that one document has the following values:

```
{ "_id" : 1, "birthday" : "10/13/1978"}
{ "_id" : 2, "birthday" : "10/13/1978"}
{ "_id" : 3, "birthday" : "10/13/1978"}
```

Using *updateMany,* we will add the "*gift*" key to each data.

```
> db.users.updateMany({"birthday" : "10/13/1978"},
... {"$set" : {"gift" : "Happy Birthday!"}})
```

Querying Documents: The *find()* method is used to perform queries in MongoDB. Querying returns a subset of documents in a collection. Which documents get returned is determined by the first argument to find, which is a document specifying the query criteria.

An **empty query document** (i.e., *db.movies.find()*) will match everything in the "*movies*" collection. We begin restricting our search when we start adding key/value pairs to the query document. For example: to find all documents where the value for "*age*" is ***27***, we can add that key/value pair to the query document:

```
> db.users.find({"age" : 27})
```

If we have a string we want to match, such as a "*username*" key with the value "*john*", we use that key/value pair instead:

```
> db.users.find({"username" : "John"})
```

Multiple conditions can be strung together by adding more key/value pairs to the query document, which gets interpreted as "*condition1 AND condition2*"

```
> db.users.find({"username" : "John", "age" : 24})
```

B. Query Selector

Query conditional: "*$lt*", "*$lte*", "*$gt*", and "*$gte*" are all comparison operators, corresponding to <, <=, >, and >=, respectively.

For example, to look for users who are **between the ages of 18 and 30**, we can do this:

```
> db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})
```

To query for documents where a key's value is **not equal** to a certain value, you can use conditional operator, "*$ne*", which stands for "**not equal**." If you want to find all users who do not have the username "*john*", you can query for them using this:

```
> db.users.find({"username" : {"$ne" : "joe"}})
```

OR Queries: There are two ways to do an *OR* query in MongoDB. "*$in*" can be used to query for a variety of values for a single key. "*$or*" is more general; it can be used to query for any of the given values across multiple keys.
For instance, suppose we're running a raffle and the winning ticket numbers are *725*, *542*, and *390*. To find all three of these documents, we can construct the following query:

```
> db.raffle.find({"ticket_no" : {"$in" : [725, 542, 390]}})
```

Now we want to find documents where "*ticket_no*" is *725* or "*winner*" is *true*. For this type of query, we'll need to use the "*$or*" conditional. "*$or*" takes an array of possible criteria. In the raffle case, using "*$or*" would look like this:

```
> db.raffle.find({"$or" : [{"ticket_no" : 725}, {"winner" : true}]})
```

$not: is useful to find all documents that don't match a given pattern. For example, we want to retrieve the employee's documents whose salary is **not greater than ₱10,000**.

```
> db.users.find({salary: {$not: {$gt: 10000}}}).pretty()
```

Regular Expressions: "*$regex*" provides regular expression capabilities for pattern matching strings in queries. Regular expressions are useful for flexible string matching. For example, we want to find all users with the name "*John*" or "*john*," we can use a regular expression to do case-insensitive matching:

```
> db.users.find( {"name" : {"$regex" : /john/i } })
```

Regular expression flags (e.g., i) are allowed but not required. If we want to match not only various capitalizations of "*john*," but also "*johnny*," we can continue to improve our regular expression by doing this:

```
> db.users.find({"name" : /johnny?/i})
```

Querying Arrays: Querying for elements of an array is designed to behave the way querying for scalars (string, numbers, date, etc.) does. For example, if the array is a list of fruits, like this:

```
> db.food.insertOne({"fruit" : ["apple", "banana", "peach"] })
```

The following query will successfully match the document:

```
> db.food.find({"fruit" : "banana"}
```

$all: If you need to match arrays by more than one element, you can use *"$all".* This allows you to match a list of elements. For example, assume we create a collection with three elements:

```
> db.food.insertOne({"_id" : 1, "fruit" : ["apple", "banana", "peach"] })
> db.food.insertOne({"_id" : 2, "fruit" : ["apple","kumquat", "orange"] })
> db.food.insertOne({"_id" : 3, "fruit" : ["cherry", "banana", "apple"] })
```

Then we can find all documents with both "apple" and "banana" elements by querying with "*$all*".

```
> db.food.find({fruit : {$all : ["apple", "banana"]}})
{"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
{"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}
```

If we want to query for a specific element of an array, you can specify an index using the syntax *key.index*:

```
> db.food.find({"fruit.2" : "peach"})
```

$size - matches any array with the number of elements specified by the argument. Here's an example:

```
> db.food.find({"fruit" : {"$size" : 3}})
```

$slice - an operator that can be used to return a subset of elements for an array key
For example, suppose we had a blog post document and we wanted to **return the first 10 comments**:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : 10}})
```

Alternatively, if we wanted the **last 10 comments**, we could use *−10*:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -10}})
```

It can also return pages in the middle of the results by taking an offset and the number of elements to return:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : [23, 10]}})
```

This would skip the first 23 elements and **return the 24th through 33rd**. If there were fewer than 33 elements in the array, it would return as many as possible.

Querying on Embedded Documents: Embedded document or nested documents are those types of documents which contain a document inside another document.
For example, assume that we have a document having the following elements:

```
{
        "name" : {
        "first" : "John",
        "last" : "Escalona"
        },
"age" : 45
}
```

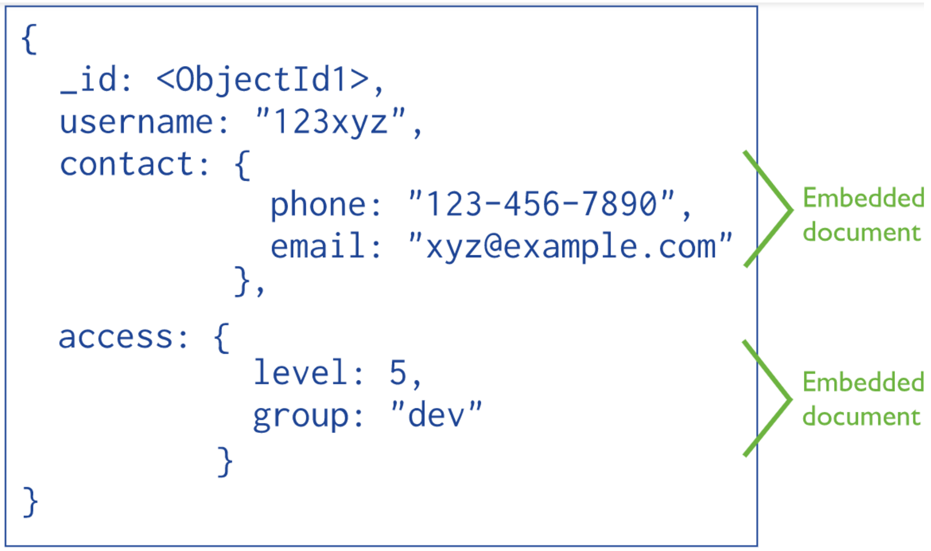
You can query for embedded keys using dot notation:

```
> db.people.find({"name.first" : "John", "name.last" : "Escalona"})
```

C. Data Model

Data Modelling: In MongoDB, data has a flexible schema. It is totally different from the SQL database, where you had to determine and declare a table's schema before inserting data. MongoDB collections do not enforce document structure. The main challenge in data modeling is balancing the need of the application, the performance characteristics of the database engine, and the data retrieval patterns.

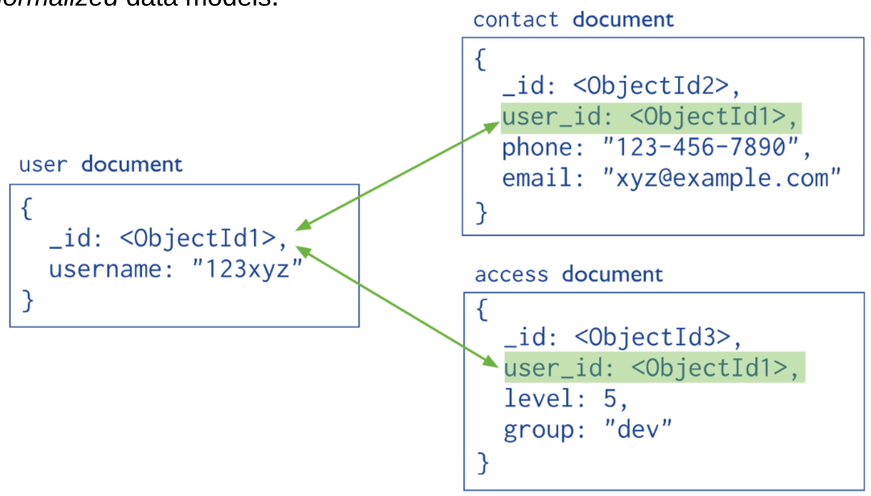
Embedded Data: *Embedded documents* capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures in a field or array within a document. These *denormalized* data models allow applications to retrieve and manipulate related data in a single database operation.



```
{
   _id: <ObjectId1>,
   username: "123xyz",
   contact: {
                phone: "123-456-7890",
                email: "xyz@example.com"
             },
   access: {
               level: 5,
               group: "dev"
            }
}
```

***Figure 1**. Example of an Embedded Documents*

References: References store the relationships between data by including links or references from one document to another. Applications can resolve these references to access the related data. Broadly, these are *normalized* data models.



```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

***Figure 2**. Example of a normalized data model using references*

REFERENCES

MongoDB (n.d.). *Query and Update Operators*. Retrieved from https://docs.mongodb.com/manual/reference/operator/

Chodorow, K., Brazil, E., and Bradshaw, S. (2019). *MongoDB: The definitive guide (3rd ed.)*. O'Reilly Media, Inc.