MongoDB Query (Part 2)

Replacing Documents

```
> db.users.insertMany([
  {"_id": 2, "name": "Jon Snow", "email": "Jon.Snow@got.es"},
  {"_id": 3, "name": "Joffrey Baratheon", "email": "Joffrey.Baratheon@got.es"},
  {"_id": 5, "name": "Margaery Tyrell", "email": "Margaery.Tyrell@got.es"},
  {"_id": 6, "name": "Khal Drogo", "email": "Khal.Drogo@got.es"}
])
{ "acknowledged" : true, "insertedIds" : [ 2, 3, 5, 6 ] }
> db.users.find()
{ "_id" : 2, "name" : "Jon Snow", "email" : "Jon.Snow@got.es" }
{ "_id" : 3, "name" : "Joffrey Baratheon", "email" : "Joffrey.Baratheon@got.es" }
{ "_id" : 5, "name" : "Margaery Tyrell", "email" : "Margaery.Tyrell@got.es" }
{ "_id" : 6, "name" : "Khal Drogo", "email" : "Khal.Drogo@got.es" }
```

Now, suppose the user **`Margaery Tyrell`** gets married to **`Joffrey Baratheon`**, and she wishes to change her surname to her husband's

```
> db.users.replaceOne(
  {"_id" : 5},
  {"name": "Margaery Baratheon", "email": "Margaery.Baratheon@got.es"}
)
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Here, the first argument is the query filter to identify the document to be replaced, and the second argument is the new document.

the following replace command will have the same effect as the previous one, as long as there is only one user with the name of Margaery Tyrell. If there is more than one document that matches the query, then only the first one will be replaced:

```
>db.users.replaceOne(
  {"name": "Margaery Tyrell },
  {"name": "Margaery Baratheon", "email": "Margaery.Baratheon@got.es"}
)
```

_id fields are immutable in MongoDB. Immutable fields are like normal fields; however, once assigned with a value, their value cannot be changed again.

Upsert Using Replace

However, in real-world scenarios, you will mostly be doing these operations in large numbers. On a large-scale system, performing a two-step update or insert operation for each of the records will be very time-consuming and error prone. However, having a dedicated command, you can simply prepare and execute an upsert command for each of the records you receive and let MongoDB do the update or insert.

```
> db.users.find()
{"_id": 2, "name": "Jon Snow", "email": "Jon.Snow@got.es"}
{"_id": 3, "name": "Joffrey Baratheon", "email": "Joffrey.Baratheon@got.es"}
{"_id": 5, "name": "Margaery Baratheon", "email": "Margaery.Baratheon@got.es"}
{"_id": 6, "name": "Khal Drogo", "email": "Khal.Drogo@got.es"}

>db.users.replaceOne(
  {"name" : "Margaery Baratheon"},
  {"name": "Margaery Tyrell", "email": "Margaery.Tyrell@got.es"},
  { upsert: true }
)

>db.users.replaceOne(
  {"name" : "Tommen Baratheon"},
  {"name": "Tommen Baratheon", "email": "Tommen.Baratheon@got.es"},
  { upsert: true }
)
```

Replacing Using findOneAndReplace()

MongoDB provides another operation, **`findOneAndReplace()`**, to perform the same operations. However, it provides more options. Its main features are as follows:

- As the name indicates, it finds one document and replaces it.
- If more than one document is found matching the query, the first one will be replaced.
- A sort option can be used to influence which document gets replaced if more than one document is matched.
- By default, it returns the original document.
- If the option of **`{returnNewDocument: true}`** is set, the newly added document will be returned.
- Field projection can be used to include only specific fields in the document returned in response.

```
db.movies.insertMany([
   { "_id": 1011, "title" : "Macbeth" },
   { "_id": 1513, "title" : "Macbeth" },
   { "_id": 1651, "title" : "Macbeth" },
   { "_id": 1819, "title" : "Macbeth" },
   { "_id": 2117, "title" : "Macbeth" }
])

db.movies.findOneAndReplace(
   {"title" : "Macbeth"},
   {"title" : "Macbeth", "latest" : true},
   {
      sort : {"_id" : -1},
      projection : {"_id" : 0}
   }
)
```

you found the document for a movie by its title and replaced it with another document that contains an additional field—that is, **latest : true.**

```
db.movies.findOneAndReplace(
   {"title" : "Macbeth"},
   {"title" : "Macbeth", "latest" : true},
   {
      sort : {"_id" : -1},
      projection : {"_id" : 0},
      returnNewDocument : true
   }
)
```

```
> db.movies.find({"title" : "Macbeth"})
{ "_id" : 1011, "title" : "Macbeth" }
{ "_id" : 1513, "title" : "Macbeth" }
{ "_id" : 1651, "title" : "Macbeth" }
{ "_id" : 1819, "title" : "Macbeth" }
{ "_id" : 2117, "title" : "Macbeth", "latest" : true }
```

Replace versus Delete and Re-Insert

the following potential shortfalls of using it over dedicated replace functions:

1. First of all, in the delete and insert method, the data is transferred over the wire multiple times. This involves the drivers or clients to parse the data in multiple stages. This will slow down the overall performance.
2. When multiple clients are constantly reading and writing to your collections, concurrency issues may arise. As an example, say you have successfully deleted a record and before you insert the new record, some other client accidentally inserts a different record with the same **`_id`**.
3. Your database client or driver may lose its connection to the database in the middle of two operations. For example, the delete operation was successful but insertion could not happen. To avoid such issues, you will have to run your commands in a transaction so that the failure of one operation can revert the previously successful operations in the same transaction.

The dedicated replace functions, on the other hand, are effectively atomic and are therefore safe to use in concurrent environments.

Modify Fields

Updating a Document with updateOne()

```
> db.movies.insertMany([
  {"_id": 1, "title": "Macbeth", "year": 2014, "type": "series"},
  {"_id": 2, "title": "Inside Out", "year": 2015, "type": "movie", "num_mflix_comments": 1},
  {"_id": 3, "title": "The Martian", "year": 2015, "type": "movie", "num_mflix_comments": 1},
  {"_id": 4, "title": "Everest", "year": 2015, "type": "movie", "num_mflix_comments": 1}
])
{ "acknowledged" : true, "insertedIds" : [ 1, 2, 3, 4 ] }

db.movies.updateOne(
   {"title" : "Macbeth"},
   {$set : {"year" : 2015}}
)
```

The second argument is a document that specifies a new field of **`year`** and its value.

Modifying More Than One Field

```
db.movies.updateOne(
  {"title" : "Macbeth"},
  {$set : {"type" : "movie", "num_mflix_comments" : 1}}
)

> db.movies.find({"title" : "Macbeth"}).pretty()

db.movies.updateOne(
  {"title" : "Macbeth"},
  {$set : {"year" : 2015, "year" : 2015, "year" : 2016, "year" : 2017}}
)
```

when the same field is provided multiple times, the update happens from left to right.

Multiple Documents Matching a Condition

If the given query condition matches more than one document, only the first document will be modified

```
        db.movies.updateOne(
          {"type" : "movie"},
          {$set : {"flag" : "modified"}}
        )
        { "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Upsert with updateOne()

`updateOne()` also supports upserts with an additional flag in the command.

```
        db.movies.updateOne(
          {"title" : "Sicario"},
          {$set : {"year" : 2015}},
          {"upsert" : true}
)
```

Updating a Document with findOneAndUpdate()

`findOneAndUpdate()` function, which is capable of doing everything that **`updateOne()`** does with a few additional features

Returning a New Document in Response

```
db.movies.findOneAndUpdate(
  {"title" : "Macbeth"},
  {$set : {"num_mflix_comments" : 15}},
  {"returnNewDocument" : true}
)
```

by setting the flag `returnNewDocument` to `true`, the response shows the modified document, which also confirms that the count of comments has been modified correctly.

```
db.movies.findOneAndUpdate(
  {"title" : "Macbeth"},
  {$set : {"num_mflix_comments" : 20}},
  {
    "projection" : {"_id" : 0, "num_mflix_comments" : 1},
    "returnNewDocument" : true
  }
)
```

Sorting to Find a Document

```
db.movies.findOneAndUpdate(
  {"type" : "movie"},
  {$set : {"latest" : true}},
  {
    "returnNewDocument" : true,
    "sort" : {"_id" : -1}
  }
)
```

Updating Multiple Documents with updateMany()

```
db.movies.updateMany(
  {"year" : 2015},
  {$set : {"languages" : ["English"]}}
)
```

Update Operators

Increment ($inc)

```
db.movies.findOneAndUpdate(
  {"title" : "Macbeth"},
  {$inc : {"num_mflix_comments" : 3, "rating" : 1.5}},
  {returnNewDocument : true}
)

db.movies.findOneAndUpdate(
  {"title" : "Macbeth"},
  {$inc : {"num_mflix_comments" : -2, "rating" : -0.2}},
  {returnNewDocument : true}
)
```

Multiply ($mul)

```
db.movies.findOneAndUpdate(
  {"title" : "Macbeth"},
  {$mul : {"rating" : 2}},
  {returnNewDocument : true}
)

db.movies.findOneAndUpdate(
  {"title" : "Macbeth"},
  {$mul : {"box_office_collection" : 16.3}},
  {returnNewDocument : true}
)
```

This update operation multiplies a nonexistent field **`box_office_collection`** by a given value:

Rename ($rename)

# First insert a new field "**imdb_rating**"

```
> db.movies.findOneAndUpdate(
  {"title" : "Macbeth"},
  {$set : {"imdb_rating" : 6.6}},
  {returnNewDocument : true}
)
{
  "_id" : 1,
  "title" : "Macbeth",
  "year" : 2017,
  "type" : "movie",
  "num_mflix_comments" : 21,
  "flag" : "modified",
  "rating" : 2.6,
  "box_office_collection" : 0,
  "imdb_rating" : 6.6
}

db.movies.findOneAndUpdate(
  {"title" : "Macbeth"},
  {$rename : {"num_mflix_comments" : "comments", "imdb_rating" : "rating"}},
  {returnNewDocument : true}
)
```

Using this operator, a field can also be moved to and from nested documents.

```
db.movies.findOneAndUpdate(
  {"title" : "Macbeth"},
  {$rename : {"rating" : "imdb.rating"}},
  {returnNewDocument : true}
)
```

Current Date ($currentDate)

#The operator $currentDate is used to set the value of a given field as the current date or timestamp
# Providing a field name with a value of true will insert the current date as a **Date**. Alternatively, a $type operator can be used to explicitly specify the value as a **date** or **timestamp**

```
db.movies.findOneAndUpdate(
  {"title" : "Macbeth"},
  {$currentDate : {
    "created_date" : true,
       "last_updated.date" : {$type : "date"},
       "last_updated.timestamp" : {$type : "timestamp"},
  }},
  {returnNewDocument : true}
)
```

The preceding **`findOneAndUpdate`** operation sets three fields using the **`$currentDate`** operator. The field **`created_date`** has a value of true, which defaults to a **`Date`** type. The other two fields use a dot notation and explicit **`$type`** declaration.

Removing Fields ($unset)

```
> db.movies.find({"title" : "Macbeth"}).pretty()
{
  "_id" : 1,
  "title" : "Macbeth",
  "year" : 2017,
  "type" : "movie",
  "flag" : "modified",
  "box_office_collection" : 0,
  "comments" : 21,
  "imdb" : {
    "rating" : 6.6
  },
  "created_date" : ISODate("2020-06-26T01:22:35.457Z"),
  "last_updated" : {
    "date" : ISODate("2020-06-26T01:22:35.457Z"),
  "timestamp" : Timestamp(1593134555, 1)
  }
}

db.movies.findOneAndUpdate(
  {"title" : "Macbeth"},
  {$unset : {
    "created_date" : "",
    "last_updated" : "dummy_value",
    "box_office_collection": 142.2,
    "imdb" : null,
    "flag" : ""
  }},
  {returnNewDocument : true}
)
```

# you are trying to remove multiple fields and providing them with different values, and you will observe that their values have no impact

Setting When Inserted ($setOnInsert)

The operator **`$setOnInsert`** is similar to **`$set`**; however, it only sets the given fields when an insert happens during an **`upsert`** operation. It has no impact when the **`upsert`** operation results in the update of existing documents.

```
db.movies.findOneAndUpdate(
  {"title":"Macbeth"},
  {
    $rename:{"comments":"num_mflix_comments"},
    $setOnInsert:{"created_time":new Date()}
  },
  {
    upsert : true,
    returnNewDocument:true
  }
)
```

The output shows that **`$setOnInsert`** did not change the document, however, the field **`comment`** is now renamed to **`num_mflix_comments`**. Also, the field **`created_time`** is not added because the upsert operation was used to update an existing document.

```
db.movies.findOneAndUpdate(
  {"title":"Spy"},
  {
    $rename:{"comments":"num_mflix_comments"},
    $setOnInsert:{"created_time":new Date()}
  },
  {
    upsert : true,
    returnNewDocument:true
  }
)
```

The only difference between this snippet and the previous one is that this operation finds a movie named **`Spy`**, which is not present in our collection. Because of the upsert, the operation will result in adding a document to the collection.