Janusz Starczewski

Scripting languages in web applications
Data Access in MongoDB

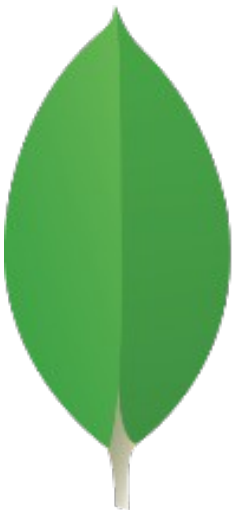
MongoDB

Hu**mongo**us

Document-oriented

Open Source

BSON

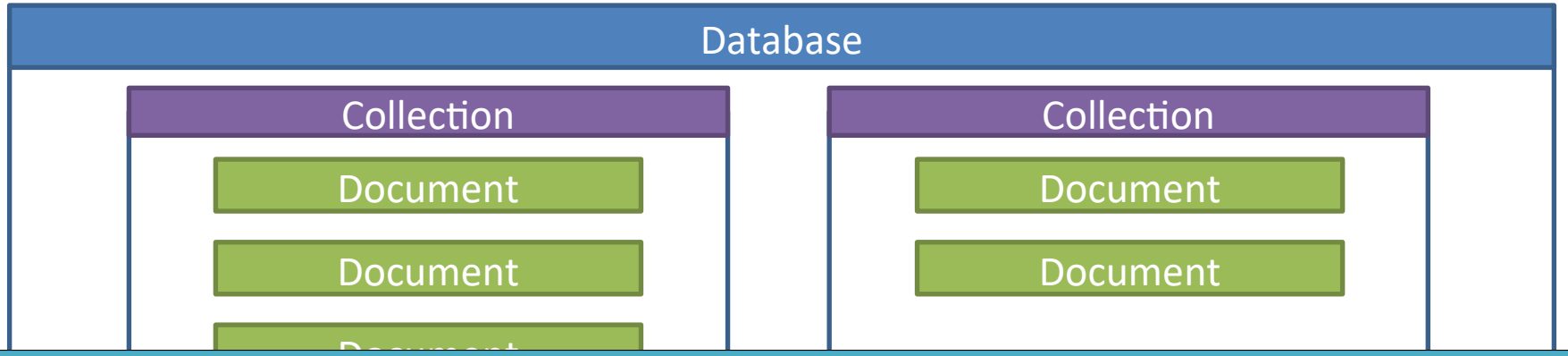


mongoDB®

What is MongoDB?

- MongoDB is
  – Created by 10gen
  (term coined from hu**mongo**us)
  – an open source, document-oriented database designed
  – stores BSON (JSON-like) documents
  – Schema-less

MongoDB Database



RDBMS	MongoDB
Database	Database
Table	Collection
Column	Field
Row	Document
Join	Embedded Document Linking across Document
Foreign Key	**Reference**

ecommercedb.sales

Document


```
{
  "_id": {
    "$oid": "5bd761dcae323e45a93ccfe8"
  },
  "saleDate": {
    "$date": "2015-03-23T21:06:49.506Z"
  },
  "items": [
    "printer paper",
    "notepad",
    "pens",
    "backpack",
    "envelopes",
    "binder"
  ],
  "price": 46.45,
  "storeLocation": "Denver",
  "customer": {
    "gender": "M",
    "age": 42,
    "email": "cauho@witwuta.sv",
    "satisfaction": 4
  },
  "couponUsed": true,
  "purchaseMethod": "Online"
}
```


JSON array

JSON object

MongoDB Shell (mongosh)

The MongoDB Shell, mongosh, is a fully functional JavaScript and Node.js 14.x REPL environment for interacting with MongoDB deployments. You can use the MongoDB Shell to test queries and operations directly with your database.

Create DB

```
db

use myNewDB

db.myCollection.insertOne( { x: 1 } );
```

CRUD create, read, update, delete doc ops

To **insert** document single/multiple, use
`db.collection.insertOne()`
`db.collection.insertMany()`

To **query** documents in a collection, use `db.collection.find()`

To **update** documents single/multiple, use
`db.collection.updateOne()`
`db.collection.updateMany()`
or to replace, use `db.collection.replaceOne()`

To **delete** documents single/multiple, use
`db.collection.deleteOne()`
`db.collection.deleteMany()`

Insert single doc


```
use sample_mflix
db.movies.insertOne(
  {
    title: "The Favourite",
    genres: [ "Drama", "History" ],
    runtime: 121,
    rated: "R",
    year: 2018,
    directors: [ "Yorgos Lanthimos" ],
    cast: [ "Olivia Colman", "Emma Stone", "Rachel Weisz" ],
    type: "movie"
  }
)
```

`insertOne()` returns a document that includes the newly inserted document's _id field value.

To retrieve the inserted document, read the collection:

```
db.inventory.find( { title: "The Favourite" } )
```

To ensure you return the document you inserted, you can instead query by _id.

Insert multiple doc


```
use sample_mflix db.movies.insertMany([
  {
    title: "Jurassic World: Fallen Kingdom",
    genres: [ "Action", "Sci-Fi" ],
    runtime: 130,
    rated: "PG-13",
    year: 2018,
    directors: [ "J. A. Bayona" ],
    cast: [ "Chris Pratt", "Bryce Dallas Howard", "Rafe Spall" ],
    type: "movie"
  },
  {
    title: "Tag",
    genres: [ "Comedy", "Action" ],
    runtime: 105,
    rated: "R",
    year: 2018,
    directors: [ "Jeff Tomsic" ],
    cast: [ "Annabelle Wallis", "Jeremy Renner", "Jon Hamm" ],
    type: "movie"
  }
])
```

To read documents in the collection: `db.movies.find( {} )`

Read all docs

```
use sample_mflix

db.movies.find()
```

This operation is equivalent to the following SQL statement:

```
SELECT * FROM movies
```

Read doc with equality cond


```
use sample_mflix
db.movies.find( { "title": "Titanic" } )
```

This operation corresponds to the following SQL statement:

```
SELECT * FROM movies WHERE title = "Titanic"
```

Read doc using ops

```
{ <field1>: { <operator1>: <value1> }, ... }
```


```
use sample_mflix

db.movies.find( {rated: {$in: ["PG", "PG-13"] }})
```

This operation corresponds to the following SQL statement:

```
SELECT * FROM movies WHERE rated in ("PG", "PG-13")
```

Read doc using LOGICAL ops


```
use sample_mflix

db.movies.find( {
 year: 2010,
 $or: [{"awards.wins": {$gte: 5}}, {genres:"Drama"}]
})

db.movies.find({
   countries: "Mexico", "imdb.rating": {$gte: 7}
})
```

use QUOTATION mark
in DOT notation

Read EMBEDDED docs

querry filter embedded doc:

```
{ size: { h: 14, w: 21, uom: "cm" } }
```

equality match:

```
{ "size.uom": "in" }
```

query op:

```
{ "size.h": { $lt: 15 } }
```

AND cond:

```
{ "size.h": { $lt: 15 }, "size.uom": "in",
status: "D" }
```

Update ops


```
<update_op>: { <field1>: <value1>, … },
```

`$currentDate`	Sets the value of a field to current date, either as a Date or a Timestamp.
`$inc`	Increments the value of the field by the specified amount.
`$min`	Only updates the field if the specified value is less than the existing field value.
`$max`	Only updates the field if the specified value is greater than the existing field value.
`$mul`	Multiplies the value of the field by the specified amount.
`$rename`	Renames a field.
`$set`	Sets the value of a field in a document.
`$setOnInsert`	Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.
`$unset`	Removes the specified field from a document.

Update single doc


```
use sample_mflix
db.movies.updateOne( { title: "Tag" },
{
  $set: {
    plot: "One month every year, five highly competitive friends
           hit the ground running for a no-holds-barred game of
  tag"
  }
  { $currentDate: { lastUpdated: true } }
})
```

Uses the **`$set`** operator to update the value of the plot field for the movie Tag.

Uses the **`$currentDate`** operator to update the value of the `lastUpdated` field to the current date. If lastUpdated field does not exist, `$currentDate` will create the field.

Update multiple docs


```
use sample_airbnb

db.listingsAndReviews.updateMany(
   { security_deposit: { $lt: 100 } },
   {
     $set: { security_deposit: 100, minimum_nights: 1 }
   }
```

The update operation uses the `$set` operator to update the value of the `security_deposit` field to 100 and the value of the `minimum_nights` field to 1.

Update-REPLACE doc


```
db.accounts.replaceOne(
   { account_id: 371138 },
   { account_id: 893421, limit: 5000, products:
[ "Investment", "Brokerage" ] }
)
```

Run the following command to read your updated document:

```
db.accounts.findOne( { account_id: 893421 } )
```

Delete SINGLE doc


```
use sample_mflix

db.movies.deleteOne( { cast: "Brad Pitt" } )
```

Deletes the first document from
the `sample_mflix.movies` collection where the `cast` array
contains `"Brad Pitt"`:

Delete ALL docs

```
use sample_mflix


db.movies.deleteMany({})
```

Delete MULTIPLE docs


```
use sample_mflix
```

```
db.movies.deleteMany( { title: "Titanic" } )
```

AGGREGATION PIPELINES

Common uses for aggregation include:

- Grouping data by a given expression.
- Calculating results based on multiple fields and storing those results in a new field.
- Filtering data to return a subset that matches a given criteria.
- Sorting data.

Aggregation Pipeline syntax


```
db.<collection>.aggregate([
   {
     <$stage1>
   },
   {
     <$stage2>
   }
   ...
])
```

MySQL vs MongoDB Aggregation

MySQL	MongoDB
WHERE	$match
GROUP BY	$group
ORDER BY	$sort
SELECT expressions FROM..	$project
LIMIT	**$limit**

- Example..

```
SELECT purchaseMethod, sum(price) as
totalprice
FROM  salesDB.sales
WHERE couponUsed = FALSE
GROUP BY purchaseMethod
ORDER BY sum(price) ASC
LIMIT 2;
```

```
db.sales.aggregate(
[   { $match: { couponUsed: FALSE }},
    { $group: { _id: "$purchaseMethod",
     totalprice: { $sum: "$price" }}},
    { $sort: { totalprice: 1 }},
    { $limit: 2 }
])
```

Export Aggregate

Click Export To Language to see the code.

```
db.sales.aggregate{
            [{ $project: {items: 1,storeLocation:1,customer:1}},
             { $match: {storeLocation: "Seattle"}}]
              }
```

Aggregation Pipeline exmpl by Atlas sample dataset

suppress the _id field from the documents


```
db.movies.aggregate([
  // First Stage
  { $project: { _id: 0, genres: 1, imdb: 1, title: 1 } },
  // Second Stage: $unwind deconstructs an array field from the input
documents to output a document for each element.
  { $unwind: "$genres" },
  // Third Stage: $group groups input documents by the specified _id
expression
  { $group:
    { _id: "$genres",
      averageGenreRating: { $avg: "$imdb.rating" }
    }
  },
  // Fourth Stage: $sort sorts pipeline in descending order „-1”
  { $sort: { averageGenreRating: -1 } }
] )
```

Return the Specified Fields

- Select $project stage and copy the following expression into the panel:

  { items: 1, storeLocation: 1, customer: 1 }

- The operation corresponds to the following SQL statement:

```
SELECT items, storeLocation, customer
FROM salesdb.sales;
```

The _id field is, by default, included in the output documents.
You can remove by setting the field to 0.

Ref: https://docs.atlas.mongodb.com/data-explorer/cloud-agg-pipeline

Specify Equality Condition

- $match stage filters the documents to pass only the documents that match the specified condition(s)
- Syntax :
  $match: { <field1>: <value1>, ... }
- The below code corresponds to the following SQL statement:

```
{ storeLocation: "Seattle"}
```

```
SELECT items, storeLocation, customer
FROM salesdb.sales WHERE storeLocation = “Seattle”;
```

$match work as WHERE statement in a SQL query.

Specify Conditions Using Operators

- Syntax :

Name	Description	Name	Description
$eq	equal to	$lt	less than
$gt	greater than	$lte	less than or equal to
$gte	greater than or equal to	$ne	not equal to

For example,

corresponds to the following SQL statement:

```
SELECT * FROM MIS2502.sales
                WHERE price > 200;
```

Does order matter?

- What happens when we type in the following code? What happens when we switch the order of $project and $match?

```
$ project:   { items: 1, storeLocation: 1, customer: 1   }
$ match:    {  price:  {  $gt: 200  }  }
```

Unlike SQL, in the aggregation pipeline, each stage transforms the documents as they pass through the pipeline. Therefore, order matters!!!

AND and OR Conditions

- Syntax :

$match: { $and(or) : [{ <condition1> }, { <condition2> } , …}] }

$and and $or operator performs a logical operation on an array of two or more <conditions>.

```
$match: { $and: [
                  { price: { $gt: 180} } ,
                  { storeLocation: "Seattle"}
                  ]}
```

The operation corresponds to the following SQL statement:

```
SELECT * FROM salesDB.sales
WHERE price > 180 AND storeLocation = “Seattle”;
```

Sort Results

Using $sort stage, we can specify the sort order of the returned documents.

```
$sort:  { price: 1 }
```

The operation corresponds to the following SQL statement:

```
SELECT * FROM salesDB.sales ORDER By price ASC;
```

We can set the field to 1 (-1), to specify ascending (descending) order for a field,.

Limit Results

Using $limit stage, we can specify the number of the returned documents.

$limit: 3

The operation corresponds to the following SQL statement:

```
SELECT * FROM salesDB.sales LIMIT 3;
```

So, what may happen if we put $limit on the first stage?

Aggregation

$group documents by some specified expression and outputs to the next stage a document for each distinct grouping

- Syntax :

```
$group: { _id: <expression>, <field1>: { <accumulator1> : <expression1> }, ... }
```

```
$group:  { _id: "$purchaseMethod",
                totalprice: { $sum: "$price"}  }
```

corresponds to the following SQL statement:

```
SELECT purchaseMethod, sum(price) as totalprice
FROM salesDB.sales
Group by purchaseMethod;
```

Aggregation

Similar to GROUP BY in SQL, the output documents can contain computed filed

```
$group:  { _id: "$purchaseMethod",
           totalprice: { $sum: "$price"},
           avgprice: { $avg: "$price"},
           maxprice: { $max: "$price"},
           number: { $sum:1} }
```

corresponds to the following SQL statement:

```
SELECT purchaseMethod, sum(price) as totalprice,
avg(price) as avgprice, max(price) as maxprice, count(price)
as number
FROM salesDB.sales
```

In $group, we use **$sum: 1** to count the number of documents.

Group by null

Grouping _id with null will calculate the total price and the average quantity as well as counts for all documents in the collection.

```
$group:  { _id: null,
           totalprice: { $sum: "$price"},
           avgprice: { $avg: "$price"},
           maxprice: { $max: "$price"},
           number: { $sum:1} }
```

corresponds to the following SQL statement:

```
SELECT sum(price) as totalprice, avg(price) as avgprice,
max(price) as maxprice, count(price) as number
FROM salesDB.sales;
```

Summary

- Given a semi structured database, we now should be able to create a NoSQL statement to answer a question

- Understand how each stage in aggregation tap works and the relationship with SQL keywords
  - $project
  - $match
  - $sort
  - $limit
  - $group