**MongoDB Hands-On**

Records in a MongoDB database are called documents, and the field values may include numbers, strings, booleans, arrays, or even nested documents.

Example Document

{

title: "Post Title 1",

body: "Body of post.",

category: "News",

likes: 1,

tags: ["news", "events"],

date: Date()

}

**Change or Create a Database**

You can change or create a new database by typing use then the name of the database.

Example : Create a new database called "blog":

**use blog**

We are now in the blog database.

**Create Collection**

There are 2 ways to create a collection.

**Method 1**

You can create a collection using the createCollection() database method.

Example

**db.createCollection("posts")**

**Method 2**

You can also create a collection during the insert process.

Example :We are here assuming object is a valid JavaScript object containing post data:

**db.posts.insertOne(object)**

This will create the "posts" collection if it does not already exist.

**Insert Documents**

There are 2 methods to insert documents into a MongoDB database.

**insertOne()**

To insert a single document, use the insertOne() method.

This method inserts a single object into the database.

Example

db.posts.insertOne({

title: "Post Title 1",

body: "Body of post.",

category: "News",

likes: 1,

tags: ["news", "events"],

date: Date()

})

**insertMany()**

To insert multiple documents at once, use the insertMany() method.

This method inserts an array of objects into the database.

Example

db.posts.insertMany([

{

title: "Post Title 2",

body: "Body of post.",

category: "Event",

likes: 2,

tags: ["news", "events"],

date: Date()

},

{

title: "Post Title 3",

body: "Body of post.",

category: "Technology",

likes: 3,

tags: ["news", "events"],

date: Date()

},

{

title: "Post Title 4",

body: "Body of post.",

category: "Event",

likes: 4,

tags: ["news", "events"],

date: Date()

}

])

**MongoDB  Find**

**Find Data**

There are 2 methods to find and select data from a MongoDB collection, find() and findOne().

**find()**

To select data from a collection in MongoDB, we can use the find() method.

This method accepts a query object. If left empty, all documents will be returned.

### Example

db.posts.find()

**findOne()**

To select only one document, we can use the findOne() method.

This method accepts a query object. If left empty, it will return the first document it finds.

### Example

db.posts.findOne()

**Querying Data**

To query, or filter, data we can include a query in our find() or findOne() methods.

### Example

db.posts.find( {category: "News"} )

**Projection**

Both find methods accept a second parameter called projection.

This parameter is an object that describes which fields to include in the results.

### Example

This example will only display the title and date fields in the results.

db.posts.find({}, {title: 1, date: 1})

We use a 1 to include a field and 0 to exclude a field.

### Example

This time, let's exclude the \_id field.

db.posts.find({}, {\_id: 0, title: 1, date: 1})

Let's exclude the category field. All other fields will be included in the results.

### Example

db.posts.find({}, {category: 0})

We will get an error if we try to specify both 0 and 1 in the same object.

### Example

db.posts.find({}, {title: 1, date: 0})

**MongoDB Update**

**Update Document**

To update an existing document we can use the updateOne() or updateMany() methods.

The first parameter is a query object to define which document or documents should be updated.

The second parameter is an object defining the updated data.

**updateOne()**

The updateOne() method will update the first document that is found matching the provided query.

### Example

db.posts.find( { title: "Post Title 1" } )

Now let's update the "likes" on this post to 2. To do this, we need to use the $set operator.

### Example

db.posts.updateOne( { title: "Post Title 1" }, { $set: { likes: 2 } } )

Check the document again and you'll see that the "like" have been updated.

### Example

db.posts.find( { title: "Post Title 1" } )

## Insert if not found

If you would like to insert the document if it is not found, you can use the upsert option.

### Example

Update the document, but if not found insert it:

db.posts.updateOne(

{ title: "Post Title 5" },

{

$set:

{

title: "Post Title 5",

body: "Body of post.",

category: "Event",

likes: 5,

tags: ["news", "events"],

date: Date()

}

},

{ upsert: true }

)

## updateMany()

The updateMany() method will update all documents that match the provided query.

### Example

Update likes on all documents by 1. For this we will use the $inc (increment) operator:

db.posts.updateMany({}, { $inc: { likes: 1 } })

Now check the likes in all of the documents and you will see that they have all been incremented by 1.

**MongoDB Delete**

## Delete Documents

We can delete documents by using the methods deleteOne() or deleteMany().

These methods accept a query object. The matching documents will be deleted.

## deleteOne()

The deleteOne() method will delete the first document that matches the query provided.

### Example

db.posts.deleteOne({ title: "Post Title 5" })

## deleteMany()

The deleteMany() method will delete all documents that match the query provided.

### Example

db.posts.deleteMany({ category: "Technology" })

**MongoDB Query Operators**

## MongoDB Query Operators

There are many query operators that can be used to compare and reference document fields.

### Comparison

The following operators can be used in queries to compare values:

* $eq: Values are equal
* $ne: Values are not equal
* $gt: Value is greater than another value
* $gte: Value is greater than or equal to another value
* $lt: Value is less than another value
* $lte: Value is less than or equal to another value
* $in: Value is matched within an array

### Logical

The following operators can logically compare multiple queries.

* $and: Returns documents where both queries match
* $or: Returns documents where either query matches
* $nor: Returns documents where both queries fail to match
* $not: Returns documents where the query does not match

**MongoDB Update Operators**

## MongoDB Update Operators

There are many update operators that can be used during document updates.

### Fields

The following operators can be used to update fields:

* $currentDate: Sets the field value to the current date
* $inc: Increments the field value
* $rename: Renames the field
* $set: Sets the value of a field
* $unset: Removes the field from the document

**MongoDB Aggregation Pipelines**

## Aggregation Pipelines

Aggregation operations allow you to group, sort, perform calculations, analyze data, and much more.

Aggregation pipelines can have one or more "stages". The order of these stages are important. Each stage acts upon the results of the previous stage.

### Example

db.posts.aggregate([

// Stage 1: Only find documents that have more than 1 like

{

$match: { likes: { $gt: 1 } }

},

// Stage 2: Group documents by category and sum each categories likes

{

$group: { \_id: "$category", totalLikes: { $sum: "$likes" } }

}

])

## Sample Data

To demonstrate the use of stages in a aggregation pipeline, we will load sample data into our database.

From the MongoDB Atlas dashboard, go to Databases. Click the ellipsis and select "Load Sample Dataset". This will load several sample datasets into your database.

In the next sections we will explore several aggregation pipeline stages in more detail using this sample data.

# **MongoDB Aggregation $group**

## Aggregation $group

This aggregation stage groups documents by the unique \_id expression provided.

### Example

In this example, we are using the "sample\_airbnb" database loaded from our sample data in the [Intro to Aggregations](https://www.w3schools.com/mongodb/mongodb_aggregations_intro.php) section.

db.listingsAndReviews.aggregate(

[ { $group : { \_id : "$property\_type" } } ]

)

This will return the distinct values from the property\_type field.

# **MongoDB Aggregation $limit**

## Aggregation $limit

This aggregation stage limits the number of documents passed to the next stage.

### Example

In this example, we are using the "sample\_mflix" database loaded from our sample data in the [Intro to Aggregations](https://www.w3schools.com/mongodb/mongodb_aggregations_intro.php) section.

db.movies.aggregate([ { $limit: 1 } ])

This will return the 1 movie from the collection.

# **MongoDB Aggregation $project**

## Aggregation $project

This aggregation stage passes only the specified fields along to the next aggregation stage.

This is the same projection that is used with the find() method.

### Example

In this example, we are using the "sample\_restaurants" database loaded from our sample data in the [Intro to Aggregations](https://www.w3schools.com/mongodb/mongodb_aggregations_intro.php) section.

db.restaurants.aggregate([

{

$project: {

"name": 1,

"cuisine": 1,

"address": 1

}

},

{

$limit: 5

}

])

This will return the documents but only include the specified fields.

Notice that the \_id field is also included. This field is always included unless specifically excluded.

We use a 1 to include a field and 0 to exclude a field.

# **MongoDB Aggregation $sort**

## Aggregation $sort

This aggregation stage groups sorts all documents in the specified sort order.

Remember that the order of your stages matters. Each stage only acts upon the documents that previous stages provide.

### Example

In this example, we are using the "sample\_airbnb" database loaded from our sample data in the [Intro to Aggregations](https://www.w3schools.com/mongodb/mongodb_aggregations_intro.php) section.

db.listingsAndReviews.aggregate([

{

$sort: { "accommodates": -1 }

},

{

$project: {

"name": 1,

"accommodates": 1

}

},

{

$limit: 5

}

])

This will return the documents sorted in descending order by the accommodates field.

The sort order can be chosen by using 1 or -1. 1 is ascending and -1 is descending.

# **ongoDB Aggregation $match**

## Aggregation $match

This aggregation stage behaves like a find. It will filter documents that match the query provided.

Using $match early in the pipeline can improve performance since it limits the number of documents the next stages must process.

### Example

In this example, we are using the "sample\_airbnb" database loaded from our sample data in the [Intro to Aggregations](https://www.w3schools.com/mongodb/mongodb_aggregations_intro.php) section.

db.listingsAndReviews.aggregate([

{ $match : { property\_type : "House" } },

{ $limit: 2 },

{ $project: {

"name": 1,

"bedrooms": 1,

"price": 1

}}

])

This will only return documents that have the property\_type of "House".

# **MongoDB Aggregation $addFields**

## Aggregation $addFields

This aggregation stage adds new fields to documents.

### Example

In this example, we are using the "sample\_restaurants" database loaded from our sample data in the [Intro to Aggregations](https://www.w3schools.com/mongodb/mongodb_aggregations_intro.php) section.

db.restaurants.aggregate([

{

$addFields: {

avgGrade: { $avg: "$grades.score" }

}

},

{

$project: {

"name": 1,

"avgGrade": 1

}

},

{

$limit: 5

}

])

This will return the documents along with a new field, avgGrade, which will contain the average of each restaurants grades.score.

# **MongoDB Aggregation $count**

## Aggregation $count

This aggregation stage counts the total amount of documents passed from the previous stage.

### Example

In this example, we are using the "sample\_restaurants" database loaded from our sample data in the [Intro to Aggregations](https://www.w3schools.com/mongodb/mongodb_aggregations_intro.php) section.

db.restaurants.aggregate([

{

$match: { "cuisine": "Chinese" }

},

{

$count: "totalChinese"

}

])

This will return the number of documents at the $count stage as a field called "totalChinese".

# **MongoDB Aggregation $lookup**

## Aggregation $lookup

This aggregation stage performs a left outer join to a collection in the same database.

There are four required fields:

* from: The collection to use for lookup in the same database
* localField: The field in the primary collection that can be used as a unique identifier in the from collection.
* foreignField: The field in the from collection that can be used as a unique identifier in the primary collection.
* as: The name of the new field that will contain the matching documents from the from collection.

### Example

In this example, we are using the "sample\_mflix" database loaded from our sample data in the [Intro to Aggregations](https://www.w3schools.com/mongodb/mongodb_aggregations_intro.php) section.

db.comments.aggregate([

{

$lookup: {

from: "movies",

localField: "movie\_id",

foreignField: "\_id",

as: "movie\_details",

},

},

{

$limit: 1

}

])

This will return the movie data along with each comment.

# **MongoDB Aggregation $out**

## Aggregation $out

This aggregation stage writes the returned documents from the aggregation pipeline to a collection.

The $out stage must be the last stage of the aggregation pipeline.

### Example

In this example, we are using the "sample\_airbnb" database loaded from our sample data in the [Intro to Aggregations](https://www.w3schools.com/mongodb/mongodb_aggregations_intro.php) section.

db.listingsAndReviews.aggregate([

{

$group: {

\_id: "$property\_type",

properties: {

$push: {

name: "$name",

accommodates: "$accommodates",

price: "$price",

},

},

},

},

{ $out: "properties\_by\_type" },

])

The first stage will group properties by the property\_type and include the name, accommodates, and price fields for each. The $out stage will create a new collection called properties\_by\_type in the current database and write the resulting documents into that collection.

**Indexing & Search**

## Indexing & Search

MongoDB Atlas comes with a full-text search engine that can be used to search for documents in a collection.

[Atlas Search](https://www.mongodb.com/docs/atlas/atlas-search?utm_campaign=w3schools_mdb&utm_source=w3schools&utm_medium=referral) is powered by Apache Lucene.

## Creating an Index

We'll use the Atlas dashboard to create an index on the "sample\_mflix" database from the sample data that we loaded in the [Intro to Aggregations](https://www.w3schools.com/mongodb/mongodb_aggregations_intro.php) section.

1. From the Atlas dashboard, click on your **Cluster name** then the **Search** tab.
2. Click on the **Create Search Index** button.
3. Use the **Visual Editor** and click Next.
4. Name your index, choose the Database and Collection you want to index and click Next.
   * If you name your index "default" you will not have to specify the index name in the $search pipeline stage.
   * Choose the sample\_mflix database and the movies collection.
5. Click **Create Search Index** and wait for the index to complete.

## Running a Query

To use our search index, we will use the $search operator in our aggregation pipeline.

### Example

db.movies.aggregate([

{

$search: {

index: "default", // optional unless you named your index something other than "default"

text: {

query: "star wars",

path: "title"

},

},

},

{

$project: {

title: 1,

year: 1,

}

}

])

The first stage of this aggregation pipeline will return all documents in the movies collection that contain the word "star" or "wars" in the title field.

The second stage will project the title and year fields from each document.

**MongoDB Schema Validation**

## Schema Validation

By default MongoDB has a flexible schema. This means that there is no strict schema validation set up initially.

Schema validation rules can be created in order to ensure that all documents a collection share a similar structure.

## Schema Validation

MongoDB supports [JSON Schema](http://json-schema.org/) validation. The $jsonSchema operator allows us to define our document structure.

### Example

db.createCollection("posts", {

validator: {

$jsonSchema: {

bsonType: "object",

required: [ "title", "body" ],

properties: {

title: {

bsonType: "string",

description: "Title of post - Required."

},

body: {

bsonType: "string",

description: "Body of post - Required."

},

category: {

bsonType: "string",

description: "Category of post - Optional."

},

likes: {

bsonType: "int",

description: "Post like count. Must be an integer - Optional."

},

tags: {

bsonType: ["string"],

description: "Must be an array of strings - Optional."

},

date: {

bsonType: "date",

description: "Must be a date - Optional."

}

}

}

}

})

This will create the posts collection in the current database and specify the JSON Schema validation requirements for the collection.