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

Evaluation

The following operators assist in evaluating documents.

$regex: Allows the use of regular expressions when evaluating field values

$text: Performs a text search

$where: Uses a JavaScript expression to match documents

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

Array

The following operators assist with updating arrays.

$addToSet: Adds distinct elements to an array

$pop: Removes the first or last element of an array

$pull: Removes all elements from an array that match the query

$push: Adds an element to an array

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.

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 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 section.

From the Atlas dashboard, click on your Cluster name then the Search tab.

Click on the Create Search Index button.

Use the Visual Editor and click Next.

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.

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 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.

MongoDB Data API

MongoDB Data API

The MongoDB Data API can be used to query and update data in a MongoDB database without the need for language specific drivers.

Language drivers should be used when possible, but the MongoDB Data API comes in handy when drivers are not available or drivers are overkill for the application.

Read & Write with the MongoDB Data API

The MongoDB Data API is a pre-configured set of HTTPS endpoints that can be used to read and write data to a MongoDB Atlas database.

With the MongoDB Data API, you can create, read, update, delete, or aggregate documents in a MongoDB Atlas database.

Cluster Configuration

In order to use the Data API, you must first enable the functionality from the Atlas UI.

From the MongoDB Atlas dashboard, navigate to Data API in the left menu.

Select the data source(s) you would like to enable the API on and click Enable the Data API.

Access Level

By default, no access is granted. Select the access level you'd like to grant the Data API. The choices are: No Access, Read Only, Read and Write, or Custom Access.

Data API Key

In order to authenticate with the Data API, you must first create a Data API key.

Click Create API Key, enter a name for the key, then click Generate API Key.

Be sure to copy the API key and save it somewhere safe. You will not get another chance to see this key again.

Sending a Data API Request

We can now use the Data API to send a request to the database.

In the next example, we'll use curl to find the first document in the movies collection of our sample_mflix database. We loaded this sample data in the Intro to Aggregations section.

To run this example, you'll need your App Id, API Key, and Cluster name.

You can find your App Id in the URL Endpoint field of the Data API page in the MongoDB Atlas UI.

Example

```
curl --location --request POST 'https://data.mongodb-api.com/app/<DATA API APP ID>/endpoint/data/v1/action/findOne' \
--header 'Content-Type: application/json' \
--header 'Access-Control-Request-Headers: *' \
--header 'api-key: <DATA API KEY>' \
--data-raw '{
   "dataSource":"<CLUSTER NAME>",
   "database":"sample_mflix",
   "collection":"movies",
   "projection": {"title": 1}
}'
```

Data API Endpoints

In the previous example, we used the findOne endpoint in our URL.

There are several endpoints available for use with the Data API.

All endpoints start with the Base URL: https://data.mongodb-api.com/app/<Data API App ID>/endpoint/data/v1/action/

Find a Single Document

Endpoint

POST Base_URL/findOne

The findOne endpoint is used to find a single document in a collection.

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "filter": <query filter>,
  "projection": <projection>
}
```

Find Multiple Documents

Endpoint

POST Base_URL/find

The find endpoint is used to find multiple documents in a collection.

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "filter": <query filter>,
  "projection": <projection>,
  "sort": <sort expression>,
  "limit": <number>,
  "skip": <number>
}
```

Insert a Single Document

Endpoint

POST Base_URL/insertOne

The insertOne endpoint is used to insert a single document into a collection.

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "document": <document>
}
```

Insert Multiple Documents

Endpoint

POST Base_URL/insertMany

The insertMany endpoint is used to insert multiple documents into a collection.

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "documents": [<document>, <document>, ...]
}
```

Update a Single Document

Endpoint

POST Base_URL/updateOne

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "filter": <query filter>,
  "update": <update expression>,
  "upsert": true|false
}
```

Update Multiple Documents

Endpoint

POST Base_URL/updateMany

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "filter": <query filter>,
  "update": <update expression>,
  "upsert": true|false
}
```

Delete a Single Document

Endpoint

POST Base_URL/deleteOne

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "filter": <query filter>
}
```

Delete Multiple Documents

Endpoint

POST Base_URL/deleteMany

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "filter": <query filter>
}
```

Aggregate Documents

Endpoint

POST Base_URL/aggregate

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "pipeline": [<pipeline expression>, ...]
}
```

MongoDB Node.js Database Interaction

Node.js Database Interaction

For this tutorial, we will use a MongoDB Atlas database. If you don't already have a MongoDB Atlas account, you can create one for free at MongoDB Atlas.

We will also use the "sample_mflix" database loaded from our sample data in the Intro to Aggregations section.

MongoDB Node.js Driver Installation

To use MongoDB with Node.js, you will need to install the mongodb package in your Node.js project.

Use the following command in your terminal to install the mongodb package:

```
npm install mongodb
```

We can now use this package to connect to a MongoDB database.

Create an index.js file in your project directory.

index.js

```
const { MongoClient } = require('mongodb');
```

Connection String

In order to connect to our MongoDB Atlas database, we'll need to get our connection string from the Atlas dashboard.

Go to Database then click the CONNECT button on your Cluster.

Choose Connect your application then copy your connection string.

Example:
mongodb+srv://<username>:<password>@<cluster.string>.mongodb.net/myFirstDatabase?retryWrites=true&w=majority

You will need to replace the <username>, <password>, and <cluster.string> with your MongoDB Atlas username, password, and cluster string.

Connecting to MongoDB

Let's add to our index.js file.

index.js

```
const { MongoClient } = require('mongodb');

const uri = "<Your Connection String>";
const client = new MongoClient(uri);

async function run() {
  try {
    await client.connect();
    const db = client.db('sample_mflix');
    const collection = db.collection('movies');

    // Find the first document in the collection
    const first = await collection.findOne();
    console.log(first);
  } finally {
    // Close the database connection when finished or an error occurs
    await client.close();
  }
}
run().catch(console.error);
```

Run this file in your terminal.

```
node index.js
```

You should see the first document logged to the console.

CRUD & Document Aggregation

Just as we did using mongosh, we can use the MongoDB Node.js language driver to create, read, update, delete, and aggregate documents in the database.

Expanding on the previous example, we can replace the collection.findOne() with find(), insertOne(), insertMany(), updateOne(), updateMany(), deleteOne(), deleteMany(), or aggregate().

Give some of those a try.

MongoDB Charts

MongoDB Charts lets you visualize your data in a simple, intuitive way.

MongoDB Charts Setup

From the MongoDB Atlas dashboard, go to the Charts tab.

If you've never used Charts before, click the Activate Now button. This will take about 1 minute to complete.

You'll see a new dashboard. Click the dashboard name to open it.

Creating a Chart

Create a new chart by clicking the Add Chart button.

Visually creating a chart is intuitive. Select the data sources that you want to use.

Example:

In this example, we are using the "sample_mflix" database loaded from our sample data in the Intro to Aggregations section.

Under Data Source, select the Movies collection.

Let's visualize how many movies were released in each year.

Drag the Year field to the Y Axis field and set the Bin Size to 1.

Drag the _id field to the X Axis field and make sure COUNT is selected for the Aggregate.

You should now see a bar chart with the number of movies released in each year.