TD 1: Data Management, Aggregations, and Indexing with MongoDB

Sidi Biha

2024-2025

1 Introduction

This document provides a comprehensive tutorial on using MongoDB to manage data. The focus is on:

- Importing data from both CSV and JSON files.
- Running queries including filtering, projection, sorting, and aggregations.
- Demonstrating advanced aggregation stages like `$unwind` and `$group`.
- Showcasing the importance of using indexes for performance.

We will use two publicly available datasets:

1. **BAC Results CSV File:** `https://github.com/binorassocies/rimdata/blob/main/data/bac_results_2015-2024.csv`
2. **NASA Meteorite Landings JSON Dataset:** `https://data.nasa.gov/resource/y77d-th95.json`

   This dataset provides an array of meteorite landing objects, including fields such as `name`, `id`, `nametype`, `recclass`, `mass`, `fall`, `year`, `reclat`, and `reclong`.

2 Prerequisites and Setup

2.1 Installing MongoDB

Make sure you have the docker compose env installed and running on your machine.

2.2 Downloading the CSV and JSON Data

- **BAC CSV File:** `https://github.com/binorassocies/rimdata/blob/main/data/bac_results_2015-2024.csv`
- **NASA Meteorite Landings JSON File:** `https://data.nasa.gov/resource/y77d-th95.json`

  You can download this file directly with a tool like `curl`:

```
curl -o meteorite_landings.json https://data.nasa.gov/resource/y77d-th95.
    json
```

Save these files in directories accessible from your mongodb docker container (under the data folder).

3 Importing CSV Data into MongoDB

Use the `mongoimport` command to load the CSV file into a MongoDB collection.

3.1 Example Command for CSV

```
docker exec -it mongodb mongoimport --db bac_db -u admin -p adminXlab2025! --
    authenticationDatabase=admin --collection bac_results --type csv --
    headerline --file /data/db/bac_results_2015-2024.csv
```

Listing 1: Import CSV Data

Explanation:

- `-db bac_db`: Specifies the database name (here, `bac_db`).
- `-collection bac_results`: Specifies the target collection.
- `-type csv`: Indicates that the file is a CSV.
- `-headerline`: Uses the first line of the CSV file as the field names.
- `-file ...`: The path to the CSV file.

After importing, connect to the MongoDB shell:

```
docker exec -it mongodb mongosh -u admin -p adminXlab2025! --
    authenticationDatabase=admin
```

Listing 2: Connecting to MongoDB Shell

Then switch to the database:

```
use bac_db
```

Listing 3: Switch Database

4 Importing the NASA Meteorite Landings JSON Dataset into MongoDB

Once you have downloaded the `meteorite_landings.json` file, you can import it into MongoDB as follows:

4.1 Example Command for JSON

```
docker exec -it mongodb mongoimport --db nasa_db -u admin -p adminXlab2025! --
    authenticationDatabase=admin --collection meteorites --jsonArray --file /
    data/db/meteorite_landings.json
```

Listing 4: Import JSON Data

Key points:

- `-db nasa_db`: Specifies the database name (here, `nasa_db`).
- `-collection meteorites`: Target collection name.
- `-file /path/to/meteorite_landings.json`: Path to the downloaded JSON file.
- `-jsonArray`: Use this because the JSON file is an array of documents.

Connect to the shell and switch to the `nasa_db` database:

```
docker exec -it mongodb mongosh -u admin -p adminXlab2025! --
    authenticationDatabase=admin
use nasa_db
```

Listing 5: Switch Database for Meteorites

5 Basic Querying: Filtering and Projection

After importing your data, you can query it using the `find()` method.

5.1 Filtering Documents (BAC Results)

Assuming the CSV contains a field called `result`, filter records with a certain value:

```
db.bac_results.find({ result: "Admis" })
```

Listing 6: Filtering by result (BAC)

5.2 Projection (BAC Results)

To display only specific fields (e.g., `year` and `result`), use projection:

```
db.bac_results.find({ result: "Admis" }, { year: 1, result: 1, _id: 0 })
```

Listing 7: Projection Example (BAC)

5.3 Filtering and Projection (NASA Meteorite Landings)

Each meteorite object can contain fields such as `name`, `mass`, `fall`, `year`, `reclat`, and `reclong`. For example, to find meteorites that fell (rather than were found) and only show their `name` and `fall` status, you can do:

```
db.meteorites.find(
  { fall: "Fell" },
  { name: 1, fall: 1, _id: 0 }
)
```

Listing 8: Filtering Meteorites by fall status

6 Aggregation Framework (BAC Dataset)

MongoDB's aggregation framework lets you perform complex data processing and analytics. Let's start with the BAC results dataset.

6.1 Basic Aggregation: Grouping and Counting by Year

Assuming the CSV contains a field called `year`, you can count how many records exist for each year:

```
db.bac_results.aggregate([
  {
    $group: {
      _id: "$year",
      count: { $sum: 1 }
    }
  },
  { $sort: { _id: 1 } }
```

```
])
```

Listing 9: Aggregation: Grouping by Year

6.2 Advanced Aggregation Examples

6.2.1 Sorting with Aggregation

This example first filters documents with `result` = "Admis", sorts them by `year` in descending order, and finally projects only the `year` and `result` fields.

```
db.bac_results.aggregate([
  { $match: { result: "Admis" } },
  { $sort: { year: -1 } },
  { $project: { year: 1, result: 1, _id: 0 } }
])
```

Listing 10: Aggregation: Sort, Match, and Project

6.2.2 Advanced Projection

Here, we add a computed field `passed` using the `$cond` operator to indicate whether a student passed (i.e., `result` equals "Admis"):

```
db.bac_results.aggregate([
  {
    $project: {
      year: 1,
      result: 1,
      passed: { $cond: { if: { $eq: [ "$result", "Admis" ] }, then: true, else:
     false } }
    }
  }
])
```

Listing 11: Aggregation: Advanced Projection

6.2.3 Unwind Example

If your documents contain an array field (for example, `subjects`), you can use `$unwind` to deconstruct the array. In this hypothetical example, we assume there is a field `subjects` which is a string of subjects separated by commas. We can first split this string into an array, then unwind and group:

```
db.bac_results.aggregate([
  {
    $project: {
      year: 1,
      subjectsArray: { $split: [ "$subjects", ", " ] }
    }
  },
  { $unwind: "$subjectsArray" },
  {
    $group: {
      _id: "$subjectsArray",
      count: { $sum: 1 }
    }
  }
])
```

Listing 12: Aggregation: Unwind and Group Subjects

6.2.4 Grouping with Multiple Fields and Sorting

This example groups documents by both `year` and `result` to count the occurrences for each combination, and then sorts the results by year (ascending) and by count (descending):

```
db.bac_results.aggregate([
  {
    $group: {
      _id: { year: "$year", result: "$result" },
      count: { $sum: 1 }
    }
  },
  { $sort: { "_id.year": 1, count: -1 } }
])
```

Listing 13: Aggregation: Group by Multiple Fields and Sort

7 Aggregation Framework (NASA Meteorite Landings)

Now, let's explore the meteorites dataset. A typical meteorite document can look like this (simplified example):

```
{
  "name": "Aachen",
  "id": "1",
  "nametype": "Valid",
  "recclass": "L5",
  "mass": "21",
  "fall": "Fell",
  "year": "1880-01-01T00:00:00.000",
  "reclat": "50.775",
  "reclong": "6.08333"
}
```

7.1 Example: Count Meteorites by fall status

We can group by `fall` to see how many meteorites were `Fell` vs. `Found`, and then sort alphabetically by the fall status:

```
db.meteorites.aggregate([
  {
    $group: {
      _id: "$fall",
      totalMeteorites: { $sum: 1 }
    }
  },
  { $sort: { "_id": 1 } }
])
```

Listing 14: Aggregation: Group by fall status

7.2 Example: Aggregating by recclass

Meteorites are classified by `recclass`. We can group by `recclass` to count how many meteorites fall into each classification, sorted in descending order by count:

```
db.meteorites.aggregate([
  {
    $group: {
      _id: "$recclass",
      total: { $sum: 1 }
    }
  },
  { $sort: { total: -1 } },
  { $limit: 10 }
])
```

Listing 15: Aggregation: Group by recclass

7.3 Example: Filtering by Mass Range

If you want to see only meteorites that exceed a certain mass (say 10,000 grams) and group them by `fall` status:

```
db.meteorites.aggregate([
  {
    $match: {
      mass: { $gt: "10000" }
    }
  },
  {
    $group: {
      _id: "$fall",
      totalHeavyMeteorites: { $sum: 1 }
    }
  },
  { $sort: { totalHeavyMeteorites: -1 } }
])
```

Listing 16: Aggregation: Filter by mass and Group by fall

8 Using Indexes in MongoDB

Indexes are essential for enhancing query performance, especially with large datasets.

8.1 Importance of Indexes

- **Performance:** Indexes allow MongoDB to locate data quickly without scanning every document.
- **Efficiency:** They reduce the workload on your database, speeding up read operations.
- **Query Optimization:** The query planner uses indexes to optimize execution paths.

8.2 Creating an Index

If you frequently query the `year` field in the BAC dataset, create an index on it:

```
db.bac_results.createIndex({ year: 1 })
```

Listing 17: Creating an Index on year (BAC)

Similarly, for the meteorites dataset, if you often query by `mass` or `fall`, you can create indexes:

```
db.meteorites.createIndex({ mass: 1 })
```

Listing 18: Creating an Index on mass (Meteorites)

8.3 Checking the Index

Verify the created indexes using:

```
db.bac_results.getIndexes()
db.meteorites.getIndexes()
```

Listing 19: Check Created Indexes

8.4 Impact on Query Performance

After creating the index, inspect the query performance using the `explain()` method:

```
db.bac_results.find({ year: 2020 }).explain("executionStats")
db.meteorites.find({ fall: "Fell" }).explain("executionStats")
```

Listing 20: Using explain() for Query Performance

This shows details on whether the query used the index and how many documents were scanned.

9 Additional MongoDB Features

9.1 Flexible Schema

MongoDB collections are schema-less, meaning that documents in the same collection can have different structures. This flexibility is useful when dealing with evolving data.

9.2 Replication and High Availability

MongoDB supports replica sets, which consist of multiple MongoDB servers providing redundancy and high availability.

9.3 Sharding

For horizontal scaling, MongoDB offers sharding, which distributes data across multiple servers.

9.4 Transactions

MongoDB supports multi-document transactions to ensure data consistency, especially during operations that span multiple documents or collections.