Exercise 1

by

Madeline Zaytsoff

ID: 40177534

Submitted in partial fulfillment of

CART 451

Concordia University

Fall 2023

Reflection: Data Access & Import

The data set I pulled from Kaggle contains information on horror movies released from the 1950s onwards. My familiarity with the genre made it an easier data set to parse for my first solo attempt at accessing MongoDB. I didn't want to spend too much time trying to do something with unfamiliar data sets so I could focus on technical development. As a bonus, I could also use the database to start finding horror movies I hadn't watched.

Importing the data wasn't too difficult as I opted to use the *Compass* app to upload my data. I tried using terminal commands at first, but quickly became overwhelmed trying to visualize everything without a GUI. What I did enjoy a lot about using the *Compass* app was that I could cut out data I wouldn't need as well as define each column of data.

Reflection: MongoDB Queries

Query 1

Not much remarkable about this first attempt other than it was the one I used to test if everything was working. I created a single parameter query to find one specific movie in the dataset I had uploaded to Mongo. What was interesting was the care I needed to take in defining the data set once I uploaded it through the *Compass* app. I hadn't realized I would be needing to filter out junk data I wouldn't be needing. In hindsight, this makes complete sense as not every piece of data in a dataset would be valuable. I just hadn't considered I would have this much control over the datasets I would be accessing. After my data was sorted, I used the findOne() function to locate a single film and access all its information.

Query 2

I wanted to know how to show me films that were above a certain popularity threshold. I started with find() but I kept running into issues with not getting anything back. Switching over to aggregate(), I found success by using an operator ($gte) in my match to only return films whose scores were above 1000. I finalized the query by using $limit to narrow the returned results down to only 2.

Query 3

This query took me the second longest to research. My database featured a section for languages, and I wanted to see how many there were. More than that, I wanted to know what each of these were and how they were spelt. After some time, I came across the distinct() function which would return to me what I needed. Just by selecting "original_language" as the filed to be parsed, I got the abbreviations for all languages that existed in the database.

Query 4

Getting more comfortable with aggregater(), I wanted to see if it was possible to switch around data types. I used $match to limit films by runtime, and then replaced the _id field with the titles of films. It worked and the debug code is relaying only the information regarding titles and runtimes. What makes me nervous about using this solution in the future is I feel this may screw something up. The problem is I don't know what that issue could be so I will try researching the ramifications through online forums or class.

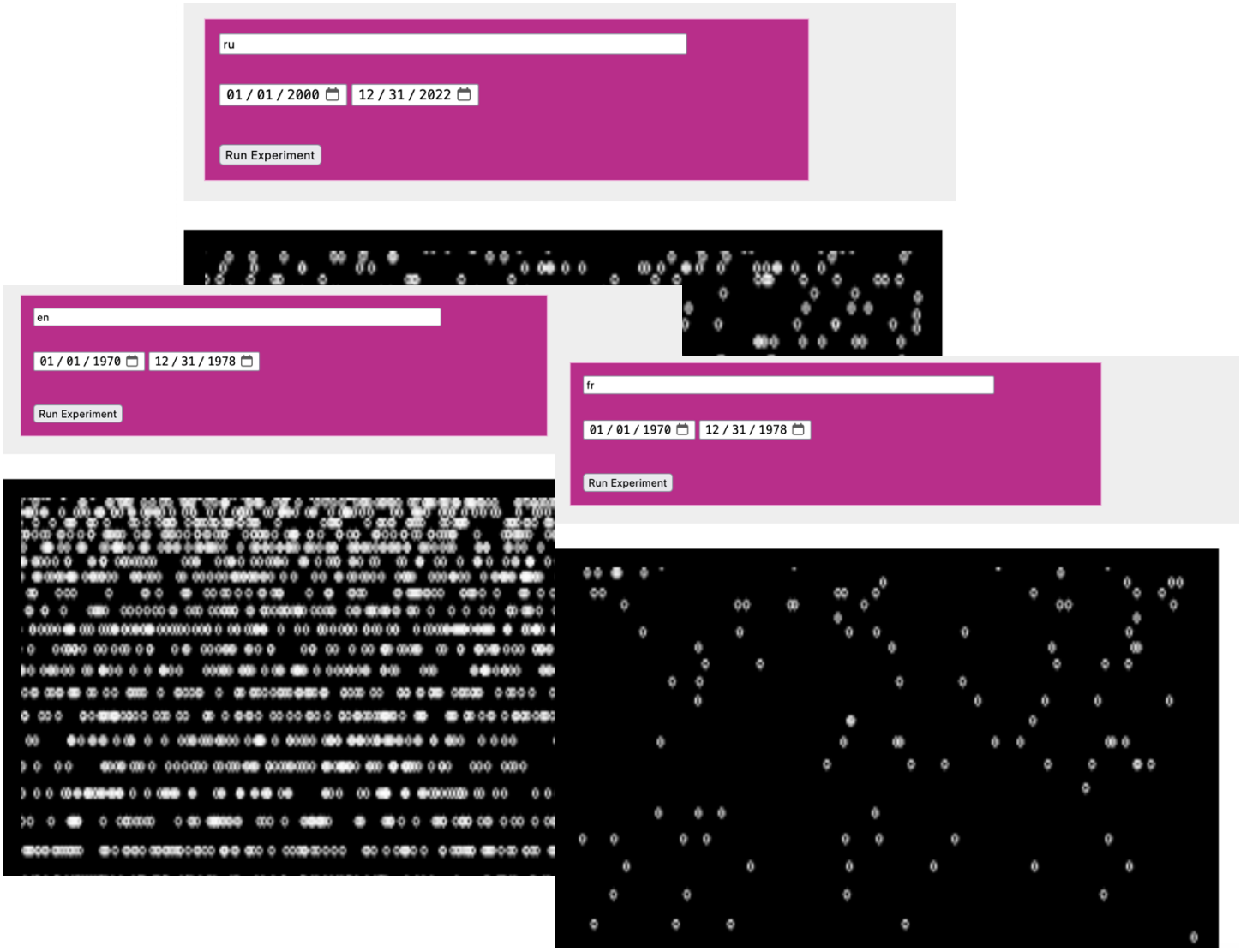
Query 5

This was to be the culmination of my previous attempts along with packaging everything neatly. Figuring out how to not only parse the data but clean up fields like dates took up quite a bit of time. For starters, MonogoDB documentation keeps saying to use something called

ISODate() for accessing date data. Turns out Mongo has their own function called Date() which was needed in place of ISODate(). Formatting the date past this required the use of an operator called $dateToString which was where I could clean up the formatting to my liking.

The toughest part of this query was understanding the difference between $project and $group. I spent all my time up until this point thinking $group did what $project was supposed to and kept getting errors. I now get that $project is used to combine your previous operators and showing only the data you want.

Reflection: Express & Mongo



My exercise takes a query and converts the returned data into an amount of raindrops. The user can select a language, a start date, and an end date. The returned value is the accumulated movie score of all films within scope.

```
//RUN SERVER QUERY//
async function queryServerButton() {

  //GET CURRENT DOCUMENT VALUES//
  let langSel = document.getElementById('languageField').value;
  let dateStart = document.getElementById('dateStartField').value;
  let dateEnd = document.getElementById('dateEndField').value;

  //SEND QUERY TO MONGO//
  fetch('/sendData', {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify({data1: langSel, data2: dateStart, data3: dateEnd}),
  })
  .then(response => response.json())
  .then(data => {changeDropCount(data.popularity);})
  .catch(err => {console.error('Error:', err)})

  //HTML TROUBLESHOOTING//
  console.log("Lang = " + langSel);
  console.log("Start Date = " + dateStart);
  console.log("End Date = " + dateEnd);
}
```

Inside my non-server code, I collect the values from the front facing html fields before sending them off. The amount returned by the query is found in data.popularity and is later added to the variable that controls rain amount.

```
//MONGO CONNECTION//
async function run() {

    try {

        app.post('/sendData', async(req, res) => {

            //INITIAL CONNECTION TO UPLOADED MONGO DB//
        await client.connect();
        await client.db("exercise1").command({ping:1});

        //ACCESS COLLECTION//
        const db = await client.db("exercise1");
        const horrorFilmsDB = await db.collection('horrorFilms');
        console.log("success");

        let data1 = req.body.data1;
        let data2 = req.body.data2;
        let data3 = req.body.data3;


        //EXERCISE 1 CODE//
        let dbQuery = await horrorFilmsDB.aggregate([
            {$match:    {original_language: data1}},
            {$match:    {release_date: {$gte: new Date(data2), $lte: new Date(data3)}}},
            {$group:    {_id: null, pop_val: {$sum: "$popularity"}}},
            {$project:  {_id: 0, popularity: {$round: ["$pop_val", 0]}}}
        ]).toArray();

        res.send(dbQuery[0]);
    })
```

Server-side code has me parsing the data queried by the front end. The filtration method goes as follows:

1. I filter data by original language.
2. I filter data that's greater or equal to the start date, but less or equal to the end date.
3. I group the sum of popularity into a new value called pop_val.
4. I return a project that only contains a rounded value of popularity.