Your Tasks

Datatake

In this project you'll be working with a subset of the MovieLens Dataset. Unlike the data set you worked on long ago in Project 1, the table here won't be organized in tables of records but rather as collections of documents! Documents are similar to records in the sense that they are used to group together pieces of data, but unlike the records we covered for SQL databases, documents can have fields that *are not* primitive data types. For example, the following document has three fields, two of which aren't primitive data types:

```
{
    "dataA": 1, // a regular primitive data type
    "anArray": [1,2,3], // an array!
    "nestedDocument": {"dataB": 2, "dataC": 3} // a doc inside a doc!
}
```

The following section will introduce you to the dataset.

Understanding the Dataset

This section will be helpful to reference as you're completing the tasks. A nice way to get an example of a record for any of the four collections (for example, `ratings`) is to call `db.ratings.findOne()` in the mongo shell.

```
$ mongo movies
MongoDB shell version v4.4.1
connecting to: mongodb://127.0.0.1:27017/movies?compressors=disabled&gssapiServi
Implicit session: session { "id" : UUID("cd49cd34-6c93-4548-aa20-b57e9f45d975")
MongoDB server version: 4.4.1

> db.ratings.findOne()
{
    "_id" : ObjectId("5fb32f37766efe011e6af587"),
    "userId" : 1,
    "movieId" : 783,
    "rating" : 2,
    "timestamp" : 1260759148
}
```

Movies Metadata

The `movies_metadata` collection contains documents with metadata for every movie in the MovieLens Dataset. There are many possible fields in each document, but some you may find useful for completing this project are:

- `movieId (int)`: A unique identifier for the movie. Corresponds to the field of the same name in `keywords`, `ratings`, and `credits`.
- `title (string)`: The title of the movie.
- `tagline (string)`: A short phrase usually used to advertise the movie.
- `release_date (int)`: The date of the film's release, as a UNIX Timestamp.
- `budget (inconsistent)`: The movie's production budget in USD. See question 2iii for more details on this field's type.
- `vote_average (number)`: The average rating given by viewers on a scale from 0 to 10.
- `vote_count (int)`: The total number of ratings given.
- `revenue (int)`: The movie's revenue in USD
- `runtime (int)`: The length of the movie in minutes
- `genres (array)`: 0 or more documents each containing the following:
  - `name (string)`: the name of one of the movie's genres
  - `id (int)`: a unique identifier for the genre

Example document:

```
// Extra fields omitted for brevity
{
    "budget" : 30000000,
    "genres" : [
        { "id" : 16, "name" : "Animation" },
        { "id" : 35, "name" : "Comedy" },
        { "id" : 10751, "name" : "Family" }
    ],
    "release_date" : 815040000,
    "revenue" : 373554033,
    "runtime" : 81,
    "status" : "Released",
    "title" : "Toy Story",
    "vote_average" : 7.7,
    "vote_count" : 5415,
    "movieId" : 862,
    "tagline" : ""
}
```

Keywords

The `keywords` collection contains documents with keywords related to certain movies. Each document in the collection has the following attributes:

- `movieId (int)`: A unique identifier for the movie. Corresponds to the field of the same name in `movies_metadata`, `ratings`, and `credits`
- `keywords (array)`: 0 or more documents each containing the following:
  - `id (int)`: A unique identifier for a keyword
  - `name (string)`: A keyword associated with the movie, for example "based on novel", "pirate", and "murder".

Example document:

```
{
    "keywords" : [
        { "id" : 3633, "name" : "dracula" },
        { "id" : 11931, "name" : "spoof" }
    ],
    "movieId" : 12110
}
```

Ratings

The `ratings` collection contains roughly 10,000 ratings submitted by specific viewers. Each document consists of the following format:

- `userId (int)`: A unique identifier for the user who gave the rating
- `movieId (int)`: A unique identifier for the movie. Corresponds to the field of the same name in `movies_metadata`, `keywords`, and `credits`
- `rating (number)`: The rating the user gave the movie, from 0 to 5.
- `timestamp (int)`: UNIX timestamp of when this rating was given

```
{
    "userId" : 1,
    "movieId" : 9909,
    "rating" : 2.5,
    "timestamp" : 1260759144
}
```

Credits

The `credits` collection contains details on writers, actors, directors, and technicians who worked on the production of the movies in the data set. Each document consists of the following:`

- `movieId (int)`: A unique identifier for the movie. Corresponds to the field of the same name in `movies_metadata`, `keywords`, and `ratings`
- `cast (array)`: 0 or more documents of the each containing the following fields:
  - `id (int)`: A unique identifier for the actor who played the character
  - `character (string)`: The name of the character in the movie
  - `name (string)`: The name of the actor as listed in the movie's credits
- `crew (array)`: 0 or more documents each containing the following fields:
  - `id (int)`: A unique identifier for the crew member
  - `department (string)`: The department the crew member worked in
  - `job (string)`: The job title of the crew member (e.g. "Audio Technician", "Director")
  - `name (string)`: The name of the crew member as listed in the movie's credits

Example document:

```
// Extra fields for cast and crew omitted for brevity
{
    "cast" : [
        {
            "character" : "Max Goldman",
            "id" : 6837,
            "name" : "Walter Matthau"
        },
        ... // Other cast members
    ],
    "crew" : [
        {
            "department" : "Directing",
            "id" : 26502,
            "job" : "Director",
            "name" : "Howard Deutch",
        },
        ... // Other crew members
    ],
    "movieId" : 15602
}
```

Building your first query

This task will walk you through step by step how to construct a query in MongoDB, and introduce you to some helpful operations for use in the rest of the tasks. If you're already comfortable doing queries in MongoDB feel free to skip to the end of this section to complete the corresponding question in `query/q0.js`.

Querying from the ratings collection

Inside the file `query/q0.js` you should see the following:

```
// Task 0 (Building your first query)

db.todo.aggregate([
    // TODO: Write your query here
]);
```

Try replacing the `todo` on line 3 so that the line becomes `db.ratings.aggregate([` instead. This tells mongo that we want to pull in documents from the `ratings` collection. Now try running the the following in a terminal: `python3 test.py q0 --view`. This should run the query and give something similar to the following output (note that the _id field will likely differ, which is fine):

```
Showing up to the first 10 results of the query
{ "_id" : ObjectId("5fb11b4a82f9cebdc2acaea0"), "userId" : 1, "movieId" : 9909,
{ "_id" : ObjectId("5fb11b4a82f9cebdc2acaea1"), "userId" : 1, "movieId" : 783, "
....
```

This query attempts to read every single document in the `ratings` collection, and would be analogous to something like the following:

```
SELECT * FROM ratings;
```

Match

The query from the earlier section attempts to read every document in the `ratings` collections. This next query will selectively match only specific documents matching a specific property.

```
db.ratings.aggregate([
    // Match documents with certain timestamps
    {
        $match: {
            timestamp: {
                $gte:  838857600, // >= 838857600
                $lt: 849398400    //  < 849398400
            }
        }
    }
]);
```

It may be useful to see how this would look as a SQL query:

```
SELECT * FROM ratings WHERE timestamp >= 838857600 AND timestamp < 849398400;
```

After pasting in the new query into `query/q0.js` and running `test.py q0 --view` you should see output resembling the following:

```
Showing up to the first 10 results of the query
{ "_id" : ObjectId("5fb11b4a82f9cebdc2acc11b"), "userId" : 24, "movieId" : 400,
{ "_id" : ObjectId("5fb11b4a82f9cebdc2acc11c"), "userId" : 24, "movieId" : 949,
...
```

Lets break down the query.

- `db.ratings.aggregate([ ... ])` : This tells mongo that we'll be running an "aggregate" operation. We can pass in a list `[ ... ]` known as the "pipeline", which will execute a series of operations. Each operation in the pipeline is known as a "stage", and each stage operates on the output of the previous stage.
- `{$match: ...}` : This is the only element of the pipeline so far. This tells mongo that we want documents from the collection that match certain properties, much like a `WHERE` clause.
- `{timestamp: { $gte: ..., $lt: ...}}` : This tells mongo that we only want documents where the timestamp field has a value greater than or equal to `838857600` and less than `849398400`.
- `838857600` and `849398400`: The value in the `timestamp` field is the number of seconds that have elapsed since January 1, 1970. This is more commonly known as "Unix time", "POSIX time", "Epoch time" or "UNIX Timestamp" and is a common way to represent times in computer systems. The two timestamps we use in the query correspond to October 1st, 1996 and December 1st, 1996 respectively.
  - There's no shortage of online tools like this one to convert between human readable times and UNIX time.
  - If you've ever heard of rumors about the world ending in January, 2038, that has to do with the way these timestamps are stored!

Group

Next we'll extend our pipeline to perform an aggregate similar to what we might attempt in SQL.

```
db.ratings.aggregate([
    // Match documents with certain timestamps
    {$match: {timestamp: { $gte:  838857600, $lt: 849398400}}},
    // Perform an aggregation
    {
        $group: {
            _id: "$movieId", // Group by the field movieId
            min_rating: {$min: "$rating"}, // Get the min rating for each group
            max_rating: {$max: "$rating"}, // Get the max rating for each group
            count: {$sum: 1} // Get the count for each group
        }
     }
]);
```

The equivalent SQL expression would look like the following:

```
SELECT
    movieId as _id,
    MAX(rating) as max_rating,
    MIN(rating) as min_rating,
    COUNT(*) as count
FROM ratings
WHERE timestamp >= 838857600 AND timestamp < 849398400
GROUP BY movieId;
```

After pasting in the new query into `query/q0.js` and running `test.py q0 --view` :

```
{ "_id" : 1094, "min_rating" : 3, "max_rating" : 3, "count" : 1 }
{ "_id" : 830, "min_rating" : 2, "max_rating" : 5, "count" : 2 }
{ "_id" : 474, "min_rating" : 2, "max_rating" : 5, "count" : 16 }
{ "_id" : 62, "min_rating" : 3, "max_rating" : 5, "count" : 12 }
{ "_id" : 64, "min_rating" : 3, "max_rating" : 3, "count" : 1 }
{ "_id" : 231, "min_rating" : 1, "max_rating" : 5, "count" : 32 }
...
```

A group "stage" in the pipeline always takes the following form:

```
{
  $group:
    {
      _id: <expression>, // Group by the field or value in <expression>
      <field1>: { <accumulator1> : <expression1> },
      ... // as many copies of the above as you want
    }
 }
```

In our above example, we grouped by the `movieId` field. To indicate that we were referring to a field and not the string literal `"movieId"` we prefixed it `$` . After that we had three expressions representing values we wanted to compute in the aggregate. The first two expressions computed the `min` and `max` values of the `rating` field.

The last column looks a bit peculiar: `count: {$sum: 1}` . This assigns the count field to the accumulated sum of the value `1` . This means we add together `n` copies of the value `1` where `n` is the number of documents in each group, giving the total count of documents in each group.

Sort and Limit

Next lets try to see the top 10 movies with the most ratings. Continuing to build off our previous query:

```
db.ratings.aggregate([
    // Match documents with certain timestamps
    {$match: {timestamp: { $gte:  838857600, $lt: 849398400}}},
    // Perform an aggregation
    {
        $group: {
            _id: "$movieId", // Group by the field movieId
            min_rating: {$min: "$rating"}, // Get the min rating for each group
            max_rating: {$max: "$rating"}, // Get the max rating for each group
            count: {$sum: 1} // Get the count for each group
        }
     },
     // Sort in descending order of count, break ties by ascending order of _id
     {$sort: {count: -1, _id: 1}},
     // Limit to only the first 10 documents
     {$limit: 10}
]);
```

Our equivalent query in SQL now looks like:

```
SELECT
    movieId as _id,
    MAX(rating) as max_rating,
    MIN(rating) as min_rating,
    COUNT(*) as count
FROM ratings
WHERE timestamp >= 838857600 AND timestamp < 849398400
GROUP BY movieId
ORDER BY count DESC, _id ASC
LIMIT 10;
```

After pasting in the new query into `query/q0.js` and running `test.py q0 --view` :

```
{ "_id" : 480, "min_rating" : 2, "max_rating" : 5, "count" : 48 }
{ "_id" : 356, "min_rating" : 1, "max_rating" : 5, "count" : 47 }
{ "_id" : 590, "min_rating" : 3, "max_rating" : 5, "count" : 45 }
{ "_id" : 457, "min_rating" : 3, "max_rating" : 5, "count" : 44 }
{ "_id" : 296, "min_rating" : 1, "max_rating" : 5, "count" : 42 }
{ "_id" : 592, "min_rating" : 2, "max_rating" : 5, "count" : 42 }
{ "_id" : 150, "min_rating" : 2, "max_rating" : 5, "count" : 41 }
{ "_id" : 380, "min_rating" : 2, "max_rating" : 5, "count" : 39 }
{ "_id" : 589, "min_rating" : 1, "max_rating" : 5, "count" : 37 }
{ "_id" : 165, "min_rating" : 2, "max_rating" : 5, "count" : 35 }
```

The sort stage always takes the following form:

`{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }`

The fields refer to which field of the output to sort on. The sort order is always either `1` to indicate ascending order or `-1` for descending order.

The limit stage just needs to specify a number to limit the number of possible documents.

Lookup

Next, we'll do the equivalent of a join in a SQL database to figure out what movies we're the IDs correspond to. Building again off the previous query:

```
db.ratings.aggregate([
    // Match documents with certain timestamps
    {$match: {timestamp: { $gte:  838857600, $lt: 849398400}}},
    // Perform an aggregation
    {
        $group: {
            _id: "$movieId", // Group by the field movieId
            min_rating: {$min: "$rating"}, // Get the min rating for each group
            max_rating: {$max: "$rating"}, // Get the max rating for each group
            count: {$sum: 1} // Get the count for each group
        }
     },
     // Sort in descending order of count, break ties by ascending order of _id
     {$sort: {count: -1, _id: 1}},
     // Limit to only the first 10 documents
     {$limit: 10},
     // Perform a "lookup" on a different collection
     {
         $lookup: {
             from: "movies_metadata", // Search inside movies_metadata
             localField: "_id", // match our _id
             foreignField: "movieId", // with the "movieId" in movies_metadata
             as: "movies" // Put matching rows into the field "movies"
         }
     }
]);
```

Normally this is where we would put the expected output, but if you tried it out yourself you should know it looks like a complete mess! It may help to look at a single formatted document here. Try running `python3 test.py q0 --format`, which should give a cleaned up version of the first document returned by the query:

```
{
    "_id" : 480,
    "min_rating" : 2,
    "max_rating" : 5,
    "count" : 48,
    "movies": [ // Notice that "movies" is an array!
        {
            "title": "Some Movie",
            "budget": 123456789,
            // Full contents of a document from
            // movies_metadata
        }
    ]
```

There are two things to note here:

- The "movies" field corresponds to an *array* of matching documents, not a single document. If there were multiple documents in `movies_metadata` with a matching id, they would all appear in that list. If there were no matching documents, `movie` would map to an empty list.
- The contents of `movie` are *entire documents.* In SQL this would be like having an entire record stored in a single column! As the name NoSQL implies though, we're not bound to the rule of SQL that all values must be atomic, and so we're allowed to have this nested structure.

Project

The result of the last query was a bit messy, so lets do some cleaning up. We'll create a new field, `name` , to store the name of the movie, rename `count` and get rid of the column field `_id` .

```
db.ratings.aggregate([
    // Match documents with certain timestamps
    {$match: {timestamp: { $gte:  838857600, $lt: 849398400}}},
    // Perform an aggregation
    {
        $group: {
            _id: "$movieId", // Group by the field movieId
            min_rating: {$min: "$rating"}, // Get the min rating for each group
            max_rating: {$max: "$rating"}, // Get the max rating for each group
            count: {$sum: 1} // Get the count for each group
        }
     },
     // Sort in descending order of count, break ties by ascending order of _id
     {$sort: {count: -1, _id: 1}},
     // Limit to only the first 10 documents
     {$limit: 10},
     // Perform a "lookup" on a different collection
     {
         $lookup: {
             from: "movies_metadata", // Search inside movies_metadata
             localField: "_id", // match our _id
             foreignField: "movieId", // with the "movieId" in movies_metadata
             as: "movies" // Put matching rows into the field "movies"
         }
     },
     {
        $project: {
                _id: 0, // explicitly project out this field
                title: {$first: "$movies.title"}, // grab the title of first mov
                num_ratings: "$count", // rename count to num_ratings
                max_rating: 1,
                min_rating: 1
        }
     }
]);
```

After pasting in the new query into `query/q0.js` and running `test.py q0 --view` :

```sql
-- Note that the closest equivalent to $lookup is a LEFT OUTER JOIN, not an
-- INNER JOIN!
SELECT
    MAX(rating) as max_rating,
    MIN(rating) as min_rating,
    m.title as title,
    COUNT(*) as num_ratings
FROM ratings as r LEFT OUTER JOIN movies_metadata as m ON r.movieId = m.id
WHERE timestamp >= 838857600 AND timestamp < 849398400
GROUP BY r.movieId
ORDER BY num_ratings DESC, r._id ASC
LIMIT 10;
```

A few things to note here:

- The 0's and 1's stand for "drop" and "keep" respectively, i.e. "_id" gets dropped because its corresponding value in the project was 0.
- To get the title of the first movie in the "movies" array we used `{$first: "$movies.title"}`. We used `.title` to access the `title` attribute of each document in `movies`. Try running the query without the `$first` operator (`"title": "$movies.title"`) and see what happens.
- You can rename columns using a project.

You should now see some output that looks like this:

```
{ "min_rating" : 2, "max_rating" : 5, "title" : "Jurassic Park", "num_ratings" :
{ "min_rating" : 1, "max_rating" : 5, "title" : "Forrest Gump", "num_ratings" :
{ "min_rating" : 3, "max_rating" : 5, "title" : "Dances with Wolves", "num_ratin
{ "min_rating" : 3, "max_rating" : 5, "title" : "The Fugitive", "num_ratings" :
{ "min_rating" : 2, "max_rating" : 5, "title" : "Batman", "num_ratings" : 42 }
{ "min_rating" : 1, "max_rating" : 5, "title" : "Pulp Fiction", "num_ratings" :
{ "min_rating" : 2, "max_rating" : 5, "title" : "Apollo 13", "num_ratings" : 41
{ "min_rating" : 2, "max_rating" : 5, "title" : "True Lies", "num_ratings" : 39
{ "min_rating" : 1, "max_rating" : 5, "title" : "Terminator 2: Judgment Day", "n
{ "min_rating" : 2, "max_rating" : 5, "title" : "Pretty Woman", "num_ratings" :
```

Congrats, you've built your first query! You can run `python3 test.py q0` to test that the provided solution works. This section may seem a bit intimidating, but its mostly there for you to reference back to as you work through the remaining tasks.

Your Tasks

Note on grading: question 0 and the questions in Task 1 are worth 1 point each, while the questions in Tasks 2 and 3 are worth 2 points each.

Task 1: Basics

i. After spending over a year social distancing, you find yourself reading a lot of marvel comics and watching a lot of Disney movies. Find the IDs of all movies labeled with the keyword "mickey mouse" **or** "marvel comic" by writing a query on the `keywords` collection. Order your output in ascending order of `movieId`. The output documents should have the following fields:

```
{"movieId": <number>}
```

- Hint: start by trying to find movieIds labeled with just "mickey mouse", then use the $or operator to match documents with either of the labels.
- Hint: you may find the $elemMatch operator useful here to select the appropriate documents. For example, the following query would match any movie with English listed as one of its spoken languages:

```
db.movies_metadata.aggregate([
  // Use elemMatch in the $match stage to find movies  with English
  // as a spoken language
    {$match: {spoken_languages: {$elemMatch: {name: "English"}}}},
    {$project: {title: 1, _id: 0}} // clean up output
])
```

ii. We're interested in the best comedy films to watch. Return the id, title, average vote, and vote count of the top 50 comedy movies ordered from highest to lowest by average vote, breaking ties by descending order of vote count, and any further ties in ascending order of `movieId`. Only include movies with 50 or more votes. The output documents should have the following fields:

```
{
    "title" : <string>,
    "vote_average" : <number>,
    "vote_count" : <number>,
    "movieId" : <number>
}
```

- Useful operators: $elemMatch, $gte for matching
- Hint: genre names are case sensitive!

iii. Do movies get more good reviews than bad reviews? Is it the other way around? We want to know! For each possible rating find how many times that rating was given. Include the rating and the number of the times the rating was given and output in descending order of the rating. The output documents should have the following fields:

```
{
    "count": <number>,
    "rating": <number>
}
```

- Hint: the building your first query section gives an example of how to get a count

iv. You've discovered a critic who always seems to know exactly which movies you would love and which ones you would hate. Their true name is a mystery, but you know their user id: `186`. Find critic 186's five most recent movie reviews, and create create a document with the following fields:

```
{
    "movieIds": [most recent movieId, 2nd most recent, ... , 5th most recent],
    "ratings": [most recent rating, 2nd most recent, ..., 5th most recent],
    "timestamps": [most recent timestamp, 2nd most recent, ... , 5th most recent
}
```

- Useful operators: Look into the $push operator
- Hint: You may find it helpful to see what happens when you group by null.

Task 2: Movie Night

i. The TAs are having a movie night but they're having trouble choosing a movie! Luckily, Joe has read about IMDb's Weighted Rating which assigns movies a score based on demographic filtering. The weighted ranking (WR) is calculated as follows:

$$\mathrm{WR} = (\frac{v}{v+m})R + (\frac{m}{v+m})C$$

- v is the number of votes for the movie
- m is the minimum votes required to be listed in the chart
- R is the average rating of the movie (this is stored in the field `vote_average`)
- C is the mean vote across the whole report. For the purposes of this question this value is approximately 7, which you can hardcode into your query.

We would like to set a minimum number of votes to make sure the score is accurate. For this question we will assume the minimum votes required to be listed is 1838. Return the 20 highest rated movies according to this formula. The output should contain three fields: `title` with the title of the movie, `vote_count` with the number of votes the movie received, and `score` which contains the WR for the associated movie rounded to two decimal places. How many movies can you recognize on this list? Sort in descending order of `score`, and break ties in descending order of `vote_count` and ascending order of `title`. Your output documents should have the following fields:

```
{
    "title": <string>,
    "vote_count": <number>,
    "score": <number>
}
```

- Useful operators: Look up what the `$add`, `$multiply`, `$divide`, and `$round` operators do (you may find this question reminiscent of Scheme from 61a!)

ii. The TAs consider the prospect of making their own feature film on the beauty and joy of databases, and want to think of a catchy tagline. Run the following to see some examples taglines:

```
db.movies_metadata.aggregate({$project: {"_id": 0, "tagline": 1}})
```

Notice how the second one is "Roll the dice and unleash the excitement!" We want to see the 20 most common words (length > 3) across all taglines in descending order. In order to do this, we

would need to split our sample tagline into its constituent words ("Roll", "the", "dice", "and", "unleash", "the", "excitement!").

To make things interesting, we will limit the words to length >3 to remove filler words, prepositions, and some pronouns (in the previous example, remove "the" and "and"). We also want to trim off any surrounding punctuation (periods, commas, question marks, or exclamation points) in a word and set all words to lowercase (our final set of words that will be included in our table for our example tagline is "roll, "dice", "unleash", "excitement", without the exclamation mark). Do not trim off any other punctuation character that is not mentioned. Order your output by descending order of `count`. Your output documents should have the following fields:

```
{
    "_id": <string>,
    "count": <number>
}
```

Can you guess what the most popular words might be?

- Hint: Look up the `$unwind` stage to see what it does.
- Useful operators:
  - $split can be used to convert a string to an array. For example splitting the string "a proper copper coffee pot" by " " (a space) will create the array ["a", "proper", "copper", "coffee", "pot"]
  - $toLower converts a string to lowercase
  - $trim can be used to trim off surrounding punctuation marks
  - $strLenCP can be used to get the length of a string. Make sure to check for length *after* removing punctuation marks!

iii. How much does it cost to make a movie? The TAs were hoping to write a query for this but realized something that will haunt them for the rest of their lives... Mongo's lack of schema requirements means that the budget field of documents metadata isn't always an integer! Even worse, sometimes the field doesn't even exist! It looks like whoever prepared the data set always did one of the following:

- If they didn't know the budget of a given movie they did one of the following:
  - Set the `budget` field to `false`
  - Set the `budget` field to `null`
  - Set the `budget` field to an empty string: `""`
  -

    - Excluded the `budget` field from the document
- If they did know the budget of a given movie they did the following:
    - Set the `budget` field to a number value, for example `186`
    - Set the `budget` field to a string with prefix `$`, for example `"$186"`
    - Set the `budget` field to a string with the the postfix "USD", for example `"186 USD"`

Group the budgets by their value rounded to the nearest multiple of ten million, and return the count for each rounded value. Additionally include an extra group `"unknown"` for the count of movies where the budget was not known. Order by ascending order of rounded budget. Your output documents should have the following fields:

```
{
    "budget": <number or "unknown">,
    "count": <number>
}
```

- Useful operators: You may find the following useful: $ne, $and, $cond, $isNumber, $toInt and $round. Trim will also be useful here for removing prefixes and postfixes.
- Hint: You can check if a field is present in a document by checking whether the field is equal to `undefined`

Task 3: Paparazzi

i. Comic book writer Stan Lee was known to make cameos in film adaptations of his works. Find the release date, title, and the name of the character Lee played for every movie Lee has appeared in. Order the results in descending order of release date. Your output documents should have the following fields:

```
{
    "title": <string>,
    "release_date": <number>,
    "character": <string>
}
```

- Useful operators: you may find $unwind (used in 2ii) handy here
- Hint: Stan Lee's id in the credits collection is `7624`.

ii. Director Wes Anderson is known for his unique visual and narrative style, and frequently collaborates with certain actors. Find the 5 actors who have appeared the most often in movies where Anderson is listed on the crew with the title "Director". Your output should include the actor's name, id, and the number of times the actor has collaborated with Anderson. Order in descending order of the number of collaborations. Break ties in ascending order of the actor's id. Your output documents should have the following fields:

```
{
    "count": <number>, // number of times collaborated
    "id": <string>,
    "name": <string>
}
```

- Hint: Wes Anderson's id in the credits collection is `5655`. To get started, try to match all documents in `credits` where he is listed as the director.
- Hint: Like the above question, $unwind can be useful here.
- Hint: You may need to group by multiple fields for this question. For example, `{$group: _id: {val1: "$field.val1", val2: "$field.val2"}}` will group documents by both val1 and val2, similar to how the expression `GROUP BY val1, val2` would in SQL.

You're done!

Congrats, you're finished with the last project! There are **no hidden tests** for this assignment, so whatever score you see on the autograder after this will be your score on this assignment. Follow the instructions in the next section to submit your work.

Testing Tips

Running Queries

You can run your answers through mongo directly by running `mongo movies` to open the database and then entering your query directly:

```
$ mongo movies
db.replace_with_a_collection.aggregate([ ... your query ... ])
```

This can help you catch any syntax errors in your queries. Alternatively you can run a query directly through the testing script by pasting in your query into the appropriate file (we'll use `query/q0.js` as an example) and running `python3 test.py q0 --view`. This will print up to ten of the query's results.

You can request more results with the `--batch_size` flag (i.e. `python3 test.py q0 --view --batch_size 20` will give the first twenty results).

Formatting output

If you find yourself dealing with large, hard to read documents, you can view a formatted version of the first document by using `--format` instead of `--view`. For example, using the provided query from the Building your first query section of the spec:

```
$ python3 test.py q0 --format
Showing formatted first document of the query
{
    "min_rating": 2,
    "max_rating": 5,
    "title": "Jurassic Park",
    "num_ratings": 48
}
```

To run a test, from within the `sp24-proj6-yourname` directory:

```
$ python3 test.py     # This runs all of the tests
$ python3 test.py 3ii # This would run tests for only q3ii
```

Format Matching

Before we run a full test on your output, we check that the format of your output matches what we're expecting. Format in this context means that all the field names we expect are there, there are no extra field names, and that the types corresponding to the field names match. Here's an example of some mismatched format for `diffs/q2i.diff`

```
EXTRA FIELDS
- foo

MISMATCHED TYPES
- mismatch on field `movieId`:
    - expected type: `number` (example: `63`)
    - actual type: `null`, (example: `null`)

FORMAT MISMATCH
- Example of expected document:
{
    "movieId": 63
}

- Your document:
{
    "foo": "bar",
    "movieId": null
}
```

The above output tells you two things:

- One of your documents had an extra field. In this case, looking at the "Your document" section, your output had an extra field called "foo"
- One of your documents has a mismatched type for one of its fields. In this case, look at the "Your document" section, your output had a value of type null for the field "movieId", when it should have been a number.

Diffs

If you pass the format check, we'll run a diff against your query and the expected output. Become familiar with the UNIX diff format, if you're not already, because our tests saves a simplified diff for any query executions that don't match in `diffs/`. As an example, the following output for `diffs/q2ii.diff:`:

```
+ {"_id": "only", "count": 521}
  {"_id": "just", "count": 481}
- {"_id": "about", "count": 535}
```

This indicates that:

-

- your output has an extra document `{"_id": "about", "count": 535}` (the `-` at the beginning means the expected output *doesn't* include this line but your output has it)
- your output is missing the document `{"_id": "only", "count": 521}` (the plus at the beginning means the expected output *does* include those lines but your output is missing it).
- If there is neither a `+` nor `-` at the beginning then it means that the line is in both your output and the expected output (your output is correct for that line).

If you care to look at the query outputs directly, ours are located in the `expected_output` directory. Your output should be located in your solution's `your_output` directory once you run the tests.

Reformatting

When we generate the diffs we'll be doing some basic reformatting to reorder things that don't have an inherent order to make sure that the results are consistent with each other. These include:

- field names will be rearranged to be in alphabetical order
- arrays when we don't ask for an explicit order to them

Previous
Getting Started

Next
Submitting the Assignment

Last updated 3 months ago