Loading the Data

In order to load the data, I first placed it into a postgres database in a relational schema that roughly matched the one on the assignment. In order to do this, I tried a few different import methods before finding one that worked.

The first thing I tried was to import the data from the files directly into postgres using the `COPY` command, like so:

```
COPY names(nconst, primaryName, birthYear, deathYear, primaryProfession,
knownForTitles)
FROM '/Users/josh/Documents/code/cs4200-monorepo/ps3/name.basics.tsv'
DELIMITER E'\t' CSV HEADER NULL '\N';
```

For some tables, such as the one above - this worked! However, I found that for tables that had an array data type (which were most of them) - this approach did not work with the array format in the tsv, and I wasn't able to find a way to change the delimiters in the copy command to get it to work without modifying the input files. I tried to modify the input files, but they were so large they crashed vscode whenever I opened them, and I wasn't able to find a great regex to use to find/replace all of the arrays with the format I needed.

Next, I decided to try and write a node script that would read in a file, and execute batched inserts on the database. Using a scripting language easily allowed me to modify the string formatting so that postgres would accept the input, but then I ran into a different problem - node couldn't load the files into memory without overflowing the heap. This led me to try reading the file in as a stream, which finally worked. Below is the script I used to import one of the tables (for brevity, I'm going to exclude the rest, but they all follow the same format)

```
const readMovieNames = async () => {
  const instream = fs.createReadStream(
    "/Users/josh/Documents/code/cs4200-monorepo/ps3/title.basics.tsv"
  );
  const rl = readline.createInterface(instream, null);

  let rowsToInsert: Array<Array<string>> = [];

  rl.on("line", async (line) => {
    const lineValues = line.split("\t");
    const dataToInsert = [];
    lineValues.forEach((l) => {
      if (l.includes(`\N`)) {
        dataToInsert.push(null);
      } else {
        dataToInsert.push(l);
      }
    });
    const genresString = dataToInsert.pop();
    dataToInsert.push(`{${genresString}}`);
```

```
    rowsToInsert.push(dataToInsert);

    if (rowsToInsert.length >= BATCH_SIZE) {
      let query1 = format(
        "INSERT INTO titles (tconst, titleType, primaryTitle,
originalTitle, isAdult, startYear, endYear, runtimeMinutes, genres) VALUES
%L",
        rowsToInsert
      );
      rowsToInsert = [];

      let res = await client.query(query1).catch((e) => {
        console.error(e);
      });
      console.log(res);
      console.log("Query executed");
    }
  });
};

const main = async () => {
  await client.connect();
  await readMovieNames();
};
```

At this point, I'd finally gotten the data into postgres. Now, I needed to export it as a nested json structure into a file so that it could be imported by mongodb.

At first, I tried a query like this

```
copy
  (select array_to_json(array_agg(json_strip_nulls(row_to_json(t))))
   from
     (select titles.tconst as _id,
             titleType as type,
             primaryTitle as title,
             startYear,
             endYear,
             originalTitle,
             runtimeMinutes as runtime,
             genres,
             averageRating as avgRating,
             numVotes,

        (select array_to_json(array_agg(row_to_json(d)))
         from
           (select directors.nconst as _id,
                   primaryName as name,
                   birthYear,
                   deathYear
            from directors
```

```
            left join names on directors.nconst = names.nconst
            where tconst = titles.tconst ) d) as directors,

          (select array_to_json(array_agg(row_to_json(d)))
           from
             (select writers.nconst as _id,
                     primaryName as name,
                     birthYear,
                     deathYear
              from writers
              left join names on writers.nconst = names.nconst
              where tconst = titles.tconst ) d) as writers,

          (select array_to_json(array_agg(row_to_json(d)))
           from
             (select principals.nconst as _id,
                     primaryName as name,
                     birthYear,
                     deathYear
              from principals
              left join names on principals.nconst = names.nconst
              where tconst = titles.tconst
                and principals.category='producer') d) as producers,

          (select array_to_json(array_agg(row_to_json(d)))
           from
             (select principals.nconst as _id,
                     primaryName as name,
                     birthYear,
                     deathYear
              from principals
              left join names on principals.nconst = names.nconst
              where tconst = titles.tconst
                and (principals.category='actor'
                     or principals.category='actress')) d) as actors
      from titles
      left join ratings on titles.tconst = ratings.tconst
      where isAdult=false) t) TO '/tmp/movies.json';
```

However, after about 25 minutes of churning away, postgres told me it ran out of memory, and absolutely nothing was saved in the output file! This was a similar problem to the one I had encountered while using node, so I decided that instead of aggregating all of the objects into an array in the output file, what if I just have postgres write them to file individually, so that it doesn't need so much heap space? To do this, I changed the first line of the query (after the `copy` keyword) to this:

```
(select (json_strip_nulls(row_to_json(t)))
...
```

And this worked! At this point, I started importing the data with the following command:

```
mongoimport --uri mongodb+srv://<USERNAME>:
<PASSWORD>@<CLUSTERURL>.mongodb.net/imdb --collection movies --type json -
-file /tmp/movies.json --drop --verbose=4 --parseGrace=skipField
```

However, I pretty quickly ran into an error about 1% of the way into the data - there was a problem with a misformatted escape character in the json outputted by postgres. When I grepped over the file to see how frequent this was, I found over 20000 instances of this malformed escape character. To fix this, I used the following bash command

```
sed -i '' -e 's/\\"/\"/g' /tmp/movies.json
```

This found all of the poorly escaped double quotes and fixed them. After this, I was able to run the import successfully (at least until I hit the mongodb storage limit and had to upgrade).

The import took around 2.5 hours. However, I strongly suspect this was actually bottlenecked by my ISP and my low upload speed. I was pretty much maxing out the upload speed the whole time it was uploading (I checked) - and am fairly certain that this would have been much faster if I had a higher upload bandwidth.

The Queries

A

Living actors whose name starts with Phi and who did not participate in any movie in 2014.

```
db.movies.aggregate(
  [
    {
      "$match": {
        "type": "movie"
      }
    },
    {
      "$unwind": {
        "path": "$actors",
        "includeArrayIndex": "string",
        "preserveNullAndEmptyArrays": true
      }
    },
    {
      "$project": {
        "title": true,
        "startyear": true,
        "endyear": true,
        "type": true,
        "actors": true
      }
```

```
    },
    {
      "$group": {
        "_id": "$actors._id",
        "name": {
          "$first": "$actors.name"
        },
        "birthyear": {
          "$first": "$actors.birthyear"
        },
        "deathyear": {
          "$first": "$actors.deathyear"
        },
        "movies": {
          "$push": "$$ROOT"
        }
      }
    },
    {
      "$match": {
        "name": /^Phi/,
        "movies.startyear": {
          "$not": {
            "$eq": 2014
          }
        },
        "deathyear": {
          "$eq": null
        }
      }
    },
    {
      "$project": {
        "name": true
      }
    }
  ],
  { "allowDiskUse": true }
)
```

This query took ~31.14 seconds to run, and these are some of the results that were returned:

```
[
  { "_id": "nm0001275", "name": "Philip Glass" },
  { "_id": "nm0001311", "name": "Philip Baker Hall" },
  { "_id": "nm0001911", "name": "Philip Anglim" },
  { "_id": "nm0002015", "name": "Phil Collins" },
  { "_id": "nm0004882", "name": "Phil Donahue" },
  { "_id": "nm0006573", "name": "Philippe Leroy" },
  { "_id": "nm0008723", "name": "Philippe Abia" },
  { "_id": "nm0009223", "name": "Phil Abrams" },
  { "_id": "nm0012497", "name": "Philippe Adrien" },
```

```
  { "_id": "nm0015370", "name": "Philip Akin" },
  { "_id": "nm0020086", "name": "Philippe Allard" },
  { "_id": "nm0022815", "name": "Philip Altice" },
  { "_id": "nm0024456", "name": "Philippe Ambrosini" },
  { "_id": "nm0029721", "name": "Philip Angelotti" },
  { "_id": "nm0029798", "name": "Philippe Angers" },
  { "_id": "nm0030611", "name": "Phil Anselmo" },
  { "_id": "nm0030983", "name": "Philip Anthony" },
  { "_id": "nm0037674", "name": "Philip Artemus" },
  { "_id": "nm0039459", "name": "Phil Askham" },
  { "_id": "nm0042253", "name": "Philippe Aussant" }
]
```

B

Producers who have produced more than 50 talk shows in 2017 and whose name contain Gill.

```
db.movies.aggregate([
  {
    "$match": {
      "genres": "Talk-Show",
      "producers.name": {
        "$regex": ".*Gill.*"
      },
      "startyear": {
        "$lte": 2017
      }
    }
  },
  {
    "$unwind": {
      "path": "$producers",
      "includeArrayIndex": "string",
      "preserveNullAndEmptyArrays": true
    }
  },
  {
    "$group": {
      "_id": "$producers._id",
      "name": {
        "$first": "$producers.name"
      },
      "count": {
        "$sum": 1
      }
    }
  },
  {
    "$match": {
      "count": {
        "$gte": 50
```

```
      },
      "name": {
        "$regex": ".*Gill.*"
      }
    }
  }
])
```

This query executed in 13.2 seconds and produced this result set:

```
[
  { "_id": "nm8230849", "name": "Ryan Gill", "count": 79 },
  { "_id": "nm1861174", "name": "Dominic Gillette", "count": 128 }
]
```

C

Average runtime for movies that were written by living members whose names contain Bhardwaj

```
db.movies.aggregate([
  {
    "$unwind": {
      "path": "$writers",
      "includeArrayIndex": "string",
      "preserveNullAndEmptyArrays": true
    }
  },
  {
    "$match": {
      "type": "movie",
      "writers.name": {
        "$regex": ".*Bhardwaj.*"
      },
      "writers.deathyear": {
        "$exists": false
      }
    }
  },
  {
    "$group": {
      "_id": "$writers._id",
      "name": {
        "$first": "$writers.name"
      },
      "avgRuntime": {
        "$avg": "$runtime"
      },
      "movies": {
        "$push": {
```

```
            "_id": "$_id",
            "title": "$title",
            "runtime": "$runtime"
          }
        }
      }
    }
])
```

Here is a snippet of the output produced by this query. It executed in ~13.4 seconds. I included an array of movies containing the id, title, and runtime of each movie by the writer so we could check that the average is correct and manually verify these results.


```
[
  {
    "_id": "nm2637116",
    "name": "Pravesh Bhardwaj",
    "avgRuntime": 118,
    "movies": [
      { "_id": "tt1646217", "title": "Mr. Singh/Mrs. Mehta", "runtime":
118 }
    ]
  },
  {
    "_id": "nm9523134",
    "name": "Pradeep Bhardwaj",
    "avgRuntime": 136.66666666666666,
    "movies": [
      { "_id": "tt12320574", "title": "Bir Bikram 2", "runtime": 131 },
      { "_id": "tt7676212", "title": "Jhyanakuti", "runtime": 138 },
      { "_id": "tt7814604", "title": "Panche Baja" },
      { "_id": "tt7869094", "title": "Radha" },
      { "_id": "tt8374600", "title": "So Simple" },
      { "_id": "tt8682226", "title": "Chhodi Gaye Paap Lagla" },
      { "_id": "tt8728132", "title": "Hifajat" },
      { "_id": "tt8905928", "title": "Changaa Chait", "runtime": 141 }
    ]
  },
  {
    "_id": "nm7354230",
    "name": "Munish Bhardwaj",
    "avgRuntime": 108,
    "movies": [
      { "_id": "tt5343678", "title": "Moh Maya Money", "runtime": 108 }
    ]
  },
  {
    "_id": "nm5224655",
    "name": "Prasad Bhardwaja",
    "avgRuntime": 94,
    "movies": [{ "_id": "tt2349700", "title": "Bhagaude", "runtime": 94 }]
```

```
    }
]
```

D

Producers with the greatest number of long-run movies produced (runtime greater than 120 minutes).

```
db.movies.aggregate([
  {
    "$match": {
      "runtime": {
        "$gt": 120
      },
      "producers": {
        "$exists": true
      }
    }
  },
  {
    "$unwind": {
      "path": "$producers",
      "preserveNullAndEmptyArrays": true
    }
  },
  {
    "$group": {
      "_id": "$producers._id",
      "name": {
        "$first": "$producers.name"
      },
      "movies": {
        "$push": {
          "_id": "$_id",
          "title": "$title"
        }
      },
      "count": {
        "$sum": 1
      }
    }
  },
  {
    "$sort": {
      "count": -1
    }
  }
])
```

This query executed in 10.2 seconds. Here is a sample of the result set:

```
[
  {
    "_id": "nm0573093",
    "name": "Vince McMahon",
    "movies": [
      ...
    ],
    "count": 140
  },
  {
    "_id": "nm8024422",
    "name": "Claire Mooney",
    "movies": [
      ...
    ],
    "count": 90
  },
  {
    "_id": "nm1727403",
    "name": "Acun Ilicali",
    "movies": [
      ...
    ],
    "count": 88
  },
  {
    "_id": "nm9941948",
    "name": "Maxwell James",
    "movies": [
      ...
    ],
    "count": 75
  }
]
```

E

Sci-Fi movies directed by James Cameron and acted in by Sigourney Weaver

```
db.movies.find({
  "directors.name": "James Cameron",
  "actors.name": "Sigourney Weaver",
  "genres": "Sci-Fi"
})
```

This query executed in 14 seconds, and returned 1 movie (some irrelevant data was removed for brevity):

```
{
  _id: 'tt0090605',
  type: 'movie',
  title: 'Aliens',
  originaltitle: 'Aliens',
  genres: [ 'Action', 'Adventure', 'Sci-Fi' ],
  directors: [ { _id: 'nm0000116', name: 'James Cameron', birthyear: 1954
} ],
  writers:
   [ { _id: 'nm0000116', name: 'James Cameron', birthyear: 1954 },
     ... ],
  actors:
   [ { _id: 'nm0000244', name: 'Sigourney Weaver', birthyear: 1949 },
     ... ]
  ...
}
```

Execution Plans

All of the plans used a basic collection scan (iterating over all documents in the collection), and then applied the aggregate functions, sorts, projections, etc. after.

A


```
{
  "stages": [
    {
      "$cursor": {
        "query": { "type": "movie" },
        "fields": {
          "actors": 1,
          "endyear": 1,
          "startyear": 1,
          "title": 1,
          "type": 1,
          "_id": 1
        },
        "queryPlanner": {
          "plannerVersion": 1,
          "namespace": "imdb.movies",
          "indexFilterSet": false,
          "parsedQuery": { "type": { "$eq": "movie" } },
          "queryHash": "2A1623C7",
          "planCacheKey": "2A1623C7",
          "winningPlan": {
            "stage": "COLLSCAN",
            "filter": { "type": { "$eq": "movie" } },
            "direction": "forward"
          },
          "rejectedPlans": []
```

```
          }
        }
      },
      {
        "$unwind": {
          "path": "$actors",
          "preserveNullAndEmptyArrays": true,
          "includeArrayIndex": "string"
        }
      },
      {
        "$project": {
          "_id": true,
          "title": true,
          "type": true,
          "endyear": true,
          "actors": true,
          "startyear": true
        }
      },
      {
        "$group": {
          "_id": "$actors._id",
          "name": { "$first": "$actors.name" },
          "birthyear": { "$first": "$actors.birthyear" },
          "deathyear": { "$first": "$actors.deathyear" },
          "movies": { "$push": "$$ROOT" }
        }
      },
      {
        "$match": {
          "$and": [
            { "name": { "$regex": "^Phi" } },
            { "movies.startyear": { "$not": { "$eq": 2014 } } },
            { "deathyear": { "$eq": null } }
          ]
        }
      },
      { "$project": { "_id": true, "name": true } }
    ]
  }
```

This plan used a collection scan, and then proceeded to compute all of the aggregated data and projections

B

```
{
  "queryPlanner": {
    "plannerVersion": 1,
    "namespace": "imdb.movies",
```

```
      "indexFilterSet": false,
      "parsedQuery": {
        "$and": [
          { "genres": { "$eq": "Talk-Show" } },
          { "startyear": { "$lte": 2017 } },
          { "producers.name": { "$regex": ".*Gill.*" } }
        ]
      },
      "queryHash": "60825D19",
      "planCacheKey": "60825D19",
      "winningPlan": {
        "stage": "COLLSCAN",
        "filter": {
          "$and": [
            { "genres": { "$eq": "Talk-Show" } },
            { "startyear": { "$lte": 2017 } },
            { "producers.name": { "$regex": ".*Gill.*" } }
          ]
        },
        "direction": "forward"
      },
      "rejectedPlans": []
    }
  }
```


This query started off with a collection scan, and then proceeded to the aggregation plan afterwards.

c


```
{
  "stages": [
    {
      "$cursor": {
        "query": { "type": { "$eq": "movie" } },
        "fields": { "runtime": 1, "title": 1, "writers": 1, "_id": 1 },
        "queryPlanner": {
          "plannerVersion": 1,
          "namespace": "imdb.movies",
          "indexFilterSet": false,
          "parsedQuery": { "type": { "$eq": "movie" } },
          "queryHash": "2A1623C7",
          "planCacheKey": "2A1623C7",
          "winningPlan": {
            "stage": "COLLSCAN",
            "filter": { "type": { "$eq": "movie" } },
            "direction": "forward"
          },
          "rejectedPlans": []
        }
      }
    },
```

```
      {
        "$unwind": {
          "path": "$writers",
          "preserveNullAndEmptyArrays": true,
          "includeArrayIndex": "string"
        }
      },
      {
        "$match": {
          "$and": [
            { "writers.name": { "$regex": ".*Bhardwaj.*" } },
            { "writers.deathyear": { "$not": { "$exists": true } } }
          ]
        }
      },
      {
        "$group": {
          "_id": "$writers._id",
          "name": { "$first": "$writers.name" },
          "avgRuntime": { "$avg": "$runtime" },
          "movies": {
            "$push": { "_id": "$_id", "title": "$title", "runtime":
"$runtime" }
          }
        }
      }
    ]
}
```

This plan used a collection scan, and then proceeded to compute all of the aggregated data and projections

D

```
{
  "stages": [
    {
      "$cursor": {
        "query": {
          "runtime": { "$gt": 120 },
          "producers": { "$exists": true }
        },
        "fields": { "producers": 1, "title": 1, "_id": 1 },
        "queryPlanner": {
          "plannerVersion": 1,
          "namespace": "imdb.movies",
          "indexFilterSet": false,
          "parsedQuery": {
            "$and": [
              { "runtime": { "$gt": 120 } },
              { "producers": { "$exists": true } }
```

```
            ]
          },
          "queryHash": "89B919B1",
          "planCacheKey": "89B919B1",
          "winningPlan": {
            "stage": "COLLSCAN",
            "filter": {
              "$and": [
                { "runtime": { "$gt": 120 } },
                { "producers": { "$exists": true } }
              ]
            },
            "direction": "forward"
          },
          "rejectedPlans": []
        }
      }
    },
    { "$unwind": { "path": "$producers", "preserveNullAndEmptyArrays":
true } },
    {
      "$group": {
        "_id": "$producers._id",
        "name": { "$first": "$producers.name" },
        "movies": { "$push": { "_id": "$_id", "title": "$title" } },
        "count": { "$sum": { "$const": 1 } }
      }
    },
    { "$sort": { "sortKey": { "count": -1 } } }
  ]
}
```

This plan used a collection scan, and then proceeded to compute all of the aggregated data and projections

E

```
{
  "winningPlan": {
    "stage": "COLLSCAN",
    "filter": {
      "$and": [
        { "actors.name": { "$eq": "Sigourney Weaver" } },
        { "directors.name": { "$eq": "James Cameron" } },
        { "genres": { "$eq": "Sci-Fi" } }
      ]
    },
    "direction": "forward"
  }
}
```

This query was a straightforward one - mongo did a collection scan looking at three fields specified, since none of them were indexed

Indexes

Queries C & D

I theorized that for some of the queries (C & D) - indexes would not help, since I'm starting them off using aggregated functions as the first step in the plan inside of my initial match step. In order to verify this, I decided to try and create indexes on the fields I was filtering by for Query D - the runtime field and producers documents.

First I tried to create an ascending index on runtime. `db.movies.createIndex({runtime: 1})` This did change the query plan, and the runtime... but for the worse! It switched to using an IXSCAN, but this almost doubled the query time.


```
{
  "winningPlan": {
    "stage": "FETCH",
    "filter": { "producers": { "$exists": true } },
    "inputStage": {
      "stage": "IXSCAN",
      "keyPattern": { "runtime": 1 },
      "indexName": "runtime_1",
      "isMultiKey": false,
      "multiKeyPaths": { "runtime": [] },
      "isUnique": false,
      "isSparse": false,
      "isPartial": false,
      "indexVersion": 2,
      "direction": "forward",
      "indexBounds": { "runtime": ["(120, inf.0]"] }
    }
  }
}
```


Next, I tried to create an ascending index on runtime and on producers. `db.movies.createIndex({runtime: 1, producers: 1})`. Mongos query processor chose not to use this index, and instead used the plain index on runtime, placing the plan using this new index in the rejected plans field when I ran the explain() command on the query.

This happened because with the way I wrote most of my queries, such as D - I'm not giving it an input to search for - if I was doing this, the indexes would be very useful. Instead, I'm giving it a filter, and saying `get all X with X.y > z and X.j is present`. This isn't something that I can hash and find easily in an index - I'd have to iterate over all the data anyways to see if the condititons are met for each entry. That's indexes aren't applicable for most of my queries.

The other queries I was able to optimize using indexes.

Query A

Query A begins off by filtering down the movies to this with type="movie". I decided that an index on movies.type would likely improve the runtime of this query, and created one using `db.movies.createIndex({"type": 1})`. Running the same query after adding this index showed a noticeable improvement, with an execution time of 19.2 seconds, slightly more than a 50% improvement on the original runtime.

Query B

The first step in Query B's pipeline was a match that filtered by Genre, amoung other things. I suspected that an index on Genre could improve the query time so I created one `db.movies.createIndex({"genre": 1})`. Running the same query after adding this index showed a significant improvement - it was down to 4.2s execution time, from 13.2 seconds - an improvement of 300%. Running the explain command on the query indicated that it was using an IXSCAN instead of a collection scan, using the new index that I created.

Query E

For query E, I am giving it concrete data to search. It can hash this data and use an index to locate this data very quickly.

Since all three fields I am filtering by are parallel arrays, it is not possible to create one index on all of them. I chose to create one index on actors to start, since most movies have more actors than directors, so this is the lowest hanging fruit to optimize.

```
db.movies.createIndex({'actors.name': 1})
```

The difference this made was huge - this brought the execution time down from ~14 seconds to a whopping 0 milliseconds. This is because it went from iterating over millions of records, to iterating over only the records that had Sigourney Weaver as an actor. I confirmed using the explain() command that it was indeed using this new index and an IXSCAN