MongoDB Aggregation Pipeline

Transform the documents into aggregated results

October 1, 2021

Aggregation Pipeline

What is it?

- The aggregation pipeline lets us transform data through a series of stages
- These stages can be thought of as “functions”, where some work is done
- The data gets transformed during each stage and the result is returned to the next stage in the pipeline
- One way to think about this is like woodworking. You start with a tree, and through a series of steps you get something that is useful like a chair or a table.

What problem does is solve?

- When working with data, most of the time you only need a small portion of the data you get back from your API.
- This wastes a lot of network bandwidth and CPU cycles for the front end to process and filter that data into something it can use.
- Stages can be ran individually, but doing so can create some unwieldy code.

How does it solve this problem

- The aggregation pipeline allows developers to transform a large amount of data into aggregated results tailored specifically to our use case.
- The nature of the aggregation pipeline in MongoDB is such that we can run multiple stages in order, passing the result to the next stage
- MongoDB will intelligently prioritize these steps for optimization, saving us network bandwidth and making APIs that use our database more efficient.

How can we use this in the real world?

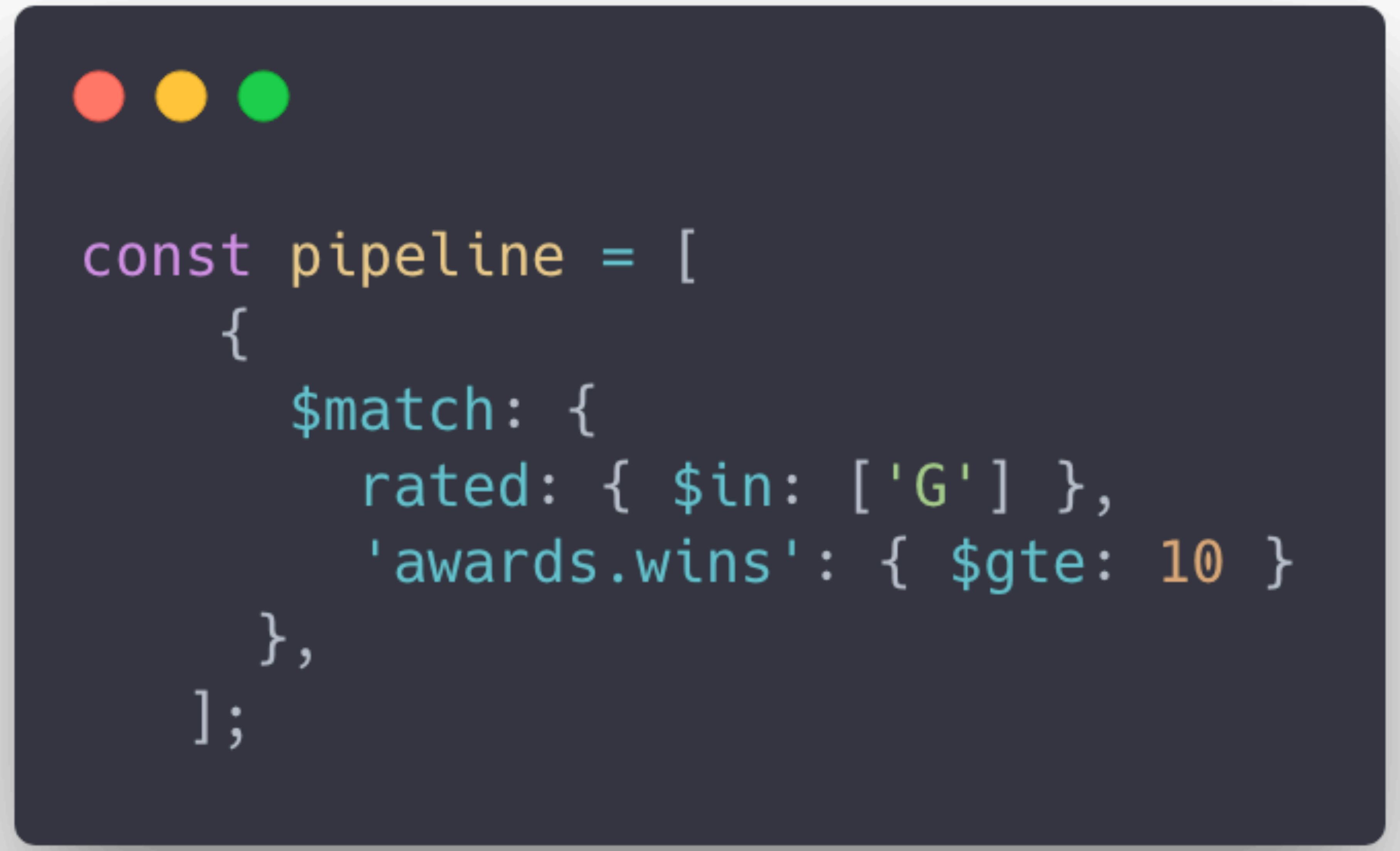
Student activity

- In the upcoming activity, you will finish a partially completed aggregation pipeline to help people find out what movie they should watch.
- Imagine you have a database of movies and you have very specific taste.
- Wouldn't it be cool if you could grab a list of all the best movies in the last 10 years sorted by their metacritic score?
- In order to jump into this activity, lets go over some of the stages that you might want to use complete the activity.

Stage 1 - $match

$match

- Filters documents based on specific criteria.
- In this example, we want only movies that have a rating equal to `G` using the $in operator. This checks for equality.
- We also use the $gte operator to match movies that won at least 10 awards

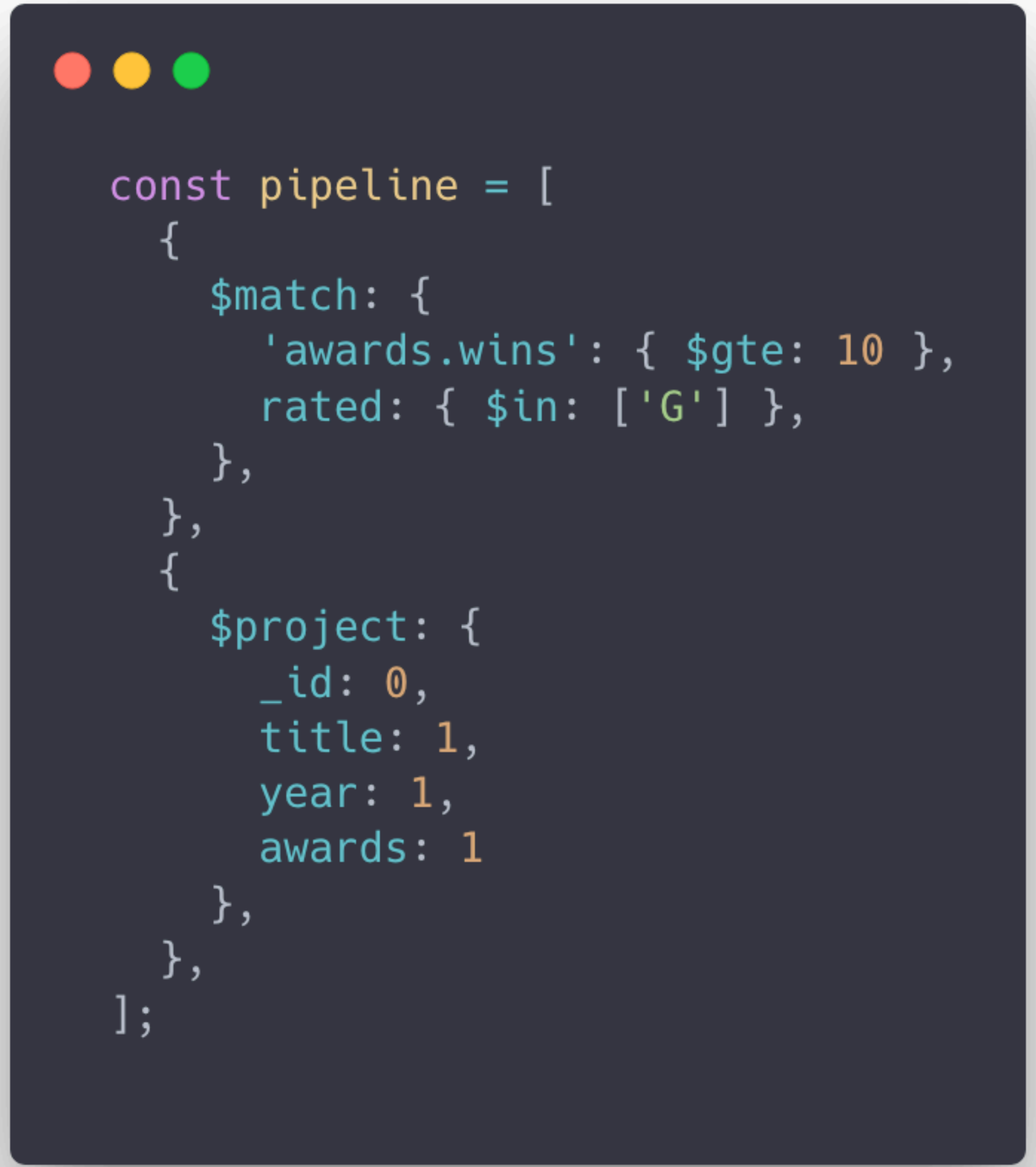


```
const pipeline = [
    {
      $match: {
        rated: { $in: ['G'] },
        'awards.wins': { $gte: 10 }
      },
    },
  ];
```

Stage 2 - $project

$project

- Specifies fields should and/or shouldn't be passed along to the next stage
- In this example, we don't care about the _id field, but we do care about the title, the year and the awards.
- Each field can be set to either 0 meaning we don't want to include, and 1 to represent true, or that we want to include

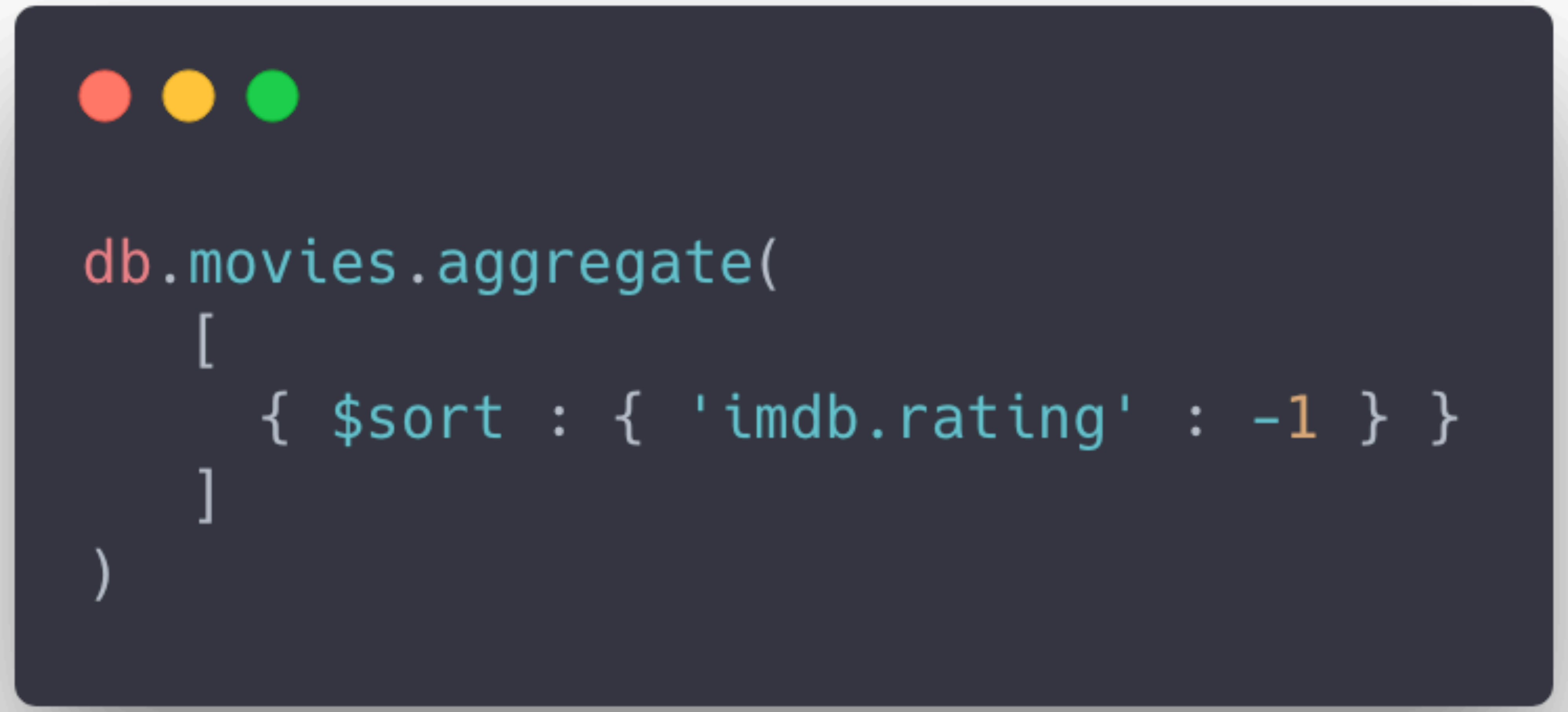


```
const pipeline = [
  {
    $match: {
      'awards.wins': { $gte: 10 },
      rated: { $in: ['G'] },
    },
  },
  {
    $project: {
      _id: 0,
      title: 1,
      year: 1,
      awards: 1
    },
  },
];
```

Stage 3 - $sort

$sort

- Selects all the input documents and sorts them in ascending or descending order
- The -1 represents sort in descending order.
- Conversely, the 1 represents sorting in ascending order

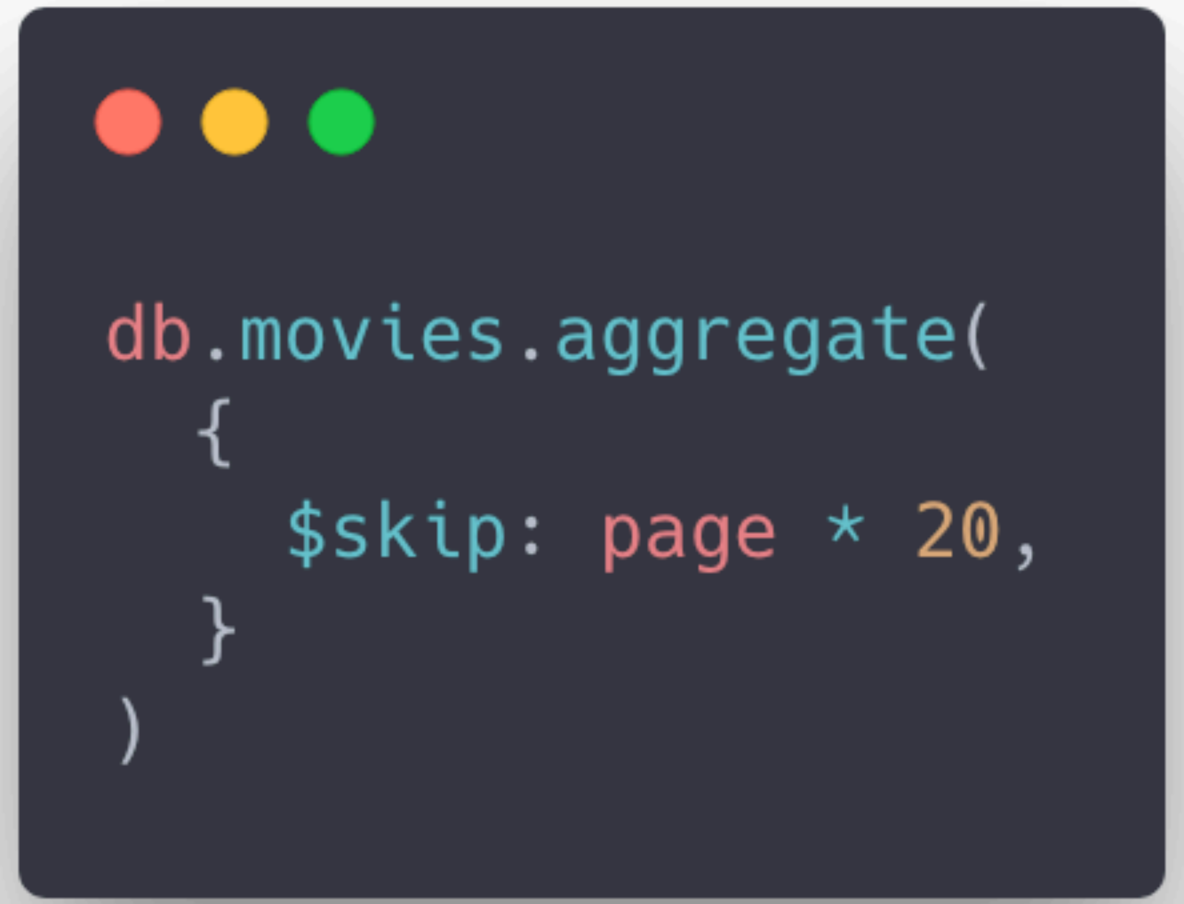


```
db.movies.aggregate(
    [
      { $sort : { 'imdb.rating' : -1 } }
    ]
)
```

Stage 4 - $skip

$skip

- Skips over the specified number of documents that pass into the stage and passes the remaining documents to the next stage in the pipeline.
- This is used in the activity to emulate pagination functionality

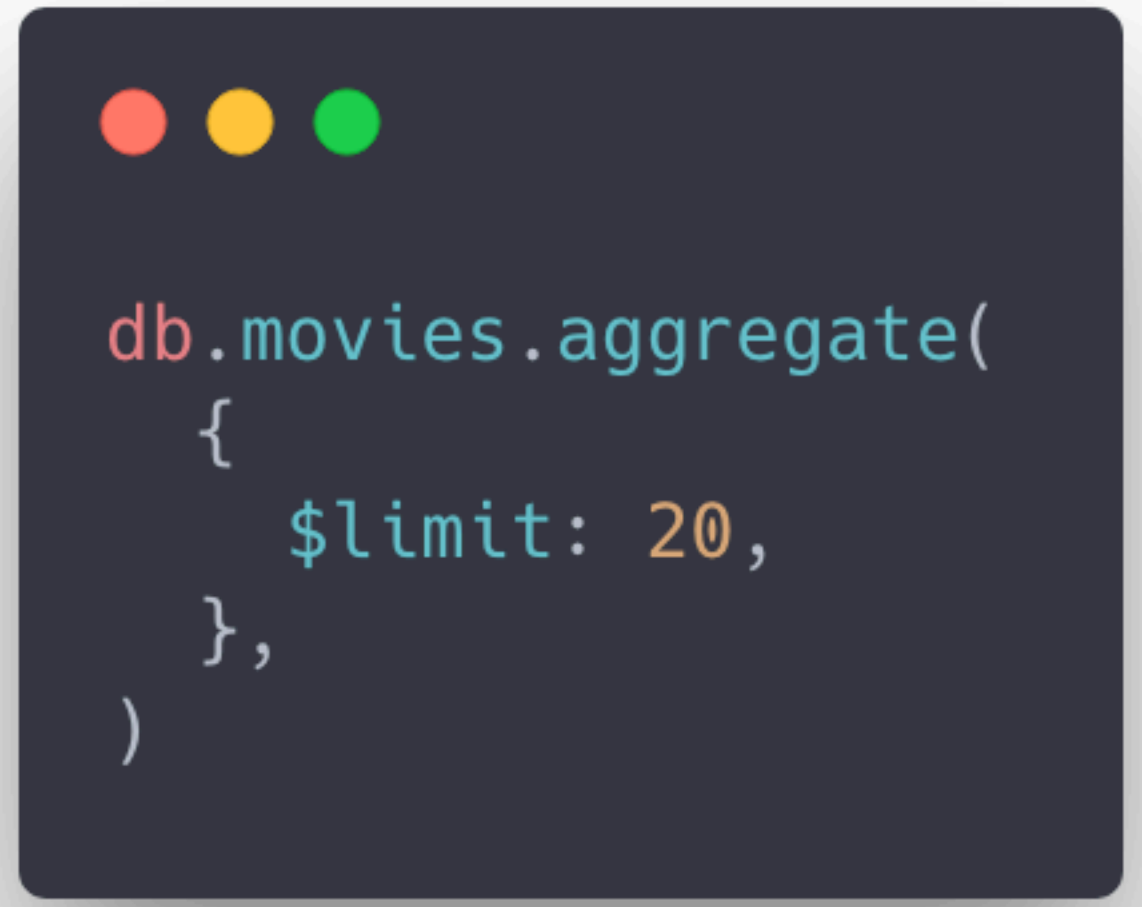


```
db.movies.aggregate(
  {
    $skip: page * 20,
  }
)
```

Stage 5 - $limit

$limit

- Limits the number of documents that are passed to the next stage in the pipeline
- When using this with $sort, make sure you have at least one field that contains unique values to allow it to function properly



```
db.movies.aggregate(
  {
    $limit: 20,
  },
)
```

Activity time!