Aggregation pipeline

Aggregation pipeline allows you to group,sort,perform calculations,analyze data and much more

Aggregation pipeline can have one or more stages .The order of these stages are important .Each stages acts upon the result of previous stage

Agendas:

- **Execute Aggregation Pipeline and its operations (pipeline must contain $match, $group, $sort, $project,**
- **$skip etc. students encourage to execute several queries to demonstrate various aggregation operators)**

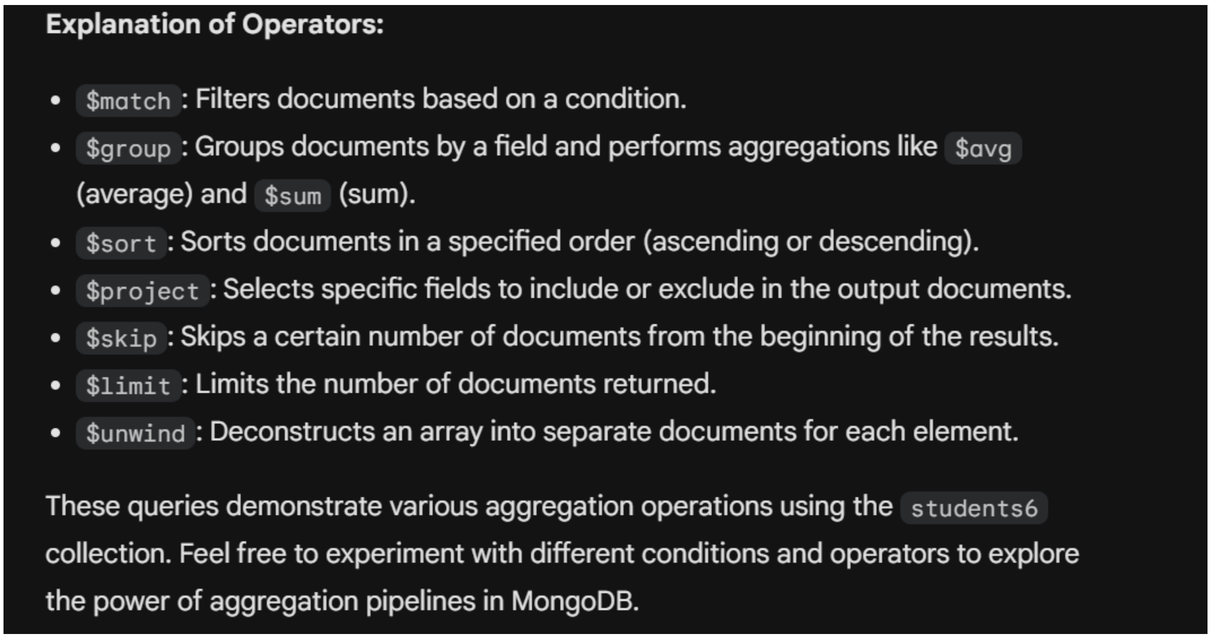
Next step we should build a new dataset and import "students6"

First download the file or image of students 6

Later import it.

- Download collection here
- Upload the new collection with name "students6"

Lets see the explaination of each commands present here:



Explanation of Operators:

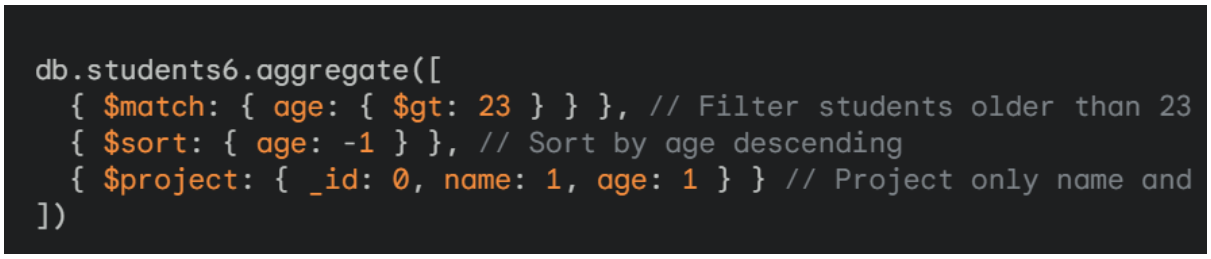
- `$match`: Filters documents based on a condition.
- `$group`: Groups documents by a field and performs aggregations like `$avg` (average) and `$sum` (sum).
- `$sort`: Sorts documents in a specified order (ascending or descending).
- `$project`: Selects specific fields to include or exclude in the output documents.
- `$skip`: Skips a certain number of documents from the beginning of the results.
- `$limit`: Limits the number of documents returned.
- `$unwind`: Deconstructs an array into separate documents for each element.

These queries demonstrate various aggregation operations using the `students6` collection. Feel free to experiment with different conditions and operators to explore the power of aggregation pipelines in MongoDB.

1 program:

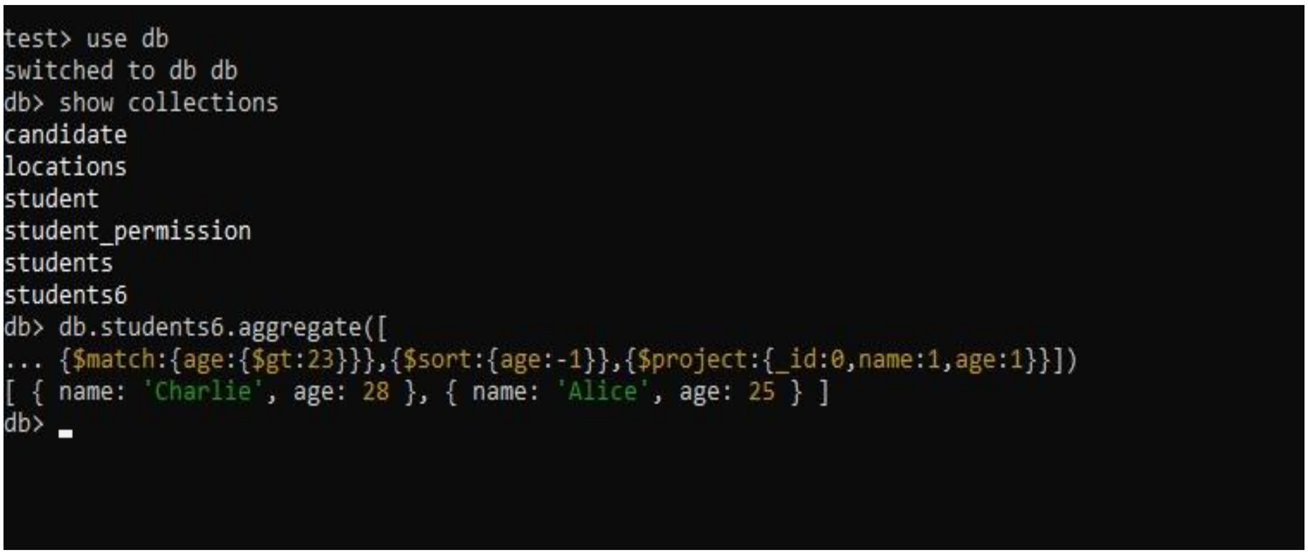
1. Find students with age greater than 23, sorted by age in descending order, and only return name and age

Input:



```
db.students6.aggregate([
  { $match: { age: { $gt: 23 } } }, // Filter students older than 23
  { $sort: { age: -1 } }, // Sort by age descending
  { $project: { _id: 0, name: 1, age: 1 } } // Project only name and
])
```

Output:



```
test> use db
switched to db db
db> show collections
candidate
locations
student
student_permission
students
students6
db> db.students6.aggregate([
... {$match:{age:{$gt:23}}},{$sort:{age:-1}},{$project:{_id:0,name:1,age:1}}])
[ { name: 'Charlie', age: 28 }, { name: 'Alice', age: 25 } ]
db> _
```

Explanation:

It appears to be a MongoDB aggregation pipeline that calculates the average score of students and then filters the results to show only students with an average score greater than 85, skipping the first result. Let's break down the code step by step:

db.students6.aggregate([...]): This line uses the aggregate function on the students6 collection. The aggregate function allows you to perform complex data manipulations on collections in MongoDB.

{$project: { ... }}: The first stage of the pipeline uses the $project operator. The $project operator allows you to specify which fields you want to include or exclude from the output documents. In this case, the $project operator is being used to:

Create a new field called averageScore by using the $avg aggregation operator on the scores field. The $avg operator calculates the average of an array of values.

{$match: {averageScore: {$gt: 85}}}: The second stage of the pipeline uses the $match operator. The $match operator allows you to filter the output documents based on a specified condition. In this case, the $match operator is being used to filter the documents so that only documents where the averageScore is greater than 85 are included in the output.

{$skip: 1}: The third stage of the pipeline uses the $skip operator. The $skip operator allows you to skip a certain number of documents at the beginning of the output. In this case, the $skip operator is being used to skip the first document in the output.

So, when you run this aggregation pipeline, it will first calculate the average score for each student in the students6 collection. Then, it will filter the results to only include students with an average score greater than 85. Finally, it will skip the first document in the filtered results and return the remaining documents.

In the output you provided, it only shows one document after the pipeline is run, which means there was only one student in the students6 collection with an average score greater than 85 after skipping the first result. The student's name is David and their average score is 93.33333333333333.

Sure, the code snippet you provided is a MongoDB aggregation pipeline that filters and sorts student documents based on their age. Let's break down the pipeline step by step:

db.students.aggregate([...]): This line utilizes the aggregate function on the students collection to perform a series of data transformations.

{$match:{age:{$gt:23}}}: This initial stage filters documents using the $match operator. The $match operator filters documents based on a specified condition. In this case, it selects documents where the field age is greater than ($gt) 23.

{$sort:{age:-1}}: The $sort operator rearranges the output documents based on a specified field. Here, the $sort operator sorts documents in descending order (-1) according to the age field. So, documents with higher ages will appear first in the output.

{$project:{_id:0,name:1,age:1}}: This final stage employs the $project operator to specify the output document structure. The $project operator allows you to include or exclude fields from the output documents. In this case, it's configured to:

Exclude the _id field by setting it to 0. This removes the object identifier from the output.

Include the name field by setting it to 1.

Include the age field by setting it to 1.

The output showcases two documents:

{ name: "Charlie", age: 28 }

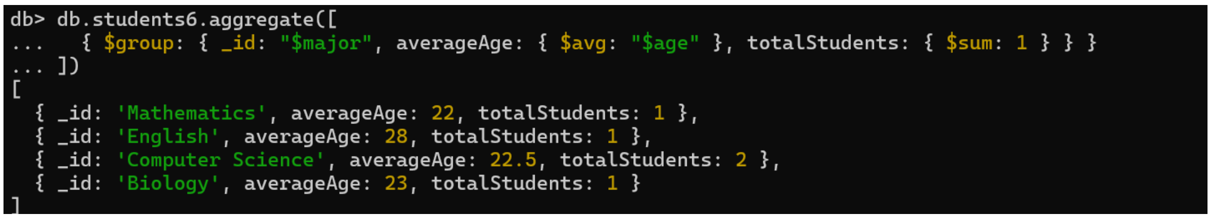
{ name: "Alice", age: 25 }

Since the pipeline sorts by age in descending order, Charlie, who has a higher age (28), is listed first.

2program:

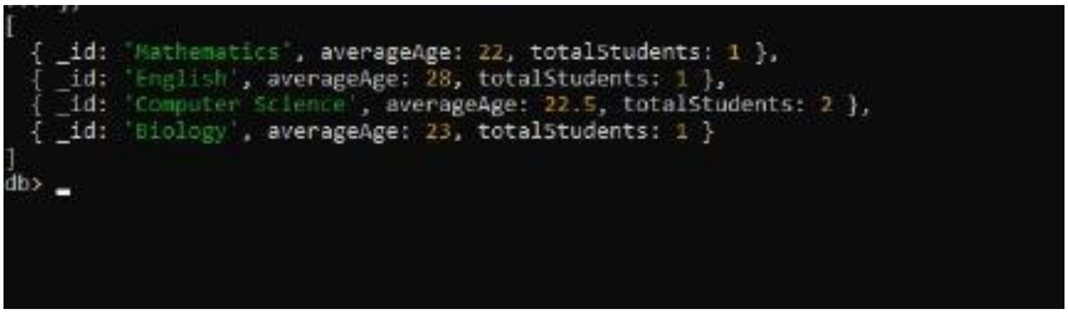
1. **B. Find students with age less than 23, sorted by name in ascending order, and only return name and score**

Input:



```
db> db.students6.aggregate([
...   { $group: { _id: "$major", averageAge: { $avg: "$age" }, totalStudents: { $sum: 1 } } }
... ])
[
  { _id: 'Mathematics', averageAge: 22, totalStudents: 1 },
  { _id: 'English', averageAge: 28, totalStudents: 1 },
  { _id: 'Computer Science', averageAge: 22.5, totalStudents: 2 },
  { _id: 'Biology', averageAge: 23, totalStudents: 1 }
]
```

Output:



```
[
  { _id: 'Mathematics', averageAge: 22, totalStudents: 1 },
  { _id: 'English', averageAge: 28, totalStudents: 1 },
  { _id: 'Computer Science', averageAge: 22.5, totalStudents: 2 },
  { _id: 'Biology', averageAge: 23, totalStudents: 1 }
]
db> _
```

Explanation:
The code in the image appears to be written in JavaScript and uses MongoDB aggregation framework. It's likely querying a database called "students6" to find average student age and total students enrolled in each subject.
Here's a breakdown of the code:

db.students6.aggregate - This line initiates the aggregation operation on the "students6" collection in the database.

`[$group: {...}] - This defines a stage in the aggregation pipeline where documents are grouped together based on a specified field. In this case the documents are grouped by their "id" field.

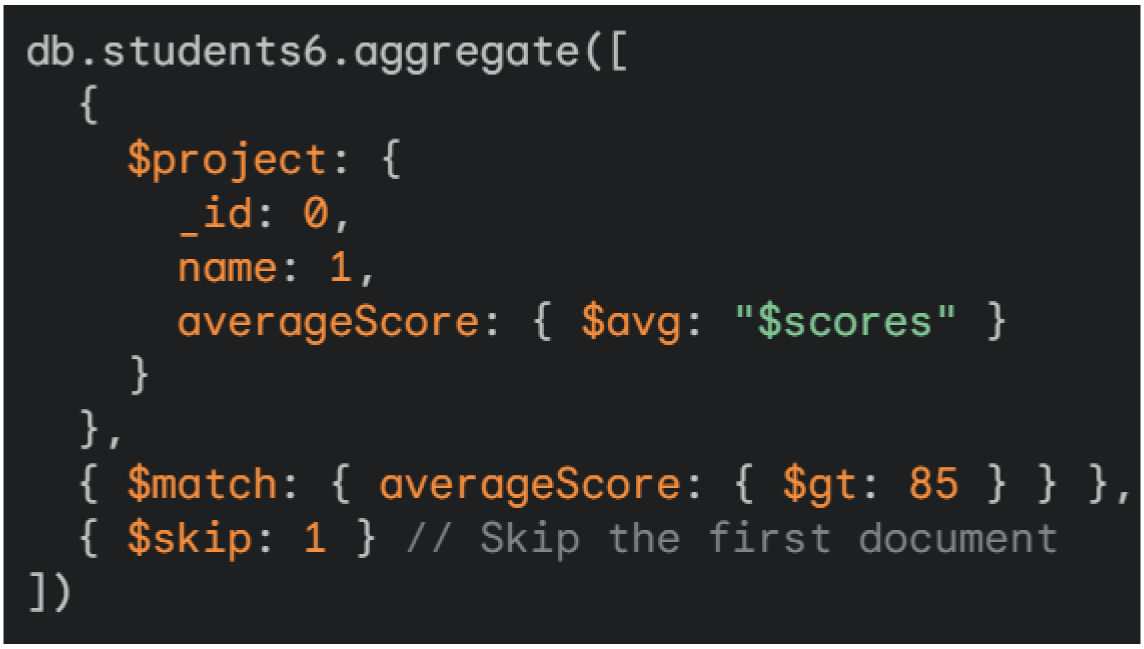
id: "Snajor", averageAge: {$avg: "Sage"}, totalstudents: {$sum: 1}} - This line defines the group documents. Here, "id" is set to

“Snajor”, “averageAge” is calculated using the aggregation operator$avgon the field “Sage”, and “totalstudents” is the sum of all the documents in the group using the aggregation operator$sum”.

The lines following 1 db> appear to be the output of the query, potentially showing average age and total students enrolled in four subjects: “Mathematics”, “English”, “Computer Science” and “Biology”.

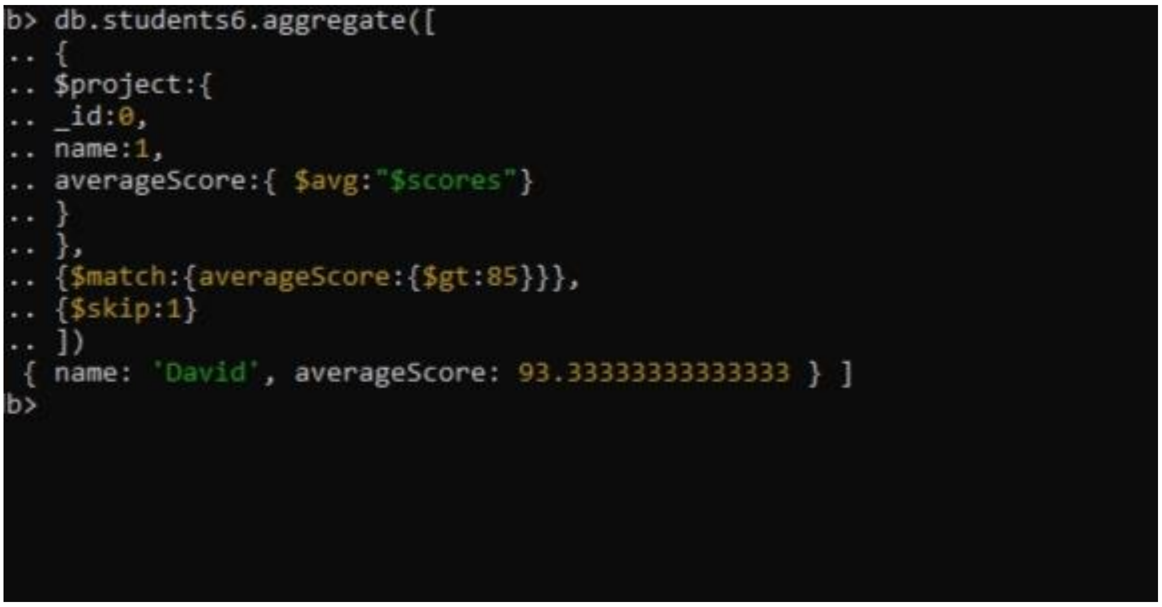
Program 3:
Find students with an average score (from scores array) above 85 and skip the first document

Input:



```
db.students6.aggregate([
  {
    $project: {
      _id: 0,
      name: 1,
      averageScore: { $avg: "$scores" }
    }
  },
  { $match: { averageScore: { $gt: 85 } } },
  { $skip: 1 } // Skip the first document
])
```

Output:



```
b> db.students6.aggregate([
.. {
.. $project:{
.. _id:0,
.. name:1,
.. averageScore:{ $avg:"$scores"}
.. }
.. },
.. {$match:{averageScore:{$gt:85}}},
.. {$skip:1}
.. ])
 { name: 'David', averageScore: 93.33333333333333 } ]
b>
```

explaination:

The code you sent appears to be a MongoDB aggregation pipeline that calculates the average score of students and then filters the results to show only students with an average score greater than 85, skipping the first result. Let's break down the code step by step:

db.students6.aggregate([...]): This line uses the aggregate function on the students6 collection. The aggregate function allows you to perform complex data manipulations on collections in MongoDB.

{$project: { ... }}: The first stage of the pipeline uses the $project operator. The $project operator allows you to specify which fields you want to include or exclude from the output documents. In this case, the $project operator is being used to:

Exclude the _id field by setting it to 0.

Include the name field by setting it to 1.

Create a new field called averageScore by using the $avg aggregation operator on the scores field. The $avg operator calculates the average of an array of values.

{$match: {averageScore: {$gt: 85}}}: The second stage of the pipeline uses the $match operator. The $match operator allows you to filter the output documents based on a specified condition. In this case, the $match operator is being used to filter the documents so that only documents where the averageScore is greater than 85 are included in the output.

{$skip: 1}: The third stage of the pipeline uses the $skip operator. The $skip operator allows you to skip a certain number of documents at the beginning of the output. In this case, the $skip operator is being used to skip the first document in the output.

So, when you run this aggregation pipeline, it will first calculate the average score for each student in the students6 collection. Then, it will filter the results to only include students with an average score greater than 85. Finally, it will skip the first document in the filtered results and return the remaining documents.

In the output you provided, it only shows one document after the pipeline is run, which means there was only one student in the students6 collection with an average score greater than 85 after skipping the first result. The student's name is David and their average score is 93.33333333333333.