Experiment:6

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)

AGGREGATE PIPELINES:

An aggregate pipeline in MongoDB is a framework that allows for the processing and transformation of documents within a collection through a sequence of stages. Each stage in the pipeline performs a specific operation on the input documents and passes the results to the next stage. This enables complex data aggregation tasks such as filtering, grouping, sorting, reshaping, and calculating aggregate values like sums and averages. The stages are processed in a defined order, and MongoDB's aggregation framework provides a wide range of operators to handle various data transformation needs. By leveraging aggregate pipelines, developers can efficiently perform data analysis and reporting within the database, reducing the need for multiple queries and enhancing performance.

AGENDA:

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

LET'S BUILD NEW DATABASE:

Upload the new collection with name "students6".

```
_id: 4
name : "David"
age : 20
major : "Computer Science"
▼ scores : Array (3)
    0: 98
    1: 95
    2: 87
```

EXPLAINATION OF OPERATORS:

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.

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

To retrieve students older than 23, sorted by age in descending order, and displaying only their names and ages in MongoDB, you can utilize an aggregate pipeline. This involves filtering the documents using the **`$match`** stage to select students with an age greater than 23, sorting the results in descending order using the **`$sort`** stage, and specifying the fields to return using the **`$project`** stage to include only the `name` and `age` fields. The resulting query effectively filters, sorts, and projects the necessary information in one efficient operation.

```
test> use db
switched to db db
db> show collections
candidates
foo
locations
players
students
students_permission
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>
```

To find students older than 23, sorted by age in descending order, and displaying only 3 their names and ages from the `students6` collection in MongoDB, you can use the following **aggregate pipeline**. First, the **`$match`** stage filters the documents to include only those with an age greater than 23. Next, the **`$sort`** stage arranges the filtered results by age in descending order. Finally, the **`$project`** stage specifies that only the `name` and `age` fields should be included in the output, excluding the `_id` field. Executing this pipeline will return documents such as **`{ name: 'Charlie', age: 28 }`** and **`{ name: 'Alice', age: 25 }`,** effectively meeting the query requirements in a single, streamlined operation.

Find students with age less than 23, sorted by age in ascending order, and only return name and score:

To find students under the age of 23, sorted by their ages in ascending order, and display only their names and scores in MongoDB, you would structure a query to filter for students where their age meets the criteria, sort these students by their age from youngest to oldest, and then specify to retrieve only their names and scores. This ensures you get a clear list of younger students with their respective academic scores, without including unnecessary details like identifiers or other fields.

```
db> db.students6.aggregate([
... {$match:{age:{$lt:23}}},
... {$sort:{age:1}},
... {$project:{_id:0,name:1,scores:1}}
... ])
[
  { name: 'David', scores: [ 98, 95, 87 ] },
  { name: 'Bob', scores: [ 90, 88, 95 ] }
]
db>
```

To find students younger than 23, sorted by name in ascending order, and return only their names and scores, you can use a MongoDB **aggregate pipeline**. First, the **`$match`** stage filters the documents to include only those with an age less than 23. Next, the **`$sort`** stage arranges these filtered results by name in ascending order. Finally, the **`$project`** stage specifies that only the `name` and `scores` fields should be included in the output, excluding the `_id` field. This query ensures that the resulting documents are neatly organized and include only the relevant information, such as **`{ name: 'Bob', scores: [90, 88, 95] }` and `{ name: 'David', scores: [98, 95, 87] }`.**

Group students by major, calculate average age and total number of students in each major:

To group students by their major, calculate the average age, and count the total number of students in each major using **MongoDB's aggregate framework**, you can construct a pipeline with two main stages. First, the `$group` stage aggregates documents based on the `major` field, computing the average age using `$avg` and counting the students using **`$sum: 1`.** The `_id` in the **`$group`** stage specifies the field to group by, in this case, `major`. The second stage, **`$project`**, reshapes the output to include fields like **`major`, `averageAge`, and `totalStudents`**, using **`$project`** to exclude the `_id` field and rename fields as needed. This pipeline efficiently summarizes student data by major, providing insights into the average age and the total number of students for each major represented in the dataset.

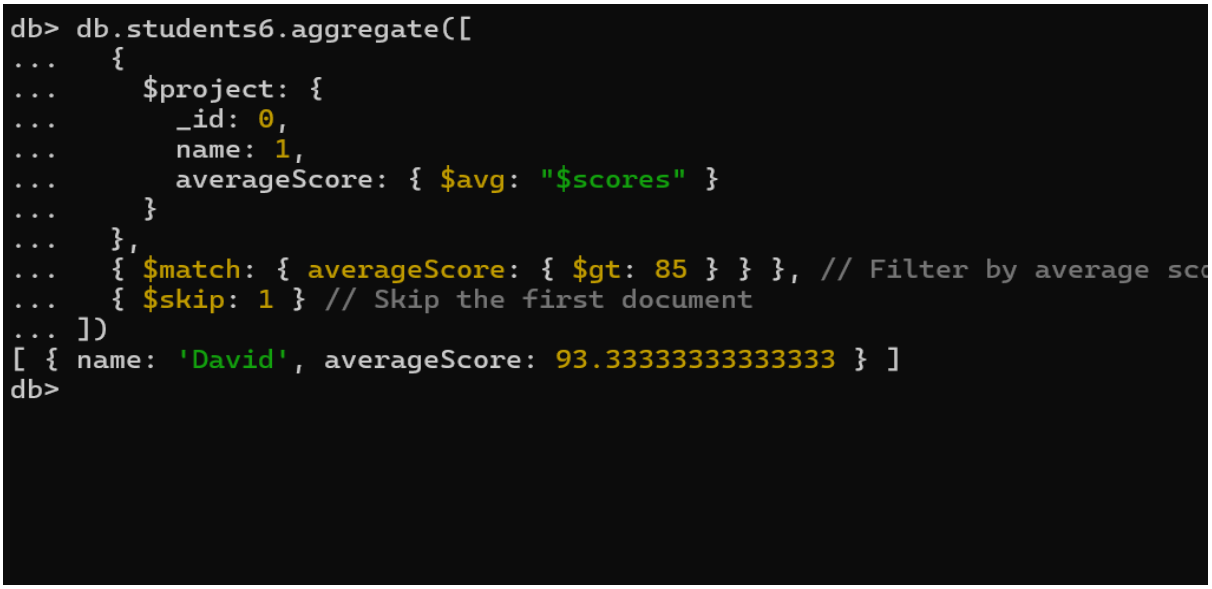
```
db> db.students6.aggregate([
...   { $group: { _id: "$major", averageAge: { $avg: "$age" }, totalStudents: { $sum: 1 } } }
... ])
[
  { _id: 'Mathematics', averageAge: 22, totalStudents: 1 },
  { _id: 'English', averageAge: 28, totalStudents: 1 },
  { _id: 'Computer Science', averageAge: 22.5, totalStudents: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

This MongoDB aggregation query groups students by their majors and computes two main statistics for each major. The `_id` field represents the major name, while **`averageAge`** calculates the average age of students within each major. For instance, students in English have an average age of 28, those in Mathematics average 22 years old, and Computer Science students average 22.5 years. Additionally, **`totalStudents` counts** how many students belong to each major; for example, there's one student in English, one in Mathematics, two in Computer Science, and one in Biology. This aggregation provides a concise summary of student demographics across different majors, showing both average age and total enrollment per major in the dataset.

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

In MongoDB, to find students whose average score (computed from their `scores` array) 5 is above 85 and skip the first matching document, you can construct an aggregate pipeline. First, the **`$group` stage** groups documents by student names (`_id: "$name"`) and computes the average score using **`$avg: "$scores"`**. Next, the **`$match`** stage filters these groups to include only those with an **`averageScore` greater than 85**. Lastly, the **`$skip`** stage skips the first document that meets the criteria, allowing you to start the results from the

second matching document onward. This pipeline efficiently identifies and skips over the initial result, focusing on students whose average scores exceed 85, providing a streamlined way to query and analyze student performance data in MongoDB.

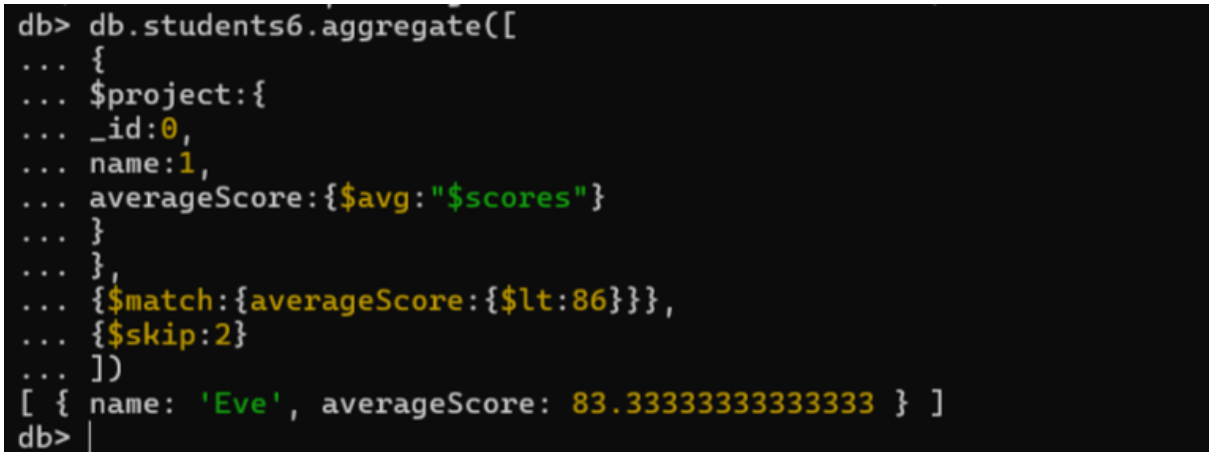


This MongoDB aggregate pipeline begins by projecting fields to reshape the output, specifically including only the `name` and computing the `averageScore` from the `scores` array for each student. After projecting, the pipeline moves to the **`$match` stage**, filtering to include only students whose computed `averageScore` is greater than 85. In this case, David is the only student who meets this criterion, with an `averageScore` of approximately 93.33. Finally, the **`$skip` stage** skips the first result, ensuring that the output starts from the second document onward. This pipeline effectively identifies and excludes the initial match, focusing on students who excel with average scores exceeding 85, providing a concise summary of high-performing students in the dataset.

Find students with an average score (from scores array) below 86 and skip the first 2 documents:

In MongoDB, the aggregate pipeline to find students whose average score from their `scores` array is below 86 and skip the first two matching documents begins with a **`$project`** 6 stage. Here, the pipeline reshapes the output to include only the `name` field and calculates the `averageScore` using **`$avg: "$scores"`**. Following this, the pipeline moves to a `$match` stage, filtering students based on their computed `averageScore` being less than 86. After filtering, the **`$skip`** stage is applied to skip the first two documents that meet the criteria,

ensuring that the query starts from the third document onward in the dataset. This pipeline effectively identifies and excludes the initial matches, focusing on students with below-average scores as per the specified threshold, providing a streamlined approach to analyzing student performance data in MongoDB.



This MongoDB aggregate pipeline first uses the **`$project` stage** to reshape the output, focusing on the `name` field and calculating the `averageScore` using `$avg: "$scores"` for each student in the `students6` collection. After projecting, the pipeline proceeds to the **`$match`** stage, filtering students whose computed `averageScore` is below 86. In this case, only Eve meets this criterion, with an `averageScore` of approximately 83.33. Finally, the **`$skip`** stage skips the first two matching documents, ensuring that the output starts from the third document onward. This pipeline effectively identifies and excludes the initial matches, providing a concise summary of students with below-average scores below 86 in the dataset.