7.AGGREGATION PIPELINE

Aggregation pipeline:

The aggregation pipeline is a framework for data aggregation in MongoDB, modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into aggregated results. The pipeline provides a powerful way to manipulate and transform data, allowing complex queries to be constructed through a series of stages.

Each stage in the pipeline performs an operation on the input documents, passing the results to the next stage. This allows for complex data transformations and aggregations to be composed through a sequence of simple operations.

Operations are:

- $Match
- $group
- $project
- $sort
- $limit
- $skip
- $unwind
- $lookup

Syntax for $match:

```
{ $match: { field: value } }
```

Syntax for $group:

```
{ $group: { _id: "$field", total: { $sum: "$anotherField" } } }
```

Syntax for $project:

```
{ $project: { field1: 1, field2: 1, computedField: { $sum: ["$field1", "$field2"] } } }
```

Syntax for $sort:

```
{ $sort: { field: 1 } } // 1 for ascending, -1 for descending
```

Syntax for $limit:

```
{ $limit: 5 }
```

Syntax for $skip:

```
{ $skip: 5 }
```

Syntax for $unwind:

```
{ $unwind: "$arrayField" }
```

Syntax for $lookup:

```
{ $lookup: { from: "otherCollection", localField: "fieldFromInput", foreignField: "fieldFromOtherCollection", as: "outputArrayField" } }
```

Example Aggregation Pipeline:

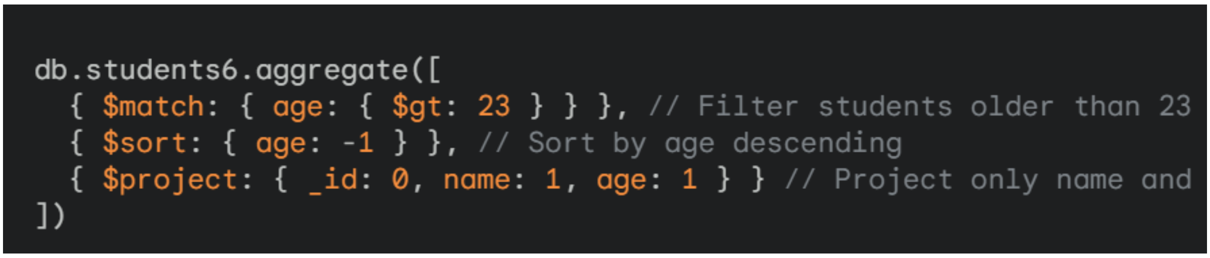
```
db.orders.aggregate([ { $match: { status: "completed" } }, // Stage 1: Filter completed orders
```

{ $group: { _id: "$customerId", totalSales: { $sum: "$amount" } } }, // Stage 2: Group by customerId and calculate total sales

{ $sort: { totalSales: -1 } }, // Stage 3: Sort by total sales in descending order

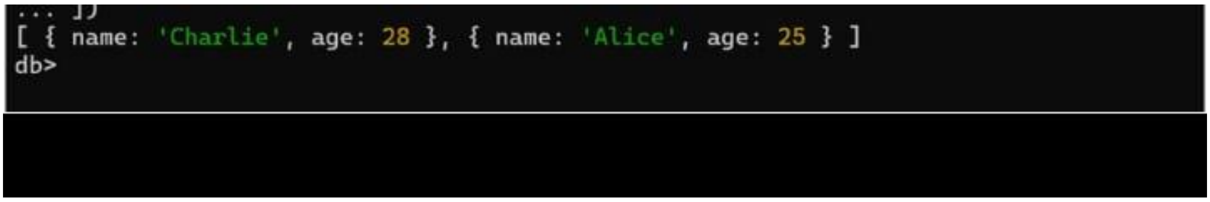
{ $limit: 10 } // Stage 4: Limit to the top 10 customers])

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



```
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:



```
... ])
[ { name: 'Charlie', age: 28 }, { name: 'Alice', age: 25 } ]
db>
```

Explaination:

This code is an example of a MongoDB aggregation pipeline applied to a collection named `students6`. It demonstrates how to filter, sort, and project documents within this collection. Here's an explanation of each stage in the pipeline:

1. **$match**: This stage filters the documents to include only those where the `age` field is greater than 23. The `$gt` operator stands for "greater than".

{ $match: { age: { $gt: 23 } } } // Filter students older than 23

$sort: This stage sorts the resulting documents by the `age` field in descending order. The `-1` indicates descending order.

{ $sort: { age: -1 } } // Sort by age descending

{ $project: { _id: 0, name: 1, age: 1 } } // Project only name and age

Here is the complete aggregation pipeline:

db.students6.aggregate([{ $match: { age: { $gt: 23 } } }, // Stage 1: Filter students older than 23

{ $sort: { age: -1 } }, // Stage 2: Sort by age in descending order

{ $project: { _id: 0, name: 1, age: 1 } } // Stage 3: Project only the name and age fields])

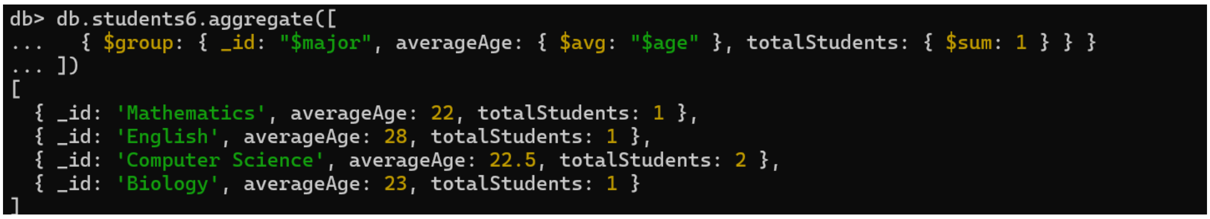
Breakdown of the Execution:

1. **Stage 1**: The `$match` stage filters out students who are 23 years old or younger. Only students older than 23 are passed to the next stage.
2. **Stage 2**: The `$sort` stage orders these filtered students by their age, from the oldest to the youngest.
3. **Stage 3**: The `$project` stage restructures the documents to include only the `name` and `age` fields, omitting the `_id` field.

The result of this aggregation pipeline will be a list of students older than 23, sorted by their age in descending order, with only their names and ages included in the output.

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

Input and Output:



```
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 }
]
```

Explaination:

This code is another example of a MongoDB aggregation pipeline applied to the `students6` collection. It demonstrates how to group documents by a specific field and calculate aggregate values for each group. Here's an explanation of the code:

1. **$group**: This stage groups the documents by the `major` field and calculates two aggregate values for each group:
   - `averageAge`: The average age of students in each major.
   - `totalStudents`: The total number of students in each major.

   { $group: { _id: "$major", // Group by the 'major' field

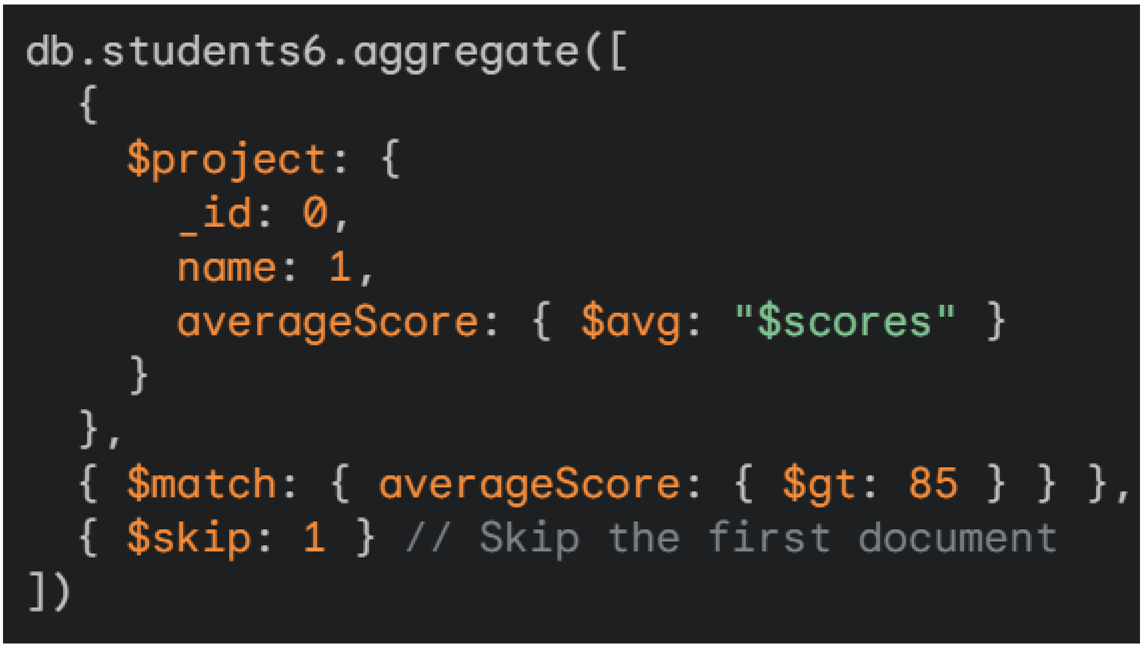
   averageAge: { $avg: "$age" }, // Calculate the average age of students in each major totalStudents: { $sum: 1 } // Count the total number of students in each major } }

Breakdown of the Execution:

1. **Grouping**: The `$group` stage groups the documents by the value of the `major` field. Each unique value of `major` becomes a group.
2. **Calculating `averageAge`**: For each group, the `averageAge` field is calculated using the `$avg` operator, which computes the average of the `age` field within the group.

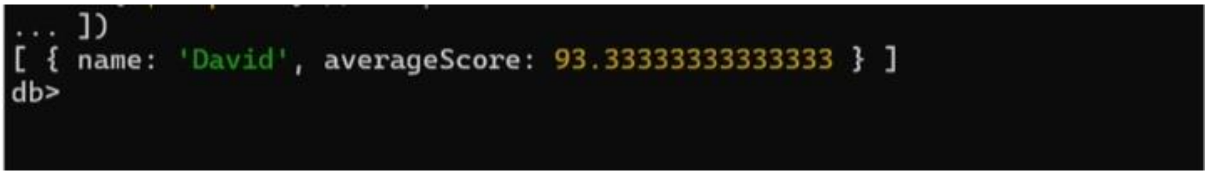
3. **Counting `totalStudents`**: For each group, the `totalStudents` field is calculated using the `$sum` operator, which increments by 1 for each document in the group, effectively counting the number of documents (students) in each group.

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



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

Output:



```
... ])
[ { name: 'David', averageScore: 93.33333333333333 } ]
db>
```

Explaination:

It demonstrates how to project certain fields, filter documents based on computed values, and skip a specified number of documents. Here's an explanation of each stage:

1. **$project**: This stage reshapes each document to include only the `name` and a computed field `averageScore`, while excluding the `_id` field.

   ```
   $project: {
     _id: 0,  // Exclude the _id field
     name: 1,  // Include the name field
     averageScore: { $avg: "$scores" }  // Compute
   the average of the scores array and include it as
   averageScore
     }
   }
   ```

2. **$match**: This stage filters the documents to include only those where the `averageScore` is greater than 85.

   ```
   { $match: { averageScore: { $gt: 85 } } }
   ```

3. **$skip**: This stage skips the first document in the resulting set of documents.

   ```
   { $skip: 1 }  // Skip the first document
   ```

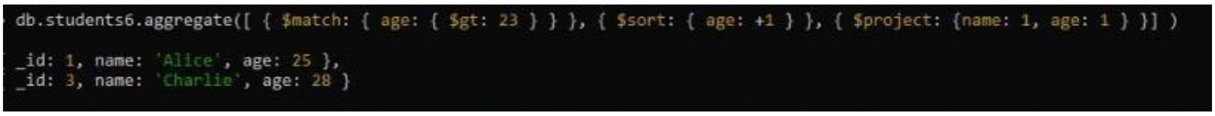
Breakdown of the Execution

1. **Stage 1 ($project)**: The `$project` stage transforms each document to:
   - Exclude the `_id` field.
   - Include the `name` field.
   - Compute the `averageScore` field by calculating the average of the values in the `scores` array for each document.
2. **Stage 2 ($match)**: The `$match` stage filters the documents to pass only those where the `averageScore` is greater than 85.

3. **Stage 3 ($skip)**: The `$skip` stage skips the first document in the filtered set of documents.

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

Input and output:



```
db.students6.aggregate([ { $match: { age: { $gt: 23 } } }, { $sort: { age: +1 } }, { $project: {name: 1, age: 1 } }] )

_id: 1, name: 'Alice', age: 25 },
_id: 3, name: 'Charlie', age: 28 }
```

Explaination:

This MongoDB code uses the `aggregate` method to perform a series of operations on the `students6` collection. Here is a step-by-step explanation of each stage in the aggregation pipeline:

1. **$match**: This stage filters the documents to include only those where the `age` field is greater than 23.
   - This selects documents where `age` is greater than 23.
2. **$sort**: This stage sorts the filtered documents by the `age` field in ascending order.
   - `age: +1` means sorting by `age` in ascending order.
3. **$project**: This stage specifies the fields to include in the output documents.
   - This includes only the `name` and `age` fields in the output documents.

Output Explanation:

The output documents are:

```
{ _id: 1, name: 'Alice', age: 25 }
{ _id: 3, name: 'Charlie', age: 28 }
```

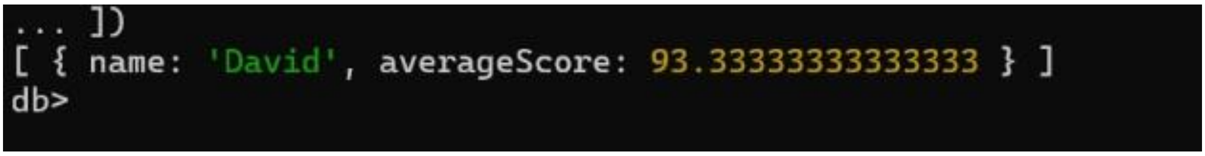
- These documents are the result of filtering (`$match`), sorting (`$sort`), and projecting (`$project`) the original documents in the `students6` collection.
- Only the documents with `age` greater than 23 are included.
- The documents are sorted by `age` in ascending order.
- Only the `name` and `age` fields are included in the final output.

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

Input:

db.students6.aggregate([{$project:{_id:0,name:1,averageScore:{$avg :”$scores”}}},{$match:{averageScore:{$gt:85}]}]){$skip:1}])

Output:



```
... ])
[ { name: 'David', averageScore: 93.33333333333333 } ]
db>
```

Explaination:

This MongoDB code snippet uses the `aggregate` method to perform a series of operations on the `students6` collection. Let's break down each stage of the aggregation pipeline:

1. **$project**: This stage creates a new document for each document in the collection. It includes only the `name` field and a new field

averageScore which is calculated as the average of the scores array.

- _id: 0 excludes the _id field from the output documents.
- name: 1 includes the name field in the output documents.
- averageScore: { $avg: "$scores" } calculates the average of the values in the scores array for each document and includes it as averageScore.

- **$match**: This stage filters the documents to include only those where the averageScore field is greater than 85.
  - This selects documents where averageScore is greater than 85.
- **$skip**: This stage skips the first document in the filtered results.
  - This skips the first document in the result set.

Output Explanation:

The output document is:

```
[{name:'David',averageScore:93.33333333333333}]
```

- This document is the result of projecting ($project), filtering ($match), and skipping ($skip) the original documents in the students6 collection.
- Only the documents with averageScore greater than 85 are included.
- The first document in the filtered results is skipped.
- The final output includes the name and averageScore fields.