CLASS 7

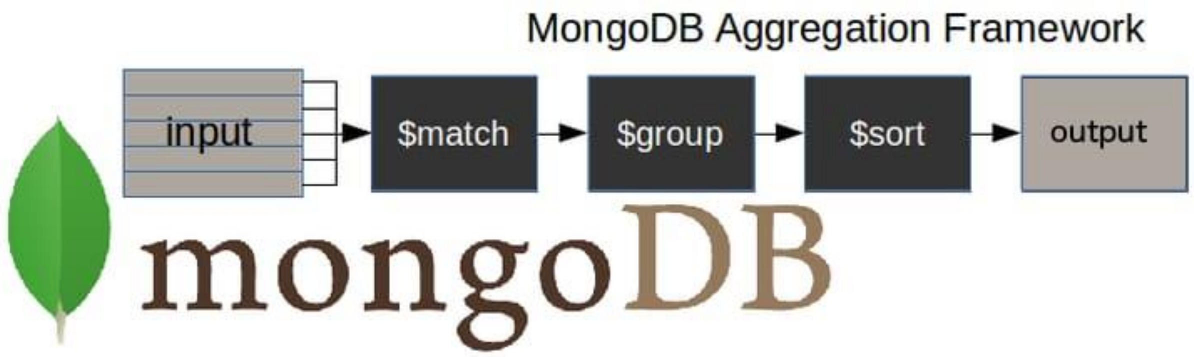
AGGREGATION PIPELINE

AGGREGATION PIPELINE:

Aggregation is a way of processing a large number of documents in a collection by means of passing them through different stages. The stages make up what is known as a pipeline. The stages in a pipeline can filter, sort, group, reshape and modify documents that pass through the pipeline.

How does the Mongodb aggregation pipeline work?

Here is a diagram to illustrate a typical MongoDB aggregation pipeline**.**



- **$match stage** – filters those documents we need to work with, those that fit our needs
- **$group stage** – does the aggregation job
- **$sort stage** – sorts the resulting documents the way we require (ascending or descending)
- **$limit:** limits the number of documents returned.
- **$skip:** Skips the first n documents and passes the remaining documents to the next stage in the pipeline.
- **$unwind:** Deconstructs an array field from the input documents to output a document for each element.
- **$lookup:** Performs a left outer join to a collection in the same database to filter in documents from the "joined" collection for processing.
- **$addFields:** Adds new fields to documents
- **$replaceRoot**: Replaces the input document with the specified embedded document.

The input of the pipeline can be a single collection, where others can be merged later down the pipeline.

The pipeline then performs successive transformations on the data until our goal is achieved.

This way, we can break down a complex query into easier stages, in each of which we complete a different operation on the data. So, by the end of the query pipeline, we will have achieved all that we wanted.

This approach allows us to check whether our query is functioning properly at every stage by examining both its input and the output. The output of each stage will be the input of the next.

Syntax:

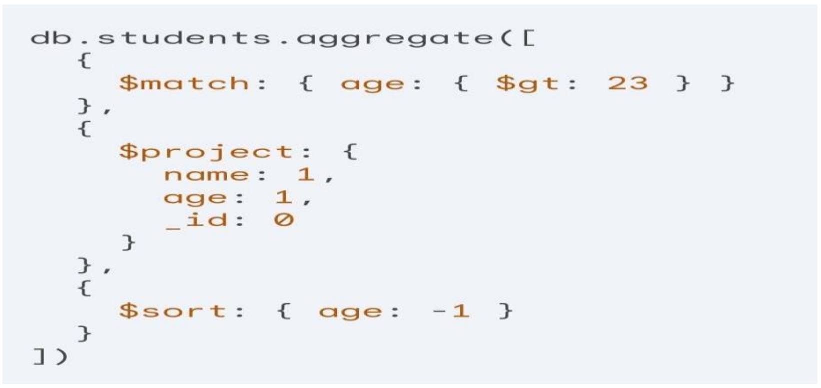
```
db.<collection>.aggregate([
  {
    <$stage1>
  },
  {
    <$stage2>
  },
  ...
])
```

In this syntax,
<collection> refers to the name of the collection on which you want to run the aggregation pipeline.
<$stage1>, <$stage2>, and so on represent the different stages of the pipeline, such as $match, $group, and $sort.
Each stage transforms the documents as they pass through the pipeline.

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

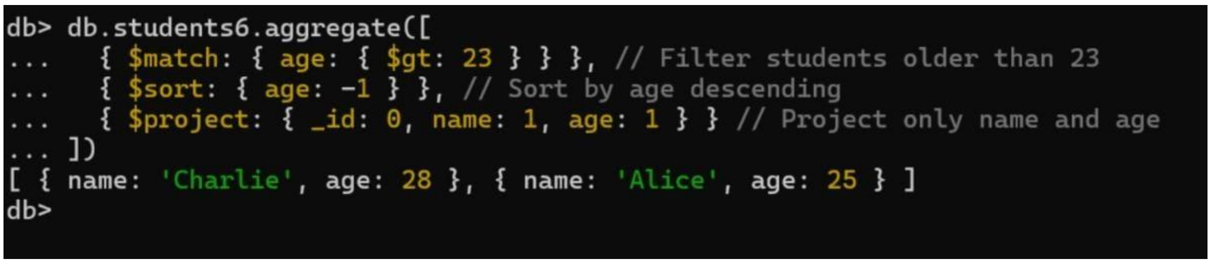
Understanding the Requirement:

- Find students with age greater than 23.
- Sort them by age in descending order.
- Return only name and age.



```
db.students.aggregate([
   {
      $match: { age: { $gt: 23 } }
   },
   {
      $project: {
         name: 1,
         age: 1,
         _id: 0
      }
   },
   {
      $sort: { age: -1 }
   }
])
```

OUTPUT:

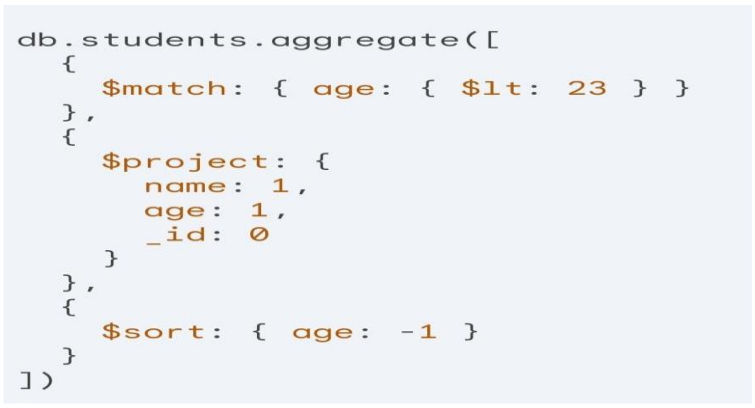


```
db> db.students6.aggregate([
...   { $match: { age: { $gt: 23 } } }, // Filter students older than 23
...   { $sort: { age: -1 } }, // Sort by age descending
...   { $project: { _id: 0, name: 1, age: 1 } } // Project only name and age
... ])
[ { name: 'Charlie', age: 28 }, { name: 'Alice', age: 25 } ]
db>
```

Explanation:

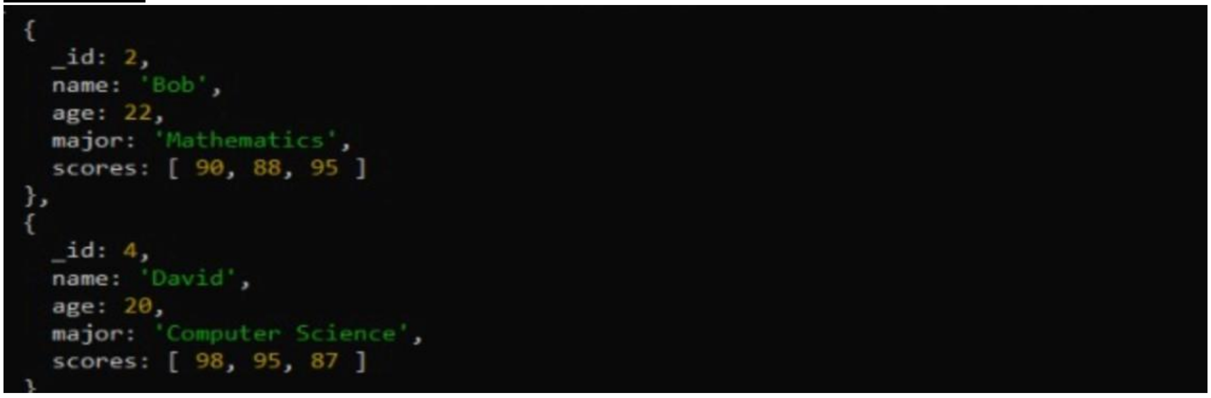
1. **$match:** Filters documents where the "age" field is greater than 23.
2. **$project:** Selects only the "name" and "age" fields, excluding the default "_id" field.
3. **$sort:** Sorts the documents by the "age" field in descending order.

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



```
db.students.aggregate([
  {
    $match: { age: { $lt: 23 } }
  },
  {
    $project: {
      name: 1,
      age: 1,
      _id: 0
    }
  },
  {
    $sort: { age: -1 }
  }
])
```

OUTPUT:



```
{
  _id: 2,
  name: 'Bob',
  age: 22,
  major: 'Mathematics',
  scores: [ 90, 88, 95 ]
},
{
  _id: 4,
  name: 'David',
  age: 20,
  major: 'Computer Science',
  scores: [ 98, 95, 87 ]
```

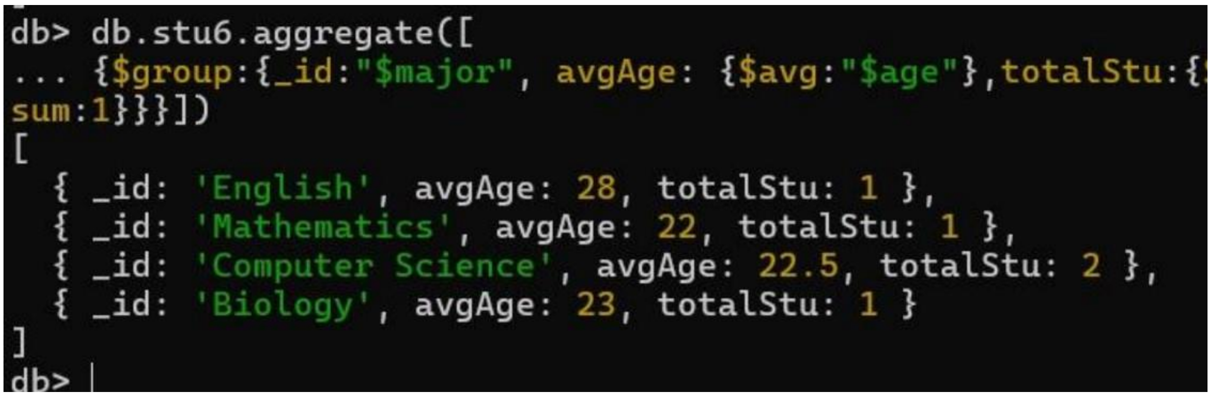
Explanation:

- **$match:** Only students with an age less than 23 are selected.
- **$project:** We only want the "name" and "age" fields, so we specify them. The _id field is excluded.
- **$sort:** The remaining documents are sorted by "age" in descending order (oldest first).

3. **Grouping the students by major & calculating their average age with a total number of students in each major.**

   Understanding the Requirement
   - Group students based on their major.
   - Calculate the average age for each major.
   - Count the total number of students in each major.

$group:
- _id: Defines the grouping criteria. In this case, we're grouping by the "major" field.
- averageAge: Calculates the average age of students within each group using the $avg accumulator.
- totalStudents: Counts the total number of students in each group using the $sum accumulator with a constant value of 1.

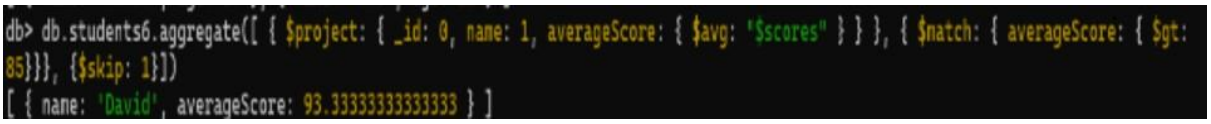


```
db> db.stu6.aggregate([
... {$group:{_id:"$major", avgAge: {$avg:"$age"},totalStu:{$
sum:1}}}])
[
  { _id: 'English', avgAge: 28, totalStu: 1 },
  { _id: 'Mathematics', avgAge: 22, totalStu: 1 },
  { _id: 'Computer Science', avgAge: 22.5, totalStu: 2 },
  { _id: 'Biology', avgAge: 23, totalStu: 1 }
]
db>
```

<u>Output Structure</u>
The output will be a list of documents, each representing a major. Each document will contain:
- _id: The major name.
- averageAge: The average age of students in that major.
- totalStudents: The total number of students in that major

This aggregation pipeline efficiently groups students by their major, calculates the average age, and counts the total number of students for each major.

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

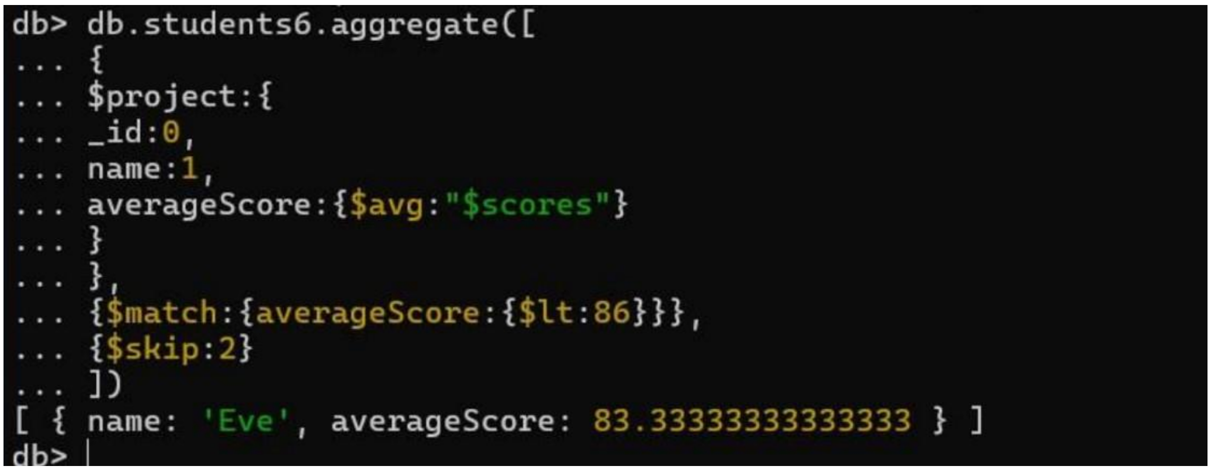


```
db> db.students6.aggregate([ { $project: { _id: 0, name: 1, averageScore: { $avg: "$scores" } } }, { $match: { averageScore: { $gt:
85}}}, {$skip: 1}])
[ { name: 'David', averageScore: 93.33333333333333 } ]
```

<u>Explanation:</u>
This pipeline is identical to the previous response as it accurately addresses the given requirements:
1. **$project**: Calculates the average score for each student using $avg on the "scores" array and includes the "name" and calculated "averageScore" fields.
2. **$match**: Filters documents where the "averageScore" is greater than 85.
3. **$skip**: Skips the first document in the result set.

This pipeline efficiently finds students with an average score above 85 and returns all but the first matching student.

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



```
db> db.students6.aggregate([
... {
... $project:{
... _id:0,
... name:1,
... averageScore:{$avg:"$scores"}
... }
... },
... {$match:{averageScore:{$lt:86}}},
... {$skip:2}
... ])
[ { name: 'Eve', averageScore: 83.33333333333333 } ]
db> |
```

Explanation:

1. **$project**: Calculates the average score for each student using $avg on the "scores" array and includes the "name" and calculated "averageScore" fields.
2. **$match**: Filters documents where the "averageScore" is less than 86.
3. **$skip**: Skips the first two documents in the result set.

This pipeline effectively finds students with an average score below 86 and returns all matching students except for the first two.