CLASS 7

AGGREGATION PIPELINE

AGGREGATION PIPELINE:

An aggregation pipeline in MongoDB is a sequence of stages that process documents and return computed results**.** Each stage of the aggregation pipeline transforms the document as the documents pass through it. However, once an input document passes through a stage, it doesn't necessarily produce one output document. Some stages may generate more than one document as an output.

MongoDB provides the db.collection.aggregate() method in the mongo shell and the db.aggregate() command to run the aggregation pipeline.

A stage can appear multiple times in a pipeline, with the exception of $out, $merge, and $geoNear stages. In this article, we will discuss in brief the seven major stages that you will come across frequently when aggregating documents in MongoDB. For a list of all available stages, see Aggregation Pipeline Stages.

- $project

  Reshapes each document in the stream, e.g., by adding new fields or removing existing fields. For each input document, output one document.

- $match

  Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. For each input document, the output is either one document (a match) or zero document (no match).

- $group

  Groups input documents by a specified identifier expression and apply the accumulator expression(s), if specified, to each group. $group consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field (group id) and, if specified, accumulated fields.

- $sort:

  Reorders the document stream by a specified sort key. The documents are unmodified, except for the order of the documents. For each input document, the output will be one document.

- $skip

  Skips the first *n* documents where *n* is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, the output is either zero document (for the first *n* documents) or one document (after the first *n* documents).

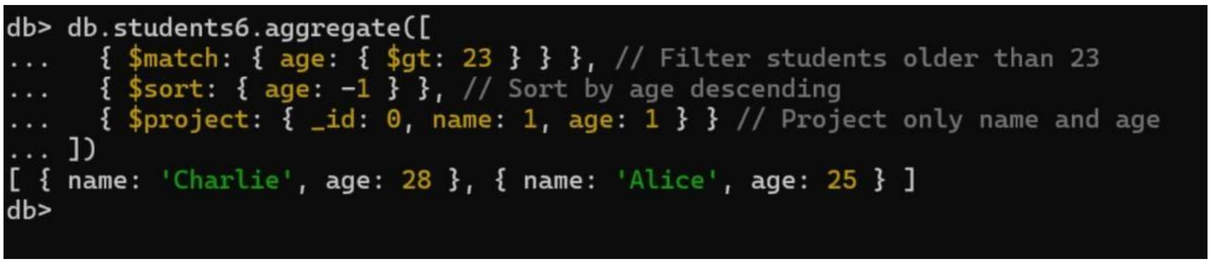
- $limit

  Passes the first *n* documents unmodified to the pipeline where *n* is the specified limit. For each input document, the output is either one document (for the first *n* documents) or zero document (after the first *n* documents).

- $unwind

  Breaks an array field from the input documents and outputs one document for *each* element. Each output document will have the same field, but the array field is replaced by an element value per document. For each input document, outputs *n* documents where *n* is the number of array elements and can be zero for an empty array.

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

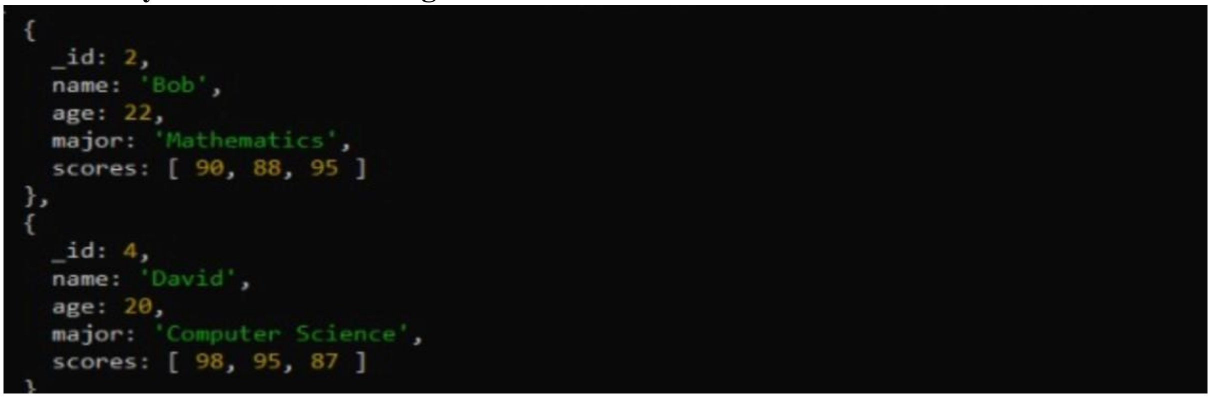


```
db> db.students6.aggregate([
...   { $match: { age: { $gt: 23 } } }, // Filter students older than 23
...   { $sort: { age: -1 } }, // Sort by age descending
...   { $project: { _id: 0, name: 1, age: 1 } } // Project only name and age
... ])
[ { name: 'Charlie', age: 28 }, { name: 'Alice', age: 25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



```
 {
   _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 ]
 }
```

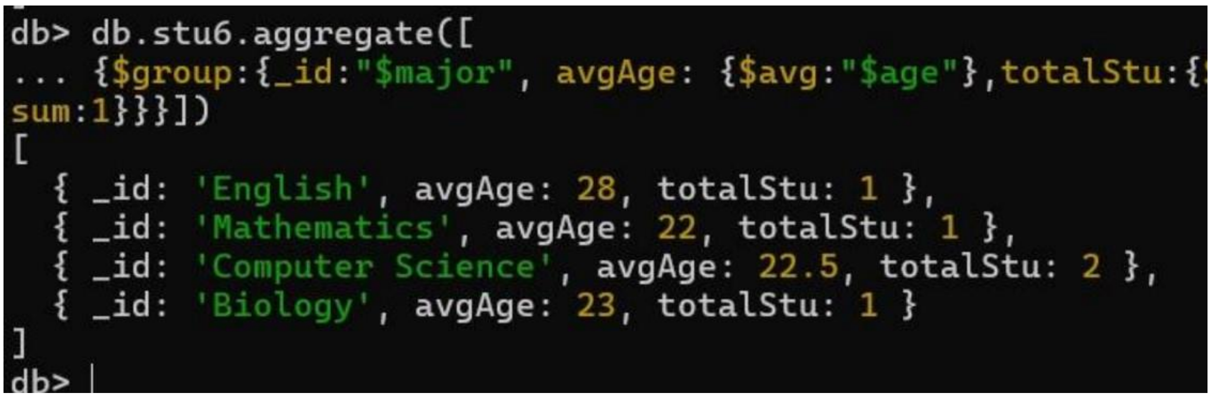
Explanation:

- **$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.**

$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.



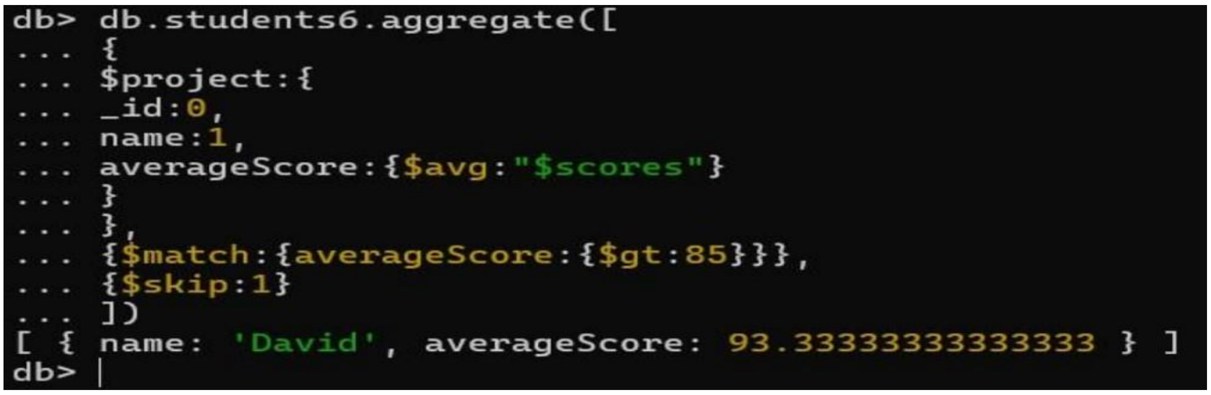
```
db> db.stu6.aggregate([
... {$group:{_id:"$major", avgAge: {$avg:"$age"},totalStu:{$
sum:1}}}])
[
  { _id: 'English', avgAge: 28, totalStu: 1 },
  { _id: 'Mathematics', avgAge: 22, totalStu: 1 },
  { _id: 'Computer Science', avgAge: 22.5, totalStu: 2 },
  { _id: 'Biology', avgAge: 23, totalStu: 1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

Output Structure

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 } ]
db> |
```

Explanation:

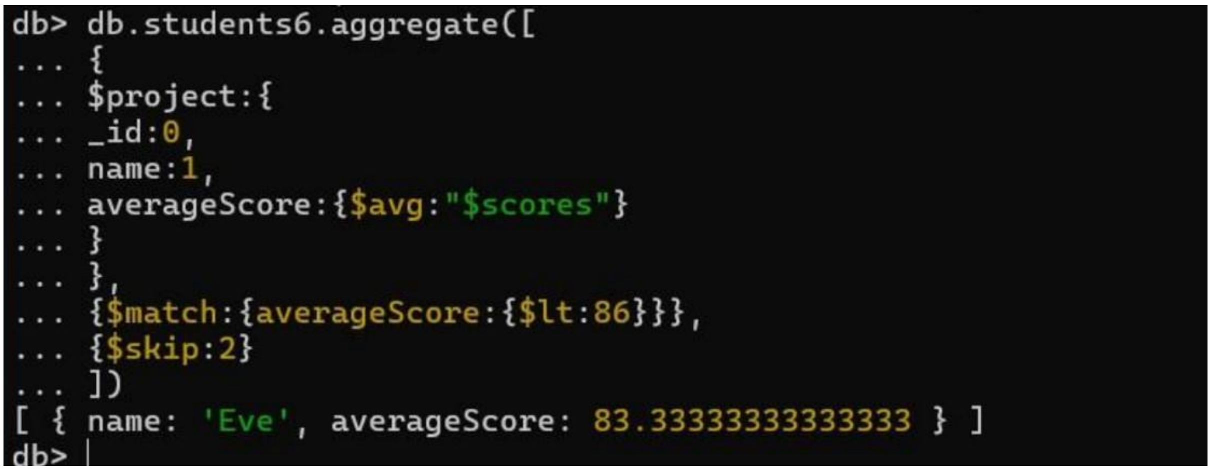
$project: This stage is like a filter, selecting only the necessary information from each student document.

- name: We want to include the student's name in the final result.
- averageScore: We calculate the average of the "scores" array for each student using $avg.
- **$match**: This stage acts as a gatekeeper, allowing only documents that meet a specific criteria to pass through.
    - `averageScore: { $gt: 85 }`: Only students with an average score greater than 85 will proceed to the next stage.
    -
- **$skip**: This stage is like skipping the first student in the filtered results.
    - `1`: Skips the first document, effectively giving us the second and subsequent high-scoring students.

In essence: The pipeline processes each student document, calculates their average score, filters out students with scores below 85, and then discards the top-scoring student from the remaining results.

The final output will be a list of students with an average score above 85, excluding the highest-scoring 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.