CLASS-7

AGGREGATION PIPELINES

An aggregation pipeline is a sequence of stages that process data from a MongoDB collection. Each stage performs a specific operation on the data, such as filtering, transforming, or aggregating. The output of one stage becomes the input for the next stage, allowing you to chain multiple operations together to achieve complex data processing tasks.

Components of an Aggregation Pipeline

An aggregation pipeline consists of the following components:

1. Stages: These are the individual operations that make up the pipeline. MongoDB provides a range of built-in stages, such as $match, $project, $filter, $group, and more.
2. Operators: These are the specific functions or expressions used within a stage to perform the desired operation. For example, the $match stage uses the $eq operator to filter documents based on a specific condition.
3. Expressions: These are the values or calculations used as input to an operator. For example, in the $project stage, you might use an expression like {$concat: ["$firstName", " ", "$lastName"]} to create a new field.

Types of Aggregation Stages

MongoDB provides several types of aggregation stages, including:

1. Filter Stages: These stages filter out documents that don't match a specific condition. Examples include $match and $filter.
2. Transformation Stages: These stages transform or modify the data in some way. Examples include $project, $addFields, and $replaceRoot.

3. Aggregation Stages: These stages perform aggregation operations, such as grouping, sorting, and calculating values. Examples include $group, $sort, and $facet.

4. Array Stages: These stages operate on arrays within documents. Examples include $unwind and $arrayElemAt.

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.

Example:

1. **Finding students with ages greater than 23, sorted by age in descending order, and only returning name and age.**

```
db> db.stu6.aggregate([
... {$match: {age: { $gt:23}}},
... {$sort:{age:-1}},
... {$project:{_id:0,name:1, age:1}}])
[ { name: 'Charlie', age: 28 }, { name: 'Alice', age: 25 } ]
db>
```

$match- filters students' age older than 23

$sort- sorts the age by descending order

$project- projects only name and age

2. Finding students with ages lesser than 23, sorted by name in ascending order, and only returning names and scores.

```
db> db.stu6.aggregate([ { $match: { age: { $lt: 23 } } }, {
$sort: { name: 1 } }, { $project: { _id: 0, name: 1, scores:
 1 } }])
[
  { name: 'Bob', scores: [ 90, 88, 95 ] },
  { name: 'David', scores: [ 98, 95, 87 ] }
]
db>
```

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

```
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>
```

4. **Finding students with an average score above 75 and skipping the first document**

```
db> db.stu6.aggregate([
... {
... $project: {
... _id:0,name:1,averageScore: {$avg: "$scores"}}},
... { $match: { averageScore: {$gt:75}}},
... {$skip:1}])
[
  { name: 'Bob', averageScore: 91 },
  { name: 'Charlie', averageScore: 82 },
  { name: 'David', averageScore: 93.33333333333333 },
  { name: 'Eve', averageScore: 83.33333333333333 }
]
db>
```

5. **Finding students with an average score below 86 and skipping the first 2 documents**

```
db> db.stu6.aggregate([ { $project: { _id: 0, name: 1, avera
geScore: { $avg: "$scores" } } }, { $match: { averageScore:
{ $lt: 86 } } }, { $skip: 2 }])
[ { name: 'Eve', averageScore: 83.33333333333333 } ]
db>
```

Benefits of Aggregation Pipelines

Aggregation pipelines offer several benefits, including:

- Flexibility: Pipelines can be customized to perform complex data processing tasks.
- Efficiency: Pipelines can reduce the amount of data transferred between the client and server.
- Scalability: Pipelines can handle large datasets and scale horizontally.
- Expressiveness: Pipelines provide a concise and expressive way to perform data transformations and aggregations.