Aggregation pipeline

Agenda:

- 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)

What is aggregation pipeline?

The aggregation pipeline in MongoDB is a framework for performing data aggregation operations. It processes documents in a collection through a sequence of stages, where each stage performs an operation on the data and passes the results to the next stage. This allows for complex data transformations and aggregations.

Explination of opertions:

$match: This operator acts as a meticulous gatekeeper, allowing only documents that satisfy a set of conditions to pass through.

• $redact: This operator is a more nuanced filter, employing complex expressions to determine which documents deserve to be included.

$group: This operator is the cornerstone of grouping, enabling you to create categories and perform calculations or aggregations on each group of documents.

• $sum: This operator, a tireless accountant, meticulously calculates the total for a particular field across all documents within a group.

• $avg: This operator, a skilled statistician, computes the average value of a field within each group.

• $min and $max: These operators, acting as scouts, identify the minimum and maximum values of a field within a group, respectively.

• $project: This operator, a sculptor of data, allows you to meticulously choose which fields to keep or discard, crafting the desired structure of your output.

$unset: This operator, a data eraser, meticulously removes unwanted fields from the output, leaving a clean and focused presentation.

• $unwind: This operator, a deconstructionist, meticulously expands an array into separate documents, enabling you to work with each element individually.

• $size: This operator, a keen measurer, meticulously determines the size of an array within a document.

• $slice: This operator, a precise extractor, meticulously selects and extracts a specific portion of an array, allowing you to focus on the relevant elements

• $sort: This operator, an organizer, meticulously arranges the output documents in a specific order, ensuring a logical presentation.

• $limit: This operator, a guardian of results, meticulously restricts the number of documents returned, preventing potential data overload.

• $skip: This operator, a selective gatekeeper, meticulously bypasses a specified number of documents at the beginning, allowing you to focus on later portions of the results.

• String manipulation operators: These operators, like linguistic artisans, meticulously transform and manipulate string data within documents.

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}}},
{$sort:{age:-1}},
{$project:{_id:0,name:1,age:1}}
])
```

The pipeline starts by finding all students in the students6 collection who are older than 23 years old. Then, it sorts these students in descending order of their age, so the oldest students appear first. Finally, the output only shows the student's name and age, discarding the automatically generated _id field.

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

```
db. students6. aggregate([
{$match:{age:{$gt:23} } },
{$sort:{age:1}},
{$project:{_id:0,name:1,age:1}}
])
```

This MongoDB aggregation pipeline performs a series of operations on the `students6` collection.

1. **$match**:

   ```
   {$match: {age: {$gt: 23}}}
   ```

   This stage filters the documents to include only those where the `age` field is greater than 23.

2. **$sort**:

   ```
   {$sort: {age: 1}}
   ```

   This stage sorts the filtered documents by the `age` field in ascending order (`1` signifies ascending order).

3. **$project**:

   ```
   {$project: {_id: 0, name: 1, age: 1}}
   ```

   This stage modifies the output to include only the `name` and `age` fields, excluding the `_id` field (since `_id: 0` is specified).

```
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 }
```

(i).Find students with age less than or equle to 22, sorted by name in ascending order, and only return name and score

```
db. students6. aggregate([
{$match:{age:{$lte:22}}},
{$sort:{age:1}},
{$project:{_id:0,name:1,scores:1}}
])
```

```
db> db.students6.aggregate([ { $match: { age: { $lte: 22 } } }, { $sort: { age: +1 } }, { $project: {_id:0,name: 1, scores: 1 } }] )
[
  { name: 'David', scores: [ 98, 95, 87 ] },
  { name: 'Bob', scores: [ 90, 88, 95 ] }
]
```

(ii).Find students with age less than or equle to 22, sorted by name in descending order, and only return name and score

```
b> db.students6.aggregate([ { $match: { age: { $lte: 22 } } }, { $sort: { age: -1 } }, { $project: {_id:0,name: 1, scores: 1 } }] )

{ name: 'Bob', scores: [ 90, 88, 95 ] },
{ name: 'David', scores: [ 98, 95, 87 ] }
```

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

```
db.students6.aggregate([
{$group:{_id: “$major”,averageAge:{$avg:“age”},totalStudents:{$sum:1}}}
])
```

This pipeline starts by grouping all students in the students6 collection based on their declared major. Within each major group, it calculates two statistics:

- The average age of students in that major (averageAge).
- The total number of students in that major (totalStudents).

Output:

```
[
  { _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 }
]
```

(i). Group students by major, calculate average age is less than and total number of students in each major:

```
db.students6.aggregate([
{$group:{_id: “$major”,averageAge:{$avg:“$lt”},totalStudents:{$sum:1}}}
])
```

```
> db.students6.aggregate([ { $group: { _id: "$major", averageAge: { $avg: "$lt" }, totalStudents: { $sum: 1 } } }] )

{ _id: 'Computer Science', averageAge: null, totalStudents: 2 },
{ _id: 'Biology', averageAge: null, totalStudents: 1 },
{ _id: 'English', averageAge: null, totalStudents: 1 },
{ _id: 'Mathematics', averageAge: null, totalStudents: 1 }
```

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

(i).first Find students with an average score (from scores array)

```
db.students6.aggregate([
 { $project:{ _id:0, name:1, averageScore:{$avg:“$scores”}}}
])
```

This stage projects the documents to include the `name` field and a new field `averageScore`, which is calculated as the average of the values in the `scores` array.

The `_id` field is excluded (since `_id: 0` is specified).

```
> db.students6.aggregate([{$project:{_id:0,name:1,averageScore:{$avg:"$scores"}
. }
. }])

{ name: 'Alice', averageScore: 85 },
{ name: 'Bob', averageScore: 91 },
{ name: 'Charlie', averageScore: 82 },
{ name: 'David', averageScore: 93.33333333333333 },
{ name: 'Eve', averageScore: 83.33333333333333 }
```

This gives the student name and average score.

(i) .Next find students with an average score (from scores array) above 85

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

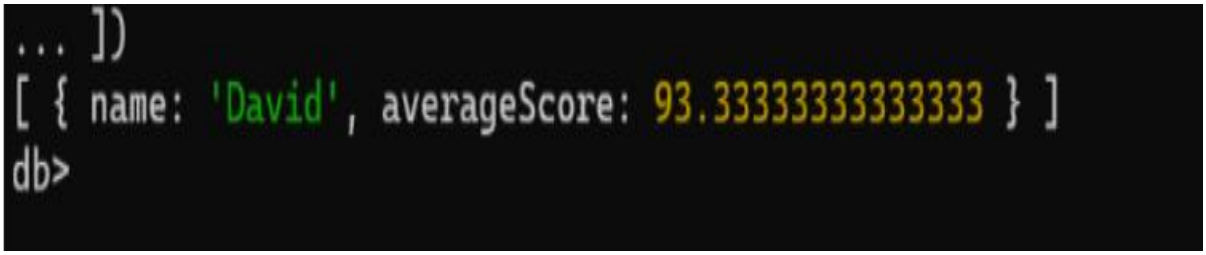
The pipeline starts by iterating through each student document in the students6 collection. For each student, it calculates the average score from the "scores" array (assuming it exists and contains numerical values).It then creates a new document for each student that includes only the name and the calculated averageScore.

```
]
db> db.students6.aggregate([{ $project: { _id: 0, name: 1, averageScore: { $avg: "$scores" } } },
... {$match:{averageScore:{$gt:85}}}])
[
  { name: 'Bob', averageScore: 91 },
  { name: 'David', averageScore: 93.33333333333333 }
]
```

(ii). Next 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}])
```

the $match stage filters the documents first. Since $skip comes later, it tries to skip a document from the already filtered results.



4.Find students with an average score (from scores array) grater than or equal to 85 and skip the first 2 documents

db.students6.aggregate([{ $project:{ _id:0, name:1, averageScore:{$avg:“$scores”}} }, {$match:{averageScore:{$lte:85}]}]) {$skip:2}]