Experiment:6

Aggregation pipeline

1. Accumulator Operators:

These operators shine when you need to perform calculations on groups of documents. They act like tireless accountants, processing data within each group. Here are some popular examples:

- `$sum`: Tallies up numeric values in a particular field across all documents within a group. Imagine calculating the total sales for each product category.
- `$avg`: Calculates the average value of a numeric field within a group. Useful for finding the average order value per customer.
- `$count`: Simply counts the number of documents in a group. Perfect for getting group sizes or finding the number of users in each country.
- `$first`: Retrieves the first document from a group, often used in combination with sorting to get the earliest or minimum value.
- `$last`: Similar to `$first`, but grabs the last document in a group, helpful for finding the latest or maximum value.
- `$push`: Adds elements to an array within the output documents for each group. Like accumulating items bought by each customer.

2. Projection Operators:

These operators are the architects of your output documents. They control which fields are included and how they are presented:

- `$project`: The maestro of projection, allowing you to specify fields to include or exclude. You can even use expressions to transform existing fields.
- `$alias`: Renames a field in the output for clarity. Like changing "product_id" to "productId".

3. Array Operators:

Working with arrays within documents? These operators are your helping hand:

- `$addToSet`: Ensures unique elements are added to an array within the output documents for each group. No duplicates allowed!
- `$push`: Appends elements to the end of an array within the output documents for each group. Adding purchased items to a "cart" array.
- `$size`: Determines the number of elements within an array field. Counting the number of items in a shopping cart.

4. Arithmetic Operators:

For numeric calculations within the pipeline, these operators are your go-to:

- `$add`: Performs addition on numeric fields. Useful for calculating total order amount (price * quantity).
- `$subtract`: Subtracts values from numeric fields. Finding the price difference after a discount.
- `$multiply`: Multiplies numeric fields. Calculating the total cost (price * quantity).
- `$divide`: Divides numeric fields. Finding the average order value (total_sales / number_of_orders).

5. Comparison Operators:

Need to filter or group based on comparisons? These operators come to the rescue:

- `$eq`: Checks for equality between two values. Filtering products matching a specific category ID.
- `$gt`: Selects documents where a field's value is greater than another value. Finding orders with a total amount exceeding a threshold.
- `$gte`: Similar to `$gt`, but includes the equal case (greater than or equal to).
- `$lt`: Selects documents where a field's value is less than another value. Finding customers below a certain age range.
- `$lte`: Similar to `$lt`, but includes the equal case (less than or equal to).

Lets Build new Dataset

- Download collection here
- Upload the new collection with name “students6”

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
])
```

- `db.students6.aggregate([])`: This line initiates the aggregation pipeline. It specifies that we're working with the `students6` collection and using the `aggregate` method to perform the pipeline operations.
- `{ $match: { age: { $gt: 23 } } }`: This is the first stage of the pipeline, using the `$match` operator. It filters the documents based on the condition specified within. Here, we're filtering for documents where the `age` field is greater than (`$gt`) 23.

- `{ $sort: { age: 1 } }`: This is the second stage, using the `$sort` operator. It sorts the documents based on the specified field. In this case, we're sorting by the `age` field in ascending order (1).

- `{ $project: { id: 0, name: 1, age: 1 } }`: This is the third and final stage, using the `$project` operator. It controls which fields are included in the output documents and how they are presented. Here, we're specifying that we only want to include the `name` and `age` fields (1) and exclude the `id` field (0).

this pipeline starts with all documents in the students6 collection, filters for students older than 23, sorts them by age (youngest to oldest), and then presents only the name and age for each student who meets the criteria.

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>
```

1B. Find students with age less than 23, sorted by name in ascending order, and only return name and score.

```
db> db.s6.aggregate([ { $match: { age: { $lt: 23 } } }, /* Filter students older than 23*/ { $sort:
{ name:1 } }, /* Sort by age ascending*/ { $project: {name: 1,score:1 } } /* Project only name and a
ge*/] )
```

- `db> db.s6.aggregate([`: This line initiates the aggregation pipeline. It specifies that we're working with the `s6` collection and using the `aggregate` method to perform the pipeline operations.

- `{ $match: { age: { $lt: 23}}},`: This is the first stage of the pipeline, using the `$match` operator. It filters the documents based on the condition specified within. Here, we're filtering for documents where the `age` field is less than (<) 23.

- `{ $sort: { name: 1} },`: This is the second stage, using the `$sort` operator. It sorts the documents based on the specified field. In this case, we're sorting by the `name` field in ascending order (1).

- `{ $project: {name: 1, score: 1 } }`: This is the third and final stage, using the `$project` operator. It controls which fields are included in the output documents and how they are presented. Here, we're specifying that we only want to include the `name` and `score` fields (1) and exclude any other fields that might exist in the document (implicit by not mentioning them).

This pipeline starts with all documents in the `students6` collection, filters for students younger than 23, sorts them by name (A to Z), and then presents only the name and score for each student who meets the criteria.

Output:

```
[ { _id: 2, name: 'Bob' }, { _id: 4, name: 'David' } ]
```

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

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

The code you provided uses the MongoDB aggregation pipeline to calculate average age and total number of students enrolled in each major. Here's a breakdown line by line:

db> db.students6.aggregate([: This line initiates the aggregation pipeline. It specifies that we're working with the students6 collection and using the aggregate method to perform the pipeline operations.

{ $group: { _id: "$major", averageAge: { $avg: "$age" }, totalStudents: { $sum: 1}}}: This is the first and only stage of the pipeline. It uses the $group operator, which is powerful for grouping documents together based on shared fields and performing calculations on those groups. Here's what each part does:

- _id: "$major": This defines the grouping criteria. It groups documents together based on the value of the "major" field. So, all documents with the same major will be put into one group.
- averageAge: { $avg: "$age" }: This calculates the average age for each group. It uses the $avg accumulator operator to compute the average value of the "age" field within each group.
- totalStudents: { $sum: 1 }: This calculates the total number of students in each group. It uses the $sum accumulator operator to add up the value of 1 (implicitly assumed to be the count of 1 for each document) for each document in the group.

]): This closing bracket signifies the end of the aggregation pipeline.

The pipeline essentially groups students by their major, calculates the average age and total number of students within each major, and returns the results. The final output may look similar to this:

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 }
]
```

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
])
```

1. db.students6.aggregate(): This line initiates the aggregation pipeline. It specifies that we'll be performing aggregation operations on the students6 collection within the current database.
2. [: This square bracket marks the beginning of an array that defines the stages of the aggregation pipeline. Each stage within the pipeline transforms the data and feeds the results to the subsequent stage.
3. { $project: { ... } }: This defines the first stage of the pipeline that utilizes the $project operator. The $project operator allows you to specify which fields to include or exclude from the output documents.
    - _id: 0: This instruction excludes the _id field from the output documents. By default, MongoDB assigns a unique identifier (_id) to each document. Here, we're indicating that we don't need this field in the results.
    - name: 1: This includes the name field in the output documents.
    - averageScore: { $avg: "$scores" }: This calculates the average score for each student and creates a new field named averageScore in the output documents. The $avg operator computes the average value of an array of numbers. In this case, it takes the scores array (which holds the individual scores for each student) and calculates the mean.
4. },: This comma separates the first stage ($project) from the second stage ($match) within the aggregation pipeline.
5. { $match: { averageScore: { $gt: 85 } } }: This defines the second stage that utilizes the $match operator. The $match operator filters the documents based on a specified condition.
    - averageScore: This refers to the new field (averageScore) created in the previous stage, which holds the calculated average score for each student.
    - { $gt: 85 }: This is the matching criteria. The $gt operator stands for "greater than". Here, it filters the documents where the averageScore is greater than 85.

6. },: This comma separates the second stage ($match) from the third stage ($skip) within the aggregation pipeline.
7. { $skip: 1 } // Skip the first document: This defines the third and final stage that utilizes the $skip operator. The $skip operator allows you to skip a specified number of documents from the beginning of the results.
    - 1: This value indicates that we want to skip the first document after applying the filter in the previous stage.
8.]: This square bracket marks the end of the aggregation pipeline array.

 Calculates the average score for each student in the students6 collection.
 Filters the documents to include only those students with an average score greater than 85.

 Skips the first document from the filtered results (if there are multiple documents with an average score exceeding 85).

The final output includes documents with the name field and the newly created averageScore field, showcasing students who have an average score above 85 (excluding the first document that met this criteria, if applicable).

Output:

```
db> db.students6.aggregate([
...   {
...     $project: {
...       _id: 0,
...       name: 1,
...       averageScore: { $avg: "$scores" }
...     }
...   },
...   { $match: { averageScore: { $gt: 85 } } }, // Filter by average sc
...   { $skip: 1 } // Skip the first document
... ])
[ { name: 'David', averageScore: 93.33333333333333 } ]
db>
```

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

```
db> db.s6.aggregate([{$project:{_id:0,name:1,averageScore:{$avg:
"$scores"}}},{$match:{averageScore:{$lt:86}}},{$skip:2}])
```

(Here s6 means student6 collection)

1. db> db.s6.aggregate([: This line initiates the aggregation pipeline. It specifies that we'll be performing aggregation operations on the s6 collection within the current database (db). The square bracket [marks the beginning of an array that defines the stages of the aggregation pipeline. Each stage transforms the data and feeds the results to the subsequent stage.
2. {$project: {_id: 0, name: 1, averageScore: {$avg: "$scores"}}},: This defines the first stage of the pipeline that utilizes the $project operator. The $project operator allows you to specify which fields to include or exclude from the output documents, and you can also perform transformations on existing fields.
    - _id: 0: This instruction excludes the _id field from the output documents. By default, MongoDB assigns a unique identifier (_id) to each document. Here, we're indicating that we don't need this field in the results.
    - name: 1: This includes the name field in the output documents.
    - averageScore: {$avg: "$scores"}: This calculates the average score for each student and creates a new field named averageScore in the output documents. The $avg operator computes the average value of an array of numbers. In this case, it takes the scores array (which holds the individual scores for each student) and calculates the mean.
3. ,: This comma separates the first stage ($project) from the second stage ($match) within the aggregation pipeline.
4. {$match: {averageScore: {$lt: 86}}},: This defines the second stage that utilizes the $match operator. The $match operator filters the documents based on a specified condition.
    - averageScore: This refers to the new field (averageScore) created in the previous stage, which holds the calculated average score for each student.
    - {$lt: 86}: This is the matching criteria. The $lt operator stands for "less than". Here, it filters the documents where the averageScore is less than 86. So, this stage keeps only the documents where students have an average score below 86.
5. ,: This comma separates the second stage ($match) from the third stage ($skip) within the aggregation pipeline.
6. {$skip:2}]: This defines the third and final stage that utilizes the $skip operator. The $skip operator allows you to skip a specified number of documents from the beginning of the results.
    - 2: This value indicates that we want to skip the first two documents after applying the filter in the previous stage.
7.]: This square bracket] marks the end of the aggregation pipeline array.

8. Calculates the average score for each student in the s6 collection. Excludes the _id field and includes the name field and the newly created averageScore field in the output documents.
9. Filters the documents to include only those students with an average score less than 86.
10. Skips the first two documents from the filtered results (if there are multiple documents with an average score below 86).
11. The final output includes documents with only the name and averageScore fields, showcasing students who have an average score lower than 86, excluding the first two documents that met this criteria (if applicable).

Output:

```
[ { name: 'Eve', averageScore: 83.33333333333333 } ]
db>
```