Aggregation Pipeline

Aggreation pipeline:

In MongoDB, an aggregation pipeline is a powerful framework for performing complex data processing tasks on collections. It allows you to efficiently group, filter, transform, and summarize your data.

Syntax:

```
db.<collection_name>.aggregate([
{ <stage1_definition> },
{ <stage2_definition> },
 ...
 { <stageN_definition> }
])
```

- **<collection_name>:** Replace this with the actual name of your MongoDB collection.
- **[]:** This represents the array containing the pipeline stages.
- **{}:** Each element within the array defines a single stage in the pipeline.
- **<stageN_definition>:** This represents the configuration for each stage using specific operators and expressions. (Details will vary depending on the chosen operation)

Benefits:

- **Modular Design:** Breaking down complex queries into smaller, manageable stages improves readability and maintainability.
- **Efficiency:** Aggregation pipelines can process large datasets efficiently by performing operations on the server-side.
- **Flexibility:** You can chain various stages together to achieve intricate data transformations and calculations.

Aggregation Pipeline Operations:

Operation	Description	Example
$match	Filters documents based on a condition.	Filters orders with a total amount greater than $100. { $match: { total: { $gt: 100 } } }

$group	Groups documents together based on shared fields and performs calculations on the grouped data.	Calculates average order value per customer. { $group: { _id: "$customer", averageOrder: { $avg: "$total" } } }
$sort	Sorts the results based on a specified field (ascending or descending).	Sorts products by price (ascending). { $sort: { price: 1 } }
$project	Selects or transforms specific fields for the output documents	Includes only name and price fields from products. { $project: { _id: 0, name: 1, price: 1 } }
$limit	Limits the number of documents returned in the final results.	Returns only the top 5 most expensive products. { $limit: 5 }
$lookup	Performs a join-like operation to retrieve data from another collection.	Looks up the product category details for each product document.
$unwind	Deconstructs an array field from a document, creating a separate document for each element in the array.	Unwinds an "orders" array containing order details within a customer document.
$addFields	Creates new fields with calculated values based on existing fields or expressions.	Calculates a discount price for each product.

Download collection name “students6”

Q: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)

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

Explanation:

1. **{$match:{age:{$gt:23}}}**:

- This stage applies a filter using the $match operator.
- It selects documents where the age field is greater than ($gt) 23.
- Only documents satisfying this criteria will be passed to the next stage.

2. **{$sort:{age:-1}}**:
   - This stage sorts the filtered documents using the $sort operator.
   - It sorts based on the age field in descending order (-1).
   - Documents with higher age values will appear first in the output.
3. **{$project:{_id:0,name:1,age:1}}:**
   - This stage uses the $project operator to specify the output document structure.
   - It excludes the _id field by setting it to 0.
   - It includes the name and age fields by setting them to 1.
   - Only these two fields will be present in the final output documents.

Find students with age less than 24, sorted by age in descending order, and return _id name and age

```
db> db.students6.aggregate([ { $match: { age: { $lt: 24 } } }, { $sort: { age: -1 } }, { $project: { _id: 1, name: 1, age: 1 } }] )
[
  { _id: 5, name: 'Eve', age: 23 },
  { _id: 2, name: 'Bob', age: 22 },
  { _id: 4, name: 'David', age: 20 }
]
```

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

```
db> db.students6.aggregate([ { $match: { age: { $lt: 23 } } }, { $sort: { age: 1 } }, { $project: { _id: 0, name: 1, age: 1 } }] )
[ { name: 'David', age: 20 }, { name: 'Bob', age: 22 } ]
db>
```

- **{$sort: { age: 1 } }**: This stage sorts the filtered documents by the age field in ascending order (1). Documents with younger ages appear first.

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}}}])
[
  { _id: 'Mathematics', averageAge: 22, totalStudents: 1 },
  { _id: 'Computer Science', averageAge: 22.5, totalStudents: 2 },
  { _id: 'Biology', averageAge: 23, totalStudents: 1 },
  { _id: 'English', averageAge: 28, totalStudents: 1 }
]
```

- **{$group:{...}}**: This stage applies the $group operator to group documents based on a specific field.
    - **_id:** "$_major": Groups documents based on the value of the major field. Documents with the same major will be placed in the same group.
    - **averageAge: {$avg: "$age"}:** Calculates the average age within each group. It uses the $avg operator on the age field.
    - **totalStudents: {$sum: 1}:** Counts the number of documents in each group. The $sum operator with a value of 1 increments a counter for each document in the group.

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 } ]
```

- **averageScore: { $avg: "$scores" }:** Calculates the average score for each student. It uses the $avg operator on the scores array field (assuming it exists in each student document).
- **{$skip: 1}**: This stage is optional and skips the first matching document after the filtering stage. Since filtering happens before skipping, only documents with an average score above 85 are considered for skipping.

Error:(skip:-1)

```
db> db.students6.aggregate([ { $project: { _id: 0, name: 1, averageScore: { $avg: "$scores" } } }, { $match: { averageScore: { $gt:
85}}}, {$skip: -1}])
MongoServerError[Location5107200]: invalid argument to $skip stage: Expected a non-negative number in: $skip: -1
```

- A negative value for $skip isn't supported in MongoDB aggregation pipelines. It expects a non-negative number representing the number of documents to skip at the beginning of the results.

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: { averageSc
ore: { $lt:86}}}, {$skip:2}])
[ { name: 'Eve', averageScore: 83.33333333333333 } ]
db>
```

- **$skip**: The $skip stage remains the same, instructing the pipeline to skip the first two documents that meet the filtering criteria (average score < 86).