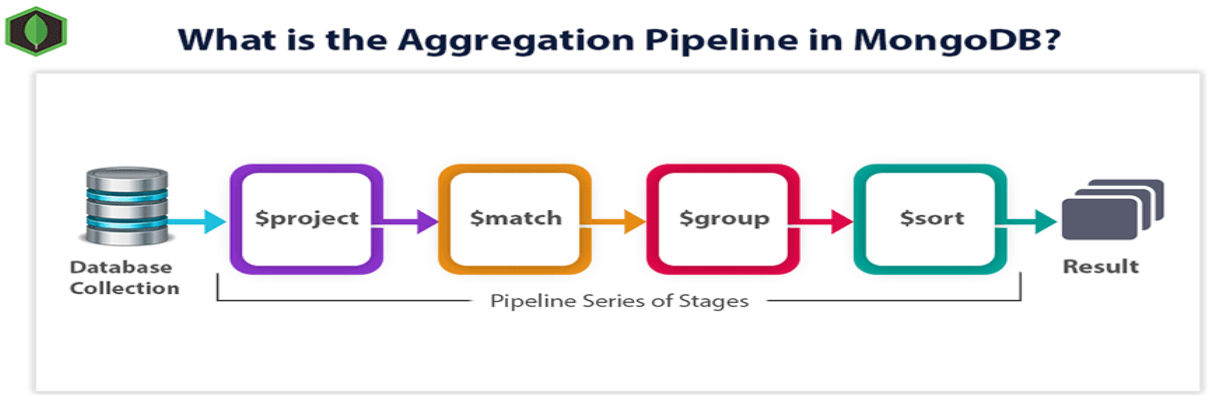
EXPRIMENT 06

Aggregation pipeline

The **MongoDB Aggregation Pipeline** allows you to perform a sequence of data processing operations on the documents in a collection.The aggregation pipeline consists of a series of stages, where each stage represents a specific operation or transformation. **The output of each stage serves as the input for the next stage**, allowing you to create complex data processing workflows. The aggregation pipeline stages are processed in sequence, providing a flexible and expressive way to analyze and manipulate data.



6 . Execute Aggregation Pipeline and its operations (pipeline must contain $match, $group, $sort, $project, $skip etc.

$match	-	Filter (reduce) the number of documents that is passed to the next stage
$group	-	Group documents by a distinct key, the key can also be a compound key
$project	-	Pass documents with specific fields or newly computed fields to the next stage
$sort	-	Returns the input documents in sorted order
$limit	-	Limit the number of documents for the next stage
$unwind	-	Splits an array into into one document for each element in the array
$out	-	As the last stage, creates/replaces an unsharded collection with the input documents

$match:

Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.

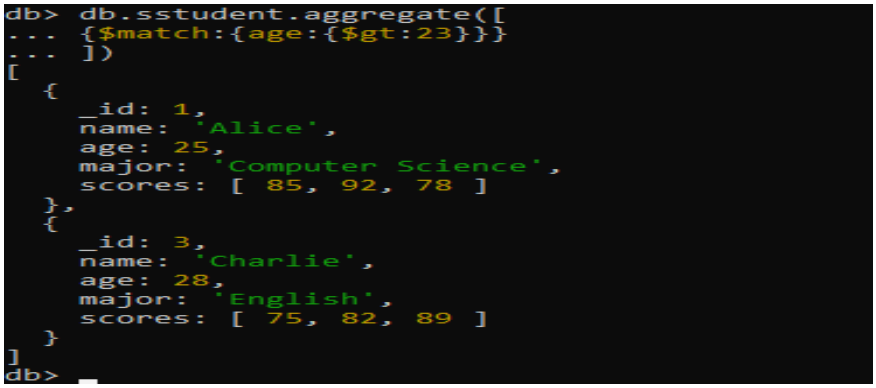
- ✓ **Find students with age greater than 23**

Input:

```
db> db.sstudent.aggregate([
... {$match:{age:{$gt:23}}}
... ])
```

This code snippet extracts documents from a collection named "sstudent" where the "age" is greater than 23. It uses MongoDB's aggregation framework to filter data.The program is called "student aggregate" and it is used to aggregate students' scores.

Output:



- **db:** This refers to the database object in the Node.js shell.
- **db.sstudent:** This specifies the collection named “sstudent” within the database.
- **.aggregate():** This method is used to perform aggregation operations on the collection.
- **$match:** This is a stage that filters the documents in the collection.
- **{age:{$gt:23}}:** This is the filter condition. It specifies that the document’s “age” field must have a value greater than (>$gt) 23.

$sort:

Sorts all input documents and returns them to the pipeline in sorted order. For the field or fields to sort by, set the sort order to `1` or `-1` to specify an ascending or descending sort respectively.

Sorted by age in descending order

Input:

The query is likely filtering for documents in a collection named "sstudent" where the "age" field is greater than 23.

Output:

```
db> db.sstudent.aggregate([ {$sort:{age:-1}}] )
[
  {
    _id: 3,
    name: 'Charlie',
    age: 28,
    major: 'English',
    scores: [ 75, 82, 89 ]
  },
  {
    _id: 1,
    name: 'Alice',
    age: 25,
    major: 'Computer Science',
    scores: [ 85, 92, 78 ]
  },
  {
    _id: 5,
    name: 'Eve',
    age: 23,
    major: 'Biology',
    scores: [ 80, 77, 93 ]
  },
  {
    _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 ]
  }
]
```

- Each line represents a student.
- There are five students listed: Charlie, Alice, Eve, Bob, and David.
- Each student has an `_id`, name, age, major, and an array of scores.

The data is sorted by age in descending order. So, Charlie, the first student listed, is the oldest at 28 years old. David, the last student listed, is the youngest at 20 years old.

Some popular pipleline operations:

`$match`	Filter documents
`$project`	Reshape documents
`$group`	Summarize documents
`$unwind`	Expand arrays in documents
`$sort`	Order documents
`$limit/$skip`	Paginate documents
`$redact`	Restrict documents
`$geoNear`	Proximity sort documents
`$let,$map`	Define variables

$project:

Passes along the documents with the requested fields to the next stage in the pipeline. The specified fields can be existing fields from the input documents or newly computed fields.

Input:

```
db> db.sstudent.aggregate([ {$project:{_id:0,name:1,age:1}}] )
```

The code filters a MongoDB collection ("sstudent") to only show "name" and "age" fields, excluding the default "_id" field.

Output:

```
db> db.sstudent.aggregate([ {$project:{_id:0,name:1,age:1}}] )
[
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 22 },
  { name: 'Charlie', age: 28 },
  { name: 'David', age: 20 },
  { name: 'Eve', age: 23 }
]
db>
```

- **.aggregate():** This method initiates the aggregation pipeline.
- **({$project: {_id: 0, name: 1, age: 1}}):** This is the project stage of the aggregation pipeline. It's used to specify which fields you want to include in the output documents.
- **_id: 0:** This tells MongoDB to exclude the '_id' field from the output documents. By default, MongoDB includes the '_id' field in all documents.
- **name: 1:** This tells MongoDB to include the 'name' field in the output documents.
- **age: 1:** This tells MongoDB to include the 'age' field in the output documents.

So in essence, this code snippet is querying the "sstudent" collection and returning only the "name" and "age" fields from each document, while excluding the "_id" field.

- ✓ **Find students with age greater than 23, sorted by age in descending order, and only return name and age**

Input:

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

This code digs through a student database (MongoDB collection), finds students over 23 (possibly sorts them by age), and shows only their name, ID, and likely some kind of page info (assuming "page" is meant to be "age").

Output:

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

- The code filters students in the "sstudent" collection to only those older than 23.
- It then projects the results to show only the "name" and "age" fields, excluding the default "_id" field.
- In short, it filters students by age and shows a simplified list of their names and ages.

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

Input:

```
db> db.sstudent.aggregate([{ $match: { age: { $lt: 23 } } }, { $sort: { age: -1 } }, { $project: { _id: 0, name: 1, age: 1 } }] )
```

This code sifts through student data (sstudent collection). It keeps only students under 23, sorts them from youngest to oldest (descending age), and shows a simplified list with just their names and ages, skipping a default ID field.

Output:

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

The code looks at a collection named "sstudent".

- It filters the students to only show those younger than 23 years old.
- It then sorts the results by age, with the youngest student first.
- Finally, it creates a simplified list that excludes the student ID and only shows the student's name and age.

✓ **Group students by major, calculate average age and total number of students in each major:**

Fig : major field in student database

Example1:

Input:

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

- It filters a student collection ("sstudent") to find students younger than 23.
- It then sorts them by age (youngest first).

- Finally, it shows only the "name" and "age" fields, excluding a default student ID field.

Output:

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

- The code groups students by their major (`id:"$major"`).
- It calculates the average age for each major (`averageAge:{$avg:"$age"}`).
- It also calculates the total number of students in each major (`totalStudents: {$sum:1}}`).
- Finally, it presents the results in a list, showing the major, average age, and total number of students for each major.

Example2:

```
]
db> db.sstudent.aggregate([ { $group: { _id: "$major", avgAge: { $sum: "$age" }, total: { $sum: 1 } } } ] )
[
  { _id: 'Biology', avgAge: 23, total: 1 },
  { _id: 'English', avgAge: 28, total: 1 },
  { _id: 'Computer Science', avgAge: 45, total: 2 },
  { _id: 'Mathematics', avgAge: 22, total: 1 }
]
db>
```

- The code groups students by their major (`id: "Smajor"`).
- It calculates the average age for each major (`avgAge: {$avg: "Sage"}`).
- It also calculates the total number of students in each major (`total: {$sum: 1}}`).

- Finally, it presents the results in a list, showing the major, average age, and total number of students for each major.

✓ Find students with an average score (from scores array) above 85

Input:

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

Output:

```
db> db.sstudent.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 }
]
db>
```

- **db.sstudent.aggregate -** This line tells MongoDB to use the aggregation framework and perform an aggregation on the "sstudent" collection.

- **[...project: {..._id: 0, name: 1, averageScore: ($avg: "Sscores")...}]** - This defines the aggregation pipeline. In this case, the pipeline contains a single stage that projects a new document with only the "name" and "averageScore" fields. The `$avg` operator calculates the average of the values in the "Sscores" field for each group.

- **[{ name: 'Alice', averageScore: 85), { name: 'Bob', averageScore: 91}, { name: 'Charlie', averageScore: 82}, { name: 'David", averageScore: 93.33333333333333 }, { name: 'Eve', averageScore: 83.33333333333333 }] -**

This is the output of the aggregation pipeline. It shows the name and average score for each student in the "sstudent" collection.

- ✓ **Find students with an average score (from scores array) above 85 and skip the first document**

Input:

```
db> db.sstudent.aggregate([ { $project: { _id: 0, name: 1, averageScore: { $avg: "$scores" } }},
...  {$match:{averageScore:{$gt:85}}},
... {$skip:1}
... ])
```

Output:

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

- **db.sstudent.aggregate([...]) -** This line tells MongoDB to use the aggregation framework and perform an aggregation on the "sstudent" collection. Aggregation pipelines allow you to process data and return results in a specific format.
- **`{ $project: { id: 0, name: 1, averageScore: { $avg: "$scores"}}}, {... } -** This defines the stages of the aggregation pipeline. In this case, the pipeline contains two stages:

The first stage **(`{ $project: ... }`)** projects a new document with only the "name" and "averageScore" fields. The **`$avg`** operator calculates the average of the values in the "scores" field for each student.

- ✓ **Find students with an average score (from scores array) above 65 and skip the first document**

```
db> db.sstudent.aggregate([{ $project: { _id: 0, name: 1, averageScore: { $avg: "$scores" } } }, { $match: { averageScore: { $gt: 65 } } }, { $skip: 1 }] )
[
  { name: 'Bob', averageScore: 91 },
  { name: 'Charlie', averageScore: 82 },
  { name: 'David', averageScore: 93.3333333333333 },
  { name: 'Eve', averageScore: 83.3333333333333 }
]
db>
```

db.sstudent.aggregate([{$project: { ... } }, { $match: { ... } }, {$skip: 1}]) - This line tells MongoDB to use the aggregation framework and perform an aggregation on the "sstudent" collection. The aggregation pipeline consists of three stages:

- The first stage **({$project: { ... }})** projects a new document with only the "name" and "averageScore" fields. The `$avg` operator calculates the average of the values in the "scores" field for each student.
- The second stage **({$match: { ... }})** filters the results based on some criteria. In this case, the ellipsis (...) hides the specific criteria, but it likely filters for students with an average score greater than a certain threshold.
- The third stage **({$skip: 1})** skips the first document in the results.

- ✓ **Find students with an average score (from scores array) above 65 and skip 2 document**

```
db> db.sstudent.aggregate([{ $project: { _id: 0, name: 1, averageScore: { $avg: "$scores" } } }, { $match: { averageScore: { $gt: 65 } } }, { $skip: 2}])
[
  { name: 'Charlie', averageScore: 82 },
  { name: 'David', averageScore: 93.33333333333333 },
  { name: 'Eve', averageScore: 83.33333333333333 }
]
```

db.sstudent.aggregate([{$project: { ... } }, { $match: { averageScore: { $gt: 65 } } }, { $skip: 2 }]) - This line tells MongoDB to use the aggregation framework and perform an aggregation on the "sstudent" collection. The aggregation pipeline consists of three stages:

- The first stage **({$project: { ... }})** projects a new document with only the "name" and "averageScore" fields. The `$avg` operator calculates the average of the values in the "scores" field for each student.
- The second stage **({$match: { averageScore: { $gt: 65 } }})** filters the results to only include students with an average score greater than 65. `The`$gt` operator stands for "greater than".
- The third stage **(`{$skip: 2}`)** skips the first two documents in the filtered results.

✓ Find students with an average score (from scores array) above 65 and skip 3 document

```
db> db.sstudent.aggregate([{ $project: { _id: 0, name: 1, averageScore: { $avg: "$scores" } } }, { $match: { averageScore: { $gt: 65 } } }, { $skip: 3}])
[
  { name: 'David', averageScore: 93.33333333333333 },
  { name: 'Eve', averageScore: 83.33333333333333 }
]
db>
```

- **db.sstudent.aggregate([...])** - This line tells MongoDB to use the aggregation framework and perform an aggregation on the "sstudent" collection. Aggregation pipelines allow you to process data and return results in a specific format.
- **{ $project: { id: 0, name: 1, averageScore: { $avg: "$scores"}}}, {... }** - This defines the stages of the aggregation pipeline. In this case, the pipeline contains two stages:
- The first stage **({ $project: ... })** projects a new document with only the "name" and "averageScore" fields. The **`$avg`** operator calculates the average of the values in the "scores" field for each student.

- ✓ **Find students with an average score (from scores array) below 86 and skip the first 2 documents**

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

db.sstudent.aggregate([{$project: { id: 0, name: 1, averageScore: { $avg: "$scores"}}}, { $match: { averageScore: { $lt: 86} } }, { $skip: 2}]) - This line tells MongoDB to use the aggregation framework and perform an aggregation on the "sstudent" collection. The aggregation pipeline consists of three stages:

- The first stage **({$project: { ... } })** projects a new document with only the "name" and "averageScore" fields. The `$avg` operator calculates the average of the values in the "scores" field for each student.
- The second stage **({$match: { averageScore: { $lt: 86} } })** filters the results to only include students with an average score less than 86.
- The **$lt** operator stands for "less than".
- The third stage **({$skip: 2})** skips the first two documents in the filtered results.