

MONGO DB

CLASS 7: AGGREGATE PIPELINES.

An aggregation pipeline in MongoDB is a way to process and analyse data by passing it through a series of stages. Each stage performs a specific operation on the data, like filtering, grouping, or sorting. The output of one stage becomes the input for the next stage, allowing you to transform the data step-by-step to get the desired results.

The MongoDB **Aggregation pipeline** is a framework for data aggregation modeled on the concept of data processing pipelines. Documents enter a **multi-stage pipeline** that transforms the documents into aggregated results.

Each stage performs an operation on the input documents and passes the results to the next stage. The stages can filter, group, and modify the documents in various ways.

It encourage to execute several queries to demonstrate various **Aggregation operators**.

Here's a brief explanation of each stage in a MongoDB aggregation pipeline:

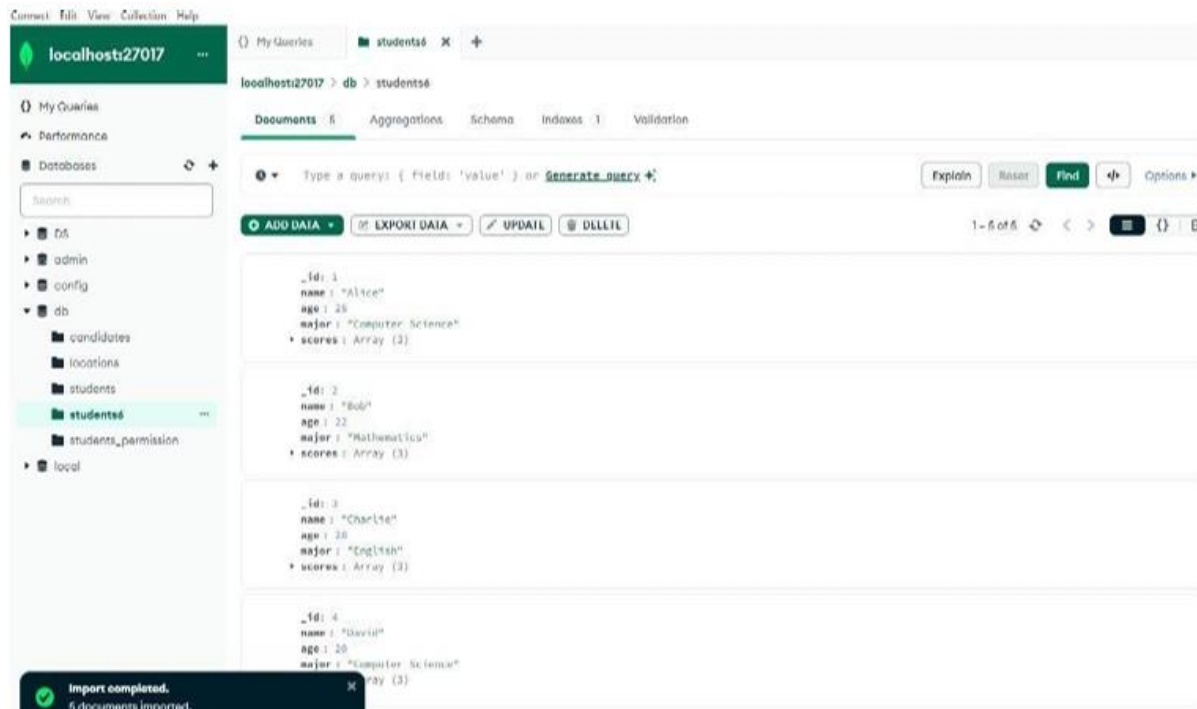
1. **\$match:** Selects documents that match certain criteria.
2. **\$group:** Groups documents by a field and performs calculations for each group.
3. **\$sort:** Sorts documents by a specified field.
4. **\$project:** Chooses which fields to include or create new fields.
5. **\$skip:** Skips a specified number of documents.
6. **\$limit:** Limits the number of documents to a specified number.
7. **\$unwind:** Breaks apart an array field to create a separate document for each item.
8. **\$lookup:** Performs a left outer join to a collection in the same database to filter in documents from the "joined" collection for processing.
9. **\$addFields:** Adds new fields to documents.
10. **\$replaceRoot:** Replaces the input document with the specified embedded document.

The order of the stages is crucial because the output of one stage becomes the input of the next.

These stages can be combined to process and analyse data step-by-step in MongoDB. The following data set has been designated as a new collection named

students6. This collection comprises records of 5 students, each with attributes such as ID, name, age, major, and scores.

Now lets import a new collection called “students6” through mongo compass.



To switch this collection have to use some commands they are

use db

show dbs

show collections

```
test> use db
switched to db db
db> show dbs
DS          40.00 KiB
admin       40.00 KiB
config      108.00 KiB
db          284.00 KiB
local       72.00 KiB
db> show collections
candidates
locations
students
students_permission
students6
```

```
[
  { "_id": 1, "name": "Alice", "age": 25, "major": "Computer Science", "scores": [85, 92, 78] },
  { "_id": 2, "name": "Bob", "age": 22, "major": "Mathematics", "scores": [90, 88, 95] },
  { "_id": 3, "name": "Charlie", "age": 28, "major": "English", "scores": [75, 82, 89] },
  { "_id": 4, "name": "David", "age": 20, "major": "Computer Science", "scores": [98, 95, 87] },
  { "_id": 5, "name": "Eve", "age": 23, "major": "Biology", "scores": [80, 77, 93] }
]
```

Here is an expanded explanation for each stage along with the examples:

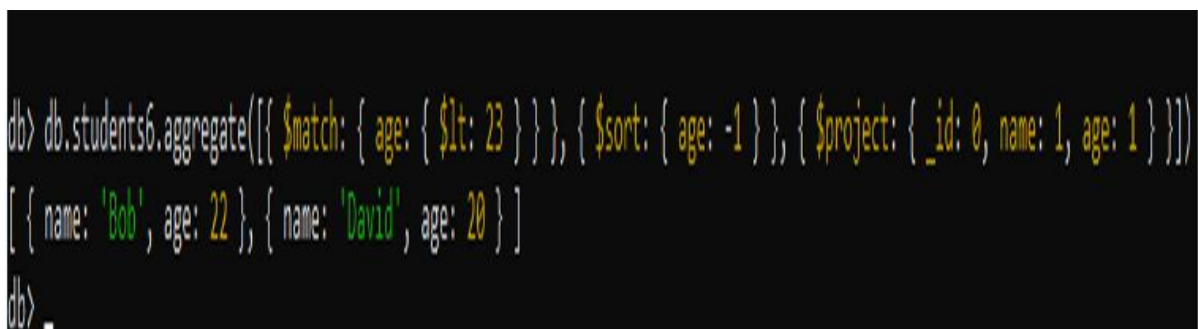
1. \$match:

The \$match stage filters documents in the pipeline. Only documents that match the specified criteria are passed to the next stage. This stage is highly efficient when used early in the pipeline as it reduces the amount of data that needs to be processed by subsequent stages.

Example:

Now to find students with age less than 23 it could be sorted by descending order to obtain only name and age we use a command

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



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

According to the output Bob and David are 22 and 20 year students respectively. Here,

\$lt: represents less than.

\$gt: represents greater than.

Age:(-1):- represents sorting in descending order.

Again Now to find students with age greater than 23 it could be sorted by descending order to obtain only name and age we use a command.

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

2. \$group:

Groups documents by a specified key and performs aggregate operations.

Example: Group by major and calculate the average age of students in each major

Now to group students by major to calculate average age and total number of students in each major using sum:2 we use a command

```
db.students6.aggregate([{$group:{_id:"$major",averageAge:{$avg:"$age"},totalStudents:{$sum:2}}])
```

```
> db.students6.aggregate([ { $group: { _id: "$major", averageAge: { $avg: "$age" }, totalStudents: { $sum: 2 } } } ] )
{ _id: 'Computer Science', averageAge: 22.5, totalStudents: 4 },
{ _id: 'English', averageAge: 28, totalStudents: 2 },
{ _id: 'Mathematics', averageAge: 22, totalStudents: 2 },
{ _id: 'Biology', averageAge: 23, totalStudents: 2 }
```

Now to group students by major to calculate average age and total number of students in each major using sum:1 we use a command

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

```
db> db.students6.aggregate([
... { $group: { _id: "$major", averageAge: { $avg: "$age" }, totalStudents: { $sum: 1 } } } ] )
[
{ _id: 'English', averageAge: 28, totalStudents: 1 },
{ _id: 'Computer Science', averageAge: 22.5, totalStudents: 2 },
{ _id: 'Mathematics', averageAge: 22, totalStudents: 1 },
{ _id: 'Biology', averageAge: 23, totalStudents: 1 }
]
```

3. \$skip :

The \$skip stage skips a specified number of documents in the pipeline. This is useful for pagination, where you might want to skip a number of documents to retrieve the next set of results.

Example:

Here to find students with an average score (from scores array) above 85 and skip the first document to do this so have to use a command is

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

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

Again now to find students with an average score (from scores array) below 86 and skip the first two document to do this so have to use a command is

```
db.students6.aggregate([{$project:{_id:0,name:1,averageScore:{$avg:"$scores"}}},{$match:{averageScore:{$lt:86}}},{$skip:2}]]
```

```
db> db.students6.aggregate([{$project:{_id:0,name:1,averageScore:{$avg:"$scores"}}},{$match:{averageScore:{$lt:86}}},{$skip:2}]]);
[ { name: 'Eve', averageScore: 83.33333333333333 } ]
```

4. \$sort:

The \$sort stage sorts all input documents and returns them in the specified order. The sort order is determined by the value of a specified field or fields. Sorting can be done in ascending or descending order and is useful for organizing output data.

5. \$project:

The \$project stage reshapes each document in the pipeline. It can include, exclude, or add new fields to the documents. Computed fields can also be created using expressions. This is useful for transforming the structure of the documents and preparing the data for further stages or final output.