Aggregation pipeline

An aggregation pipeline in MongoDB 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. The stages make up a pipeline, which is a series of operations that process the data.

Key benefits:
- Flexibility: Chain various operations together to achieve complex data manipulation.
- Efficiency: Perform multiple calculations in one go, reducing the need for separate queries.
- Clarity: Break down complex queries into smaller, easier-to-understand stages.

Here is a basic overview of some commonly used stages in an aggregation pipeline:

1.$match: Filters the documents to pass only the documents that match the specified condition(s).
{ $match: { status: "A" } }

2.$group: Groups input documents by the specified identifier expression and applies the accumulator expressions.

```
{
  $group: {
    _id: "$status",
    total: { $sum: "$amount" }
  }
}
```

3.$project: Passes along the documents with the requested fields.

```
{
  $project: {
    name: 1,
    status: 1
  }
}
```

4.$sort: Sorts all input documents and returns them to the pipeline in sorted order.
{ $sort: { age: -1 } }

5.$limit: Limits the number of documents passed to the next stage.
{ $limit: 5 }

6.$skip: Skips the first n documents where n is the specified skip number and passes the remaining documents to the next stage.
{ $skip: 10 }

7.$unwind: Deconstructs an array field from the input documents to output a document for each element.
{ $unwind: "$sizes" }

BUILD A NEW DATASET:

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

“HERE” INCLUDES:

[

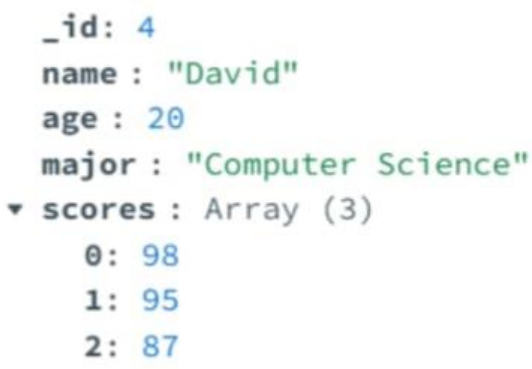
{ "_id": 1, "name": "Alice", "age": 25, "major": "Computer Science", "scores": [85, 92, 78] },
{ "_id": 2, "name": "Bob", "age": 22, "major": "Mathematics", "scores": [90, 88, 95] },

{ "_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] }

]



1.A.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
])
```

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>
```

- $match Stage:

- This stage uses the $match operator to filter the documents in the "students" collection.
- It specifies a condition where the "age" field must be greater than 23 ({ "age": { $gt: 23 } }).
- Only documents matching this criteria pass on to the next stage.

- $sort Stage:

- The $sort operator arranges the remaining documents based on the specified field.
- In this case, we sort by the "age" field ({ "age": -1 }).
- The -1 indicates descending order, meaning older students (higher age) appear first.

- $project Stage:

- The final stage utilizes the $project operator to control which fields are included in the output.
- We set "name" and "age" to 1 ({ "name": 1, "age": 1 }), indicating these fields should be included.
- Any other fields in the original documents are excluded from the final result.

1.B.FIND STUDENTS WITH AGE LESSTAHN 23 SORTD BY NAME IN ASCENDING ORDER AND ONLY RETURN NAME AND SCORE :

Same as the above but, If your database system allows, you can use another command (like find) to retrieve the transformed data after applying the pipeline. This retrieved data would contain only the "name" and "score" fields for students younger than 23, listed alphabetically by name.

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

EXPLANATION:

1.Grouped Data: Students are grouped by their major.
2.Calculated Values: Within each major group, you'll get:
o _id: The name of the major (the grouping criteria).
o avgAge: The average age of students in that major (calculated using $avg).
o totalStudents: The total number of students in that major (calculated using $sum).

3.FIND STUDENTS WITH AN AVERAGE SCOES (FROM SCORES ARRAY)ABOVE 85 AND SKIP THE FIRST DOCUMENT AGGREGATION PIPELINE FOR TOP STUDENTS(THEORY):

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

```
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>
```

1. Filtered Students :Only students with an average score exceeding 85 (after grouping and averaging) remain.

2. Skipped Document: The first document meeting the criteria (average score > 85) is skipped.

Retrieval: If your database system allows, you can use a separate command (like find) to retrieve the transformed data after applying the pipeline. This retrieved data would contain documents for students who have an average score above 85, excluding the first one that meets this criteria.

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

- **Filtered Students:** Only students with an average score below 86 remain.
- **Skipped Documents:** The first two documents meeting this criteria are excluded.

Retrieval: Use a separate command (like find) to retrieve the transformed data. This data would include documents for students with an average score below 86, starting from the third one that meets this condition.