AGGREGATION PIPELINE

AGENDA:

. 1. Data Transformation: Reshape your data by filtering, projecting specific fields, or renaming them to better suit your analysis needs.

2. Aggregation: Perform calculations on groups of documents. This could involve counting documents, finding sums, averages, minimums, maximums, etc.

3. Multi-Stage Processing: Chain multiple operations together. Each stage in the pipeline takes the output of the previous stage and transforms it further.

4. Document Manipulation: Create new fields derived from existing ones, or embed documents within other documents.

Output Generation:

The final stage defines the format and content of the results. You can choose to return the entire transformed data set or just specific aggregated value.

- Input: Raw data documents enter the pipeline.
- Stages: Each stage performs a specific operation on the data, like filtering documents, grouping them based on criteria, or calculating values.
- Processing: The output from one stage becomes the input for the next, allowing for complex transformations.
- Output: The final stage defines what gets delivered: the transformed data set or specific summarized values.

LETS BUILD A NEW DATASET:

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

“HERE” INCLUDES:

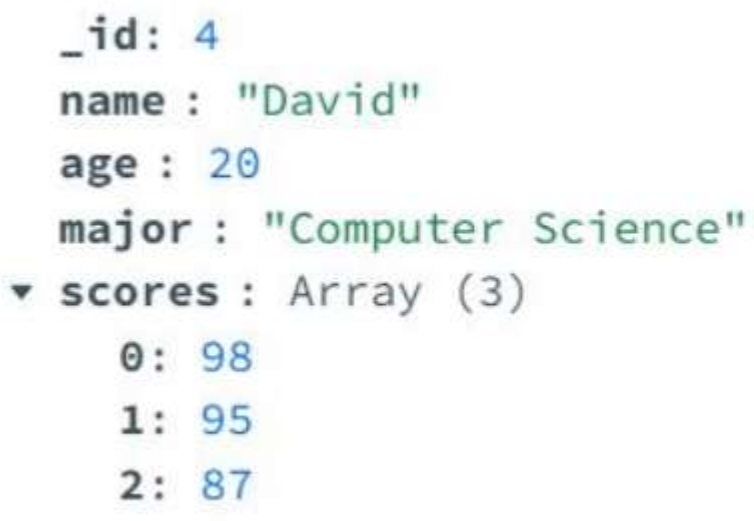
```
[

{ "_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] }]



```
  _id: 4
  name : "David"
  age : 20
  major : "Computer Science"
▾ scores : Array (3)
    0: 98
    1: 95
    2: 87
```

EXPLANATION ABOUT OPERATORS IN DETAIL:

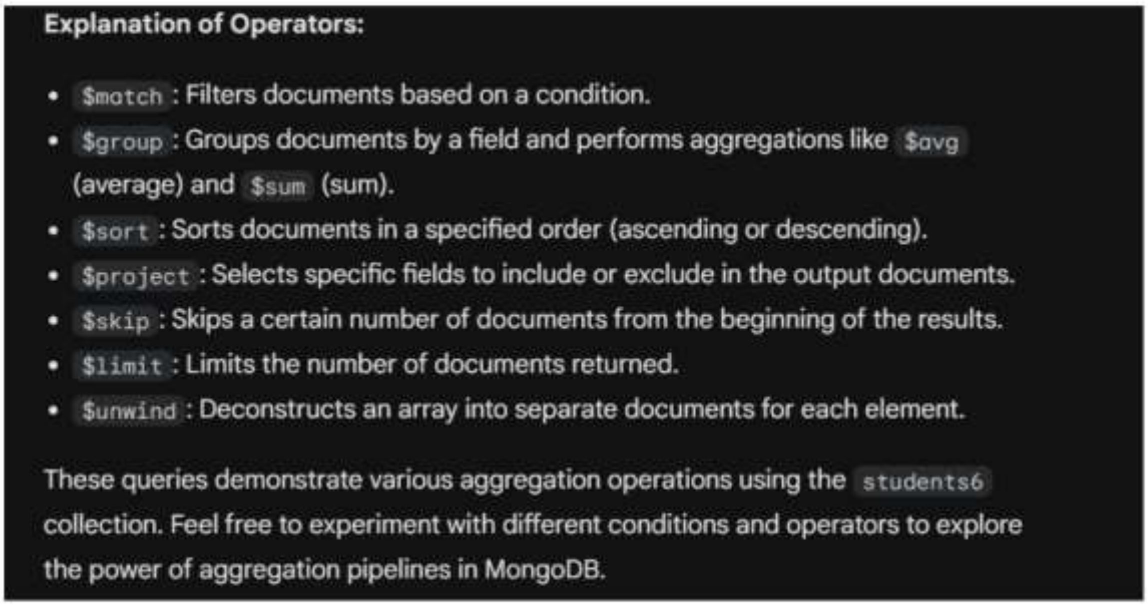
• Input: Aggregation pipelines operate on collections of documents in your database.

• Operators: These are the building blocks that perform specific actions on the data at each stage. Here are some common types:

- Filtering: $match keeps only documents matching certain criteria.
- Projection: $project selects specific fields to include or exclude from the output.
- Grouping: $group organizes documents into groups based on shared field values.

• Multi-Stage: You can chain multiple operators together. Each stage processes the output of the previous one.

• Output: The final stage defines the results. You can return the entire transformed data set or just the aggregated values.



Explanation of Operators:

- `$match` : Filters documents based on a condition.
- `$group` : Groups documents by a field and performs aggregations like `$avg` (average) and `$sum` (sum).
- `$sort` : Sorts documents in a specified order (ascending or descending).
- `$project` : Selects specific fields to include or exclude in the output documents.
- `$skip` : Skips a certain number of documents from the beginning of the results.
- `$limit` : Limits the number of documents returned.
- `$unwind` : Deconstructs an array into separate documents for each element.

These queries demonstrate various aggregation operations using the `students6` collection. Feel free to experiment with different conditions and operators to explore the power of aggregation pipelines in MongoDB.

1.FIND STUDENTS WITH AGE GREATER THAN 23, SORTED BY AGE IN DESCENDING ORDER, AND ONLY RETURN NAME AND AGE:

• **$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.

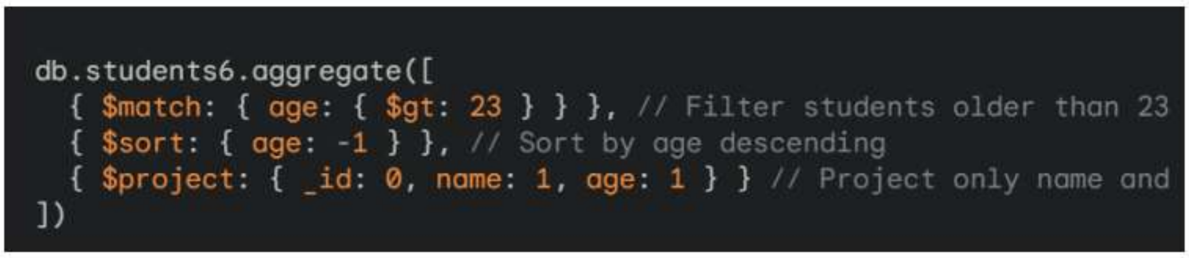
• **$sort Stage:**

- 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:**

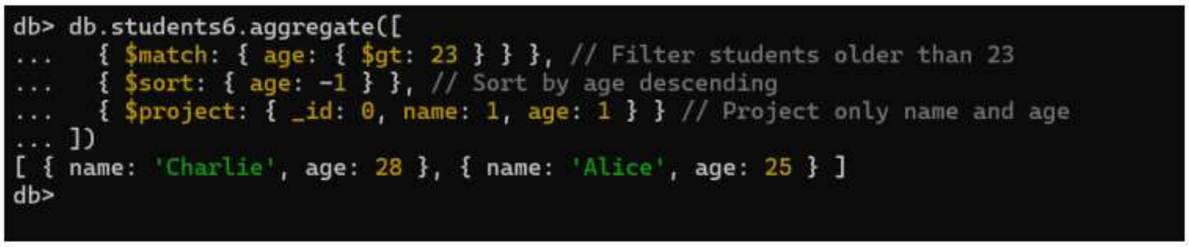
1. The final stage utilizes the $project operator to control which fields are included in the output.

2. We set "name" and "age" to 1 ({ "name": 1, "age": 1 }), indicating these fields should be included. o Any other fields in the original documents are excluded from the final result.



```
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>
```

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.

2.GROUP STUDENTS BY MAJOR ,CALCULATE AVERAGE AGE AND TOTAL NUMBER OF STUDENTS IN ECAH MAJO

AGGREGATION PIPELINE FOR STUDENT ANALYSIS THEORY:

This pipeline groups students by their major and calculates two key metrics: average age and total number of students in each major. Here's a breakdown of the stages and expected outcome:

• **STAGES**

• **$group Stage:**

1. This stage is the heart of the pipeline and utilizes the $group operator.

2.It groups documents (students) based on a specified field, "major" in this case.

3.Within each group, it performs calculations using accumulator expressions:

- ❖ _id: Sets the group identifier to the "major" field value (automatically generated).
- ❖ avgAge: Calculates the average age using the $avg accumulator on the "age" field.
- ❖ ♣ totalStudents: Counts the total number of students in the group using the $sum accumulator with a value of 1 for each document (effectively counting them).

Expected Output:

The pipeline wouldn't create separate documents, but rather transforms the data within the "students" collection. Imagine the following outcome:

1. Grouped Data: Students are grouped by their major.

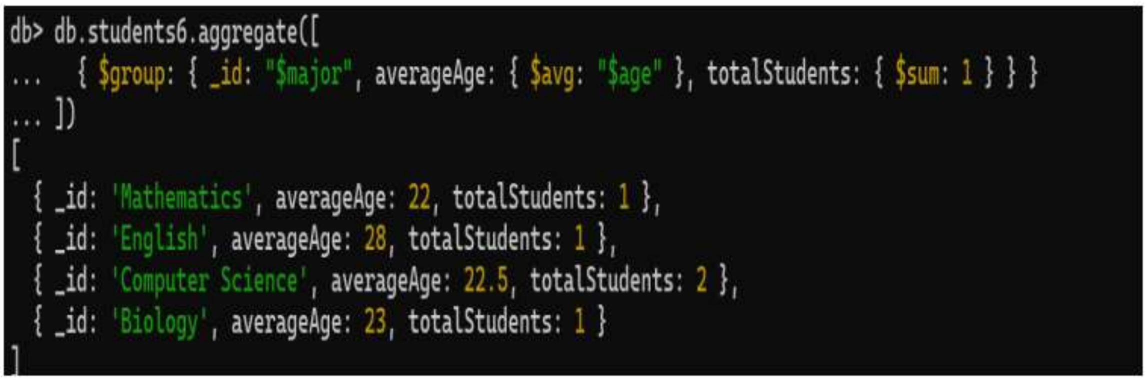
2. Calculated Values: Within each major group, you'll get:

- _id: The name of the major (the grouping criteria).
- avgAge: The average age of students in that major (calculated using $avg).
- totalStudents: The total number of students in that major (calculated using $sum).

Example Output Document (per Major):

JSON

```
{ "_id": "Computer Science",

"avgAge": 21.5,

"totalStudents": 50

}
```



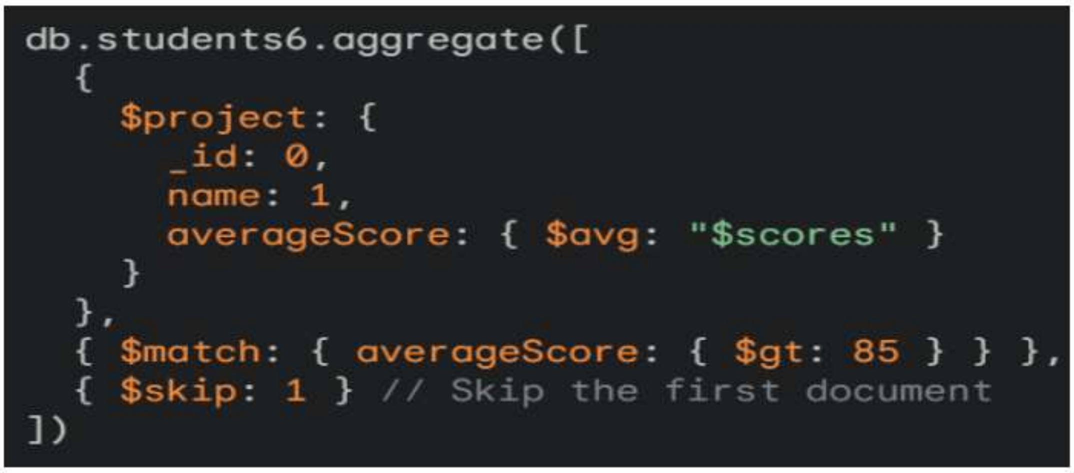
```
db> db.students6.aggregate([
...   { $group: { _id: "$major", averageAge: { $avg: "$age" }, totalStudents: { $sum: 1 } } }
... ])
[
  { _id: 'Mathematics', averageAge: 22, totalStudents: 1 },
  { _id: 'English', averageAge: 28, totalStudents: 1 },
  { _id: 'Computer Science', averageAge: 22.5, totalStudents: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

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

This pipeline identifies students with an average score (calculated from the "scores" array) above 85 and skips the first document in the results. Here's how it works:

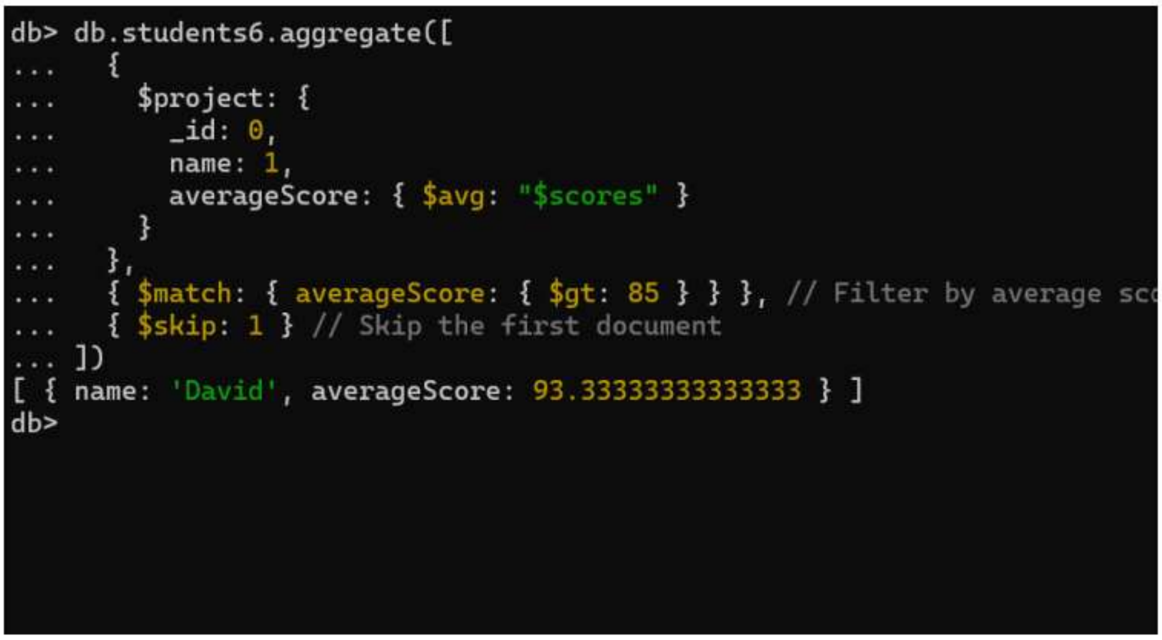
Stages:

1. **$unwind Stage (Optional):**
   - If "scores" is an array within each student document, this stage (not always necessary) unpacks the array into separate documents, one for each score.
   - It's relevant if you need to calculate the average across all individual scores, not the average of arrays.
2. **$group Stage:**
   - This stage groups documents (potentially unpacked scores) by student ID (_id).
   - It calculates the average score using $avg on the "score" field within each student group.
3. **$match Stage:**
   - This stage filters the grouped data, keeping only students with an average score greater than 85 ({ "averageScore": { $gt: 85 } }).
4. **$sort Stage (Optional):**
   - You can add this stage to sort the results further (e.g., by average score descending).
5. **$skip Stage:**
   - This stage skips the first document in the final results ({ $skip: 1 }).



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

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>
```