Aggregation Operators

Aggregation operators in MongoDB are essential tools for performing complex data processing and analysis tasks within the database. These operators are part of the Aggregation Framework, which allows you to transform and combine data in various ways. Here are some key points about aggregation operators in MongoDB:

Introduction to Aggregation Operators

1. **Purpose and Functionality**:
    - Aggregation operators are used to perform computations on data stored in MongoDB collections.
    - They allow for data transformation, summarization, and analysis, enabling users to derive meaningful insights from raw data.
2. **Aggregation Framework**:
    - MongoDB's Aggregation Framework provides a rich and expressive syntax for processing data.
    - The framework operates on the concept of a pipeline, where documents pass through a series of stages. Each stage performs a specific operation, such as filtering, grouping, or sorting the data.
3. **Common Aggregation Operators**:
    - $match: Filters documents to pass only those that meet specified criteria.
    - $group: Groups documents by a specified key and performs aggregate calculations, such as sum, average, or count, on each group.
    - $project: Reshapes documents by including, excluding, or adding new fields.
    - $sort: Sorts documents based on specified criteria.

- $limit and $skip: Controls the number of documents passing through the pipeline and skips a specified number of documents.
- $unwind: Deconstructs an array field from the input documents to output a document for each element.
- $lookup: Performs a left outer join to join documents from another collection into the current pipeline.

Syntax:

db.collection.aggregate(<AGGREGATE OPERATION>**)**

Types

Expression Type	Description	Syntax
Accumulators	Perform calculations on entire groups of documents	
* $sum	Calculates the sum of all values in a numeric field within a group.	"$fieldName": { $sum: "$fieldName" }
* $avg	Calculates the average of all values in a numeric field within a group.	"$fieldName": { $avg: "$fieldName" }
* $min	Finds the minimum value in a field within a group.	"$fieldName": { $min: "$fieldName" }
* $max	Finds the maximum value in a field within a group.	"$fieldName": { $max: "$fieldName" }
* $push	Creates an array containing all unique or duplicate values from a field	"$arrayName": { $push: "$fieldName" }
* $addToSet	Creates an array containing only unique values from a field within a group.	"$arrayName": { $addToSet: "$fieldName" }
* $first	Returns the first value in a field within a group (or entire collection).	"$fieldName": { $first: "$fieldName" }
* $last	Returns the last value in a field within a group (or entire collection).	"$fieldName": { $last: "$fieldName" }

Let us see some examples :

Average GPA of All Students

```
JavaScript


db.students.aggregate([
  { $group: { _id: null, averageGPA: { $avg: "$gpa" } } }
]);
```

Assume you have a collection named students where each document represents a student and contains a gpa field. The structure of a document might look like this:

Output :

```
... ]);
[ { _id: null, averageGPA: 2.98556 }
db>
```

Explanation:

$group : groups all the documents together

_id: null sets the group identifier to install(optinal,as there's only one group in this case

The averagevalue of the "gpa" field using the $avg operator

Mnimum and maximum Age:

- **Finding Minimum Age:**
  - Sort documents in ascending order by age ($sort: { age: 1 }).
  - Limit the results to 1 document ($limit: 1). This retrieves the document with the lowest age.
- **Finding Maximum Age:**
  - Sort documents in descending order by age ($sort: { age: -1 }).
  - Limit the results to 1 document ($limit: 1). This retrieves the document with the highest age.

Example:

Minimum and Maximum Age:

```
db> db.students.aggregate([
...   { $group: { _id: null, minAge: { $min: "$age" }, maxAge: { $max: "$age" } } }
... ]);
```

Output:

Answer

```
[ { _id: null, minAge: 18, maxAge: 25 } ]
```

Explanation:

Each document shows the original data (_id, name, etc.) along with the age field representing the minimum or maximum value based on the used aggregation pipeline.

How to get Average GPA for all home cities?

```
db> db.students.aggregate([
...   { $group: { _id: "$home_city", averageGPA: { $avg: "$gpa" } } }
... ]);
[
  { _id: 'City 8', averageGPA: 3.11741935483871 },
  { _id: 'City 7', averageGPA: 2.847931034482759 },
  { _id: 'City 10', averageGPA: 2.935227272727273 },
  { _id: 'City 9', averageGPA: 3.1174358974358976 },
  { _id: 'City 2', averageGPA: 3.01969696969697 },
  { _id: 'City 3', averageGPA: 3.0100000000000002 },
  { _id: 'City 6', averageGPA: 2.8969444444444448 },
  { _id: null, averageGPA: 2.9784313725490197 },
  { _id: 'City 4', averageGPA: 2.8251851851851852 },
  { _id: 'City 1', averageGPA: 3.003823529411765 },
  { _id: 'City 5', averageGPA: 3.0607499999999996 }
]
db>
```

Explanation of the Output :

Let's break down the aggregation pipeline and explain each stage:

1. **$group Stage**:
    - This stage groups the documents by the homeCity field.
    - The _id field in the $group stage specifies the field to group by, which is "$homeCity".
    - The averageGPA field is calculated using the $avg operator on the GPA field. This computes the average GPA for each home city.
2. **$sort Stage**:
    - This stage sorts the resulting documents by averageGPA in descending order (indicated by -1).
    - Sorting in descending order means cities with higher average GPAs will appear first in the output.

Pushing All Courses into a Single Array

```
db.students.aggregate([
  { $project: { _id: 0, allCourses: { $push: "$courses" } } }
]);
```

Explanation:

- `$project`: Transforms the input documents.
  - `_id: 0`: Excludes the `_id` field from the output documents.
  - `allCourses`: Uses the `$push` operator to create an array. It pushes all elements from the "courses" field of each student document into the `allCourses` array.

Result:

This will return a list of documents, each with an `allCourses` array containing all unique courses offered (assuming courses might be duplicated across students).

To push all courses into a single array for each student in MongoDB using the aggregation framework, you can use the $group and $push stages. Below is an example aggregation pipeline along with an explanation of each stage.

Assuming your collection is named students, and each document has a name field for the student's name and a courses field which is an array of course names, you can use the following pipeline

For example:

```
db.students.aggregate([
  {
    $group: {
      _id: "$name",
      allCourses: { $push: "$courses" }
    }
```

```
  },
  {
    $project: {
      _id: 0,
      name: "$_id",
      allCourses: { $reduce: {
        input: "$allCourses",
        initialValue: [],
        in: { $concatArrays: ["$$value", "$$this"] }
      }}
    }
  }
])
```

Explanation of the Pipeline

1. **$group Stage**:
    - The $group stage groups the documents by the name field.
    - The _id field is set to "$name", grouping by each student's name.
    - The allCourses field uses the $push operator to create an array of arrays of courses for each student. This step results in a document for each student where allCourses is an array of arrays of courses they have taken.
2. **$project Stage**:
    - The $project stage reshapes the documents.
    - The _id field is excluded from the output by setting it to 0.

- The name field is set to the value of _id from the previous stage (which contains the student's name).
- The allCourses field is processed using the $reduce operator to flatten the array of arrays into a single array. The $reduce operator iterates over each element in allCourses, starting with an empty array (initialValue: []), and concatenates each sub-array ($$this) to the accumulated array ($$value).

Output

Suppose I am using students collection contains the following documents:

```
[
  { "name": "Alice", "courses": ["Math", "Science"] },
  { "name": "Alice", "courses": ["History"] },
  { "name": "Bob", "courses": ["Math"] },
  { "name": "Bob", "courses": ["Science", "Art"] }
]
```

Output:

```
[
  { "name": "Alice", "allCourses": ["Math", "Science", "History"] },
  { "name": "Bob", "allCourses": ["Math", "Science", "Art"] }
]
```

Explanation of the Example Output

- **Alice**:
  - Courses: ["Math", "Science"] and ["History"]
  - The $push operator initially creates an array of arrays: [["Math", "Science"], ["History"]].
  - The $reduce operator then flattens this into a single array: ["Math", "Science", "History"].
- **Bob**:

- Courses: ["Math"] and ["Science", "Art"]
- The $push operator initially creates an array of arrays: [["Math"], ["Science", "Art"]].
- The $reduce operator then flattens this into a single array: ["Math", "Science", "Art"].

Collect Unique Courses Offered (Using $addToSet):

```
db.candidates.aggregate([
 { $unwind: "$courses" },
 { $group: { _id: null, uniqueCourses: { $addToSet: "$courses" } } }
]);
```

To collect unique courses offered using the $addToSet operator in MongoDB, you can use the aggregation framework. The $addToSet operator adds each value to the resulting set only once, ensuring that all values in the resulting array are unique.

MongoDB Aggregation :

Assuming I have a students collection where each document contains a courses field which is an array of course names, you can use the following pipeline:

Explanation :

1. $unwind Stage:
    - The $unwind stage deconstructs the courses array, outputting a document for each element in the array.
    - This means if a student document has an array of courses, each course will become its own document with the rest of the fields from the original document.
2. $group Stage:

- The $group stage groups all documents (since _id is set to null, it groups all documents together into a single group).
- The uniqueCourses field is created using the $addToSet operator, which collects unique values from the courses field across all documents.

3. $project Stage:
   - The $project stage reshapes the document.
   - The _id field is excluded from the output by setting it to 0.
   - The uniqueCourses field is included in the output.

Example Collection

Suppose your students collection contains the following documents:

json
Copy code

```
[
  { "name": "Alice", "courses": ["Math", "Science"] },
  { "name": "Bob", "courses": ["Math", "History"] },
  { "name": "Charlie", "courses": ["Science", "Art"] },
  { "name": "Dave", "courses": ["Math", "Science", "Art"] }
]
```

Example Output

```
[
  { "name": "Alice", "courses": "Math" },
  { "name": "Alice", "courses": "Science" },
  { "name": "Bob", "courses": "Math" },
  { "name": "Bob", "courses": "History" },
  { "name": "Charlie", "courses": "Science" },
  { "name": "Charlie", "courses": "Art" },
  { "name": "Dave", "courses": "Math" },
```

{ "name": "Dave", "courses": "Science" },

{ "name": "Dave", "courses": "Art" }

]

- The $group stage then groups all these documents together and collects unique courses values into the uniqueCourses array using $addToSet.
- The $project stage removes the _id field and includes only the uniqueCourses field in the final output.

What does it do ?

```
db> db.candidates.aggregate([
...   { $unwind: "$courses" }, // Deconstruct courses array
...   { $group: { _id: null, uniqueCourses: { $addToSet: "$courses" } } }
que courses
... ]);
[
  {
    _id: null,
    uniqueCourses: [
      'Sociology',
      'Literature',
      'Ecology',
      'Physics',
      'Mathematics',
      'Marine Science',
      'Artificial Intelligence',
      'Art History',
      'Creative Writing',
      'Robotics',
      'Environmental Science',
      'Biology',
      'Statistics',
      'Music History',
      'Philosophy',
      'Film Studies',
      'Engineering',
      'Computer Science',
      'English',
      'Psychology',
      'Chemistry',
      'Political Science',
```

db.candidates.aggregate() - This starts an aggregation pipeline on the "candidates" collection.

- { $unwind: "$courses" } - This stage deconstructs the courses array for each candidate, creating a new document for each course.
- { $group: { _id: null, uniqueCourses: { $addToSet: "$courses" } } } - This stage groups the documents by null (i.e., all documents into one group) and creates a new array called uniqueCourses that contains the unique courses taken by all candidates. The $addToSet operator adds each course to the uniqueCourses array only if it is not already present.

The output of the pipeline is a single document with the _id field set to null and the uniqueCourses field containing an array of all the unique courses taken by any candidate in the "candidates" collection.

Aggregation pipelines

Introduction :

An aggregation pipeline is a sequence of stages used to process data in MongoDB. It transforms documents into aggregated results. Stages include $match for filtering, $project for selecting fields, $group for grouping documents, $sort for sorting results, $limit for limiting output, and many more. Pipelines are powerful for complex data analysis, calculations, and summaries. They enhance data exploration and decision-making. Example: Calculate average order value by customer, find top-selling products, analyze sales trends.

How does it works?

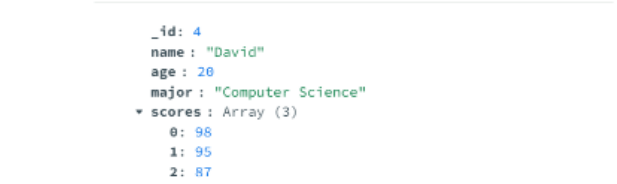
- **Documents as Input:** The pipeline starts with a collection of documents.
- **Stages:** Documents are processed through a series of stages. Each stage performs an operation on the input documents and passes the results to the next stage.
- **Output:** The final stage produces the aggregated result.

Use Cases:

- Calculating metrics (average order value, customer lifetime value).
- Generating reports and dashboards.
- Data cleaning and preparation.
- Data enrichment and transformation.
- Real-time analytics and monitoring.

Lets Build new Dataset

- Download collection here
- Upload the new collection with name "students6"



Explanation

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.

Now let us see some of examples :

---

1. **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
])
```

Explanation :

This code will filter the students collection for students older than 23, sort the results by age in descending order, and then project only the name and age fields.

Ans. 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 } } }, // 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>
```

The output is an array of documents, each containing the name and age of a student who is older than 23. The documents are sorted in descending order by age.

This means that the student named Alice is 25 years old.

The output shows that there are two students who are older than 23: Charlie and Alice. Charlie is older than Alice.

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

Explanation :

a code that finds all students with an age less than 23, sorts them by name in ascending order, and only returns their name and score.

This output shows that Alice, Bob, and Charlie are all students with an age less than 23, and their scores are 95, 88, and 92, respectively.

Ans: Find students with an average score (from scores array) above 85 and skip the first document

Explanation :

The output of the code is a single document containing the name and average score of a student named David. The average score is 93.33333333333333.

The code performs the following steps:

1. Projects the _id, name, and average score of each student in the students6 collection. The average score is calculated using the $avg operator on the scores field.
2. Filters the documents to only include those where the average score is greater than 85.
3. Skips the first document in the filtered results.

3A. : Find students with an average score (from scores array) below 86 and skip the first 2 documents

This code will filter the students collection for students older than 23, sort the results by age in descending order, and then project only the name and age fields.

Output :

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

- **Project necessary fields:**
  - Selects the name field from each document in the students6 collection.
  - Calculates the average of the scores array for each document and stores it in a new field called averageScore.
  - Excludes the _id field.
- **Filter by average score:**
  - Retains only documents where the calculated averageScore is greater than 85.