# AGGREGATION OPERATOR

Aggregation operators in MongoDB allow you to perform advanced data analysis and transformations on your collections. The aggregation framework provides an efficient and expressive way to calculate aggregations, including filtering, grouping, sorting, reshaping, and more. Below are some common aggregation operators and simulated output examples.

## Common Aggregation Operators

- **$match**: Filters the documents to pass only those that match the specified condition(s) to the next pipeline stage.

- **$group**: Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group.

- **$sort**: Sorts all input documents and returns them in the specified order.

- **$project**: Reshapes each document in the stream, such as by adding new fields or removing existing fields.

- **$limit**: Limits the number of documents passed to the next stage in the pipeline.

- **$skip**: Skips over a specified number of documents from the pipeline and passes the rest to the next stage.

- **$unwind**: Deconstructs an array field from the input documents to output a document for each element.

- **$lookup**: Performs a left outer join to an unsharded collection in the same database to filter in documents from the "joined" collection for processing.

## Example Collection

Consider a collection named `orders` with the following documents:

```
[
    { "_id": 1, "product": "A", "quantity": 10, "price": 100, "date": "2024-07-01" },
    { "_id": 2, "product": "B", "quantity": 5, "price": 200, "date": "2024-07-02" },
    { "_id": 3, "product": "A", "quantity": 8, "price": 100, "date": "2024-07-03" },
    { "_id": 4, "product": "C", "quantity": 15, "price": 300, "date": "2024-07-04" },
    { "_id": 5, "product": "B", "quantity": 7, "price": 200, "date": "2024-07-05" }
]
```

## Example Aggregation Pipeline

### 1. $match: Filter orders for product "A"

```
db.orders.aggregate([
    { $match: { product: "A" } }
])
```

Output:

```
[
    { "_id": 1, "product": "A", "quantity": 10, "price": 100, "date": "2024-07-01" },
    { "_id": 3, "product": "A", "quantity": 8, "price": 100, "date": "2024-07-03" }
]
```

2. **$group**: Group by product and calculate total quantity

```
db.orders.aggregate([
    { $group: { _id: "$product", totalQuantity: { $sum: "$quantity" } } }
])
```

Output:

```
[
    { "_id": "A", "totalQuantity": 18 },
    { "_id": "B", "totalQuantity": 12 },
    { "_id": "C", "totalQuantity": 15 }
]
```

3. **$sort**: Sort by total quantity in descending order

```
db.orders.aggregate([
    { $group: { _id: "$product", totalQuantity: { $sum: "$quantity" } } },
    { $sort: { totalQuantity: -1 } }
])
```

Output:

```
[
    { "_id": "C", "totalQuantity": 15 },
    { "_id": "A", "totalQuantity": 18 },
    { "_id": "B", "totalQuantity": 12 }
]
```

These examples illustrate how we can use MongoDB's aggregation framework to filter, group, sort, reshape, and limit your data to extract meaningful insights.

## Arithmetic Operators

Used to perform arithmetic operations on numbers:

- **$add**: Adds numbers to return the sum

```
{ $add: ["$price", "$tax"] }
```

- **$subtract**: Subtracts the second value from the first.

```
{ $subtract: ["$total", "$discount"] }
```

- **$multiply**: Multiplies numbers to return the product.

```
{ $multiply: ["$price", "$quantity"] }
```

- **$divide**: Divides the first number by the second.

```
{ $divide: ["$total", "$quantity"] }
```

- **$mod**: Returns the remainder of the division of the first number by the second.

```
{ $mod: ["$total", 10] }
```

## Array Operators

Used to perform operations on arrays:

- **$arrayElemAt**: Returns the element at the specified array index.

```
{ $arrayElemAt: ["$items", 0] }
```

- **$concatArrays**: Concatenates arrays to return a new array.

```
{ $concatArrays: ["$items", "$discountedItems"] }
```

- **$filter**: Selects a subset of the array to return based on the specified condition.

```
{
  $filter: {
    input: "$items",
    as: "item",
    cond: { $gt: ["$$item.price", 100] }
  }
}
```

- **$size**: Returns the number of elements in the array.

```
{ $size: "$items" }
```

- **$slice**: Returns a subset of an array.

```
{ $slice: ["$items", 5] }
```

## Benefits of using Aggregation Pipeline:

- **Efficient data summarization:** Allows you to perform complex calculations and data manipulation without retrieving entire documents.expand_more
- **Modular and reusable:** Each stage operates independently, making pipelines easy to understand and reuse.
- **Scalability:** Works well with large datasets as it processes documents in stages.

### Syntax

db.collection.aggregate(<AGGREGATE OPERATION>)

### Getting Started with Aggregation:

Here's a basic example using `$match` and `$group` to find the total number of orders per customer:

```
db.orders.aggregate([
  { $match: { "status": "completed" } },
  { $group: { _id: "$customer_id", total_orders: { $sum: 1 } } }
])
```

This pipeline first filters for completed orders ($match) and then groups them by customer ID (_id). Within each group, it calculates the total number of orders ($sum: 1).

By combining different operators, we can create powerful aggregation pipelines to analyze and transform your data effectively in MongoDB.

### Example Aggregation Pipeline

Here's an example pipeline that combines several of these operators:

```
db.orders.aggregate([
  { $match: { status: "A" } },
  {
    $group: {
      _id: "$cust_id",
      totalAmount: { $sum: "$amount" },
      avgAmount: { $avg: "$amount" }
    }
  },
  { $sort: { totalAmount: -1 } },
  { $limit: 5 },
  {
    $project: {
      _id: 0,
      customerId: "$_id",
      totalAmount: 1,
      avgAmount: 1
    }
  }
])
```

### Average GPA of All Students

Here's how you can calculate the average GPA of all students in MongoDB using the aggregation pipeline:

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

**Output**

```
[ { _id : null, averageGPA : 2.98556 }]
```

### Explanation:

**1.$group:** This operator groups all documents in the "students" collection into a single group (_id: null). Since we want the average GPA for all students, a single group suffices.

**2.$avg:** Within the $group stage, we use the $avg operator to calculate the average value of the "gpa" field across all documents in the group.

**3.Output:** The aggregation pipeline returns a document with two fields:

- _id: Since we set it to null, this field will be null in the output.
- average_gpa: This field will contain the calculated average GPA value.

### Minimum and Maximum Age:

Here's how you can find the minimum and maximum age in MongoDB and display the output:

```
db.students.aggregate([ { $group: { _id: 0, min_age: { $min: "$age" }, max_age: { $max: "$age" } } } ])
```

**Output:**

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

### Explanation:

1. **$project:** This operator allows you to specify which fields to include or exclude from the output documents.
2. **_id: 0:** We set the _id field to 0 to exclude it from the output as it's not relevant for this query.
3. **$min and $max:** Within the $project stage, we use:
   - $min: "$age" to find the minimum value of the "age" field across all documents.
   - $max: "$age" to find the maximum value of the "age" field across all documents.
4. **Output:** The aggregation pipeline returns a document with two fields:
   - min_age: This field will contain the minimum age found in the collection.
   - max_age: This field will contain the maximum age found in the collection.

### Pushing All Courses into a Single Array

To push all courses into a single array in MongoDB, we can use the aggregation framework. Here's a step-by-step guide to achieve this:

1. **Connect to MongoDB**: Ensure we are connected to our MongoDB instance. We can use the MongoDB shell, MongoDB Compass, or any MongoDB client library for our preferred programming language.

2. **Aggregation Pipeline**: Use the aggregation framework to group all documents and collect the courses into a single array.

```
db.students.aggregate([
  {
    $group: {
      _id: null,
      allCourses: { $push: "$courses" }
    }
  },
  {
    $project: {
      _id: 0,
      allCourses: { $reduce: {
        input: "$allCourses",
        initialValue: [],
        in: { $concatArrays: ["$$value", "$$this"] }
      }}
    }
  }
])
```

**Explanation**

- **$group**: Groups all documents (students) together with _id: null, so we can aggregate all courses. The $push operator creates an array of arrays (each student's courses).
- **$project**: Projects the result to format the output. The $reduce operator flattens the array of arrays into a single array.

### Sample Data and Output:

Let's assume our collection `students` has the following documents:

```
[
  { "_id": 1, "name": "Alice", "courses": ["Math", "Science"] },
  { "_id": 2, "name": "Bob", "courses": ["History", "Math"] },
  { "_id": 3, "name": "Charlie", "courses": ["Art", "Math", "Science"] }
]
```

When we run the aggregation pipeline, the output will be:

```
{
  "allCourses": ["Math", "Science", "History", "Math", "Art", "Math", "Science"]
```

## Execution Output:

```
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',
```