Aggregation

Aggregations in MongoDB are a powerful way to process and analyze data within the database. Aggregation operations allow you to perform data transformations, grouping, filtering, sorting, and calculations on collections, providing you with the ability to extract meaningful insights from your data.

MongoDB's aggregation framework includes various pipeline stages and operators to achieve complex data manipulations. Aggregations are commonly used for generating reports, statistics, and summaries of data. Below are some essential concepts and stages used in MongoDB aggregations:

1. Aggregation Pipeline: The aggregation pipeline is a series of stages that process documents sequentially. Each stage takes the output of the previous stage and performs specific operations on it.

2. Pipeline Stages: The aggregation pipeline consists of several stages, such as `$match`, `$group`, `$project`, `$sort`, `$limit`, `$skip`, `$unwind`, and more. Each stage performs a specific operation on the data.

3. Operators: MongoDB provides various aggregation operators to perform calculations and transformations, like `$sum`, `$avg`, `$max`, `$min`, `$addToSet`, `$push`, `$concat`, `$substr`, `$filter`, etc.

4. Grouping: The `$group` stage is used to group documents based on a specific field or set of fields and then apply aggregate functions to the grouped data.

5. Projection: The `$project` stage allows you to reshape and include/exclude specific fields from the documents in the output.

6. Sorting: The `$sort` stage is used to sort the documents based on specific fields.

7. Filtering: The `$match` stage filters documents based on specified criteria.

8. Unwinding: The `$unwind` stage is used to transform arrays into separate documents, allowing you to perform further operations on the array elements.

9. Limiting and Skipping: The `$limit` and `$skip` stages control the number of documents in the output.

10. Geospatial Aggregation: MongoDB supports geospatial aggregations, such as `$geoNear` and `$geoWithin`, for performing geospatial queries.

Overall, MongoDB's aggregation framework provides a versatile and efficient way to analyze, transform, and summarize data directly within the database, reducing the need for extensive data processing on the application side. It is particularly valuable for handling complex querying and reporting requirements.

1.Structure of an Aggregation Pipeline

```
db.collection.aggregate([
   {
      $stage1: {
         { expression1 },
         { expression2 }...
      }
},
{
      $stage2: {
         { expression1 }...
      }
   }
])
```

2.Using $match and $group Stages in a MongoDB Aggregation Pipeline

$match
The $match stage filters for documents that match specified conditions. Here's the code for $match:

```
{
  $match: {
    "field_name": "value"
  }
}
```

$group
The $group stage groups documents by a group key.

```
{
  $group:
    {
      _id: <expression>, // Group key
      <field>: { <accumulator> : <expression> }
    }
 }
```

$match and $group in an Aggregation Pipeline
The following aggregation pipeline finds the documents with a field named "state" that matches a value "CA" and then groups those documents by the group key "$city" and shows the total number of zip codes in the state of California.

```
db.zips.aggregate([
{
  $match: {
    state: "CA"
   }
},
{
  $group: {
    _id: "$city",
    totalZips: { $count : { } }
  }
}
])
```

3 Using $sort and $limit Stages in a MongoDB Aggregation Pipeline

$sort
The $sort stage sorts all input documents and returns them to the pipeline in sorted order. We use 1 to represent ascending order, and -1 to represent descending order.

```
{
   $sort: {
        "field_name": 1
    }
}
```

$limit
The $limit stage returns only a specified number of records.

```
{
  $limit: 5
}
```

$sort and $limit in an Aggregation Pipeline
The following aggregation pipeline sorts the documents in descending order, so the documents with the greatest pop value appear first, and limits the output to only the first five documents after sorting.

```
db.zips.aggregate([
{
  $sort: {
    pop: -1
  }
},
{
  $limit:  5
}
])
```

4.Using $project, $count, and $set Stages in a MongoDB Aggregation Pipeline

$project
The $project stage specifies the fields of the output documents. 1 means that the field should be included, and 0 means that the field should be suppressed. The field can also be assigned a new value.

```
{
    $project: {
        state:1,
        zip:1,
        population:"$pop",
    _id:0
  }
}
```

$set

The $set stage creates new fields or changes the value of existing fields, and then outputs the documents with the new fields.

```
{
  $set: {
    place: {
      $concat:["$city",",","$state"]
    },
    pop:10000
   }
 }
```

$count

The $count stage creates a new document, with the number of documents at that stage in the aggregation pipeline assigned to the specified field name.

```
{
 $count: "total_zips"
}
```

5.$out

In MongoDB, the `**$out**` stage is used in the aggregation pipeline to output the results of the aggregation into a new or existing collection. It allows you to save the result of an aggregation operation into a separate collection for further analysis or use.

Syntax:

```
db.sourceCollection.aggregate([
 // Your aggregation stages here
 // ...
 { $out: "outputCollection" }
])
```

- **sourceCollection is the name of the collection you want to perform the aggregation on.**
- **outputCollection is the name of the collection where you want to store the aggregation results. If this collection does not exist, MongoDB will create it.**

Here's an example of how you can use the `$out` stage in an aggregation pipeline:

Assume you have a collection named "orders" with documents like this:

```
{
  "_id": 1,
  "orderDate": "2023-07-20",
  "amount": 100
}
{
  "_id": 2,
  "orderDate": "2023-07-21",
  "amount": 150
}
{
  "_id": 3,
  "orderDate": "2023-07-22",
  "amount": 200
}
```

Now, let's say you want to calculate the total sales for each day and save the results into a new collection called "dailySales."

You can achieve this using the `$group` stage to calculate the total sales for each day and then use the `$out` stage to save the results into the "dailySales" collection:

```
db.orders.aggregate([
  {
    $group: {
      _id: "$orderDate",
      totalSales: { $sum: "$amount" }
    }
  },
  {
    $out: "dailySales"
  }
])
```

After running this aggregation query, the results will be stored in the "dailySales" collection:

```
// dailySales collection
{
  "_id": "2023-07-20",
  "totalSales": 100
}
```

```
{
  "_id": "2023-07-21",
  "totalSales": 150
}
{
  "_id": "2023-07-22",
  "totalSales": 200
}
```

Please note that the `$out` stage will replace the existing "dailySales" collection if it already exists. If you want to create a new collection, make sure to provide a new name that does not conflict with existing collections.

Keep in mind that using the `$out` stage has some implications:
- It can be a resource-intensive operation, especially for large datasets, as it rewrites the entire collection.
- The user executing the aggregation pipeline needs to have the necessary privileges to create or replace a collection.

Always use the `$out` stage with caution, considering the performance and data integrity implications.