Aggregation Pipeline

On this page

An aggregation pipeline consists of one or more **stages** that process documents:

- Each stage performs an operation on the input documents. For example, a stage can filter documents, group documents, and calculate values.
- The documents that are output from a stage are passed to the next stage.
- An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum, and minimum values.

You can update documents with an aggregation pipeline if you use the stages shown in **Updates with Aggregation Pipeline.**

NOTE

Aggregation pipelines run with the `db.collection.aggregate()` method do not modify documents in a collection, unless the pipeline contains a `$merge` or `$out` stage.

You can **run aggregation pipelines in the UI** for deployments hosted in **MongoDB Atlas.**

When you run aggregation pipelines on MongoDB Atlas deployments in the MongoDB Atlas UI, you can preview the results at each stage.

Complete Aggregation Pipeline Examples

This section shows aggregation pipeline examples that use the following pizza `orders` collection:

```
db.orders.insertMany( [
   { _id: 0, name: "Pepperoni", siz
     quantity: 10, date: ISODate( "
   { _id: 1, name: "Pepperoni", siz
     quantity: 20, date : ISODate(
   { _id: 2, name: "Pepperoni", siz
     quantity: 30, date : ISODate(
   { _id: 3, name: "Cheese", size:
     quantity: 15, date : ISODate(
   { _id: 4, name: "Cheese", size:
     quantity:50, date : ISODate( "
   { _id: 5, name: "Cheese", size:
     quantity: 10, date : ISODate(
   { _id: 6, name: "Vegan", size: "
     quantity: 10, date : ISODate(
   { _id: 7, name: "Vegan", size: "
     quantity: 10, date : ISODate(
] )
```

Calculate Total Order Quantity

The following aggregation pipeline example contains two **stages** and returns the total order quantity of medium size pizzas grouped by pizza name:

```
db.orders.aggregate( [

   // Stage 1: Filter pizza order d
   {
      $match: { size: "medium" }
   },

   // Stage 2: Group remaining docu
   {
      $group: { _id: "$name", total
   }

] )
```

The `$match` stage:

- Filters the pizza order documents to pizzas with a `size` of `medium`.
- Passes the remaining documents to the `$group` stage.

The `$group` stage:

- Groups the remaining documents by pizza `name`.
- Uses `$sum` to calculate the total order `quantity` for each pizza `name`. The total is stored in the `totalQuantity` field returned by the aggregation pipeline.

Example output:

```
[
   { _id: 'Cheese', totalQuantity: 50 }
   { _id: 'Vegan', totalQuantity: 10 },
   { _id: 'Pepperoni', totalQuantity: 2
]
```

Calculate Total Order Value and Average Order Quantity

The following example calculates the total pizza order value and average order quantity between two dates:

```
db.orders.aggregate( [

   // Stage 1: Filter pizza order d
   {
      $match:
      {
         "date": { $gte: new ISODat
      }
   },

   // Stage 2: Group remaining docu
   {
      $group:
      {
         _id: { $dateToString: { fo
         totalOrderValue: { $sum: {
         averageOrderQuantity: { $a
      }
   },

   // Stage 3: Sort documents by to
   {
      $sort: { totalOrderValue: -1
   }

] )
```

The `$match` stage:

- Filters the pizza order documents to those in a date range specified using `$gte` and `$lt`.
- Passes the remaining documents to the `$group` stage.

The `$group` stage:

- Groups the documents by date using `$dateToString`.
- For each group, calculates:
  - Total order value using `$sum` and `$multiply`.
  - Average order quantity using `$avg`.
- Passes the grouped documents to the `$sort` stage.

The `$sort` stage:

- Sorts the documents by the total order value for each group in descending order (`-1`).
- Returns the sorted documents.

Example output:

```
[
   { _id: '2022-01-12', totalOrderValue
   { _id: '2021-03-13', totalOrderValue
   { _id: '2021-03-17', totalOrderValue
   { _id: '2021-01-13', totalOrderValue
]
```

TIP

See also:

- Aggregation with User Preference Data
- Aggregation with the Zip Code Data Set
- Updates with Aggregation Pipeline

Additional Aggregation Pipeline Stage Details

An aggregation pipeline consists of one or more **stages** that process documents:

- A stage does not have to output one document for every input document. For example, some stages may produce new documents or filter out documents.
- The same stage can appear multiple times in the pipeline with these stage exceptions: `$out`, `$merge`, and `$geoNear`.
- To calculate averages and perform other calculations in a stage, use **aggregation expressions** that specify **aggregation operators**. You will learn more about aggregation expressions in the next section.

For all aggregation stages, see **Aggregation Stages.**

Aggregation Expressions and Operators

Some aggregation pipeline stages accept **expressions.** Operators calculate values based on input expressions.

In the MongoDB Query Language, you can build expressions from the following components:

Component	Example
Constants	`3`
Operators	`$add`
Field path expressions	`"$<path.to.field>"`

For example, `{ $add: [ 3, "$inventory.total" ] }` is an expression consisting of the `$add` operator and two input expressions:

- The constant `3`
- The **field path expression** `"$inventory.total"`

DOCS MENU Ask MongoDB AI

The expression returns the result of adding 3 to the value at path `inventory.total` of the input document.

Field Paths

Field path expressions are used to access fields in input documents. To specify a field path, prefix the field name or the **dotted field path** (if the field is in an embedded document) with a dollar sign `$`. For example, `"$user"` to specify the field path for the `user` field or `"$user.name"` to specify the field path to the embedded `"user.name"` field.

`"$<field>"` is equivalent to `"$$CURRENT.<field>"` where the `CURRENT` is a system variable that defaults to the root of the current

▸ Self-Managed Deployments
▸ FAQ
▸ Reference

object, unless stated otherwise in specific stages.

For more information and examples, see **Field Paths.**

Run an Aggregation Pipeline

To run an aggregation pipeline, use:

- `db.collection.aggregate()` or
- `aggregate`

Update Documents Using an Aggregation Pipeline

To update documents with an aggregation pipeline, use:

Command	`mongosh` Methods
`findAndModify`	db.collection.findOneAndUpdate() db.collection.findAndModify()
`update`	db.collection.updateOne() db.collection.updateMany() Bulk.find.update() Bulk.find.updateOne() Bulk.find.upsert()

Other Considerations

Aggregation Pipeline Limitations

An aggregation pipeline has limitations on the value types and the result size. See **Aggregation Pipeline Limits.**

Aggregation Pipelines and Sharded Collections

An aggregation pipeline supports operations on sharded collections. See **Aggregation Pipeline and Sharded Collections.**

Aggregation Pipelines as an Alternative to Map-Reduce

Starting in MongoDB 5.0, **map-reduce** is deprecated:

- Instead of **map-reduce**, you should use an **aggregation pipeline.** Aggregation pipelines provide better performance and usability than map-reduce.
- You can rewrite map-reduce operations using **aggregation pipeline stages**, such as `$group`, `$merge`, and others.
- For map-reduce operations that require custom functionality, you can use the `$accumulator` and `$function` aggregation operators. You can use those operators to define custom aggregation expressions in JavaScript.

For examples of aggregation pipeline alternatives to map-reduce, see:

- **Map-Reduce to Aggregation Pipeline**
- **Map-Reduce Examples**

Learn More

To learn more about aggregation pipelines, see:

- **Expression Operators**
- **Aggregation Stages**
- **Practical MongoDB Aggregations**

Deployment Options