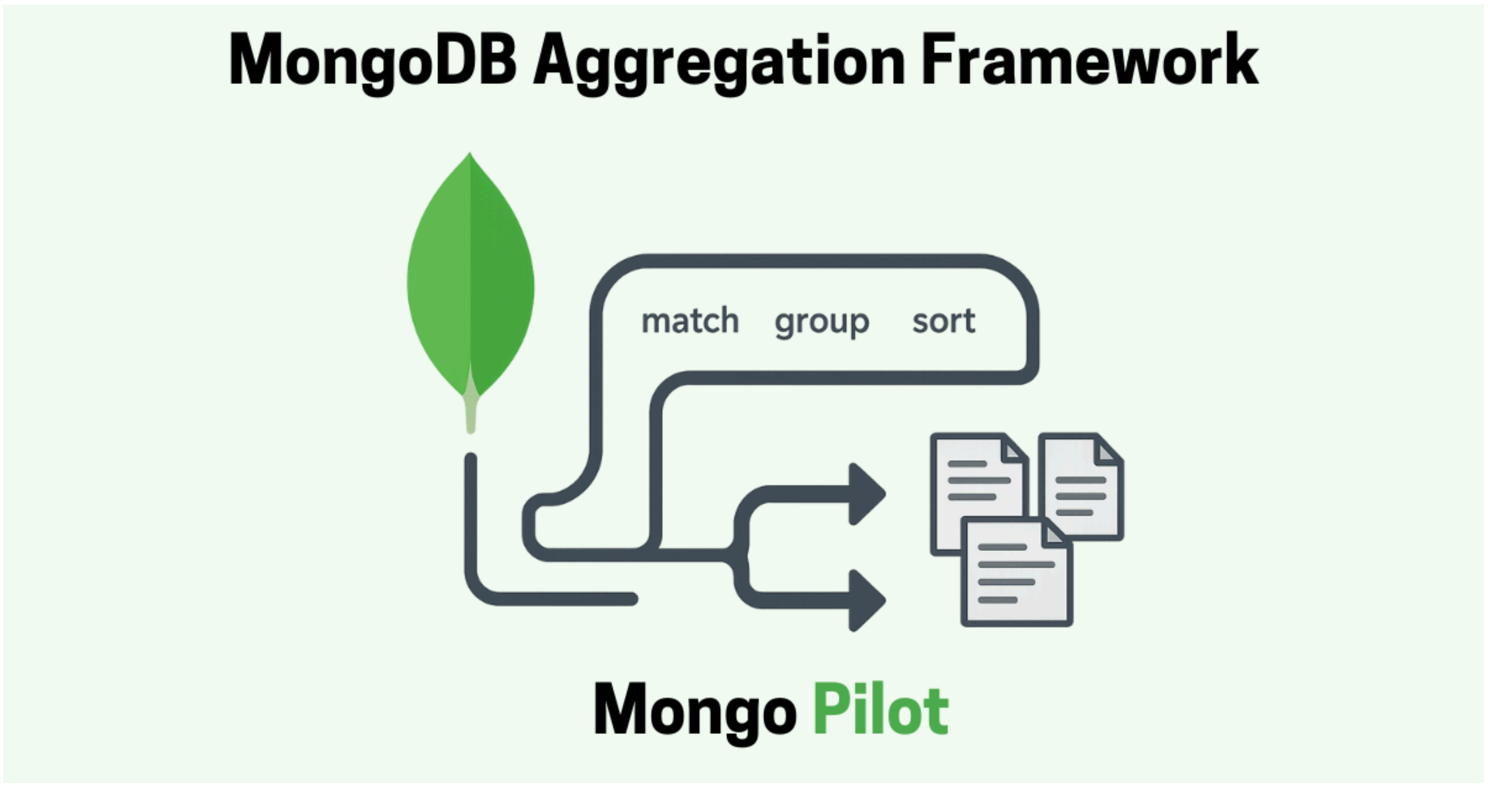
RESOURCES

# MongoDB Aggregation : A Beginner's Guide



MongoDB's aggregation framework is a powerful tool for processing and analyzing data in your database. Unlike simple queries that just filter and return documents, aggregation allows you to transform, calculate, group, and manipulate your data in sophisticated ways.In this guide, we'll walk through the essential aggregation pipeline stages using our restaurant Orders collection to demonstrate real-world applications.

👉 If you need a Cheat Sheet of the top 10 MongoDB Aggregation operators you should master with simple examples.

## Our Sample Data: The Restaurant Orders Collection

Our collection contains order records from a restaurant chain with fields like:

```
{
  "_id": { "$oid": "60a6e5d5dcb5a7853a78c123" },
  "productId": "P001",
  "productName": "Margherita Pizza",
  "category": "main",
  "price": 12.99,
  "orderId": "ORD001",
  "customerId": "CUST001",
  "orderDate": { "$date": "2023-05-15T18:23:45Z" },
  "status": "completed",
  "paymentMethod": "credit",
  "storeLocation": "Downtown",
  "quantity": 2,
  "vegetarian": true
}
```

## Understanding Aggregation Pipelines

An aggregation pipeline consists of one or more stages that process documents as they pass through the pipeline. Each stage performs a specific operation on the input documents, and the output from one stage becomes the input for the next stage.Let’s explore the most commonly used stages:

## $match stage : Filtering Documents

The $match stage filters documents to pass only those that match the specified conditions, similar to a find() query. It's most efficient when used early in the pipeline to reduce the number of documents processed by later stages.

How to use the MongoDB $match stage?

```
db.Orders.aggregate([
  {
    $match: {
      category: "main",
      vegetarian: true
    }
  }
])
```

Example output:

```
[
  {
    "_id": { "$oid": "60a6e5d5dcb5a7853a78c123" },
    "productId": "P001",
    "productName": "Margherita Pizza",
    "category": "main",
    "price": 12.99,
    "orderId": "ORD001",
    "customerId": "CUST001",
    "orderDate": { "$date": "2023-05-15T18:23:45Z" },
    "status": "completed",
    "paymentMethod": "credit",
    "storeLocation": "Downtown",
    "quantity": 2,
    "vegetarian": true
  },
  {
    "_id": { "$oid": "60a6e5d5dcb5a7853a78c129" },
    "productId": "P001",
    "productName": "Margherita Pizza",
    "category": "main",
    "price": 12.99,
    "orderId": "ORD004",
    "customerId": "CUST004",
    "orderDate": { "$date": "2023-05-16T18:45:00Z" },
    "status": "in-progress",
    "paymentMethod": "credit",
    "storeLocation": "Suburbs",
    "quantity": 1,
    "vegetarian": true
  }
  // ... other vegetarian main dishes
]
```

And here is how to easily do it with Mongo Pilot visual pipeline builder 😉

Download Mongo Pilot now

## $sort stage: Ordering Results

The $sort stage orders documents based on specified fields. This is useful when you need data in a specific sequence, like chronological order or sorted by price.

How to use $sort stage?

```
  db.Orders.aggregate([
    { $match: { category: "main" } },
    { $sort: { price: -1 } }  // -1 for descending, 1 for ascending
  ])
```

Example output:

```
[
  {
    "_id": { "$oid": "60a6e5d5dcb5a7853a78c162" },
    "productId": "P035",
    "productName": "Hawaiian Pizza",
    "category": "main",
    "price": 15.50,
    "orderId": "ORD021",
    "customerId": "CUST021",
    "orderDate": { "$date": "2023-05-28T19:10:00Z" },
    "status": "completed",
    "paymentMethod": "cash",
    "storeLocation": "Downtown",
    "quantity": 1,
    "vegetarian": false
  },
  // ... other main dishes in descending price order
]
```

Read also > How to Sort in MongoDB Aggregation Without Killing Performance

## $limit and $skip: Pagination

These stages are essential for pagination. $skip discards a specified number of documents, and $limit restricts the number of documents passed to the next stage.

How to use $limit and $skip?

```
db.Orders.aggregate([
  { $sort: { orderDate: -1 } },
  { $skip: 10 },  // Skip the first 10 documents
  { $limit: 5 }   // Return only 5 documents
])
```

Example output:

```
// Documents 11-15 when sorted by most recent orderDate
[
  {
    "_id": { "$oid": "60a6e5d5dcb5a7853a78c153" },
    "productId": "P004",
    "productName": "Pepperoni Pizza",
    "category": "main",
    "price": 14.99,
    "orderId": "ORD016",
    "customerId": "CUST016",
    "orderDate": { "$date": "2023-05-24T17:15:30Z" },
    "status": "in-progress",
    "paymentMethod": "credit",
    "storeLocation": "Uptown",
    "quantity": 1,
    "vegetarian": false
  },
  // ... 4 more documents
]
```

## $project stage: Reshaping Documents

The $project stage reshapes documents by including, excluding, or transforming fields. It helps create more focused outputs and can compute new fields.
How to use $project?

```
db.Orders.aggregate([
  {
    $project: {
      _id: 0,  // Exclude _id
      item: "$productName",
      cost: { $multiply: ["$price", "$quantity"] },
      location: "$storeLocation"
    }
  }
])
```

Example output:

```
[
  {
    "item": "Margherita Pizza",
    "cost": 25.98,
    "location": "Downtown"
  },
  {
    "item": "Chicken Wings",
    "cost": 8.99,
    "location": "Downtown"
  },
  // ... other transformed documents
]
```

## $group stage: Aggregating Data

The $group stage is at the heart of aggregation – it groups documents by a specified key and applies accumulator expressions to create group-level calculations like sums, averages, or counts. How to use $group?

```
db.Orders.aggregate([
  {
    $group: {
      _id: "$storeLocation",
      totalSales: { $sum: { $multiply: ["$price", "$quantity"] } },
      orderCount: { $sum: 1 },
      averageOrderValue: { $avg: "$price" }
    }
  }
])
```

Example output:

```
[
  {
    "_id": "Downtown",
    "totalSales": 523.45,
    "orderCount": 18,
    "averageOrderValue": 9.74
  },
  {
    "_id": "Uptown",
    "totalSales": 412.30,
    "orderCount": 15,
    "averageOrderValue": 10.25
  },
  {
    "_id": "Suburbs",
    "totalSales": 389.75,
    "orderCount": 17,
    "averageOrderValue": 8.90
  }
]
```

>Read our guide about MongoDB indexes.

## $unwind stage: Deconstruct Arrays

The $unwind stage deconstructs an array field, creating a new document for each element. It’s essential when you need to work with array elements individually.

```
// Assuming we have orders with an 'ingredients' array
db.Orders.aggregate([
  { $match: { productId: "P001" } },
  { $addFields: { ingredients: ["flour", "tomato", "cheese", "basil"] } },
  { $unwind: "$ingredients" }
])
```

Example output of an $unwind stage:

```
[
  {
    "_id": { "$oid": "60a6e5d5dcb5a7853a78c123" },
    "productId": "P001",
    "productName": "Margherita Pizza",
    // ... other fields
    "ingredients": "flour"
  },
  {
    "_id": { "$oid": "60a6e5d5dcb5a7853a78c123" },
    "productId": "P001",
    "productName": "Margherita Pizza",
    // ... other fields
    "ingredients": "tomato"
  },
  {
    "_id": { "$oid": "60a6e5d5dcb5a7853a78c123" },
    "productId": "P001",
    "productName": "Margherita Pizza",
    // ... other fields
    "ingredients": "cheese"
  },
  {
    "_id": { "$oid": "60a6e5d5dcb5a7853a78c123" },
    "productId": "P001",
    "productName": "Margherita Pizza",
    // ... other fields
    "ingredients": "basil"
  }
]
```

## $lookup stage: Performing Joins

The $lookup stage performs a left outer join with another collection, allowing you to incorporate data from related collections.

```
// Assuming we have a Customers collection
db.Orders.aggregate([
  { $limit: 2 },
  {
    $lookup: {
      from: "Customers",
      localField: "customerId",
      foreignField: "customerId",
      as: "customerDetails"
    }
  }
])
```

Example output:

```
[
  {
    "_id": { "$oid": "60a6e5d5dcb5a7853a78c123" },
    "productId": "P001",
    "productName": "Margherita Pizza",
    // ... other order fields
    "customerDetails": [
      {
        "customerId": "CUST001",
        "name": "John Smith",
        "email": "john.smith@example.com",
        "loyaltyPoints": 230
      }
    ]
  },
    // ... second document with its customer details
  ]
```

## $addFields: Enriching Documents

The $addFields stage adds new fields to documents without replacing existing fields (unlike $project which reshapes the whole document).

```
  db.Orders.aggregate([
    {
      $addFields: {
        totalPrice: { $multiply: ["$price", "$quantity"] },
        taxAmount: { $multiply: ["$price", "$quantity", 0.08] }
      }
    },
    { $limit: 2 }
  ])
```

Example output:

```
  [
    {
      "_id": { "$oid": "60a6e5d5dcb5a7853a78c123" },
      "productId": "P001",
      "productName": "Margherita Pizza",
      // ... other original fields
      "totalPrice": 25.98,
      "taxAmount": 2.08
    },
    {
      "_id": { "$oid": "60a6e5d5dcb5a7853a78c124" },
      "productId": "P002",
      "productName": "Chicken Wings",
      // ... other original fields
      "totalPrice": 8.99,
      "taxAmount": 0.72
    }
  ]
```

## $count: Document Counting

The $count stage returns a count of the number of documents at that point in the pipeline, which is cleaner than using $group with a counter.

```
db.Orders.aggregate([
  { $match: { storeLocation: "Downtown", vegetarian: true } },
  { $count: "vegetarianDowntownOrders" }
])
```

Example output:

```
[
  {
    "vegetarianDowntownOrders": 12
  }
]
```

# Combining Stages: A Real-World Example

Let’s build a complex pipeline to demonstrate how these stages work together:

```
db.Orders.aggregate([
  // Stage 1: Filter for completed orders in the last month
  {
    $match: {
      status: "completed",
      orderDate: { $gte: new Date("2023-05-01") }
    }
  },

  // Stage 2 : Project fields

  {

  $project: {

  _id: 1,
  category: 1,
  orderDate: 1,
  quantity: 1,
  storeLocation: 1,
  totalSales: {"$multiply": ["$price","$quantity"]}

  }
  },
  // Stage 3: Group by store location and category
  {
    $group: {
      _id: "$storeLocation",
      totalSales: {"$sum": "$totalSales"},
      itemCount: { $sum: "$quantity" }
    }
  },

  // Stage 4: Sort by total sales descending
  {
    $sort: {
      "totalSales": -1
    }
  },


])
```

Example output:

```
  [
    {
      "_id": "Downtown",
      "totalSales": 253.76,
      "itemCount": 18,
    },
    {
      "_id": "Uptown",
      "totalSales": 189.45,
      "itemCount": 14,

    },
    // ... other location
  ]
```

And here is how to visually build this complete aggregation pipeline in Mongo Pilot, the best MongoDB GUI:

And here is a screenshot of the complete pipeline:

You can also read : 6 Common MongoDB Query Mistakes (and How to Fix Them)

## Simplify Complex Aggregations with Mongo Pilot

While the MongoDB aggregation framework is incredibly powerful, constructing complex pipelines with multiple stages can be challenging, especially for beginners. This is where specialized tools like Mongo Pilot can help. Mongo Pilot is a MongoDB GUI that makes it easy to:

1. Build pipelines visually: Drag and drop stages to create your pipeline. Add operations, fields via the visual query builder.
2. See results in real time: Preview how each stage transforms your data
3. Export queries: Generate code after visual creation
4. Save common pipelines: Store frequently used queries for quick access (coming soon)

Instead of writing and debugging lengthy aggregation pipelines by hand, Mongo Pilot provides an intuitive visual interface that lets you focus on getting insights from your data rather than worrying about syntax.

Download Mongo Pilot now

## Conclusion

MongoDB’s aggregation framework gives you powerful tools to analyze and transform your data right within the database. The pipeline approach makes complex data processing more manageable by breaking it down into discrete, comprehensible stages. As you become more comfortable with these basic stages, you can explore more specialized operators like $facet for multi-faceted aggregations, $bucket for creating histograms, and many others. Whether you’re writing pipelines by hand or using a visual tool like Mongo Pilot, mastering aggregation will dramatically expand what you can do with MongoDB beyond simple CRUD operations.

**AymenLoukil**

Founder, Technical Entrepreneur, International Consultant, Public Speaker

## More from the Mongo Pilot Blog

### Why You Need a GUI for MongoDB

MongoDB is powerful, flexible, and widely adopted but let's be honest: working only from the shell can be… painful. You can query, aggregate, and manage your data with the CLI, but as soon as your collections grow, your productivity drops. That's where a MongoDB GUI (Graphical User Interface) comes in. Let's break down why a

## How to Sort in MongoDB Aggregation Without Killing Performance

Sorting in MongoDB aggregations seems straightforward until your query slows your server to a crawl. Why? The $sort stage can be a performance bottleneck, especially on large datasets. This post explains why and shares proven strategies to optimize sorting, keeping your queries fast and efficient. (Based on MongoDB 8.0 documentation.) Why $sort can Hurt Performance?

## What is an Index in MongoDB (and Why It Matters)

When your MongoDB queries slow down, it’s often because you’re missing the right index. And it’s one of the common MongoDB query mistakes. An index in MongoDB is like the index of a book: it helps you jump directly to the data you need instead of flipping through every page. Without indexes, MongoDB performs a

## Leave a Comment

Name *

Email *

Website

☐ Save my name, email, and website in this browser for the next time I comment.

Post Comment

✓

## The smartest MongoDB GUI

MongoPilot empowers developers to manage MongoDB databases effortlessly with a local LLM for AI-driven queries and an intuitive visual query builder.

DOWNLOAD MONGO PILOT

MongoDB GUI

Smart MongoDB GUI with AI capabilities

MongoDB GUI

Privacy Policy

Terms & Conditions

Our other solutions

Sawtly : Video and audio dubbing

Speetals : Real User Monitoring

Site Speed Consultant