MongoDB Aggregation Framework - Complete Guide

Table of Contents

Introduction to Aggregation Pipeline {#introduction}

What is MongoDB Aggregation?

MongoDB's Aggregation Framework is a powerful data processing engine that allows you to perform complex data transformations and analysis operations directly within the database. Think of it as a pipeline where data flows through multiple stages, with each stage performing a specific operation on the data.

Key Concepts Explained:

Pipeline Concept: Imagine a water pipe with multiple filters. Water enters at one end, passes through various filters (each doing something different), and clean water comes out the other end. Similarly, documents enter the aggregation pipeline, pass through various stages (like $match, $group, $sort), and transformed results come out.

Stage Concept: Each stage in the pipeline is like a workstation in a factory assembly line. Each workstation has a specific job:

- **$match stage** = Quality control (filters out unwanted documents)
- **$group stage** = Assembly (groups related documents together)
- **$sort stage** = Organization (arranges documents in order)
- **$project stage** = Packaging (shapes the final output)

Document Flow: Documents flow sequentially from one stage to the next. The output of one stage becomes the input for the next stage. This sequential processing allows for complex transformations.

Why Use Aggregation Framework?

1. **Performance**: Operations happen directly in the database, reducing network traffic
2. **Flexibility**: Can perform complex transformations that would require multiple queries
3. **Scalability**: MongoDB can distribute aggregation across multiple servers
4. **Rich Operations**: Supports mathematical, string, date, and array operations

Basic Syntax Pattern:

javascript

```javascript
db.collection.aggregate([
  { $stage1: { /* stage 1 operations */ } },
  { $stage2: { /* stage 2 operations */ } },
  { $stage3: { /* stage 3 operations */ } }
])
```

Real-World Analogy:

Think of aggregation like a data assembly line in a factory:

- **Raw materials** = Original documents in your collection
- **Assembly line** = The aggregation pipeline
- **Workstations** = Individual stages ($match, $group, etc.)
- **Finished product** = Final aggregated results

Stage 1: $match - Filter Documents {#match}

Understanding $match

The `$match` stage is your first line of defense in data processing. It acts like a bouncer at a club - it decides which documents are allowed to proceed to the next stage of the pipeline.

Why $match is Important:

1. **Performance Optimization**: By filtering early, you reduce the amount of data that subsequent stages need to process
2. **Index Utilization**: When placed at the beginning, $match can use database indexes for faster filtering
3. **Memory Efficiency**: Fewer documents mean less memory usage in subsequent stages

How $match Works:

$match uses the same query syntax as MongoDB's `find()` method. It examines each document and decides whether to pass it to the next stage based on your filter criteria.

Sample Data Setup:

javascript

```javascript
db.orders.insertMany([
  { _id: 1, customer: "Alice", total: 120, status: "delivered", items: 2, orderDate: new Date("2023-01-15") },
  { _id: 2, customer: "Bob", total: 90, status: "pending", items: 3, orderDate: new Date("2023-01-20") },
  { _id: 3, customer: "Alice", total: 200, status: "cancelled", items: 4, orderDate: new Date("2023-02-01") },
  { _id: 4, customer: "Charlie", total: 50, status: "delivered", items: 1, orderDate: new Date("2023-02-05") },
  { _id: 5, customer: "Bob", total: 300, status: "delivered", items: 5, orderDate: new Date("2023-02-10") }
])
```

Use Cases with Detailed Explanations:

1. Basic Filtering - Find Delivered Orders

javascript

```javascript
db.orders.aggregate([
  { $match: { status: "delivered" } }
])
```

Explanation: This filters the collection to only include documents where the status field equals "delivered". Out of 5 orders, only 3 will pass through (orders 1, 4, and 5).

Business Use Case: "Show me only the orders that have been successfully delivered to customers."

2. Numeric Comparisons - High-Value Orders

javascript

```javascript
db.orders.aggregate([
  { $match: { total: { $gt: 100 } } }
])
```

Explanation: The `$gt` operator finds documents where the total is greater than 100. This filters out smaller orders, leaving only orders 1, 3, and 5.

Business Use Case: "Find all high-value orders above $100 for our premium customer analysis."

3. Multiple Conditions (AND Logic)

javascript

```javascript
db.orders.aggregate([
  { $match: { customer: "Alice", total: { $gt: 100 } } }
])
```

Explanation: This applies multiple conditions simultaneously. Both conditions must be true for a document to pass through. Only Alice's orders with total > 100 will match (orders 1 and 3).

Business Use Case: "Find all high-value orders from Alice to understand her purchasing pattern."

4. OR Logic - Multiple Status Values

```javascript
db.orders.aggregate([
  {
    $match: {
      $or: [
        { status: "pending" },
        { total: { $lt: 100 } }
      ]
    }
  }
])
```

Explanation: The `$or` operator matches documents that satisfy ANY of the conditions. Documents with pending status OR total less than 100 will pass through.

Business Use Case: "Show me orders that are either still pending or are small orders under $100."

5. Array Matching - Multiple Status Values

```javascript
db.orders.aggregate([
  {
    $match: {
      status: { $in: ["delivered", "pending"] }
    }
  }
])
```

Explanation: The `$in` operator matches documents where the field value is in the specified array. This is cleaner than using $or for multiple exact matches.

Business Use Case: "Find all orders that are either delivered or pending (exclude cancelled orders)."

6. Field Comparison with $expr

javascript

```javascript
db.orders.aggregate([
 {
  $match: {
   $expr: { $gt: ["$total", { $multiply: ["$items", 40] }] }
  }
 }
])
```

Explanation: The `$expr` operator allows you to compare fields within the same document. This finds orders where the total is greater than items × 40, indicating high-value items.

Business Use Case: "Find orders where the average item value is above $40."

Performance Best Practices:

1. **Place $match Early**: Always put $match as the first or second stage in your pipeline
2. **Use Indexes**: Ensure your match criteria can use database indexes
3. **Specific Filters**: Use the most selective filters first to eliminate the most documents early

Common Mistakes to Avoid:

1. **Placing $match Late**: Don't put $match after expensive operations like $group or $lookup
2. **Complex Expressions**: Avoid complex $expr operations if simple operators work
3. **Missing Indexes**: Ensure your match criteria have appropriate indexes

---

Stage 2: $project - Shape and Transform Documents {#project}

Understanding $project

The `$project` stage is like a skilled craftsman who reshapes raw materials into finished products. It controls the structure and content of documents that flow through the pipeline, allowing you to:

- **Include or exclude fields** (like choosing which tools to put in a toolbox)
- **Rename fields** (like putting new labels on old containers)
- **Create computed fields** (like calculating totals from individual items)
- **Transform data types** (like converting text to numbers)

How $project Works:

Think of $project as a document template. You define what the output document should look like, and MongoDB transforms each input document to match that template.

Field Selection Rules:

- `1` or `true` = Include this field
- `0` or `false` = Exclude this field
- `"$fieldName"` = Reference the value of fieldName
- `{ $operation: [...] }` = Perform computation

Basic Syntax Pattern:

javascript

```javascript
{
  $project: {
    fieldToInclude: 1,
    fieldToExclude: 0,
    newFieldName: "$oldFieldName",
    computedField: { $multiply: ["$field1", "$field2"] }
  }
}
```

Detailed Use Cases:

1. Basic Field Selection

javascript

```javascript
db.orders.aggregate([
  { $project: { customer: 1, total: 1 } }
])
```

Explanation: This creates a new document shape containing only the customer and total fields. The _id field is included by default unless explicitly excluded.

Output:

javascript

```javascript
{ _id: 1, customer: "Alice", total: 120 }
{ _id: 2, customer: "Bob", total: 90 }
```

Business Use Case: "Create a simplified report showing only customer names and order totals."

2. Excluding _id Field

javascript

```javascript
db.orders.aggregate([
  { $project: { _id: 0, customer: 1, total: 1 } }
])
```

Explanation: The `_id: 0` explicitly excludes the _id field, creating cleaner output. This is very common in reporting scenarios.

Output:

javascript

```javascript
{ customer: "Alice", total: 120 }
{ customer: "Bob", total: 90 }
```

Business Use Case: "Generate a clean customer report without database-specific IDs."

3. Field Renaming

javascript

```javascript
db.orders.aggregate([
  { $project: { customerName: "$customer", orderValue: "$total", _id: 0 } }
])
```

Explanation: Using `$customer` references the value of the customer field and assigns it to a new field name. The `$` symbol tells MongoDB to use the field's value, not the literal string "customer".

Output:

javascript

```javascript
{ customerName: "Alice", orderValue: 120 }
{ customerName: "Bob", orderValue: 90 }
```

Business Use Case: "Create a report with user-friendly field names for external clients."

4. Computed Fields - Adding Tax

javascript

```javascript
db.orders.aggregate([
 {
  $project: {
   customer: 1,
   subtotal: "$total",
   tax: { $multiply: ["$total", 0.18] },
   totalWithTax: { $multiply: ["$total", 1.18] },
   _id: 0
  }
 }
])
```

Explanation: This creates new fields through mathematical operations. The tax is calculated as 18% of the total, and totalWithTax includes both the original amount and tax.

Output:

javascript

```javascript
{ customer: "Alice", subtotal: 120, tax: 21.6, totalWithTax: 141.6 }
{ customer: "Bob", subtotal: 90, tax: 16.2, totalWithTax: 106.2 }
```

Business Use Case: "Generate invoice data with tax calculations for accounting purposes."

5. Conditional Field Creation

javascript

```javascript
db.orders.aggregate([
 {
  $project: {
   customer: 1,
   total: 1,
   customerTier: {
    $cond: {
     if: { $gte: ["$total", 200] },
     then: "Premium",
     else: "Standard"
    }
   },
   _id: 0
  }
 }
])
```

Explanation: The `$cond` operator works like an if-then-else statement. If the total is greater than or equal to 200, the customer gets "Premium" tier, otherwise "Standard".

Output:

javascript

```javascript
{ customer: "Alice", total: 120, customerTier: "Standard" }
{ customer: "Alice", total: 200, customerTier: "Premium" }
{ customer: "Bob", total: 90, customerTier: "Standard" }
```

Business Use Case: "Automatically categorize customers based on their order values for targeted marketing."

6. Working with Nested Documents

javascript

```javascript
// Sample data with nested structure
db.customers.insertMany([
  {
    _id: 1,
    profile: {
      name: "Alice Johnson",
      email: "alice@example.com",
      address: { city: "New York", country: "USA" }
    },
    preferences: { newsletter: true, offers: false }
  }
])

db.customers.aggregate([
  {
    $project: {
      customerName: "$profile.name",
      email: "$profile.email",
      city: "$profile.address.city",
      wantsNewsletter: "$preferences.newsletter",
      _id: 0
    }
  }
])
```

Explanation: You can access nested fields using dot notation. `$profile.name` accesses the name field inside the profile object.

Business Use Case: "Flatten nested customer data for easier reporting and analysis."

7. Array Element Access

javascript

```javascript
db.orders.aggregate([
 {
  $project: {
   customer: 1,
   firstItem: { $arrayElemAt: ["$items", 0] },
   itemCount: { $size: "$items" },
   _id: 0
  }
 }
])
```

Explanation: `$arrayElemAt` gets a specific element from an array (0 = first element), while `$size` returns the number of elements in the array.

Business Use Case: "Show the first item in each order and the total number of items."

Important Concepts:

Field Reference vs. Literal Values:

- `"$fieldName"` = The VALUE stored in fieldName
- `"fieldName"` = The literal string "fieldName"
- `fieldName: 1` = Include the field named fieldName

Computed Field Performance:

- Simple operations (like $multiply) are fast
- Complex operations (like $regex) can be slower
- Consider creating computed fields early if they're used in multiple stages

Common Patterns:

Data Type Conversion:

javascript

```javascript
{
  $project: {
    customer: 1,
    totalString: { $toString: "$total" },
    totalInt: { $toInt: "$total" },
    isHighValue: { $toBool: { $gte: ["$total", 100] } }
  }
}
```

String Manipulation:

javascript

```javascript
{
  $project: {
    fullName: { $concat: ["$firstName", " ", "$lastName"] },
    emailUpper: { $toUpper: "$email" },
    nameLength: { $strLenCP: "$firstName" }
  }
}
```

Stage 3: $group - Group and Aggregate Data {#group}

Understanding $group

The `$group` stage is like a skilled accountant who organizes financial records into meaningful summaries. It takes a collection of individual documents and groups them based on common characteristics, then performs calculations on each group.

Key Concepts:

Grouping Logic: Think of $group like organizing books in a library:

- **Group by Genre**: All mystery books together, all romance books together
- **Calculate Statistics**: How many books in each genre, average pages, etc.
- **Create Summaries**: Total books, most popular genre, etc.

The _id Field in $group:

- The `_id` field specifies what to group by
- `_id: "$customer"` = Group by customer field
- `_id: null` = Group all documents together
- `_id: { field1: "$field1", field2: "$field2" }` = Group by multiple fields

Common Accumulator Operators Explained:

Operator	Purpose	Example Use Case
`$sum`	Add up numbers	Total sales, count items
`$avg`	Calculate average	Average order value, mean score
`$max`	Find maximum	Highest price, latest date
`$min`	Find minimum	Lowest price, earliest date
`$first`	First value in group	First order date
`$last`	Last value in group	Most recent order
`$push`	Collect values in array	All items bought
`$addToSet`	Collect unique values	Unique categories

Detailed Use Cases:

1. Basic Grouping - Customer Total Spending

javascript

```javascript
db.orders.aggregate([
 {
  $group: {
   _id: "$customer",
   totalSpent: { $sum: "$total" },
   orderCount: { $sum: 1 }
  }
 }
])
```

Explanation:

- `_id: "$customer"` groups documents by customer name
- `$sum: "$total"` adds up all the total values for each customer
- `$sum: 1` counts documents (adds 1 for each document in the group)

Output:

javascript

```javascript
{ _id: "Alice", totalSpent: 320, orderCount: 2 }
{ _id: "Bob", totalSpent: 390, orderCount: 2 }
{ _id: "Charlie", totalSpent: 50, orderCount: 1 }
```

Business Use Case: "Calculate total spending per customer for loyalty program analysis."

2. Statistical Analysis - Order Value Analysis

javascript

```javascript
db.orders.aggregate([
 {
  $group: {
   _id: "$status",
   averageOrderValue: { $avg: "$total" },
   maxOrderValue: { $max: "$total" },
   minOrderValue: { $min: "$total" },
   totalOrders: { $sum: 1 }
  }
 }
])
```

Explanation:

- Groups orders by their status (delivered, pending, cancelled)
- Calculates various statistics for each status group
- Provides comprehensive overview of order patterns

Output:

javascript

```javascript
{ _id: "delivered", averageOrderValue: 156.67, maxOrderValue: 300, minOrderValue: 50, totalOrders: 3 }
{ _id: "pending", averageOrderValue: 90, maxOrderValue: 90, minOrderValue: 90, totalOrders: 1 }
{ _id: "cancelled", averageOrderValue: 200, maxOrderValue: 200, minOrderValue: 200, totalOrders: 1 }
```

Business Use Case: "Analyze order patterns by status to identify delivery issues or cancellation trends."

3. Array Aggregation - Collecting Values

javascript

```javascript
db.orders.aggregate([
 {
  $group: {
   _id: "$customer",
   allOrderTotals: { $push: "$total" },
   uniqueStatuses: { $addToSet: "$status" },
   firstOrderDate: { $first: "$orderDate" },
   lastOrderDate: { $last: "$orderDate" }
  }
 }
])
```

Explanation:

- `$push` creates an array containing ALL values (including duplicates)
- `$addToSet` creates an array containing only UNIQUE values
- `$first` and `$last` require sorted data to be meaningful

Output:

javascript

```javascript
{
 _id: "Alice",
 allOrderTotals: [120, 200],
 uniqueStatuses: ["delivered", "cancelled"],
 firstOrderDate: ISODate("2023-01-15"),
 lastOrderDate: ISODate("2023-02-01")
}
```

Business Use Case: "Track customer order history and status patterns over time."

4. Multi-Field Grouping - Advanced Segmentation

javascript

```javascript
db.orders.aggregate([
 {
  $group: {
   _id: {
    customer: "$customer",
    status: "$status"
   },
   totalAmount: { $sum: "$total" },
   orderCount: { $sum: 1 }
  }
 }
])
```

Explanation:

- Groups by BOTH customer AND status
- Creates separate groups for each customer-status combination
- Useful for detailed customer behavior analysis

Output:

javascript

```javascript
{ _id: { customer: "Alice", status: "delivered" }, totalAmount: 120, orderCount: 1 }
{ _id: { customer: "Alice", status: "cancelled" }, totalAmount: 200, orderCount: 1 }
{ _id: { customer: "Bob", status: "pending" }, totalAmount: 90, orderCount: 1 }
{ _id: { customer: "Bob", status: "delivered" }, totalAmount: 300, orderCount: 1 }
```

Business Use Case: "Analyze customer behavior patterns by order status for targeted follow-up campaigns."

5. Global Aggregation - Overall Statistics

javascript

```javascript
db.orders.aggregate([
 {
  $group: {
   _id: null,
   totalOrders: { $sum: 1 },
   totalRevenue: { $sum: "$total" },
   averageOrderValue: { $avg: "$total" },
   highestOrder: { $max: "$total" },
   lowestOrder: { $min: "$total" }
  }
 }
])
```

Explanation:

- `_id: null` puts ALL documents into a single group
- Calculates business metrics across the entire dataset
- Useful for dashboard KPIs and overall business health

Output:

javascript

```javascript
{
 _id: null,
 totalOrders: 5,
 totalRevenue: 760,
 averageOrderValue: 152,
 highestOrder: 300,
 lowestOrder: 50
}
```

Business Use Case: "Generate executive dashboard showing overall business performance metrics."

6. Conditional Aggregation - Advanced Calculations

javascript

```javascript
db.orders.aggregate([
 {
  $group: {
   _id: "$customer",
   totalSpent: { $sum: "$total" },
   deliveredOrders: {
    $sum: {
     $cond: [{ $eq: ["$status", "delivered"] }, 1, 0]
    }
   },
   deliveredValue: {
    $sum: {
     $cond: [{ $eq: ["$status", "delivered"] }, "$total", 0]
    }
   }
  }
 }
])
```

Explanation:

- `$cond` allows conditional logic within aggregation
- Only counts/sums values that meet specific criteria
- Provides more nuanced business insights

Output:

javascript

```javascript
{ _id: "Alice", totalSpent: 320, deliveredOrders: 1, deliveredValue: 120 }
{ _id: "Bob", totalSpent: 390, deliveredOrders: 1, deliveredValue: 300 }
{ _id: "Charlie", totalSpent: 50, deliveredOrders: 1, deliveredValue: 50 }
```

Business Use Case: "Calculate actual delivered value vs. total order value for revenue recognition."

Time-Based Grouping:

7. Monthly Sales Analysis

javascript

```javascript
db.orders.aggregate([
 {
  $group: {
   _id: {
    year: { $year: "$orderDate" },
    month: { $month: "$orderDate" }
   },
   monthlyRevenue: { $sum: "$total" },
   monthlyOrders: { $sum: 1 },
   avgOrderValue: { $avg: "$total" }
  }
 },
 { $sort: { "_id.year": 1, "_id.month": 1 } }
])
```

Explanation:

- Groups orders by year and month
- Calculates key metrics for each time period
- Sorted by chronological order for trend analysis

Business Use Case: "Track monthly sales trends and seasonal patterns."

Performance Considerations:

1. **Memory Usage**: $group operations are memory-intensive
2. **Index Usage**: Group operations can't use indexes efficiently
3. **Order Matters**: Use $match before $group to reduce documents
4. **Field Selection**: Use $project after $group to shape output

Common Patterns:

Top-N Analysis:

javascript

```javascript
db.orders.aggregate([
 { $group: { _id: "$customer", total: { $sum: "$total" } } },
 { $sort: { total: -1 } },
 { $limit: 3 }
])
```

Percentage Calculations:

javascript

```javascript
db.orders.aggregate([
 {
  $group: {
   _id: "$status",
   count: { $sum: 1 }
  }
 },
 {
  $group: {
   _id: null,
   statusCounts: { $push: { status: "$_id", count: "$count" } },
   totalOrders: { $sum: "$count" }
  }
 }
])
```

Stage 4: $sort - Sort Documents {#sort}

Understanding $sort

The `$sort` stage is like a librarian organizing books on shelves. It arranges documents in a specific order based on one or more fields, making data easier to analyze and present.

How Sorting Works:

Sort Direction:

- `1` or `ascending` = A to Z, 0 to 9, earliest to latest
- `-1` or `descending` = Z to A, 9 to 0, latest to earliest

Sort Priority: When sorting by multiple fields, MongoDB sorts by the first field, then breaks ties using the second field, and so on.

Memory and Performance Considerations:

Memory Limits:

- $sort operations are limited to 100MB of memory by default
- For larger datasets, ensure you have appropriate indexes
- Use $match before $sort to reduce the number of documents

Index Usage:

- $sort can use indexes when it's early in the pipeline

- Compound indexes can support multi-field sorts
- Sort order must match index order for optimal performance

Detailed Use Cases:

1. Basic Single-Field Sort

javascript

```
db.orders.aggregate([
  { $sort: { total: 1 } } // Ascending order
])
```

Explanation: Sorts all orders from lowest to highest total value. This is useful for finding budget-friendly options or identifying small orders.

Output Order:

javascript

```
{ _id: 4, customer: "Charlie", total: 50, ... }
{ _id: 2, customer: "Bob", total: 90, ... }
{ _id: 1, customer: "Alice", total: 120, ... }
{ _id: 3, customer: "Alice", total: 200, ... }
{ _id: 5, customer: "Bob", total: 300, ... }
```

Business Use Case: "Display products from cheapest to most expensive for price-sensitive customers."

2. Descending Sort - Highest Value First

javascript

```
db.orders.aggregate([
  { $sort: { total: -1 } } // Descending order
])
```

Explanation: Sorts orders from highest to lowest value. Perfect for identifying VIP customers or premium orders that need special attention.

Business Use Case: "Show highest-value orders first for priority processing and premium customer service."

3. Multi-Field Sort - Complex Ordering

javascript

```javascript
db.orders.aggregate([
  { $sort: { customer: 1, total: -1 } }
])
```

Explanation:

- First sorts by customer name alphabetically (A to Z)
- Then within each customer, sorts by total amount (highest first)
- This creates a customer-centric view with their biggest orders first

Output Logic:

javascript

```javascript
{ customer: "Alice", total: 200, ... } // Alice's highest order
{ customer: "Alice", total: 120, ... } // Alice's lower order
{ customer: "Bob", total: 300, ... }   // Bob's highest order
{ customer: "Bob", total: 90, ... }    // Bob's lower order
{ customer: "Charlie", total: 50, ... } // Charlie's only order
```

Business Use Case: "Create customer statements showing their orders from highest to lowest value."

4. Date-Based Sorting

javascript

```javascript
db.orders.aggregate([
  { $sort: { orderDate: -1 } } // Most recent first
])
```

Explanation: Sorts orders by date with the most recent orders appearing first. Essential for time-sensitive operations and trend analysis.

Business Use Case: "Show recent orders first for customer service and fulfillment teams."

5. Sort with Computed Fields

javascript

```javascript
db.orders.aggregate([
 {
  $addFields: {
   totalWithTax: { $multiply: ["$total", 1.18] }
  }
 },
 { $sort: { totalWithTax: -1 } }
])
```

Explanation:

- First calculates tax-inclusive total
- Then sorts by the computed field
- Useful when you need to sort by derived values

Business Use Case: "Sort orders by tax-inclusive amount for accurate financial reporting."

6. Top-N Pattern - Sort + Limit

javascript

```javascript
db.orders.aggregate([
 { $sort: { total: -1 } },
 { $limit: 3 }
])
```

Explanation:

- Sorts all orders by total value (highest first)
- Limits output to top 3 orders
- This is a very common pattern for "top performers" analysis

Business Use Case: "Find the top 3 highest-value orders for management review."

7. Sorting After Grouping

javascript

```javascript
db.orders.aggregate([
 {
  $group: {
   _id: "$customer",
   totalSpent: { $sum: "$total" },
   orderCount: { $sum: 1 }
  }
 },
 { $sort: { totalSpent: -1 } }
])
```

Explanation:

- Groups orders by customer and calculates totals
- Sorts customers by their total spending
- Identifies your most valuable customers

Business Use Case: "Rank customers by total spending to identify VIP customers for special treatment."

Advanced Sorting Techniques:

8. Multi-Level Business Logic Sort

javascript

```javascript
db.orders.aggregate([
 {
  $addFields: {
   priority: {
    $switch: {
     branches: [
      { case: { $eq: ["$status", "urgent"] }, then: 1 },
      { case: { $eq: ["$status", "pending"] }, then: 2 },
      { case: { $eq: ["$status", "delivered"] }, then: 3 }
     ],
     default: 4
    }
   }
  }
 },
 { $sort: { priority: 1, orderDate: -1 } }
])
```

Explanation:

- Creates a priority field based on order status
- Sorts by priority first, then

MongoDB Aggregation Framework - Complete Guide (Continued)

Stage 4: $sort - Sort Documents {#sort} (Continued)

Advanced Sorting Techniques:

8. Multi-Level Business Logic Sort

javascript

```javascript
db.orders.aggregate([
 {
  $addFields: {
   priority: {
    $switch: {
     branches: [
      { case: { $eq: ["$status", "urgent"] }, then: 1 },
      { case: { $eq: ["$status", "pending"] }, then: 2 },
      { case: { $eq: ["$status", "delivered"] }, then: 3 }
     ],
     default: 4
    }
   }
  }
 },
 { $sort: { priority: 1, orderDate: -1 } }
])
```

Explanation:

- Creates a priority field based on order status
- Sorts by priority first, then by order date (newest first)
- Ensures urgent orders appear first regardless of date

Business Use Case: "Process orders by business priority - urgent orders first, then pending, with newest orders prioritized within each category."

9. Null Value Handling in Sorts

javascript

```javascript
db.orders.aggregate([
  {
    $sort: {
      deliveryDate: 1, // null values appear first
      total: -1
    }
  }
])
```

Explanation: MongoDB sorts null values before all other values in ascending order. This behavior is important when dealing with optional fields.

Business Use Case: "Sort orders by delivery date, showing unscheduled orders first for assignment."

Performance Optimization Tips:

Index-Optimized Sorting:

javascript

```javascript
// Create compound index for optimal sort performance
db.orders.createIndex({ status: 1, total: -1 })

// This query will use the index efficiently
db.orders.aggregate([
  { $match: { status: "delivered" } },
  { $sort: { status: 1, total: -1 } }
])
```

Memory-Efficient Sorting:

javascript

```javascript
// Bad: Sort after expensive operations
db.orders.aggregate([
  { $lookup: { /* expensive join */ } },
  { $group: { /* expensive grouping */ } },
  { $sort: { total: -1 } } // Uses lots of memory
])

// Good: Sort early, use indexes
db.orders.aggregate([
  { $sort: { total: -1 } }, // Uses index, less memory
  { $match: { status: "delivered" } },
  { $lookup: { /* expensive join */ } }
])
```

Stage 5: $limit - Restrict Number of Documents {#limit}

Understanding $limit

The `$limit` stage is like a bouncer at a popular restaurant who only lets in a specific number of customers. It restricts the number of documents that pass through to the next stage, which is essential for:

- **Performance optimization** (processing fewer documents)
- **Pagination** (showing results in pages)
- **Top-N analysis** (finding best/worst performers)
- **Sampling** (working with representative data subsets)

How $limit Works:

`$limit` takes the first N documents that reach it and discards the rest. The key insight is that the order of documents matters greatly - always use `$sort` before `$limit` to get meaningful results.

Detailed Use Cases:

1. Basic Limiting - Top Results

javascript

```javascript
db.orders.aggregate([
  { $sort: { total: -1 } },
  { $limit: 5 }
])
```

Explanation: Gets the 5 highest-value orders. Without the sort, you'd get random 5 orders, which is usually not useful.

Business Use Case: "Show the top 5 highest-value orders for executive review."

2. Sampling for Analysis

javascript

```javascript
db.orders.aggregate([
  { $sample: { size: 100 } }, // Random sample
  { $limit: 10 }            // Further limit if needed
])
```

Explanation: Takes a random sample of 100 orders, then limits to 10. Useful for data exploration and testing.

Business Use Case: "Get a small sample of orders for testing new analytics algorithms."

3. Performance Optimization

javascript

```javascript
db.orders.aggregate([
  { $match: { status: "delivered" } },
  { $limit: 1000 }, // Limit early to reduce processing
  { $group: { _id: "$customer", total: { $sum: "$total" } } }
])
```

Explanation: Limits documents early in the pipeline to reduce memory usage and processing time in subsequent stages.

Business Use Case: "Process only the first 1000 delivered orders for quick performance testing."

4. Pagination Foundation

javascript

```javascript
// First page (items 1-10)
db.orders.aggregate([
  { $sort: { orderDate: -1 } },
  { $limit: 10 }
])

// Second page (items 11-20) - Need $skip for this
db.orders.aggregate([
  { $sort: { orderDate: -1 } },
  { $skip: 10 },
  { $limit: 10 }
])
```

Explanation: Creates paginated results. The combination of $sort, $skip, and $limit enables pagination.

Business Use Case: "Display orders in pages of 10 for user interface pagination."

Stage 6: $skip - Skip Documents (Pagination) {#skip}

Understanding $skip

The `$skip` stage is like fast-forwarding through a video to skip the first part. It discards the first N documents and passes the remaining documents to the next stage. This is primarily used for:

- **Pagination** (showing page 2, 3, etc.)
- **Offset queries** (starting from a specific position)
- **Data sampling** (skipping headers or irrelevant data)

How $skip Works:

`$skip` counts documents as they arrive and discards the first N documents. The remaining documents continue through the pipeline.

Detailed Use Cases:

1. Basic Pagination

javascript

```javascript
const pageSize = 10;
const pageNumber = 2; // 0-based: 0=first page, 1=second page

db.orders.aggregate([
  { $sort: { orderDate: -1 } },
  { $skip: pageNumber * pageSize }, // Skip first 20 documents
  { $limit: pageSize }              // Take next 10 documents
])
```

Explanation:

- Sorts orders by date (newest first)
- Skips the first 20 documents (pages 0 and 1)
- Takes the next 10 documents (page 2)

Business Use Case: "Implement pagination for order history display in web application."

2. Advanced Pagination with Metadata

javascript

```javascript
function getOrdersPage(pageNum, pageSize) {
 return db.orders.aggregate([
  {
   $facet: {
    // Get the actual page data
    data: [
     { $sort: { orderDate: -1 } },
     { $skip: pageNum * pageSize },
     { $limit: pageSize }
    ],
    // Get total count for pagination info
    totalCount: [
     { $count: "count" }
    ]
   }
  },
  {
   $project: {
    data: 1,
    totalCount: { $arrayElemAt: ["$totalCount.count", 0] },
    currentPage: { $literal: pageNum },
    pageSize: { $literal: pageSize }
   }
  }
 ]);
}
```

Explanation: Uses $facet to get both the page data and total count in one query, enabling proper pagination controls.

Business Use Case: "Provide complete pagination information including total pages and current page number."

3. Skipping Headers or Metadata

javascript

```javascript
db.csvImport.aggregate([
 { $skip: 1 }, // Skip header row
 { $project: {
  name: "$field1",
  value: "$field2"
 }}
])
```

Explanation: When importing CSV data, skip the first document if it contains headers rather than data.

Business Use Case: "Process imported CSV data while skipping the header row."

4. Offset-Based Queries

javascript

```javascript
db.orders.aggregate([
  { $sort: { _id: 1 } },
  { $skip: 50 },
  { $limit: 25 }
])
```

Explanation: Gets documents 51-75 in the sorted order. Useful for batch processing large datasets.

Business Use Case: "Process orders in batches of 25 for bulk operations."

Performance Considerations:

Efficient Pagination:

javascript

```javascript
// Inefficient for large offsets
db.orders.aggregate([
  { $sort: { orderDate: -1 } },
  { $skip: 10000 }, // Very slow for large skips
  { $limit: 10 }
])

// Better approach for large datasets
db.orders.aggregate([
  { $match: { orderDate: { $lt: lastSeenDate } } },
  { $sort: { orderDate: -1 } },
  { $limit: 10 }
])
```

Explanation: Large $skip operations are slow because MongoDB must process and discard many documents. Cursor-based pagination is more efficient.

Stage 7: $lookup - Perform JOINs Between Collections {#lookup}

Understanding $lookup

The `$lookup` stage is like a detective who connects clues from different sources. It performs left outer joins between collections, allowing you to combine related data from multiple collections into a single result set.

How $lookup Works:

Think of $lookup as connecting two tables in a relational database:

- **Local Collection**: The collection you're starting with
- **Foreign Collection**: The collection you want to join data from
- **Local Field**: The field in your current collection
- **Foreign Field**: The matching field in the other collection
- **As Field**: The name of the new array field containing matched documents

Basic Syntax:

javascript

```javascript
{
 $lookup: {
  from: "foreignCollection",
  localField: "localFieldName",
  foreignField: "foreignFieldName",
  as: "resultArrayName"
 }
}
```

Sample Data Setup:

javascript

```javascript
// Orders collection
db.orders.insertMany([
  { _id: 1, customerId: 101, total: 120, status: "delivered" },
  { _id: 2, customerId: 102, total: 90, status: "pending" },
  { _id: 3, customerId: 101, total: 200, status: "cancelled" }
]);

// Customers collection
db.customers.insertMany([
  { _id: 101, name: "Alice Johnson", email: "alice@example.com", city: "New York" },
  { _id: 102, name: "Bob Smith", email: "bob@example.com", city: "Los Angeles" },
  { _id: 103, name: "Charlie Brown", email: "charlie@example.com", city: "Chicago" }
]);

// Products collection
db.products.insertMany([
  { _id: 1, name: "Laptop", price: 999, category: "Electronics" },
  { _id: 2, name: "Mouse", price: 25, category: "Electronics" },
  { _id: 3, name: "Desk", price: 200, category: "Furniture" }
]);
```

Detailed Use Cases:

1. Basic Join - Orders with Customer Info

javascript

```javascript
db.orders.aggregate([
  {
    $lookup: {
      from: "customers",
      localField: "customerId",
      foreignField: "_id",
      as: "customerInfo"
    }
  }
])
```

Explanation:

- Joins orders with customers based on customerId
- Each order gets a customerInfo array containing matching customer documents
- If no match found, customerInfo will be an empty array

Output:

javascript

```javascript
{
 _id: 1,
 customerId: 101,
 total: 120,
 status: "delivered",
 customerInfo: [
  { _id: 101, name: "Alice Johnson", email: "alice@example.com", city: "New York" }
 ]
}
```

Business Use Case: "Display order details with customer information for customer service representatives."

2. Flattening Joined Data

javascript

```javascript
db.orders.aggregate([
 {
  $lookup: {
   from: "customers",
   localField: "customerId",
   foreignField: "_id",
   as: "customerInfo"
  }
 },
 {
  $unwind: "$customerInfo"  // Convert array to object
 },
 {
  $project: {
   orderTotal: "$total",
   customerName: "$customerInfo.name",
   customerEmail: "$customerInfo.email",
   status: 1
  }
 }
])
```

Explanation:

- $unwind converts the customerInfo array into individual documents

- $project reshapes the output to include specific fields
- Results in flattened, clean output

Output:

javascript

```javascript
{
  _id: 1,
  orderTotal: 120,
  customerName: "Alice Johnson",
  customerEmail: "alice@example.com",
  status: "delivered"
}
```

Business Use Case: "Create a flat report showing order and customer details for export to Excel."

3. Multiple Lookups - Complex Joins

javascript

```javascript
db.orders.aggregate([
 // Join with customers
 {
  $lookup: {
   from: "customers",
   localField: "customerId",
   foreignField: "_id",
   as: "customer"
  }
 },
 // Join with order items (assuming we have this data)
 {
  $lookup: {
   from: "orderItems",
   localField: "_id",
   foreignField: "orderId",
   as: "items"
  }
 },
 {
  $unwind: "$customer"
 },
 {
  $project: {
   orderTotal: "$total",
   customerName: "$customer.name",
   customerCity: "$customer.city",
   itemCount: { $size: "$items" },
   status: 1
  }
 }
])
```

Explanation: Performs multiple joins to get comprehensive order information including customer details and item counts.

Business Use Case: "Create comprehensive order reports with customer and item information."

4. Advanced Lookup with Pipeline

javascript

```javascript
db.orders.aggregate([
 {
  $lookup: {
   from: "customers",
   let: { customerId: "$customerId" },
   pipeline: [
    { $match: { $expr: { $eq: ["$_id", "$$customerId"] } } },
    { $project: { name: 1, email: 1, city: 1 } } // Only include specific fields
   ],
   as: "customerDetails"
  }
 }
])
```

Explanation:

- Uses pipeline syntax for more complex join conditions
- `let` defines variables from the local document
- `$$customerId` references the variable in the pipeline
- Allows filtering and projecting within the join

Business Use Case: "Join with customers but only include necessary fields to reduce data transfer."

5. Conditional Lookup

javascript

```javascript
db.orders.aggregate([
 {
  $lookup: {
   from: "customers",
   let: { customerId: "$customerId", orderTotal: "$total" },
   pipeline: [
    { $match: { $expr: { $eq: ["$_id", "$$customerId"] } } },
    {
     $addFields: {
      isVip: { $gte: ["$$orderTotal", 200] } // Mark as VIP if order > 200
     }
    }
   ],
   as: "customerData"
  }
 }
])
```

Explanation: Adds computed fields during the lookup operation, creating context-aware joins.

Business Use Case: "Identify VIP customers based on their current order value during the join operation."

6. Aggregating Joined Data

javascript

```javascript
db.customers.aggregate([
 {
  $lookup: {
   from: "orders",
   localField: "_id",
   foreignField: "customerId",
   as: "orders"
  }
 },
 {
  $addFields: {
   totalOrders: { $size: "$orders" },
   totalSpent: { $sum: "$orders.total" },
   avgOrderValue: { $avg: "$orders.total" },
   lastOrderDate: { $max: "$orders.orderDate" }
  }
 },
 {
  $project: {
   name: 1,
   email: 1,
   city: 1,
   totalOrders: 1,
   totalSpent: 1,
   avgOrderValue: 1,
   lastOrderDate: 1
  }
 }
])
```

Explanation: Joins customer with their orders and calculates aggregate statistics for each customer.

Business Use Case: "Create customer profiles with their order history and spending statistics."

Performance Considerations:

Index Requirements:

```javascript
// Create indexes on join fields for better performance
db.customers.createIndex({ "_id": 1 }) // Usually exists by default
db.orders.createIndex({ "customerId": 1 })
```

Memory Management:

```javascript
// Good: Filter before lookup
db.orders.aggregate([
  { $match: { status: "delivered" } }, // Reduce documents before join
  { $lookup: { /* join operation */ } }
])

// Bad: Lookup everything then filter
db.orders.aggregate([
  { $lookup: { /* join operation */ } },
  { $match: { status: "delivered" } } // Filter after expensive join
])
```

Lookup Limitations:

- $lookup creates arrays, even for single matches
- Cannot lookup from sharded collections (in older MongoDB versions)
- Memory-intensive for large result sets
- Consider denormalization for frequently accessed data

Stage 8: $unwind - Deconstruct Arrays {#unwind}

Understanding $unwind

The `$unwind` stage is like unpacking a box of items and laying each item out separately. It takes documents with array fields and creates separate documents for each array element, making it easier to work with array data in subsequent pipeline stages.

How $unwind Works:

Original Document:

```javascript
{ name: "Alice", hobbies: ["reading", "swimming", "cooking"] }
```

After $unwind on hobbies:

javascript

```javascript
{ name: "Alice", hobbies: "reading" }
{ name: "Alice", hobbies: "swimming" }
{ name: "Alice", hobbies: "cooking" }
```

$unwind Options:

javascript

```javascript
{
  $unwind: {
    path: "$arrayField",
    includeArrayIndex: "indexFieldName",  // Optional: include array index
    preserveNullAndEmptyArrays: true     // Optional: keep docs with null/empty arrays
  }
}
```

Sample Data Setup:

javascript

```javascript
db.students.insertMany([
  {
    _id: 1,
    name: "Alice",
    courses: ["Math", "Science", "History"],
    grades: [85, 92, 78]
  },
  {
    _id: 2,
    name: "Bob",
    courses: ["Math", "Art"],
    grades: [95, 88]
  },
  {
    _id: 3,
    name: "Charlie",
    courses: [] // Empty array
  },
  {
    _id: 4,
    name: "Diana",
    courses: null // Null value
  }
]);
```

Detailed Use Cases:

1. Basic Array Unwinding

javascript

```javascript
db.students.aggregate([
  { $unwind: "$courses" }
])
```

Explanation: Creates a separate document for each course, enabling course-level analysis.

Output:

javascript

```javascript
{ _id: 1, name: "Alice", courses: "Math", grades: [85, 92, 78] }
{ _id: 1, name: "Alice", courses: "Science", grades: [85, 92, 78] }
{ _id: 1, name: "Alice", courses: "History", grades: [85, 92, 78] }
{ _id: 2, name: "Bob", courses: "Math", grades: [95, 88] }
{ _id: 2, name: "Bob", courses: "Art", grades: [95, 88] }
// Note: Charlie and Diana are excluded (empty/null arrays)
```

Business Use Case: "Analyze course enrollment by creating one record per student-course combination."

2. Preserving Documents with Empty Arrays

javascript

```javascript
db.students.aggregate([
 {
  $unwind: {
   path: "$courses",
   preserveNullAndEmptyArrays: true
  }
 }
])
```

Explanation: Keeps documents even when the array is empty or null, preventing data loss.

Output:

javascript

```javascript
{ _id: 1, name: "Alice", courses: "Math", grades: [85, 92, 78] }
{ _id: 1, name: "Alice", courses: "Science", grades: [85, 92, 78] }
{ _id: 1, name: "Alice", courses: "History", grades: [85, 92, 78] }
{ _id: 2, name: "Bob", courses: "Math", grades: [95, 88] }
{ _id: 2, name: "Bob", courses: "Art", grades: [95, 88] }
{ _id: 3, name: "Charlie", courses: null } // Preserved
{ _id: 4, name: "Diana", courses: null }   // Preserved
```

Business Use Case: "Ensure all students appear in the analysis, even those not enrolled in any courses."

3. Including Array Index

javascript

```javascript
db.students.aggregate([
 {
  $unwind: {
   path: "$courses",
   includeArrayIndex: "courseIndex"
  }
 }
])
```

Explanation: Adds a field showing the original position of each element in the array.

Output:

javascript

```javascript
{ _id: 1, name: "Alice", courses: "Math", grades: [85, 92, 78], courseIndex: 0 }
{ _id: 1, name: "Alice", courses: "Science", grades: [85, 92, 78], courseIndex: 1 }
{ _id: 1, name: "Alice", courses: "History", grades: [85, 92, 78], courseIndex: 2 }
```

Business Use Case: "Track the order of course enrollment for sequence analysis."

4. Multiple Unwinds for Parallel Arrays

javascript

```javascript
db.students.aggregate([
 { $unwind: "$courses" },
 { $unwind: "$grades" }
])
```

Explanation: Unwinds both arrays, creating a cartesian product. This works when arrays have related data at the same positions.

Warning: This creates all possible combinations, which might not be what you want!

Better Approach for Parallel Arrays:

javascript

```javascript
db.students.aggregate([
 {
  $addFields: {
   courseGradePairs: {
    $map: {
     input: { $range: [0, { $size: "$courses" }] },
     as: "index",
     in: {
      course: { $arrayElemAt: ["$courses", "$$index"] },
      grade: { $arrayElemAt: ["$grades", "$$index"] }
     }
    }
   }
  }
 },
 { $unwind: "$courseGradePairs" },
 {
  $project: {
   name: 1,
   course: "$courseGradePairs.course",
   grade: "$courseGradePairs.grade"
  }
 }
])
```

Business Use Case: "Match each course with its corresponding grade accurately."

5. Unwind with Grouping - Course Popularity

javascript

```javascript
db.students.aggregate([
 { $unwind: "$courses" },
 {
  $group: {
   _id: "$courses",
   enrollmentCount: { $sum: 1 },
   students: { $push: "$name" }
  }
 },
 { $sort: { enrollmentCount: -1 } }
])
```

Explanation:

- Unwinds courses to create student-course pairs
- Groups by course to count enrollments
- Sorts by popularity

Output:

javascript

```javascript
{ _id: "Math", enrollmentCount: 2, students: ["Alice", "Bob"] }
{ _id: "Science", enrollmentCount: 1, students: ["Alice"] }
{ _id: "History", enrollmentCount: 1, students: ["Alice"] }
{ _id: "Art", enrollmentCount: 1, students: ["Bob"] }
```

Business Use Case: "Identify most popular courses and which students are enrolled in each."

6. Unwind with Lookup - Course Details

javascript

```javascript
db.students.aggregate([
  { $unwind: "$courses" },
  {
    $lookup: {
      from: "courseDetails",
      localField: "courses",
      foreignField: "courseName",
      as: "courseInfo"
    }
  },
  { $unwind: "$courseInfo" },
  {
    $project: {
      studentName: "$name",
      courseName: "$courses",
      courseCredits: "$courseInfo.credits",
      courseDepartment: "$courseInfo.department"
    }
  }
])
```

Explanation: Unwinds courses, then joins with course details to get comprehensive information.

Business Use Case: "Create detailed enrollment report with course information for academic planning."

Advanced Patterns:

7. Conditional Unwind

javascript

```javascript
db.students.aggregate([
 {
  $addFields: {
   coursesToUnwind: {
    $cond: {
     if: { $isArray: "$courses" },
     then: "$courses",
     else: []
    }
   }
  }
 },
 { $unwind: "$coursesToUnwind" },
 {
  $project: {
   name: 1,
   course: "$coursesToUnwind"
  }
 }
])
```

Explanation: Only unwinds if the field is actually an array, providing more control over the process.

8. Nested Array Unwinding

javascript

```javascript
// Sample data with nested arrays
db.departments.insertMany([
  {
    name: "Computer Science",
    courses: [
      { name: "CS101", students: ["Alice", "Bob"] },
      { name: "CS102", students: ["Charlie", "Diana"] }
    ]
  }
]);

db.departments.aggregate([
  { $unwind: "$courses" },
  { $unwind: "$courses.students" },
  {
    $group: {
      _id: "$courses.students",
      coursesEnrolled: { $addToSet: "$courses.name" },
      department: { $first: "$name" }
    }
  }
])
```

Explanation: Unwinds nested arrays to get student-level data across departments and courses.

Performance Considerations:

Memory Usage:

- $unwind can significantly increase document count
- Use $match before $unwind to reduce input size
- Consider memory limits for large arrays

Index Usage:

- Indexes on unwound fields can improve subsequent operations
- Consider compound indexes for common query patterns

Best Practices:

javascript

```javascript
// Good: Filter before unwind
db.students.aggregate([
  { $match: { active: true } }, // Reduce documents first
  { $unwind: "$courses" }
])

// Bad: Unwind everything then filter
db.students.aggregate([
  { $unwind: "$courses" },
  { $match: { active: true } } // Filter after expensive unwind
])
```

Stage 9: $addFields/$set - Add or Update Fields {#addfields}

Understanding $addFields and $set

The `$addFields` stage (also known as `$set` in newer MongoDB versions) is like a skilled craftsman who adds new components to existing products without destroying the original. It adds new fields to documents or updates existing fields while preserving all other fields.

Key Differences:

- **$addFields**: Adds new fields or updates existing ones
- **$project**: Shapes the entire document (can remove fields)
- **$set**: Alias for $addFields (MongoDB 4.2+)

How $addFields Works:

The stage examines each document and adds or modifies fields based on your specifications. Unlike $project, it doesn't remove existing fields unless you explicitly overwrite them.

Sample Data Setup:

javascript

```javascript
db.employees.insertMany([
 {
  _id: 1,
  name: "Alice Johnson",
  salary: 75000,
  department: "Engineering",
  joinDate: new Date("2020-01-15"),
  skills: ["JavaScript", "Python", "MongoDB"]
 },
 {
  _id: 2,
  name: "Bob Smith",
  salary: 60000,
  department: "Marketing",
  joinDate: new Date("2019-06-10"),
  skills: ["SEO", "Analytics", "Content"]
 },
 {
  _id: 3,
  name: "Charlie Brown",
  salary: 90000,
  department: "Engineering",
  joinDate: new Date("2018-03-20"),
  skills: ["Java", "AWS", "Docker"]
 }
]);
```

Detailed Use Cases:

1. Basic Field Addition

javascript

```javascript
db.employees.aggregate([
 {
  $addFields: {
   annualBonus: 5000,
   company: "TechCorp"
  }
 }
])
```

Explanation: Adds fixed values to all documents. The new fields appear alongside all existing fields.

Output:

javascript

```javascript
{
 _id: 1,
 name: "Alice Johnson",
 salary: 75000,
 department: "Engineering",
 joinDate: ISODate("2020-01-15"),
 skills: ["JavaScript", "Python", "MongoDB"],
 annualBonus: 5000,
 company: "TechCorp"
}
```

Business Use Case: "Add company-wide bonus and company name to all employee records."

2. Computed Fields Based on Existing Data

javascript

```javascript
db.employees.aggregate([
 {
  $addFields: {
   monthlySalary: { $divide: ["$salary", 12] },
   totalCompensation: { $add: ["$salary", 5000] }, // salary + bonus
   experienceYears: {
    $divide: [
     { $subtract: [new Date(), "$joinDate"] },
     365.25 * 24 * 60 * 60 * 1000 // Convert milliseconds to years
    ]
   }
  }
 }
])
```

Explanation:

- Calculates monthly salary from annual salary
- Adds bonus to salary for total compensation
- Calculates years of experience from join date

Business Use Case: "Calculate derived compensation metrics for HR analytics and employee reviews."

3. Conditional Field Addition

javascript

```javascript
db.employees.aggregate([
 {
  $addFields: {
   salaryGrade: {
    $switch: {
     branches: [
      { case: { $gte: ["$salary", 80000] }, then: "Senior" },
      { case: { $gte: ["$salary", 60000] }, then: "Mid" },
      { case: { $gte: ["$salary", 40000] }, then: "Junior" }
     ],
     default: "Trainee"
    }
   },
   isHighPerformer: {
    $and: [
     { $gte: ["$salary", 70000] },
     { $eq: ["$department", "Engineering"] }
    ]
   }
  }
 }
])
```

Explanation:

- Creates salary grades based on salary ranges
- Identifies high performers based on multiple criteria
- Uses complex conditional logic

Business Use Case: "Categorize employees by performance level and salary grade for promotion planning."

4. Array Field Manipulation

javascript

```javascript
db.employees.aggregate([
 {
```

MongoDB Aggregation Framework - Complete Guide (Continued)

Stage 9: $addFields/$set - Add or Update Fields {#addfields} (Continued)

4. Array Field Manipulation

javascript

```javascript
db.employees.aggregate([
 {
  $addFields: {
   skillCount: { $size: "$skills" },
   hasJavaScript: { $in: ["JavaScript", "$skills"] },
   primarySkill: { $arrayElemAt: ["$skills", 0] },
   skillsUpperCase: {
    $map: {
     input: "$skills",
     as: "skill",
     in: { $toUpper: "$$skill" }
    }
   }
  }
 }
])
```

Explanation:

- Counts skills in the array
- Checks if specific skill exists
- Gets the first skill as primary
- Converts all skills to uppercase

Business Use Case: "Analyze employee skill sets and standardize skill naming conventions."

5. String Manipulation

javascript

```javascript
db.employees.aggregate([
 {
  $addFields: {
   firstName: { $arrayElemAt: [{ $split: ["$name", " "] }, 0] },
   lastName: { $arrayElemAt: [{ $split: ["$name", " "] }, 1] },
   initials: {
    $concat: [
     { $substr: [{ $arrayElemAt: [{ $split: ["$name", " "] }, 0] }, 0, 1] },
     { $substr: [{ $arrayElemAt: [{ $split: ["$name", " "] }, 1] }, 0, 1] }
    ]
   },
   emailAddress: {
    $concat: [
     { $toLower: { $arrayElemAt: [{ $split: ["$name", " "] }, 0] } },
     ".",
     { $toLower: { $arrayElemAt: [{ $split: ["$name", " "] }, 1] } },
     "@techcorp.com"
    ]
   }
  }
 }
])
```

Explanation:

- Extracts first and last names from full name
- Creates initials from first letters
- Generates email addresses automatically

Business Use Case: "Generate email addresses and user initials for new employee onboarding system."

6. Date Field Manipulation

javascript

```javascript
db.employees.aggregate([
 {
  $addFields: {
   joinYear: { $year: "$joinDate" },
   joinMonth: { $month: "$joinDate" },
   joinQuarter: {
    $switch: {
     branches: [
      { case: { $lte: [{ $month: "$joinDate" }, 3] }, then: "Q1" },
      { case: { $lte: [{ $month: "$joinDate" }, 6] }, then: "Q2" },
      { case: { $lte: [{ $month: "$joinDate" }, 9] }, then: "Q3" }
     ],
     default: "Q4"
    }
   },
   daysEmployed: {
    $divide: [
     { $subtract: [new Date(), "$joinDate"] },
     1000 * 60 * 60 * 24 // Convert milliseconds to days
    ]
   },
   anniversaryThisYear: {
    $dateFromParts: {
     year: { $year: new Date() },
     month: { $month: "$joinDate" },
     day: { $dayOfMonth: "$joinDate" }
    }
   }
  }
 }
])
```

Explanation:

- Extracts year, month, and quarter from join date
- Calculates days employed
- Determines anniversary date for current year

Business Use Case: "Track employee tenure and plan anniversary celebrations."

7. Nested Field Creation

javascript

```javascript
db.employees.aggregate([
 {
  $addFields: {
   profile: {
    personal: {
     name: "$name",
     initials: {
      $concat: [
       { $substr: [{ $arrayElemAt: [{ $split: ["$name", " "] }, 0] }, 0, 1] },
       { $substr: [{ $arrayElemAt: [{ $split: ["$name", " "] }, 1] }, 0, 1] }
      ]
     }
    },
    professional: {
     department: "$department",
     level: {
      $cond: {
       if: { $gte: ["$salary", 80000] },
       then: "Senior",
       else: "Junior"
      }
     },
     skillCount: { $size: "$skills" }
    },
    financial: {
     annualSalary: "$salary",
     monthlySalary: { $divide: ["$salary", 12] },
     salaryGrade: {
      $switch: {
       branches: [
        { case: { $gte: ["$salary", 80000] }, then: "A" },
        { case: { $gte: ["$salary", 60000] }, then: "B" }
       ],
       default: "C"
      }
     }
    }
   }
  }
 }
])
```

Explanation: Creates a nested structure organizing employee data into logical categories.

Business Use Case: "Restructure employee data into organized profiles for API responses."

8. Updating Existing Fields

javascript

```javascript
db.employees.aggregate([
 {
  $addFields: {
   // Update existing field with computed value
   salary: { $multiply: ["$salary", 1.1] }, // 10% raise

   // Add to existing array
   skills: { $concatArrays: ["$skills", ["Leadership"]] },

   // Update with conditional logic
   department: {
    $cond: {
     if: { $eq: ["$department", "Engineering"] },
     then: "Software Engineering",
     else: "$department"
    }
   }
  }
 }
])
```

Explanation:

- Increases salary by 10%
- Adds "Leadership" to skills array
- Updates department name conditionally

Business Use Case: "Apply annual salary increases and skill updates across employee records."

9. Complex Business Logic

javascript

```
db.employees.aggregate([
 {
   $addFields: {
     performanceMetrics: {
       salaryPercentile: {
         // This would typically use $percentile in a separate stage
         $switch: {
           branches: [
             { case: { $gte: ["$salary", 85000] }, then: "Top 10%" },
             { case: { $gte: ["$salary", 70000] }, then: "Top 25%" },
             { case: { $gte: ["$salary", 60000] }, then: "Top 50%" }
           ],
           default: "Bottom 50%"
         }
       },
       promotionEligible: {
         $and: [
           { $gte: [{ $size: "$skills" }, 3] }, // Has 3+ skills
           { $gte: [ // 2+ years experience
             { $divide: [
               { $subtract: [new Date(), "$joinDate"] },
               365.25 * 24 * 60 * 60 * 1000
             ]},
             2
           ]},
           { $lt: ["$salary", 80000] } // Not already highly paid
         ]
       },
       riskScore: {
         $add: [
           { $cond: [{ $lt: ["$salary", 50000] }, 30, 0] }, // Low salary risk
           { $cond: [{ $lt: [{ $size: "$skills" }, 2] }, 20, 0] }, // Low skills risk
           { $cond: [{ $gte: [ // Long tenure might mean stagnation
             { $divide: [
               { $subtract: [new Date(), "$joinDate"] },
               365.25 * 24 * 60 * 60 * 1000
             ]},
             5
           ]}, 15, 0] }
         ]
       }
     }
   }
 }
])
```

Explanation: Creates complex business metrics combining multiple factors for HR decision-making.

Business Use Case: "Calculate comprehensive employee performance metrics for strategic HR planning."

Performance Considerations:

Memory Efficiency:

```javascript
// Good: Use $addFields to add only needed fields
db.employees.aggregate([
  { $addFields: { monthlySalary: { $divide: ["$salary", 12] } } },
  { $match: { monthlySalary: { $gte: 5000 } } }
])

// Better: Combine with $project to limit field transmission
db.employees.aggregate([
  { $addFields: { monthlySalary: { $divide: ["$salary", 12] } } },
  { $project: { name: 1, department: 1, monthlySalary: 1 } }
])
```

Index Usage:

- $addFields doesn't affect index usage for subsequent stages
- Computed fields can't use indexes unless materialized

Stage 10: $replaceRoot/$replaceWith - Replace Document Root {#replaceroot}

Understanding $replaceRoot and $replaceWith

The `$replaceRoot` stage (also known as `$replaceWith` in newer MongoDB versions) is like a magician who makes the assistant become the main act. It replaces the entire document with a specified embedded document or computed document.

Key Concepts:

- **$replaceRoot**: Replaces the root document with a specified document
- **$replaceWith**: Alias for $replaceRoot (MongoDB 4.2+)
- **Promotion**: Moving a nested document to the root level
- **Transformation**: Creating entirely new document structures

Sample Data Setup:

javascript

```javascript
db.customers.insertMany([
 {
  _id: 1,
  customerInfo: {
   name: "Alice Johnson",
   email: "alice@example.com",
   age: 28
  },
  address: {
   street: "123 Main St",
   city: "New York",
   zipCode: "10001"
  },
  orders: [
   { orderId: 101, total: 150, date: new Date("2023-01-15") },
   { orderId: 102, total: 200, date: new Date("2023-02-20") }
  ]
 },
 {
  _id: 2,
  customerInfo: {
   name: "Bob Smith",
   email: "bob@example.com",
   age: 35
  },
  address: {
   street: "456 Oak Ave",
   city: "Los Angeles",
   zipCode: "90210"
  },
  orders: [
   { orderId: 201, total: 300, date: new Date("2023-01-10") }
  ]
 }
]);
```

Detailed Use Cases:

1. Basic Document Replacement - Promote Nested Document

javascript

```javascript
db.customers.aggregate([
 { $replaceRoot: { newRoot: "$customerInfo" } }
])
```

Explanation: Replaces the entire document with just the customerInfo nested document.

Output:

javascript

```javascript
{ name: "Alice Johnson", email: "alice@example.com", age: 28 }
{ name: "Bob Smith", email: "bob@example.com", age: 35 }
```

Business Use Case: "Extract customer information as standalone documents for a customer directory."

2. Merging Multiple Nested Documents

javascript

```javascript
db.customers.aggregate([
 {
  $replaceRoot: {
   newRoot: {
    $mergeObjects: [
     "$customerInfo",
     "$address",
     { customerId: "$_id" }
    ]
   }
  }
 }
])
```

Explanation:

- Merges customerInfo and address into a single flat document
- Adds the original _id as customerId
- Creates a flattened structure

Output:

javascript

```javascript
{
  name: "Alice Johnson",
  email: "alice@example.com",
  age: 28,
  street: "123 Main St",
  city: "New York",
  zipCode: "10001",
  customerId: 1
}
```

Business Use Case: "Create flattened customer records for export to external systems."

3. Conditional Document Replacement

javascript

```javascript
db.customers.aggregate([
  {
    $replaceRoot: {
      newRoot: {
        $cond: {
          if: { $gte: ["$customerInfo.age", 30] },
          then: {
            $mergeObjects: [
              "$customerInfo",
              { category: "Senior", discount: 0.1 }
            ]
          },
          else: {
            $mergeObjects: [
              "$customerInfo",
              { category: "Junior", discount: 0.05 }
            ]
          }
        }
      }
    }
  }
])
```

Explanation: Creates different document structures based on customer age, adding appropriate category and discount.

Business Use Case: "Generate age-based customer profiles with appropriate discounts."

4. Array Element Promotion

javascript

```javascript
db.customers.aggregate([
 { $unwind: "$orders" },
 {
  $replaceRoot: {
   newRoot: {
    $mergeObjects: [
     "$orders",
     {
      customerName: "$customerInfo.name",
      customerEmail: "$customerInfo.email",
      customerCity: "$address.city"
     }
    ]
   }
  }
 }
])
```

Explanation:

- Unwinds orders array to separate documents
- Promotes each order to root level
- Includes relevant customer information

Output:

javascript

```javascript
{
 orderId: 101,
 total: 150,
 date: ISODate("2023-01-15"),
 customerName: "Alice Johnson",
 customerEmail: "alice@example.com",
 customerCity: "New York"
}
```

Business Use Case: "Create order-centric records with customer context for order processing system."

5. Computed Document Creation

javascript

```javascript
db.customers.aggregate([
 {
  $replaceRoot: {
   newRoot: {
    customerId: "$_id",
    profile: {
     name: "$customerInfo.name",
     email: "$customerInfo.email",
     location: {
      $concat: ["$address.city", ", ", "$address.zipCode"]
     }
    },
    statistics: {
     totalOrders: { $size: "$orders" },
     totalSpent: { $sum: "$orders.total" },
     avgOrderValue: { $avg: "$orders.total" },
     lastOrderDate: { $max: "$orders.date" }
    },
    segment: {
     $switch: {
      branches: [
       { case: { $gte: [{ $sum: "$orders.total" }, 500] }, then: "Premium" },
       { case: { $gte: [{ $sum: "$orders.total" }, 200] }, then: "Standard" }
      ],
      default: "Basic"
     }
    }
   }
  }
 }
])
```

Explanation: Creates a completely new document structure with computed fields and nested organization.

Business Use Case: "Generate comprehensive customer analytics profiles for business intelligence."

6. Dynamic Field Selection

javascript

```javascript
db.customers.aggregate([
 {
  $replaceRoot: {
   newRoot: {
    $arrayToObject: {
     $filter: {
      input: {
       $objectToArray: {
        id: "$_id",
        name: "$customerInfo.name",
        email: "$customerInfo.email",
        city: "$address.city",
        orderCount: { $size: "$orders" },
        totalSpent: { $sum: "$orders.total" }
       }
      },
      as: "field",
      cond: { $ne: ["$$field.v", null] } // Only include non-null values
     }
    }
   }
  }
 }
])
```

Explanation: Dynamically creates documents with only non-null fields, providing flexible document structure.

Business Use Case: "Create clean customer records excluding any null or undefined fields."

7. Multi-Level Nesting Flattening

javascript

```javascript
// Sample data with deeper nesting
db.products.insertMany([
  {
    _id: 1,
    info: {
      details: {
        name: "Laptop",
        specs: {
          cpu: "Intel i7",
          ram: "16GB",
          storage: "512GB SSD"
        }
      },
      pricing: {
        cost: 800,
        retail: 1200,
        discount: 0.1
      }
    }
  }
]);

db.products.aggregate([
  {
    $replaceRoot: {
      newRoot: {
        productId: "$_id",
        name: "$info.details.name",
        cpu: "$info.details.specs.cpu",
        ram: "$info.details.specs.ram",
        storage: "$info.details.specs.storage",
        cost: "$info.pricing.cost",
        retail: "$info.pricing.retail",
        finalPrice: {
          $multiply: [
            "$info.pricing.retail",
            { $subtract: [1, "$info.pricing.discount"] }
          ]
        }
      }
    }
  }
])
```

Explanation: Flattens deeply nested structures into a simple, usable format with computed fields.

Business Use Case: "Transform complex product hierarchies into flat structures for catalog display."

Advanced Patterns:

8. Conditional Structure Based on Document Type

```javascript
db.mixedData.aggregate([
 {
  $replaceRoot: {
   newRoot: {
    $switch: {
     branches: [
      {
       case: { $eq: ["$type", "customer"] },
       then: {
        id: "$_id",
        name: "$data.name",
        email: "$data.email",
        category: "customer"
       }
      },
      {
       case: { $eq: ["$type", "order"] },
       then: {
        id: "$_id",
        orderId: "$data.orderId",
        total: "$data.total",
        category: "order"
       }
      }
     ],
     default: { id: "$_id", category: "unknown" }
    }
   }
  }
 }
])
```

Explanation: Creates different document structures based on document type, useful for polymorphic collections.

Business Use Case: "Normalize documents from a multi-type collection into consistent structures."

9. Template-Based Document Generation

javascript

```javascript
db.customers.aggregate([
 {
  $replaceRoot: {
   newRoot: {
    // Email template generation
    subject: {
     $concat: [
      "Welcome ",
      "$customerInfo.name",
      "! Your account is ready."
     ]
    },
    body: {
     $concat: [
      "Dear ",
      "$customerInfo.name",
      ",\n\nWelcome to our service! You have ",
      { $toString: { $size: "$orders" } },
      " orders with us totaling $",
      { $toString: { $sum: "$orders.total" } },
      ".\n\nBest regards,\nThe Team"
     ]
    },
    recipient: "$customerInfo.email",
    customerSegment: {
     $cond: {
      if: { $gte: [{ $sum: "$orders.total" }, 300] },
      then: "VIP",
      else: "Regular"
     }
    }
   }
  }
 }
])
```

Explanation: Generates personalized email templates based on customer data.

Business Use Case: "Create personalized marketing email content from customer data."

Performance Considerations:

Memory Usage:

- $replaceRoot creates entirely new documents
- Memory usage depends on the size of the new document structure
- Consider field selection to minimize memory footprint

Pipeline Optimization:

```javascript
// Good: Filter before expensive transformation
db.customers.aggregate([
  { $match: { "customerInfo.age": { $gte: 25 } } }, // Filter first
  { $replaceRoot: { newRoot: { /* complex transformation */ } } }
])

// Bad: Transform everything then filter
db.customers.aggregate([
  { $replaceRoot: { newRoot: { /* complex transformation */ } } },
  { $match: { age: { $gte: 25 } } } // Filter after expensive operation
])
```

Stage 11: $group - Group Documents and Perform Aggregations {#group}

Understanding $group

The `$group` stage is like a skilled accountant who sorts receipts into categories and calculates totals for each category. It groups documents by specified criteria and performs calculations on each group, making it one of the most powerful stages for data analysis.

Key Concepts:

- **_id**: The grouping key (what to group by)
- **Accumulators**: Operations performed on each group
- **Group vs Individual**: Works on sets of documents rather than individual documents
- **Memory**: Can be memory-intensive for large datasets

Common Accumulators:

- `$sum`: Add up values
- `$avg`: Calculate average
- `$min`/`$max`: Find minimum/maximum values
- `$first`/`$last`: Get first/last values in group
- `$push`: Collect values into array
- `$addToSet`: Collect unique values into array

- `$count`: Count documents in group

Sample Data Setup:

javascript

```javascript
db.sales.insertMany([
  { _id: 1, product: "Laptop", category: "Electronics", price: 1200, quantity: 2, date: new Date("2023-01-15"), salesPersor
  { _id: 2, product: "Mouse", category: "Electronics", price: 25, quantity: 10, date: new Date("2023-01-15"), salesPerson:
  { _id: 3, product: "Desk", category: "Furniture", price: 300, quantity: 1, date: new Date("2023-01-16"), salesPerson: "Ali
  { _id: 4, product: "Chair", category: "Furniture", price: 150, quantity: 4, date: new Date("2023-01-16"), salesPerson: "Ch
  { _id: 5, product: "Keyboard", category: "Electronics", price: 80, quantity: 5, date: new Date("2023-01-17"), salesPersor
  { _id: 6, product: "Monitor", category: "Electronics", price: 400, quantity: 3, date: new Date("2023-01-17"), salesPerson
]);
```

Detailed Use Cases:

1. Basic Grouping - Sales by Category

javascript

```javascript
db.sales.aggregate([
  {
    $group: {
      _id: "$category",
      totalSales: { $sum: { $multiply: ["$price", "$quantity"] } },
      totalQuantity: { $sum: "$quantity" },
      averagePrice: { $avg: "$price" },
      productCount: { $sum: 1 }
    }
  }
])
```

Explanation:

- Groups by category
- Calculates total sales revenue (price × quantity)
- Sums total quantity sold
- Averages product prices
- Counts number of products

Output:

javascript

```javascript
{
  _id: "Electronics",
  totalSales: 4050, // (1200*2) + (25*10) + (80*5) + (400*3)
  totalQuantity: 20,
  averagePrice: 426.25,
  productCount: 4
}
{
  _id: "Furniture",
  totalSales: 900,  // (300*1) + (150*4)
  totalQuantity: 5,
  averagePrice: 225,
  productCount: 2
}
```

Business Use Case: "Analyze sales performance by product category for inventory planning."

2. Multiple Grouping Fields - Sales by Date and Salesperson

javascript

```javascript
db.sales.aggregate([
  {
    $group: {
      _id: {
        date: "$date",
        salesPerson: "$salesPerson"
      },
      totalRevenue: { $sum: { $multiply: ["$price", "$quantity"] } },
      itemsSold: { $sum: "$quantity" },
      products: { $push: "$product" }
    }
  },
  { $sort: { "_id.date": 1, "_id.salesPerson": 1 } }
])
```

Explanation:

- Groups by both date and salesperson
- Calculates revenue and items sold for each person per day
- Collects list of products sold
- Sorts results by date and salesperson

Business Use Case: "Track daily sales performance by individual salespeople."

3. Advanced Aggregation Operations

javascript

```javascript
db.sales.aggregate([
 {
  $group: {
   _id: "$salesPerson",
   totalRevenue: { $sum: { $multiply: ["$price", "$quantity"] } },
   bestSale: { $max: { $multiply: ["$price", "$quantity"] } },
   worstSale: { $min: { $multiply: ["$price", "$quantity"] } },
   firstSale: { $first: "$$ROOT" }, // Entire first document
   lastSale: { $last: "$$ROOT" },   // Entire last document
   uniqueCategories: { $addToSet: "$category" },
   allProducts: { $push: "$product" },
   avgDealSize: { $avg: { $multiply: ["$price", "$quantity"] } }
  }
 }
])
```

Explanation:

- Groups by salesperson
- Uses multiple accumulator operations
- `$$ROOT` refers to the entire document
- `$addToSet` creates unique arrays
- `$push` creates arrays with duplicates

Business Use Case: "Generate comprehensive salesperson performance reports."

4. Conditional Grouping with $cond

```javascript
db.sales.aggregate([
 {
  $group: {
   _id: "$category",
   highValueSales: {
    $sum: {
     $cond: {
      if: { $gte: [{ $multiply: ["$price", "$quantity"] }, 500] },
      then: { $multiply: ["$price", "$quantity"] },
      else: 0
     }
    }
   },
   lowValueSales: {
    $sum: {
     $cond: {
      if: { $lt: [{ $multiply: ["$price", "$quantity"] }, 500] },
      then: { $multiply: ["$price", "$quantity"] },
      else: 0
     }
    }
   },
   highValueCount: {
    $sum: {
     $cond: {
      if: { $gte: [{ $multiply: ["$price", "$quantity"] }, 500] },
      then: 1,
      else: 0
     }
    }
   }
  }
 }
])
```

Explanation: Uses conditional logic within aggregation operations to separate high-value and low-value sales.

Business Use Case: "Analyze sales distribution between high-value and low-value transactions by category."

5. Date-Based Grouping

javascript

```javascript
db.sales.aggregate([
 {
  $group: {
   _id: {
    year: { $year: "$date" },
    month: { $month: "$date" },
    day: { $dayOfMonth: "$date" }
   },
   dailyRevenue: { $sum: { $multiply: ["$price", "$quantity"] } },
   transactionCount: { $sum: 1 },
   avgTransactionSize: { $avg: { $multiply: ["$price", "$quantity"] } },
   topSalesperson: { $first: "$salesPerson" } // Note: May not be meaningful without sorting
  }
 },
 { $sort: { "_id.year": 1, "_id.month": 1, "_id.day": 1 } }
])
```

Explanation: Groups by date components to analyze daily sales patterns.

Business Use Case: "Generate daily sales reports for trend analysis."

6. Nested Grouping - Multi-Level Analysis

javascript

```javascript
db.sales.aggregate([
 // First grouping: by category and date
 {
  $group: {
   _id: {
    category: "$category",
    date: "$date"
   },
   dailyCategoryRevenue: { $sum: { $multiply: ["$price", "$quantity"] } },
   dailyCategoryCount: { $sum: 1 }
  }
 },
 // Second grouping: by category only
 {
  $group: {
   _id: "$_id.category",
   totalRevenue: { $sum: "$dailyCategoryRevenue" },
   totalTransactions: { $sum: "$dailyCategoryCount" },
   avgDailyRevenue: { $avg: "$dailyCategoryRevenue" },
   bestDay: { $max: "$dailyCategoryRevenue" },
   worstDay: { $min: "$dailyCategoryRevenue" },
   salesDays: { $sum: 1 }
  }
 }
])
```

Explanation:

- First groups by category and date
- Then regroups by category only
- Provides both daily and overall statistics

Business Use Case: "Analyze category performance with both daily and overall metrics."

7. Complex Business Logic Grouping

javascript

```javascript
db.sales.aggregate([
 {
  $addFields: {
   revenuePerItem: { $multiply: ["$price", "$quantity"] },
   priceSegment: {
    $switch: {
     branches: [
      { case: { $gte: ["$price", 300] }, then: "Premium" },
      { case: { $gte: ["$price", 100] }, then: "Mid-Range" },
      { case: { $gte: ["$price", 50] }, then: "Budget" }
     ],
     default: "Economy"
    }
   }
  }
 },
 {
  $group: {
   _id: {
    category: "$category",
    priceSegment: "$priceSegment"
   },
   totalRevenue: { $sum: "$revenuePerItem" },
   avgPrice: { $avg: "$price" },
   totalQuantity: { $sum: "$quantity" },
   productTypes: { $addToSet: "$product" },
   salespeople: { $addToSet: "$salesPerson" }
  }
 }
])
```

Explanation: Combines computed fields with grouping to create sophisticated business analytics.

Business Use Case: "Analyze sales performance by category and price segment for strategic planning."

Performance Optimization:

Index Usage:

```javascript
// Create compound index for efficient grouping
db.sales.createIndex({ category: 1, date: 1 })

// This grouping will use the index efficiently
db.sales.aggregate([
  { $match: { category: "Electronics" } }, // Uses index
  { $group: { _id: "$date", total: { $sum: "$price" } } }
])
```

Memory Management:

```javascript
// Good: Filter before grouping
db.sales.aggregate([
  { $match: { date: { $gte: new Date("2023-01-01") } } }, // Reduce input size
  { $group: { _id: "$category", total: { $sum: "$price" } } }
])

// Use allowDiskUse for large datasets
db.sales.aggregate([
  { $group: { _id: "$category", total: { $sum: "$price" } } }
], { allowDiskUse: true })
```

Common Patterns and Best Practices {#patterns}

1. Pipeline Optimization Order

The order of stages in your aggregation pipeline significantly impacts performance:

```javascript
// Optimal order
db.collection.aggregate([
```

MongoDB Aggregation Framework - Complete Guide (Continued)

Common Patterns and Best Practices {#patterns}

1. Pipeline Optimization Order

The order of stages in your aggregation pipeline significantly impacts performance:

```javascript
// Optimal order
db.collection.aggregate([
  { $match: { status: "active" } },        // 1. Filter early (reduces data)
  { $lookup: { /* join data */ } },        // 2. Join only needed data
  { $unwind: "$arrayField" },              // 3. Unwind after filtering
  { $addFields: { computed: { /* calc */ } } }, // 4. Add computed fields
  { $group: { _id: "$category", total: { $sum: "$amount" } } }, // 5. Group
  { $sort: { total: -1 } },                // 6. Sort grouped results
  { $limit: 10 },                          // 7. Limit final results
  { $project: { _id: 0, category: "$_id", total: 1 } } // 8. Shape output
]);

// Poor order (inefficient)
db.collection.aggregate([
  { $lookup: { /* expensive join on all data */ } },
  { $unwind: "$arrayField" },
  { $addFields: { computed: { /* expensive calc */ } } },
  { $match: { status: "active" } },        // Too late - already processed everything
  { $group: { _id: "$category", total: { $sum: "$amount" } } },
  { $sort: { total: -1 } },
  { $limit: 10 }
]);
```

Key Principles:

- **Filter Early**: Use `$match` as early as possible
- **Reduce Data Volume**: Limit fields and documents before expensive operations
- **Index-Friendly**: Structure queries to use indexes effectively
- **Minimize Memory**: Use `$project` to exclude unnecessary fields

2. Error Handling and Validation Patterns

javascript

```javascript
// Robust aggregation with error handling
db.orders.aggregate([
 {
  $addFields: {
   // Handle missing or null values
   safeQuantity: { $ifNull: ["$quantity", 0] },
   safePrice: { $ifNull: ["$price", 0] },

   // Validate data types
   validatedTotal: {
    $cond: {
     if: {
      $and: [
       { $type: ["$quantity", "number"] },
       { $type: ["$price", "number"] }
      ]
     },
     then: { $multiply: ["$quantity", "$price"] },
     else: 0
    }
   },

   // Handle division by zero
   avgItemPrice: {
    $cond: {
     if: { $eq: ["$quantity", 0] },
     then: 0,
     else: { $divide: ["$total", "$quantity"] }
    }
   }
  }
 }
]);
```

3. Reusable Pipeline Components

javascript

```javascript
// Define reusable pipeline stages
const commonFilters = [
  { $match: { status: { $in: ["active", "pending"] } } },
  { $match: { date: { $gte: new Date("2023-01-01") } } }
];

const enrichmentStages = [
  {
    $lookup: {
      from: "customers",
      localField: "customerId",
      foreignField: "_id",
      as: "customerInfo"
    }
  },
  { $unwind: "$customerInfo" }
];

const computedFields = [
  {
    $addFields: {
      totalValue: { $multiply: ["$quantity", "$price"] },
      customerTier: {
        $switch: {
          branches: [
            { case: { $gte: ["$customerInfo.totalSpent", 10000] }, then: "Premium" },
            { case: { $gte: ["$customerInfo.totalSpent", 5000] }, then: "Gold" },
            { case: { $gte: ["$customerInfo.totalSpent", 1000] }, then: "Silver" }
          ],
          default: "Bronze"
        }
      }
    }
  }
];

// Use in different aggregations
db.orders.aggregate([
  ...commonFilters,
  ...enrichmentStages,
  ...computedFields,
  { $group: { _id: "$customerTier", count: { $sum: 1 } } }
]);
```

4. Dynamic Pipeline Building

javascript

```javascript
function buildSalesAnalyticsPipeline(options = {}) {
 const pipeline = [];

 // Dynamic filtering
 const matchStage = { $match: {} };
 if (options.dateRange) {
  matchStage.$match.date = {
   $gte: options.dateRange.start,
   $lte: options.dateRange.end
  };
 }
 if (options.categories) {
  matchStage.$match.category = { $in: options.categories };
 }
 if (options.minAmount) {
  matchStage.$match.amount = { $gte: options.minAmount };
 }

 if (Object.keys(matchStage.$match).length > 0) {
  pipeline.push(matchStage);
 }

 // Dynamic grouping
 const groupStage = {
  $group: {
   _id: null,
   totalRevenue: { $sum: "$amount" },
   totalTransactions: { $sum: 1 },
   avgTransactionSize: { $avg: "$amount" }
  }
 };

 if (options.groupBy) {
  groupStage.$group._id = `$${options.groupBy}`;
 }

 pipeline.push(groupStage);

 // Dynamic sorting
 if (options.sortBy) {
  const sortStage = { $sort: {} };
  sortStage.$sort[options.sortBy] = options.sortOrder || -1;
  pipeline.push(sortStage);
 }

 // Dynamic limit
```

```
  if (options.limit) {
    pipeline.push({ $limit: options.limit });
  }

  return pipeline;
}

// Usage examples
const pipeline1 = buildSalesAnalyticsPipeline({
  dateRange: { start: new Date("2023-01-01"), end: new Date("2023-12-31") },
  categories: ["Electronics", "Clothing"],
  groupBy: "category",
  sortBy: "totalRevenue",
  limit: 10
});

const pipeline2 = buildSalesAnalyticsPipeline({
  minAmount: 100,
  groupBy: "salesPerson",
  sortBy: "totalTransactions"
});
```

5. Memory-Efficient Patterns

javascript

```javascript
// Pattern 1: Streaming large datasets
db.largeCollection.aggregate([
  { $match: { /* filter criteria */ } },
  { $project: { /* only needed fields */ } },
  { $limit: 1000 }, // Process in batches
  { $group: { /* aggregation */ } }
], { allowDiskUse: true });

// Pattern 2: Incremental processing
function processInBatches(collection, batchSize = 1000) {
  let skip = 0;
  let hasMore = true;

  while (hasMore) {
    const batch = collection.aggregate([
      { $skip: skip },
      { $limit: batchSize },
      { $match: { /* your criteria */ } },
      { $group: { /* your aggregation */ } }
    ]);

    const results = batch.toArray();
    hasMore = results.length === batchSize;
    skip += batchSize;

    // Process results
    results.forEach(doc => {
      // Handle individual document
    });
  }
}

// Pattern 3: Field exclusion for memory efficiency
db.documents.aggregate([
  { $project: {
    largeTextField: 0, // Exclude large fields
    binaryData: 0,     // Exclude binary data
    // Include only what you need
    id: 1,
    name: 1,
    amount: 1
  }},
  { $group: { /* aggregation on smaller docs */ } }
]);
```

6. Advanced Lookup Patterns

javascript

```
// Pattern 1: Multiple lookups with pipeline
db.orders.aggregate([
 {
  $lookup: {
   from: "customers",
   let: { customerId: "$customerId" },
   pipeline: [
    { $match: { $expr: { $eq: ["$_id", "$$customerId"] } } },
    { $project: { name: 1, email: 1, tier: 1 } } // Only needed fields
   ],
   as: "customer"
  }
 },
 {
  $lookup: {
   from: "products",
   let: { productIds: "$items.productId" },
   pipeline: [
    { $match: { $expr: { $in: ["$_id", "$$productIds"] } } },
    { $project: { name: 1, price: 1, category: 1 } }
   ],
   as: "productDetails"
  }
 }
]);

// Pattern 2: Conditional lookup
db.users.aggregate([
 {
  $lookup: {
   from: "profiles",
   let: { userId: "$_id", userType: "$type" },
   pipeline: [
    {
     $match: {
      $expr: {
       $and: [
        { $eq: ["$userId", "$$userId"] },
        { $eq: ["$profileType", "$$userType"] }
       ]
      }
     }
    }
   ],
   as: "profile"
  }
  }
]);

// Pattern 3: Aggregation within lookup
db.customers.aggregate([
  {
    $lookup: {
      from: "orders",
      let: { customerId: "$_id" },
      pipeline: [
        { $match: { $expr: { $eq: ["$customerId", "$$customerId"] } } },
        {
          $group: {
            _id: null,
            totalOrders: { $sum: 1 },
            totalSpent: { $sum: "$amount" },
            avgOrderValue: { $avg: "$amount" },
            lastOrderDate: { $max: "$date" }
          }
        }
      ],
      as: "orderStats"
    }
  }
]);
```

7. Complex Date and Time Patterns

javascript

```
// Pattern 1: Time-based analytics
db.events.aggregate([
 {
  $addFields: {
   hour: { $hour: "$timestamp" },
   dayOfWeek: { $dayOfWeek: "$timestamp" },
   weekOfYear: { $week: "$timestamp" },
   quarter: {
    $switch: {
     branches: [
      { case: { $lte: [{ $month: "$timestamp" }, 3] }, then: "Q1" },
      { case: { $lte: [{ $month: "$timestamp" }, 6] }, then: "Q2" },
      { case: { $lte: [{ $month: "$timestamp" }, 9] }, then: "Q3" }
     ],
     default: "Q4"
    }
   }
  }
 },
 {
  $group: {
   _id: {
    quarter: "$quarter",
    dayOfWeek: "$dayOfWeek",
    hour: "$hour"
   },
   eventCount: { $sum: 1 },
   uniqueUsers: { $addToSet: "$userId" }
  }
 }
]);

// Pattern 2: Rolling window calculations
db.dailyMetrics.aggregate([
 { $sort: { date: 1 } },
 {
  $setWindowFields: {
   sortBy: { date: 1 },
   output: {
    sevenDayAvg: {
     $avg: "$value",
     window: { documents: [-6, 0] } // Current + 6 previous days
    },
    thirtyDaySum: {
     $sum: "$value",
     window: { documents: [-29, 0] } // Current + 29 previous days
```

```
        }
      }
    }
  }
]);

// Pattern 3: Business hours analysis
db.transactions.aggregate([
  {
    $addFields: {
      isBusinessHour: {
        $and: [
          { $gte: [{ $hour: "$timestamp" }, 9] },  // After 9 AM
          { $lt: [{ $hour: "$timestamp" }, 17] },  // Before 5 PM
          { $gte: [{ $dayOfWeek: "$timestamp" }, 2] }, // Monday or later
          { $lte: [{ $dayOfWeek: "$timestamp" }, 6] } // Friday or earlier
        ]
      }
    }
  },
  {
    $group: {
      _id: "$isBusinessHour",
      totalTransactions: { $sum: 1 },
      totalAmount: { $sum: "$amount" },
      avgAmount: { $avg: "$amount" }
    }
  }
]);
```

8. Data Validation and Cleansing Patterns

javascript

```
// Pattern 1: Data quality assessment
db.customers.aggregate([
 {
  $addFields: {
   dataQuality: {
    hasName: { $ne: ["$name", null] },
    hasEmail: { $ne: ["$email", null] },
    hasValidEmail: {
     $regexMatch: {
      input: { $toString: "$email" },
      regex: /^[^\s@]+@[^\s@]+\.[^\s@]+$/
     }
    },
    hasPhone: { $ne: ["$phone", null] },
    hasAddress: { $ne: ["$address", null] }
   }
  }
 },
 {
  $addFields: {
   qualityScore: {
    $add: [
     { $cond: ["$dataQuality.hasName", 1, 0] },
     { $cond: ["$dataQuality.hasEmail", 1, 0] },
     { $cond: ["$dataQuality.hasValidEmail", 1, 0] },
     { $cond: ["$dataQuality.hasPhone", 1, 0] },
     { $cond: ["$dataQuality.hasAddress", 1, 0] }
    ]
   }
  }
 },
 {
  $group: {
   _id: "$qualityScore",
   count: { $sum: 1 },
   customers: { $push: "$_id" }
  }
 }
]);

// Pattern 2: Data standardization
db.products.aggregate([
 {
  $addFields: {
   standardizedName: {
    $trim: {
        input: { $toLower: "$name" }
      }
    },
    standardizedCategory: {
      $switch: {
        branches: [
          { case: { $regexMatch: { input: "$category", regex: /electronic/i } }, then: "Electronics" },
          { case: { $regexMatch: { input: "$category", regex: /furniture/i } }, then: "Furniture" },
          { case: { $regexMatch: { input: "$category", regex: /cloth/i } }, then: "Clothing" }
        ],
        default: { $toTitle: "$category" }
      }
    },
    validatedPrice: {
      $cond: {
        if: { $and: [{ $type: ["$price", "number"] }, { $gt: ["$price", 0] }] },
        then: "$price",
        else: null
      }
    }
  }
}
]);
```

9. Performance Monitoring Patterns

javascript

```javascript
// Pattern 1: Query performance analysis
function analyzeAggregationPerformance(pipeline) {
  const startTime = new Date();

  // Add explain stage for analysis
  const explainResult = db.collection.aggregate(pipeline, { explain: true });

  // Run actual aggregation
  const results = db.collection.aggregate(pipeline).toArray();

  const endTime = new Date();
  const executionTime = endTime - startTime;

  return {
    executionTime,
    resultCount: results.length,
    explain: explainResult,
    results: results.slice(0, 10) // Sample results
  };
}

// Pattern 2: Index usage verification
db.collection.aggregate([
  { $match: { category: "Electronics" } }, // Should use index
  { $group: { _id: "$brand", count: { $sum: 1 } } }
], { explain: true });

// Pattern 3: Memory usage monitoring
db.collection.aggregate([
  { $group: { _id: "$category", count: { $sum: 1 } } }
], {
  allowDiskUse: true,
  maxTimeMS: 30000, // 30 second timeout
  cursor: { batchSize: 100 } // Control batch size
});
```

10. Testing and Validation Patterns

javascript

```
// Pattern 1: Unit testing aggregation stages
function testAggregationStage(stageName, stage, inputData, expectedOutput) {
  const testCollection = `test_${stageName}_${Date.now()}`;

  // Insert test data
  db[testCollection].insertMany(inputData);

  // Run aggregation stage
  const result = db[testCollection].aggregate([stage]).toArray();

  // Validate results
  const isValid = JSON.stringify(result) === JSON.stringify(expectedOutput);

  // Cleanup
  db[testCollection].drop();

  return {
    stageName,
    passed: isValid,
    expected: expectedOutput,
    actual: result
  };
}

// Pattern 2: Data integrity validation
db.orders.aggregate([
  {
    $addFields: {
      // Validate calculated total matches item totals
      calculatedTotal: {
        $sum: {
          $map: {
            input: "$items",
            as: "item",
            in: { $multiply: ["$$item.quantity", "$$item.price"] }
          }
        }
      }
    }
  },
  {
    $addFields: {
      totalMismatch: { $ne: ["$total", "$calculatedTotal"] }
    }
  },
  {
    $match: { totalMismatch: true }
  },
  {
    $project: {
      _id: 1,
      total: 1,
      calculatedTotal: 1,
      difference: { $subtract: ["$total", "$calculatedTotal"] }
    }
  }
]);
```

Performance Optimization Strategies {#performance}

1. Index Strategy for Aggregations

```javascript
// Create indexes that support your aggregation patterns
db.sales.createIndex({ date: 1, category: 1 });         // For date-based grouping
db.sales.createIndex({ salesPerson: 1, amount: -1 });    // For salesperson analysis
db.sales.createIndex({ category: 1, amount: 1 });        // For category totals

// Compound indexes for complex queries
db.orders.createIndex({
  customerId: 1,
  status: 1,
  date: -1
});

// Partial indexes for specific conditions
db.products.createIndex(
  { category: 1, price: 1 },
  { partialFilterExpression: { active: true } }
);
```

2. Memory Management

javascript

```javascript
// Use allowDiskUse for large datasets
db.largeCollection.aggregate([
  { $group: { _id: "$category", total: { $sum: "$amount" } } }
], { allowDiskUse: true });

// Limit memory usage with proper field selection
db.documents.aggregate([
  { $project: { _id: 1, category: 1, amount: 1 } }, // Only needed fields
  { $group: { _id: "$category", total: { $sum: "$amount" } } }
]);

// Process in batches for very large collections
const batchSize = 10000;
let skip = 0;
let hasMore = true;

while (hasMore) {
  const batch = db.collection.aggregate([
    { $skip: skip },
    { $limit: batchSize },
    { $group: { _id: "$category", count: { $sum: 1 } } }
  ]).toArray();

  hasMore = batch.length === batchSize;
  skip += batchSize;

  // Process batch results
}
```

3. Query Optimization

javascript

```javascript
// Good: Filter early and use indexes
db.orders.aggregate([
  { $match: {
    date: { $gte: new Date("2023-01-01") }, // Uses index
    status: "completed"                  // Uses index
  }},
  { $lookup: { /* join only filtered data */ } },
  { $group: { _id: "$customerId", total: { $sum: "$amount" } } }
]);

// Bad: Filter after expensive operations
db.orders.aggregate([
  { $lookup: { /* expensive join on all data */ } },
  { $unwind: "$items" },
  { $group: { _id: "$customerId", total: { $sum: "$amount" } } },
  { $match: { total: { $gte: 1000 } } } // Too late
]);
```

Conclusion

The MongoDB Aggregation Framework is a powerful tool for data processing and analysis. By understanding these patterns and best practices, you can:

- Build efficient, maintainable aggregation pipelines
- Handle complex data transformations
- Optimize performance for large datasets
- Create reusable, testable code
- Implement robust error handling

Remember to always:

- **Test with realistic data volumes**
- **Monitor performance metrics**
- **Use appropriate indexes**
- **Consider memory limitations**
- **Validate data integrity**

The key to mastering aggregations is practice and understanding your specific use cases. Start with simple pipelines and gradually build complexity as you become more comfortable with the framework.

This guide provides a comprehensive foundation for working with MongoDB's Aggregation Framework. Continue experimenting with different combinations of stages to solve your specific data processing challenges.