MongoDB Aggregations

Prof. Dr. Gerald Pirkl

December 2025

Introduction

- MongoDB's **Aggregation Pipeline** is a data processing framework designed for complex analytical queries. Performed in sequences (stages).
- Inspired by functional programming concepts: each stage transforms documents and passes them on.
- The **output** of one stage **becomes the input** of the next one, forming a composable transformation chain.
- **Stages** can filter, group, reshape, sort, compute statistics, or even join collections.
- Each stage behaves like a **transformation** function.
- Documents "flow" through the pipeline, similar to a chain of `filter()`, `map()`, `flatMap()`, `reduce()` operations.
- executed within the database engine, making it suitable for data-heavy AI/ML applications $\rightarrow$ Performance!
- **Applications**: feature engineering, preprocessing, data cleaning, statistical summaries.
- **Core Concept: Aggregation Pipeline:** a chain of processing steps (*stages*) that documents pass through sequentially.

Important Stages

$match – Filtering

Filters documents according to given conditions. It reduces the amount of data early in the pipeline, which improves performance. (Do this as soon as possible!)

```
{ $match: { status: "active" } }
```

$project – Selecting or Transforming Fields

Controls which fields appear in the output and allows creating computed fields. It is commonly used to reshape documents and reduce payload size.

```
{ $project: {
    name: 1,
    priceWithTax: { $multiply: ["$price", 1.19] }
} }
```

$addFields – Adding Computed Fields

Adds new fields to documents without removing existing ones. It is useful when adding intermediate calculations before grouping or sorting.

```
{ $addFields: {
    finalPrice: { $multiply: ["$price", 1.19] }
} }
```

$group – Grouping

Groups documents by a key and computes aggregate values like sums or averages. It is equivalent to SQL's GROUP BY and is essential for analytical queries.

```
{
  $group: {
    _id: "$category",
    total: { $sum: "$amount" },
    avg: { $avg: "$amount" }
  }
}
```

This `$group` stage groups all documents by their `category` field. Each group produces a single output document whose `_id` is the category value.
The stage then computes two aggregate metrics per group:

- `total`: the sum of all `amount` values within the group (`$sum`)
- `avg`: the average of all `amount` values within the group (`$avg`)

As a result, the output contains one document per category, along with its total and average amount.

Example Suppose the collection contains the following documents:

```
{ category: "A", amount: 10 }
{ category: "A", amount: 30 }
{ category: "B", amount: 5 }
{ category: "B", amount: 15 }
{ category: "B", amount: 20 }
```

After applying the `$group` stage:

```
{
  $group: {
    _id: "$category",
    total: { $sum: "$amount" },
    avg: { $avg: "$amount" }
  }
}
```

the output becomes:

```
{ _id: "A", total: 40, avg: 20 }
{ _id: "B", total: 40, avg: 13.333... }
```

The pipeline produces one document per category, computing the sum and average of all `amount` values within each group.

$sort – Sorting

Orders the documents by one or more fields. Sorting is typically applied before limiting output or presenting results to the user.

```
{ $sort: { createdAt: -1 } }
```

0.1 $limit – Limiting Output

Restricts the number of output documents to a specified count. It is commonly used to implement pagination or "Top N" queries.

```
{ $limit: 10 }
```

$lookup – Join Between Collections

The `$lookup` stage performs a left outer join between the current collection and another collection. It matches documents by comparing a field in the input documents (`localField`) with a field in the foreign collection (`foreignField`). All matching documents from the foreign collection are returned as an array in the specified output field (`as`). If no documents match, the array will be empty.

```
{
  $lookup: {
    from: "users",
    localField: "userId",
    foreignField: "_id",
    as: "user"
  }
}
```

Example Assume we have the following collections:
orders:

```
{ _id: 1, userId: 101, total: 50 }
{ _id: 2, userId: 102, total: 75 }
{ _id: 3, userId: 101, total: 20 }
```

users:

```
{ _id: 101, name: "Alice" }
{ _id: 102, name: "Bob" }
```

Applying the `$lookup` stage:

```
{
  $lookup: {
    from: "users",
    localField: "userId",
    foreignField: "_id",
    as: "user"
  }
}
```

results in:

```
{
  _id: 1,
  userId: 101,
  total: 50,
  user: [ { _id: 101, name: "Alice" } ]
}

{
  _id: 2,
  userId: 102,
  total: 75,
  user: [ { _id: 102, name: "Bob" } ]
}

{
  _id: 3,
  userId: 101,
  total: 20,
  user: [ { _id: 101, name: "Alice" } ]
}
```

Each order document now contains an array `user` with the matching user profile. This enables relational-style queries directly inside MongoDB.

$unwind – Unwrapping Arrays

The `$unwind` stage takes an array field and transforms each element of that array into its own separate output document. This is useful when you want to process, filter, or aggregate individual array elements rather than the entire array as a single field.
If a document contains an array with multiple items, `$unwind` will produce multiple documents — one for each element. All non-array fields are duplicated across the generated documents.

{ $unwind: "$items" }

Example Given the following input document:

```
{
  _id: 1,
  customer: "Alice",
  items: [
    { product: "Book", price: 12 },
    { product: "Pen", price: 2 },
    { product: "Bag", price: 30 }
  ]
}
```

Applying the `$unwind` stage:

```
{ $unwind: "$items" }
```

produces three separate documents:

```
{
  _id: 1,
  customer: "Alice",
  items: { product: "Book", price: 12 }
}

{
  _id: 1,
  customer: "Alice",
  items: { product: "Pen", price: 2 }
}

{
  _id: 1,
  customer: "Alice",
  items: { product: "Bag", price: 30 }
}
```

This allows further stages such as `$match`, `$group`, or `$project` to operate on the individual array elements.

$count – Counting Documents

Returns the number of documents that pass through the pipeline. It is useful for statistics, pagination, or quick summary metrics.

```
{ $count: "numDocuments" }
```

Stage	Typical Use Cases
`$match`	Filtering documents early in the pipeline, applying conditions (e.g. status, date ranges, category filters), improving performance via index usage.
`$project`	Selecting only required fields, computing derived fields, reshaping documents, reducing response size.
`$addFields` / `$set`	Adding intermediate computed fields (e.g. tax, discounts), preparing values for grouping or sorting, enriching documents with metadata.
`$group`	Summaries such as total sales per customer, average scores per module, counts per category, statistical aggregations.
`$sort`	Sorting search results, ordering analytics outputs, preparing data for pagination or ranking.
`$limit`	Implementing pagination, returning only top-N results (e.g. Top 10 customers), reducing output size.
`$lookup`	Joining related data from other collections such as user details, product information, or metadata enrichment.
`$unwind`	Flattening arrays for analysis, iterating over multiple attempts or items, preparing data for grouping or filtering by array elements.
`$count`	Computing total document counts after filtering, analytics KPIs, pagination metadata.
`$facet`	Running multiple pipelines in parallel (e.g. filters + statistics + histograms), building dashboards or combined analytics outputs.
`$bucket` / `$bucketAuto`	Creating histograms, grouping values into ranges (e.g. age groups, price tiers), statistical characterization.
`$sortByCount`	Frequency analysis (e.g. most common categories), quick distribution summaries, data profiling.

Key Operators

Arithmetic Operators

- $sum
- $avg
- $max / $min
- $multiply
- $divide

String Operators

- $concat
- $substr
- $toLower / $toUpper

Stage	**Description**	**SQL Equivalent**
`$match`	Filters documents based on conditions.	`WHERE`
`$project`	Selects fields, renames them, or computes new fields.	`SELECT ...` (with expressions)
`$addFields` / `$set`	Adds new computed fields without removing existing ones.	`SELECT ... AS ...`
`$group`	Groups documents and computes aggregates.	`GROUP BY`
`$sort`	Sorts documents by one or more fields.	`ORDER BY`
`$limit`	Restricts the number of returned documents.	`LIMIT`
`$count`	Produces a count of documents.	`COUNT(*)`
`$unwind`	Deconstructs arrays into individual documents.	No direct SQL equivalent (closest:
`$lookup`	Performs joins with another collection.	`LEFT OUTER JOIN`
`$facet`	Runs multiple pipelines in parallel and returns them together.	No direct SQL equivalent (closest:
`$bucket` / `$bucketAuto`	Groups documents into ranges (histogram bins).	`WIDTH_BUCKET()`, `NTILE()` (approx
`$sortByCount`	Groups by a field and returns counts sorted by frequency.	`GROUP BY ... ORDER BY COUNT(`
`$replaceRoot` / `$replaceWith`	Replaces the root document with a subdocument.	No direct SQL equivalent
`$setWindowFields`	Window functions (running totals, rankings, etc.).	`OVER(...)` window functions

Table 1: Overview of essential MongoDB aggregation stages and their SQL counterparts

Array Operators

- $size
- $slice
- $filter
- $push

MongoDB Example Pipeline

The following pipeline extracts students who attended the `Databases` module, flattens the data, computes the best score, and sorts the results:

```
db.students.aggregate([
  // 1. Filter (similar to filter())
  {
    $match: {
      "modules.module": "Databases"
    }
  },

  // 2. Flatten modules (similar to flatMap())
  {
    $unwind: "$modules"
  },

  // 3. Filter again to keep only "Databases"
  {
    $match: {
      "modules.module": "Databases"
    }
  },

  // 4. Compute best attempt (map() + reduce())
  {
    $project: {
      firstName: 1,
      lastName: 1,
      bestAttempt: { $min: "$modules.attempts" }
    }
  },

  // 5. Sort (like sort())
  {
    $sort: {
      bestAttempt: 1
    }
  }
]);
```

Example Input Document

```
{
  _id: 1,
  firstName: "Alice",
  lastName: "Miller",
  modules: [
    {
      module: "Databases",
      attempts: [78, 82, 75]
    },
    {
      module: "Networks",
      attempts: [88, 90]
    },
    {
      module: "Databases",
      attempts: [70, 72]
    }
  ]
}
```

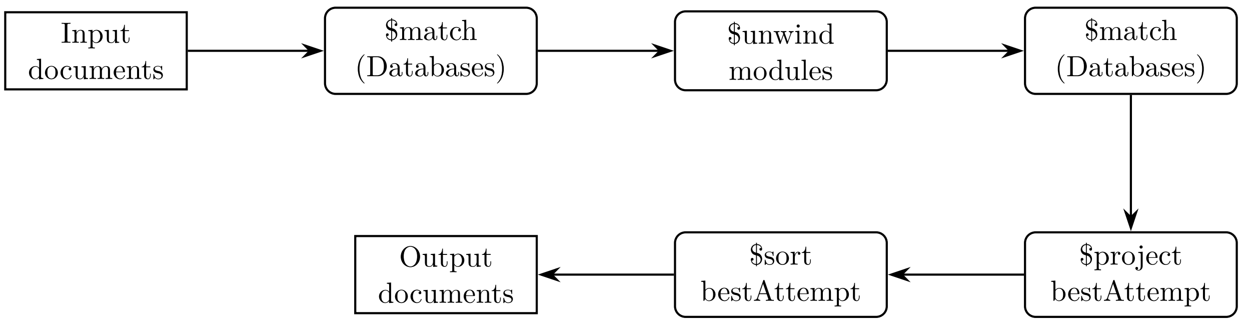

Figure 1: Example MongoDB aggregation pipeline as a sequence of stages.

Step-by-Step Explanation of pipeline:

1. **Stage 1: `$match`** Keep students who have at least one module named `"Databases"`. The example student contains two such modules, so the document passes through unchanged.

```
{
  _id: 1,
  firstName: "Alice",
  lastName: "Miller",
  modules: [
    { module: "Databases", attempts: [78, 82, 75] },
    { module: "Networks", attempts: [88, 90] },
    { module: "Databases", attempts: [70, 72] }
  ]
}
```

2. **Stage 2: `$unwind`** The `modules` array is expanded so each module becomes its own document:

```
1) {
     _id: 1,
     firstName: "Alice",
     lastName: "Miller",
     modules: { module: "Databases", attempts: [78, 82, 75] }
   }

2) {
     _id: 1,
     firstName: "Alice",
     lastName: "Miller",
     modules: { module: "Networks", attempts: [88, 90] }
   }

3) {
     _id: 1,
     firstName: "Alice",
     lastName: "Miller",
     modules: { module: "Databases", attempts: [70, 72] }
   }
```

3. **Stage 3: `$match`** Now we filter again — but this time only the unfolded module documents. Only entries with `module = "Databases"` remain:

```
1) {
     _id: 1,
     firstName: "Alice",
     lastName: "Miller",
     modules: { module: "Databases", attempts: [78, 82, 75] }
   }

2) {
     _id: 1,
     firstName: "Alice",
     lastName: "Miller",
     modules: { module: "Databases", attempts: [70, 72] }
   }
```

4. **Stage 4: `$project`** Compute the best (minimum) attempt for each module entry. Two separate results are created:

```
{
  firstName: "Alice",
  lastName: "Miller",
  bestAttempt: 75 // min of [78, 82, 75]
}

{
  firstName: "Alice",
  lastName: "Miller",
  bestAttempt: 70 // min of [70, 72]
}
```

5. **Stage 5: `$sort`** Finally, sort by best attempt (ascending):

```
{
  firstName: "Alice",
  lastName: "Miller",
  bestAttempt: 70
}

{
  firstName: "Alice",
  lastName: "Miller",
  bestAttempt: 75
}
```

Equivalent Functional Programming (Python)

Below is the same logic expressed using Python's functional style (list comprehensions, filter, map, and sorted):

```python
from itertools import chain

# 1. Filter students with Databases module
filtered_students = filter(
    lambda s: any(m["module"] == "Databases" for m in s["modules"]),
    students
)

# 2. Flatten (flatMap) and keep only Databases modules
flattened = [
    {
        "firstName": s["firstName"],
        "lastName": s["lastName"],
        "attempts": m["attempts"]
    }
    for s in filtered_students
    for m in s["modules"]
    if m["module"] == "Databases"
]

# 3. Compute best attempt
with_best_attempt = [
    {
        "firstName": x["firstName"],
        "lastName": x["lastName"],
        "bestAttempt": min(x["attempts"])
    }
    for x in flattened
]

# 4. Sort results by best attempt
result = sorted(with_best_attempt, key=lambda x: x["bestAttempt"])
```

Keep in mind, that data base operations are highly optimized. Typically DB systems are built for performance and come with large memory and processing ressource (e.g. when doing unwinding operations). Keep these operations on the DB systems

Pipeline vs. Map-Reduce

Aggregation Pipeline	**Map-Reduce**
Fast, optimized	Slow, JavaScript-based
Parallelizable	Single-step
Default method	Only useful for very complex logic

Common Pipeline Stages

- **$match** – filters documents similar to SQL WHERE.
- **$project** – reshapes documents, computes new fields.
- **$group** – aggregates documents by key.
- **$sort** – sorts output, often paired with $limit.
- **$limit** – restricts number of results.
- **$unwind** – expands arrays into individual documents.
- **$lookup** – performs joins across collections.
- **$addFields** – adds computed fields.
- **$facet** – runs multiple pipelines in parallel.
- **$bucket / $bucketAuto** – histogram groupings.

AI and Data Science

- Handles large structured and semi-structured datasets.
- Enables efficient:
  - preprocessing,
  - statistical summarization,
  - feature engineering,
  - text transformations.
- Minimizes data-transfer overhead between MongoDB and Python.

Performance Tips

- Place $match as early as possible.
- Indexes affect $match and $sort before grouping.
- Use $facet for parallel aggregations.

Tasks

Attention: these tasks are to be submitted! A python notebook is available in the moodle course. Submit the notebook including your answers in moodle. **Compare your answare against answers provided by ChatGPT.**