
Aggregation Pipelines

The `find()` method on a collection is limited. It can't group or transform the documents.

There are also single purpose methods like `estimatedDocumentCount()`, `count()`, and `distinct()` which are appended to a `find()` query making them quick to use but limited in scope.

Up until version 5.0, you could also use the map-reduce framework on MongoDB. But it's deprecated and out of scope for this course.

> Starting in MongoDB 5.0, map-reduce is deprecated: Instead of map-reduce, you should use an aggregation pipeline. Aggregation pipelines provide better performance and usability than map-reduce. https://www.mongodb.com/docs/manual/core/map-reduce/

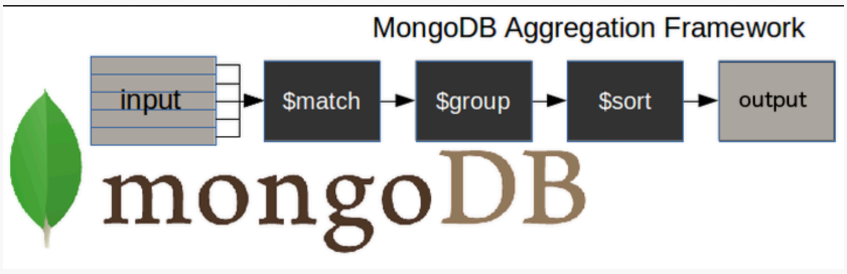
Aggregation pipelines are powerful.

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

- Each stage performs an operation on the input documents : (filtering, grouping, projecting,

transforming, ...)

- Each stage output is the input for the next stage.



from https://studio3t.com/knowledge-base/articles/mongodb-aggregation-framework/

Main stages:

- `$match` stage – filters those documents we need to work with, those that fit our needs
- `$group` stage – does the aggregation job
- `$sort` stage – sorts the resulting documents the way we require (ascending or descending)

A pipeline is executed with the `aggregate()` method and a list of JSON objects.

```Bash
db.collectionName.aggregate(pipeline, options),
```

where

```Bash
pipeline = [
        { $match : { … } },
        { $group : { … } },
        { $sort : { … } }
       ]
```

Memory constraints and outputs

Aggregation pipelines have 2 memory constraints :

1. Aggregation works in memory. Each stage can use up to 100 MB of RAM.
2. The documents returned by the aggregation query are limited to 16MB.

> The pipeline output cannot be larger than the maximum size of a MongoDB document.

But you can specify that you want the output of the pipeline as a **cursor** and not a document

For instance:

```JavaScript
db.movies.aggregate([{ $match: {} }])
```

This pipeline returns all the documents from the movies collection. If that may exceed the 16Mb limit, you should specify that you want to return a cursor

```JavaScript
db.movies.aggregate([
  { $match: {} }
], { cursor: { batchSize: 1000 }})
```

here `batchSize` controls how many documents MongoDB returns in each network round trip. It affects memory usage and network efficiency:

- Default is 101 documents
- Larger batch size = fewer round trips but more memory
- Smaller batch size = more round trips but less memory
- 1000 is a common balanced value

This doesn't affect the total number of documents returned, just how they're chunked during transmission.

create a new collection with $out

`$out` writes the results of an aggregation pipeline to a new or existing collection.

It must be the last stage in the pipeline.

for instance

```JavaScript
db.movies.aggregate([
  { $match: { "duration": { $gt: 120 } }},
  { $out: "longMovies" }  // Creates/overwrites longMovies collection
])
```

create a new field with $addFields

Use $addFields to create new fields

For instance, this pipeline adds a `heightCategory` field based on tree `height`

```JavaScript
db.movies.aggregate([
 { $addFields: {
   budgetCategory: {
     $switch: {
       branches: [
         { case: { $gte: ["$budget", 100000000] }, then: "blockbuster" },
         { case: { $gte: ["$budget", 50000000] }, then: "big-budget" },
         { case: { $gte: ["$budget", 10000000] }, then: "medium-budget" }
       ],
       default: "low-budget"
     }
   }
 }}
])
```

note the use of the `$switch` operator

Your turn: add a `profitMargin` field calculated as ((revenue - budget) / budget) * 100

You need to use the `$subtract`, `$divide` and `$multiply` operators, can't just use a direct subtraction, division or multiplication.

```JavaScript
{ $substract: [ <expression1>, <expression2> ] },
{ $divide: [ <expression1>, <expression2> ] },
{ $multiply: [ <expression1>, <expression2> ] },
```

```JavaScript
db.movies.aggregate([
 { $addFields: {
   profitMargin: {
       { $multiply: [ { $divide: [ { $subtract: ["$revenue", "$budget"] }, "$buc
   }
 }}
])
```

Check out documentation for all operators:
https://www.mongodb.com/docs/manual/reference/operator/aggregation/

$unwind : work on each element of an array

You cannot work directly on the elements of an array within a document with stages such as $group. The $unwind stage enables us to work with the values of the fields within an array.

For instance, unwind array of actors into separate documents: just list all the actors included in the `$cast` field

```JavaScript
db.movies.aggregate([
 { $match: { "cast": { $exists: true } }},
 { $unwind: "$cast" },
 { $project: {
   _id: 0,
   title: 1,
   actor: "$cast"
 }}
])
```

Your turn : Write a pipeline that creates a normalized `weightedScore` from IMDb and Metacritic ratings,

`weightedScore = (imdb.rating / 10) * (metacritic / 100)`

then uses it for filtering (weightedScore > 0.7) and sorting (desc).

```JavaScript
db.movies.aggregate([
  // Add a weighted score field combining IMDb and Metacritic
  { $addFields: {
      weightedScore: {
        $multiply: [
          { $divide: ["$imdb.rating", 10] },
          { $divide: ["$metacritic", 100] }
        ]
      }
  }},

  // Use the weighted score to filter
  { $match: {
      weightedScore: { $gt: 0.7 }
  }},

  // Sort by this new score
  { $sort: { weightedScore: -1 }}
])
```

grouping

Calculate the average `imdb.rating` per genre

This groups movies by their MPAA rating and calculates:

- Number of movies per rating (count)
- Average IMDb score for each rating (avgRating)

```JavaScript
db.movies.aggregate([
  { $group: {
      _id: "$rated",           // Group by rating (PG, R, etc.)
      count: { $sum: 1 },      // Count movies in each group
      cumulative_rating: { $sum: "$imdb.rating" },      // Count movies in each
      avgRating: { $avg: "$imdb.rating" }  // Average IMDb rating
  }}
])
```

Note the `{$sum : 1}` to count movies in each group.

you can also use : `$count`

```JavaScript
db.movies.aggregate([
  { $group: {
      _id: "$rated",
      movieCount: { $count: {} },     // Count movies in each group
      avgRating: { $avg: "$imdb.rating" }
  }}
])
```

$group aggregation operators

The $group stage supports certain expressions (operators) allowing users to perform arithmetic, array, boolean and other operations as part of the aggregation pipeline.

Operator	Meaning
$count	Calculates the quantity of documents in the given group.
$max	Displays the maximum value of a document's field in the collection.
$min	Displays the minimum value of a document's field in the collection.
$avg	Displays the average value of a document's field in the collection.
$sum	Sums up the specified values of all documents in the collection.
$push	Adds extra values into the array of the resulting document.

$push

`$push` creates an array field that collects all values from the grouped documents. It's like building a list of values from multiple documents into a single array.

for instance

```JavaScript
{ $push: "$title" }  // Creates array of all titles in the group
```

or

```javascript
db.movies.aggregate([
 // Group movies by director
 { $group: {
     _id: "$directors",
     directorMovies: {
         $push: {    // Create array of movies for each director
             title: "$title",
             year: "$year",
             rating: "$imdb.rating"
         }
     }
 }},
 // Filter directors with 3+ movies
 { $match: {
     "directorMovies.3": { $exists: true }
 }},
 // Sort by number of movies
 { $sort: {
     "directorMovies": -1
 }}
])
```

your turn : - use $push to list MPAA rating per genre

```javascript
db.movies.aggregate([
 { $unwind: "$genres" },
 { $group: {
     _id: "$genres",
     ratings: { $addToSet: "$rated" }
 }},
 { $sort: { _id: 1 }}
])
```

Difference with using `{ $addToSet: "$rated" }` instead of `{ $push: "$rated" }` ?

- now we want to count MPAA rating per genre

```javascript
????
```

mix unwind, group, addField

You turn: use $addField and $unwind to create the following pipeline

- Splits movies by genre
- Calculates average rating per genre
- Creates a structured field with genre info

- Filters genres with high average ratings (>7.5)
- Sorts results by rating

```JavaScript
db.movies.aggregate([
  // First stage: Unwind genres array
  { $unwind: "$genres" },

  // Calculate average rating per genre
  { $group: {
      _id: "$genres",
      avgRating: { $avg: "$imdb.rating" }
  }},

  // Add a field to store this info
  { $addFields: {
      genreInfo: {
        genre: "$_id",
        averageRating: "$avgRating"
      }
  }},

  // Use this field in the next stage
  { $match: {
      "genreInfo.averageRating": { $gt: 7.5 }
  }},

  // Sort by average rating
  { $sort: { "genreInfo.averageRating": -1 }}
])
```

$facet for parallel execution

`$facet` lets you run multiple aggregation pipelines in **parallel** and combine their outputs into a single result document. Each pipeline runs independently on the same input documents.

```JavaScript
db.collection.aggregate([
 // Optional: Initial stages before facet
 { $match: ... },

 { $facet: {
   "pipeline1": [
     // Stages for first parallel pipeline
   ],
   "pipeline2": [
     // Stages for second parallel pipeline
   ],
   // More named pipelines as needed
 }},

 // Optional: Post-facet stages to process combined results
 { $project: ... }
])
```

Example

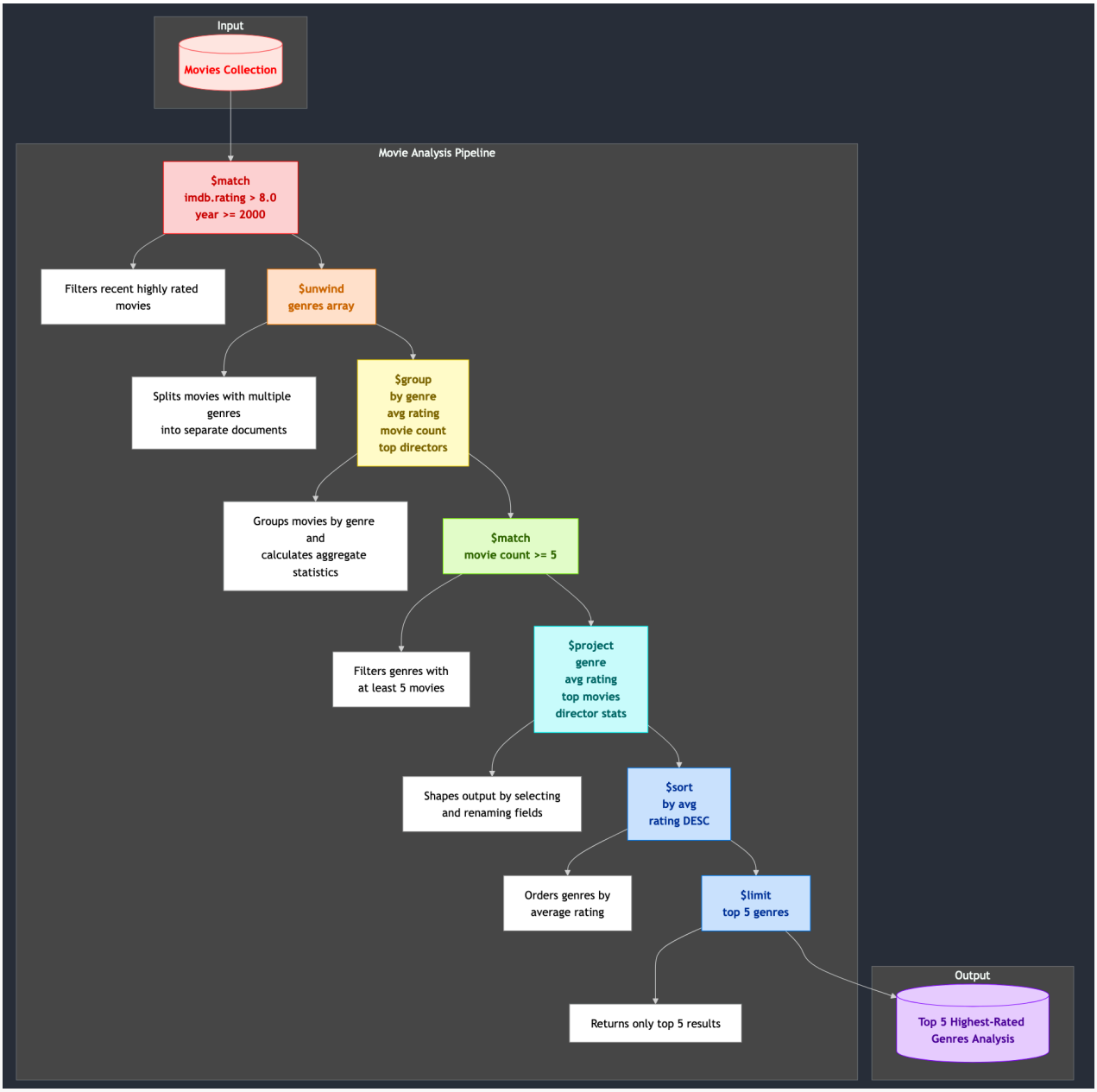
This is a find query

```JavaScript
db.movies.find(
    { imdb.rating: { $gt: 8.0 } },
    { _id: 0, title: 1, imdb.rating: 1, released: 1 }
).sort(
    { released: -1 }
).limit(
    2
)
```

This is the equivalent pipeline

```javascript
cursor = db.movies.aggregate([
    {
        $match: { imdb.rating: { $gt: 8.0 } }
    },
    {
        $project: {
            _id: 0,
            title : 1,
            imdb.rating: 1,
            released: 1
        }
    },
    {
        $sort: { released: -1 }
    },
    {
        $limit: 2
    }
])
```

A more complex aggregation pipeline



which corresponds to the pipeline

```Bash
db.movies.aggregate([
    {
        "$match": {
            "imdb.rating": { "$gt": 8.0 },
            "year": { "$gte": 2000 }
        }
    },
    {
        "$unwind": "$genres"
    },
    {
        "$group": {
            "_id": "$genres",
            "averageRating": { "$avg": "$imdb.rating" },
            "movieCount": { "$sum": 1 },
            "topDirectors": {
                "$push": {
                    "director": "$directors",
                    "movie": "$title",
                    "rating": "$imdb.rating"
                }
            }
        }
    },
    {
        "$match": {
            "movieCount": { "$gte": 5 }
        }
    },
    {
        "$project": {
            "_id": 0,
            "genre": "$_id",
            "averageRating": 1,
            "movieCount": 1,
            "topMovies": { "$slice": ["$topDirectors", 3] }
        }
    },
    {
        "$sort": {
            "averageRating": -1
        }
    },
    {
        "$limit": 5
    }
])
```

Joins and $lookups

In MongoDB, joins over collections are obtained with `$lookup`.

Here's an example of joining the movies with their comments using a `$lookup` stage.

```Bash
db.movies.aggregate([
    {
        $lookup: {
            from: "comments",           // The collection to join
            localField: "_id",          // The field in the `movies` collection
            foreignField: "movie_id",   // The field in the `comments` collectic
            as: "movie_comments"        // Name of the resulting array
        }
    },
    {
        $match: {                      // Filter the movies to include only thos
            "imdb.rating": { $gte: 8 }
        }
    },
    {
        $project: {                    // Project only the fields of interest
            _id: 0,
            title: 1,
            "imdb.rating": 1,
            movie_comments: 1
        }
    }
])
```

If we want to

- only return movies with some comments
- and only return a maximum of 2 comments per movies

```javascript
db.movies.aggregate([
    {
        $lookup: {
            from: "comments",           // The collection to join
            localField: "_id",          // The field in the movies collection
            foreignField: "movie_id",   // The field in the comments collection
            as: "movie_comments"        // The resulting array field
        }
    },
    {
        $addFields: {
            movie_comments: { $slice: ["$movie_comments", 2] }  // Limit comment
        }
    },
    {
        $match: {                       // Only include movies with at least 1 co
            movie_comments: { $ne: [] }
        }
    },
    {
        $project: {                     // Project only the fields of interest
            _id: 0,
            title: 1,
            movie_comments: 1
        }
    }
])
```

- $lookup:
  - Joins the movies collection with the comments collection based on `_id` in movies and `movie_id` in comments.
  - Adds an array field `movie_comments` containing all comments for each movie.
- $addFields:
  - Uses `$slice` to limit the `movie_comments` array to a maximum of 2 comments.
- $match:
  - Filters out movies with no comments by ensuring movie_comments is not an empty array ($ne: []).
- $project:
  - Specifies the fields to include in the final result, such as title and movie_comments.

Let's practice

And build the pipeline for the request :

> Find all movies with their comments and commenter details, showing only movies that have at least one comment, sorted by number of comments. and add the number of comments as a new field

Start with

- get the movie title, year and comments
- limit to 3 movies

```Bash
db.movies.aggregate([
  {
    $lookup: {
      from: "comments",
      localField: "_id",
      foreignField: "movie_id",
      as: "movie_comments"
    }
  },
  {
    $project: {
      title: 1,
      year: 1,
      comments: "$movie_comments.text"
    }
  },
  {
    $limit: 3
  }
])
```

This returns movies with many empty comments

- at least one comment

add condition so that it returns movies with at least one comment (not empty arrays)

```Bash
db.movies.aggregate([
  {
    $lookup: {
      from: "comments",
      localField: "_id",
      foreignField: "movie_id",
      as: "movie_comments"
    }
  },
  {
    $project: {
      title: 1,
      year: 1,
      comments: "$movie_comments.text"
    }
  },
  {
    $match : {
        comments : { $ne: [] }
    }
  },
  {
    $limit: 3
  }
])
```

- add number of comments as new field

```bash
db.movies.aggregate([
  {
    $lookup: {
      from: "comments",
      localField: "_id",
      foreignField: "movie_id",
      as: "movie_comments"
    }
  },
  {
    $project: {
      title: 1,
      year: 1,
      comments: "$movie_comments.text"
    }
  },
  {
    $match : {
        comments : { $ne: [] }
    }
  },
  {
    $limit: 3
  }
])
```

Links

- aggregation pipeline tutorial

see also

- aggregation pipeline https://www.mongodb.com/docs/manual/core/aggregation-pipeline/
- aggregation pipeline https://www.mongodb.com/resources/products/capabilities/aggregation-pipeline
- operators https://www.mongodb.com/docs/manual/reference/operator/query/gt/
- cursors https://www.mongodb.com/docs/manual/reference/method/js-cursor/
- for CRUD in python
  - https://learn.mongodb.com/courses/mongodb-aggregation-in-python
  - https://learn.mongodb.com/learn/course/mongodb-crud-operations-in-python/lesson-3-querying-a-mongodb-collection-in-python-applications/learn?client=customer&page=2
  - https://www.mongodb.com/docs/languages/python/pymongo-driver/current/aggregation/aggregation-tutorials/
  - https://learn.mongodb.com/courses/mongodb-crud-operations-in-python

- methods for pymongo: https://pymongo.readthedocs.io/en/stable/api/pymongo/cursor.html

---

Bit of practice

Let's calculate average IMDb rating and Count total movies for each year.

We define the aggregation pipeline

```python
pipeline = [
    {
        "$group": {
            "_id": "$year",  # Group by the "year" field
            "average_imdb_rating": {"$avg": "$imdb.rating"},  # Calculate averag
            "total_movies": {"$sum": 1}  # Count total movies for each year
        }
    },
    {
        "$sort": {"_id": 1}  # Sort by year in ascending order
    }
]
```

Then execute it the aggregation pipeline

```python
cursor = db.movies.aggregate(pipeline)
for doc in cursor:
    print(doc)
```

we see that we have weird values for years. Strings values with an extra `é`!

So let's find all the weird years. We can use regex

```python
cursor = db.movies.find({"year": {"$regex": "è"}})
```

but we could also check the data type of the field

```python
pipeline = [
    {"$group": {"_id": {"type": {"$type": "$year"}}, "count": {"$sum": 1}}}
]
```

Which returns

```bash
{'_id': {'type': 'string'}, 'count': 35}
{'_id': {'type': 'int'}, 'count': 21314}
```

Which shows that in MongoDB we can mix data types!

If we want to avoid counting the years that are not ints, we can add a $match clause. The initial pipeline becomes

```python
pipeline = [
    {
        "$match": {  # Ensure "year" is an integer
            "year": { "$type": "int" }
        }
    },
    {
        "$group": {
            "_id": "$year",  # Group by the "year" field
            "average_imdb_rating": {"$avg": "$imdb.rating"},  # Calculate averag
            "total_movies": {"$sum": 1}  # Count total movies for each year
        }
    },
    {
        "$sort": {"_id": 1}  # Sort by year in ascending order
    }
]
```

It looks like old movies are better than recent ones