Aggregate Syntax

The **`aggregate`** command operates on a collection like the other **Create, Read, Update, Delete** (**CRUD**) commands, like so:

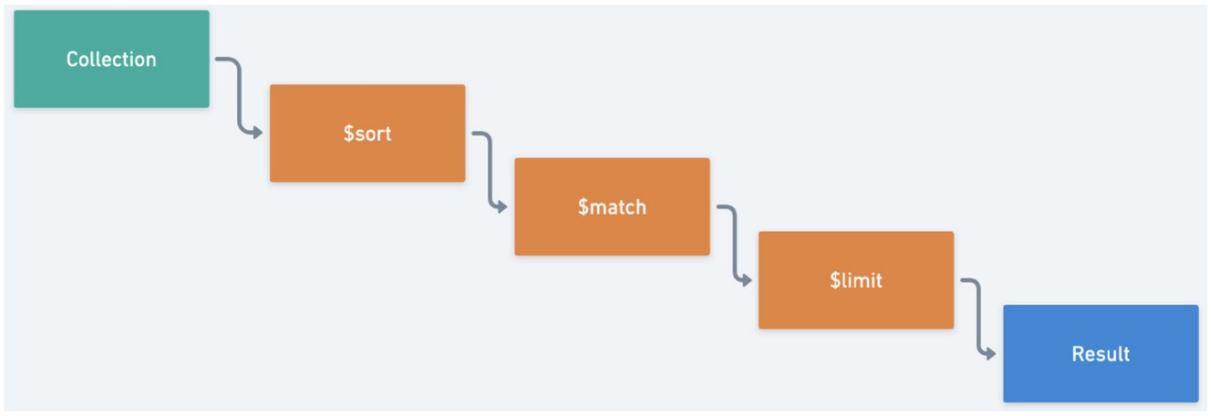
```
use sample_mflix;
var pipeline = [] // The pipeline is an array of stages.
var options = {} // We will explore the options later in the chapter.
var cursor = db.movies.aggregate(pipeline, options);
```

There are two parameters used for aggregation. The **`pipeline`** parameter contains all the logic to find, sort, project, limit, transform, and aggregate our data. The **`pipeline`** parameter itself is passed in as an array of JSON documents. You can think of this as a series of instructions to be sent to the database, and then the resulting data after the final stage is stored in a **`cursor`** to be returned to you. Each stage in the pipeline is completed independently, one after another, until none are remaining. The input to the first stage is the collection (**`movies`** in the preceding example), and the input into each subsequent stage is the output from the previous stage.
The second parameter is the **`options`** parameter. This is optional and allows you to specify the details of the configuration, such as how the aggregation should execute or some flags that are required during debugging and building your pipelines.

we create a file called **`aggregation.js`** with the following content:

```
var MyAggregation_A = function() {
   print("Running Aggregation Script Ch7.1");
   var pipeline = [];
     // This next line stores our result in a cursor.
   var cursor = db.movies.aggregate(pipeline);
     // This line will print the next iteration of our cursor.
   printjson(cursor.next())
};
MyAggregation_A();
```

pipeline is referred to as an aggregation stage:



pipeline can be passed as a variable:

```
var pipeline = [
        { $match: { "location.address.state": "MN"} },
        { $project: { "location.address.city": 1 } },
        { $sort: { "location.address.city": 1 } },
        { $limit: 3 }
    ];
```

Creating Aggregations

1. Match the theater collection to get a list of all theaters in the state of `MN` (Minnesota).
2. Project only the city in which the theater is located.
3. Sort the list by `city` name.
4. Limit the result to the first `three` theaters.

```
var simpleFindAsAggregate = function() {
    // Aggregation using match, project, sort and limit.
    print ("Aggregation Result:")
    var pipeline = [
        { $match: { "location.address.state": "MN"} },
        { $project: { "location.address.city": 1 } },
        { $sort: { "location.address.city": 1 } },
        { $limit: 3 }
    ];
    db.theaters.aggregate(pipeline).forEach(printjson);
};
simpleFindAsAggregate();
```

Exercise 7.01: Performing Simple Aggregations

"Return the top three movies in the romance genre sorted by IMDb rating, and return only movies released before 2001."

1. Translate your query into sequential stages that you can map to your aggregation stages: limit to three movies, match only romance movies, sort by IMDb rating, and match only movies released before 2001.
2. Simplify your stages wherever possible by merging duplicate stages. In this case, you can merge the two match stages: limit to three movies, sort by IMDb rating, and match romance movies released before 2001.

   It's important to remember that the order of the stages is essential and will produce incorrect results unless we rearrange them. To demonstrate this in action, we'll leave them in the incorrect order for now.

```
var pipeline = [
    { $sort: {"imdb.rating": -1}}, // Sort by IMDB rating.
    { $match: {
    genres: {$in: ["Romance"]}, // Romance movies only.
    released: {$lte: new ISODate("2001-01-01T00:00:00Z") }}},
    { $limit: 3 }, // Limit to 3 results.
    { $project: { genres: 1, released: 1, "imdb.rating": 1}}
];
db.movies.aggregate(pipeline).forEach(printjson);
```

Exercise 7.02: Aggregation Structure

"Can I reduce the number of documents being passed down the pipeline?" We will try to answer this question. **Switching “Match” and “Sort”**

```
var findTopRomanceMovies = function() {
    print("Finding top Classic Romance Movies...");
    var pipeline = [
        { $match: {
            genres: {$in: ["Romance"]}, // Romance movies only.
            released: {$lte: new ISODate("2001-01-01T00:00: 00Z") }}},
        { $sort: {"imdb.rating": -1}}, // Sort by IMDB rating.
        { $limit: 3 }, // Limit to 3 results.
        { $project: { title: 1, genres: 1, released: 1, "imdb.rating": 1}}
    ];
    db.movies.aggregate(pipeline).forEach(printjson);
}
findTopRomanceMovies();
```

Manipulating Data

The Group Stage

Previously, the most significant unit of data we could return was a single document. We can sort these documents to gain insight through a direct comparison of the documents. However, once we master the $group stage, we will be able to increase the scope of our queries to an entire collection by aggregating our documents into large logical units. Once we have the larger groups, we can apply our filters, sorts, limits, and projections just as we did on a per-document basis.

## List distinct ratings

```
    var pipeline = [
     {$group: {
        _id: "$rated"
     }}
    ];
    db.movies.aggregate(pipeline).forEach(printjson);
```

Accumulator Expressions

The $group command can accept more than just one argument. It can also accept any number of additional arguments in the following format:

```
field: { accumulator: expression},
```

```
var pipeline = [
 {$group: {
    _id: "$rated",
    "numTitles": { $sum: 1},
 }}
];
db.movies.aggregate(pipeline).forEach(printjson);
```

```
var pipeline = [
 {$group: {
    _id: "$rated",
    "sumRuntime": { $sum: "$runtime"},
 }}
];
db.movies.aggregate(pipeline).forEach(printjson);
```

```
var pipeline = [
 {$group: {
   _id: "$rated",
   "avgRuntime": { $avg: "$runtime"},
 }}
];
db.movies.aggregate(pipeline).forEach(printjson);
```

```
  var pipeline = [
 {$group: {
    _id: "$rated",
    "avgRuntime": { $avg: "$runtime"},
 }},
 {$project: {
    "roundedAvgRuntime": { $trunc: "$avgRuntime"}
 }}
];
db.movies.aggregate(pipeline).forEach(printjson);
```