Aggregation in MongoDB

Overview

MongoDB as a powerful aggregation and pipelining framework that allows for multple operations to take place over the objects in the collection in a form of a pipeline. The operations that can be performed are:

- Filtering (or matching): selecting only the documents that match a certain condition;
- Projection: selecting only portions of the documents, and performing other "document shape" transformations on them.
- Grouping and aggregation: grouping documents together by specified values, and aggregating content by each group.
- Unwinding: turning array values in a document into standalone objects.
- Sorting: reordering the order of the objects.
- Limiting and skipping: controlling which parts of the transformed collections are returned.

Pipelining

Pipelining and aggregation in MongoDB is being spearheaded by the

```
db.<collection>.aggregate(<Pipeline>)
```

command. The syntax of the `<Pipeline>` argument is:

```
<Pipeline>::=  {"$Aggregator1": <AggValue1>},...,{"$AggregatorK": <AggValueK>}
```

Here, "$Aggregator1",..., "$AggregatorK" are aggregation pipeline keywords specifying one of the operations listed above, while <AggValue1>,..., <AggValueK> are the values (usually documents, but sometimes simple values) that guide the aggregation process (parameters of each aggregation operation).

Semantics. The <Pipeline> describes the sequence of operations to be performed on the documents from a given collection.

Some types of operations: filtering, projection, unwinding, limiting and skipping can be performed one document at a time, and execution of these operations forms an actual pipeline.

Other types of operations: grouping and sorting, can be performed only after all documents in the collection have gone through the previous operation in the pipeline.

The syntax and semantics of each operation are described below.

Operation keywords. The "$Aggregator" keywords in the definition above come from the following list:

Keyword	Operation
"$match"	matching/filtering
"$project"	projection
"$group"	grouping and subsequent aggregation
"$unwind"	unwinding arrays
"$sort"	sorting/reordering
"$limit"	limiting the output to top results
"$skip"	skipping the top results

Matching/Filtering

Syntax.

```
{"$match": <QueryDoc>}
```

Here, <QueryDoc> has the same syntax as the <QueryDoc> parameter of the db.<collection>.find(<QueryDoc>) command.

Semantics. The result of the

```
db.collection.aggregate({"\$match": <QueryDoc>})
```

matches the result returned by the db.<collection>.find(<QueryDoc>) command.

Projection

Syntax.

```
{"$project": <ProjectDoc>}
```

Here, `<ProjectDoc>` is the projection document specifying how projection is made.

Projection Document syntax. A simple projection document has the form

```
{"<key1>": 1, ... , "<keyN>":1}
```

or

```
{"<key1>": 1, ... , "<keyN>":1, "_id":0}
```

The meaning of these three formats is the same as when they are used as the second argument to `db.<collection>.find()`

Examples. Consider the following collection of documents:

```
> db.ex3.find()
{ "_id" : 1, "name" : "Joe", "courses" : [ 365, 369, 466 ] }
{ "_id" : 2, "name" : "Paula", "courses" : [ 471, 365 ] }
{ "_id" : 3, "name" : "Vanda", "courses" : 365 }
{ "_id" : 4, "name" : "Morris", "courses" : [ 102, 103, 225, 357 ] }
{ "_id" : 5, "name" : "Phil", "courses" : [ 365 ] }
```

We can experiment with simple projection expressions as follows:

```
> db.ex3.aggregate({"$project": { "name":1}})
{ "_id" : 1, "name" : "Joe" }
{ "_id" : 2, "name" : "Paula" }
{ "_id" : 3, "name" : "Vanda" }
{ "_id" : 4, "name" : "Morris" }
{ "_id" : 5, "name" : "Phil" }
> db.ex3.aggregate({"$project": { "name": 1, "_id":0}})
{ "name" : "Joe" }
{ "name" : "Paula" }
{ "name" : "Vanda" }
{ "name" : "Morris" }
{ "name" : "Phil" }
```

Attribute Renaming. We can rename attributes by including the following key-value pair into the projection instructions:

```
{"<newNAme>": "$<oldName>"}
```

For example, we can rename the "name" attribute to "student"

```
> db.ex3.aggregate({"$project": { "student": "$name"}})
{ "_id" : 1, "student" : "Joe" }
{ "_id" : 2, "student" : "Paula" }
{ "_id" : 3, "student" : "Vanda" }
{ "_id" : 4, "student" : "Morris" }
{ "_id" : 5, "student" : "Phil" }

> db.ex3.aggregate({"$project": { "student": "$name", "name":1}})
{ "_id" : 1, "name" : "Joe", "student" : "Joe" }
{ "_id" : 2, "name" : "Paula", "student" : "Paula" }
{ "_id" : 3, "name" : "Vanda", "student" : "Vanda" }
{ "_id" : 4, "name" : "Morris", "student" : "Morris" }
{ "_id" : 5, "name" : "Phil", "student" : "Phil" }
```

Note: the second command allows you yo duplicate values under different keys.

Arithmetics. You can do simple arithmetics on the values of a document, and transform the results. There are five arithmetical operators available:

Operation	Experssion
Addition	"$add": [<expr1> [, <expr2>, ...<exprK>]]
Subtraction	"$subtract": [<expr1>, <expr2>]
Multiplication	"$multiply": [<expr1> [, <expr2>, ...<exprK>]]
Division	"$divide": [<expr1>, <expr2>]
Modulo Division	"$mod": [<expr1>, <expr2>]

Date expressions. You can use the following keywords to extract/transform date expressions/values that are a part of your documents:

Keyword	Extracts
"$year"	year of the date
"$month"	month of the date
"$week"	week (as a number) of the date
"$dayOfMonth"	day of month of the date
"$dayOfWeek"	day of week (as a number) of the date
"$dayOfYear"	day of year of the date
"$hour"	hour of the date-time expression
"$minute"	minute of the date-time expression
"$second"	second of the date-time expression

String expressions. The following expressions work on strings.

- "$substr": [<expr>, <start>, <nChars>]. This expression takes the string value <expr> and returns <nChars> characters starting at position <start>.
- "$concat": [<string1>,...,<stringN>]. Concatenation of all string arguments.

- `"$toLower": <string>`. Converts `<string>` to only contain lowercase characters.

- `"$toUpper": <string>`. Converts `<string>` to only contain uppercase characters.

Control. There are two conditional expressions that can be used for branching the evaluation.

- `"$cond": [<booleanExp>, <trueExp>, <falseExp>]`. If `<booleanExp>` evaluates to true, return the result of `<trueExp>`, otherwise, return the result of `<falseExp>`.

- `"$ifNull": [<expr>, <replacement>]`. If `<expr>` evaluates to `null`, return the result of `<replacement>`, otherwise returns the result of `<expr>`.

Comparisons. Complex comparisons can be placed inside control expressions and elsewhere:

- `"$cmp": [<expr1>, <expr2>]`. Returns 0 is the two expressions evaluate to the same value. Returns a negative number if first expression is less than second, and a positive number if first expression is greater than second.

- `"$strcasecmp": [<string1>, <string2>]`. Works like `"$cmp"` but ignores case for letters of Roman alphabet.

- **Comparisons**: `"$eq", "$ne", "$gt", "$gte", "$lt", "$lte": [<exp1>, <exp2>]`. Standard true false comparisons (equals, not equals, greater than, greater than or equal, less than, less than or equal respectively).

- `"$and"`, `"$or"` and `"$not"`. These work the same way as in the contents of the query documents in `find()` command.

Examples. Some examples of projection.

Working with the following data collection:

```
> db.ex7.find()
{ "_id" : 1, "state" : "CA", "class" : 10, "score" : 70 }
{ "_id" : 2, "state" : "CA", "class" : 3, "score" : 88 }
{ "_id" : 3, "state" : "CA", "class" : 5, "score" : 92 }
{ "_id" : 4, "state" : "CA", "class" : 6, "score" : 64 }
{ "_id" : 5, "state" : "CA", "class" : 3, "score" : 77 }
{ "_id" : 6, "state" : "CA", "class" : 5, "score" : 94 }
{ "_id" : 7, "state" : "NV", "class" : 5, "score" : 94 }
{ "_id" : 8, "state" : "NV", "class" : 7, "score" : 100 }
{ "_id" : 9, "state" : "NV", "class" : 10, "score" : 45 }
{ "_id" : 10, "state" : "NV", "class" : 10, "score" : 85 }
```

The query below scales each score according to the formula:

$$scaledScore = class + score * \left(1 + \frac{class}{10}\right)$$

(the query is pretty printed for your benefit)

```
> db.ex7.aggregate({"$project": {"state":1, "class":1,
                   "ScaledScore": {"$sum": ["$class",
                                    {$multiply: ["$score",
                                       {$sum: [1, {$divide: ["$class", 10]}
                                        ]}
                                     ]}
                                  ]}
                  }})
{ "_id" : 1, "state" : "CA", "class" : 10, "ScaledScore" : 150 }
{ "_id" : 2, "state" : "CA", "class" : 3, "ScaledScore" : 117.4 }
{ "_id" : 3, "state" : "CA", "class" : 5, "ScaledScore" : 143 }
{ "_id" : 4, "state" : "CA", "class" : 6, "ScaledScore" : 108.4 }
{ "_id" : 5, "state" : "CA", "class" : 3, "ScaledScore" : 103.10000000000001 }
{ "_id" : 6, "state" : "CA", "class" : 5, "ScaledScore" : 146 }
{ "_id" : 7, "state" : "NV", "class" : 5, "ScaledScore" : 146 }
{ "_id" : 8, "state" : "NV", "class" : 7, "ScaledScore" : 177 }
{ "_id" : 9, "state" : "NV", "class" : 10, "ScaledScore" : 100 }
{ "_id" : 10, "state" : "NV", "class" : 10, "ScaledScore" : 180 }
```

The following statement converts the name of the state to "CALIFORNIA" for California records and "not CALIFORNIA" for all other records.

```
> db.ex7.aggregate({"$project": {"class": 1, "score":1,
                   "state": {$cond: [{$eq: ["$state", "CA"]},
                                    {$concat: ["$state", "LIFORNIA"]},
                                    "not CALIFORNIA"
                                   ]}
                   }})
{ "_id" : 1, "state" : "CALIFORNIA", "class" : 10, "score" : 70 }
{ "_id" : 2, "state" : "CALIFORNIA", "class" : 3, "score" : 88 }
{ "_id" : 3, "state" : "CALIFORNIA", "class" : 5, "score" : 92 }
{ "_id" : 4, "state" : "CALIFORNIA", "class" : 6, "score" : 64 }
{ "_id" : 5, "state" : "CALIFORNIA", "class" : 3, "score" : 77 }
{ "_id" : 6, "state" : "CALIFORNIA", "class" : 5, "score" : 94 }
{ "_id" : 7, "state" : "not CALIFORNIA", "class" : 5, "score" : 94 }
{ "_id" : 8, "state" : "not CALIFORNIA", "class" : 7, "score" : 100 }
{ "_id" : 9, "state" : "not CALIFORNIA", "class" : 10, "score" : 45 }
{ "_id" : 10, "state" : "not CALIFORNIA", "class" : 10, "score" : 85 }
```

Grouping and Aggregation

Syntax.

```
{"$group": {<groupAttributes>, <aggregations>}}
```

Here, `<groupAttributes>` is the syntax for specifying which attributes to group by, and `<aggregations>` sets up the various aggregation operations on the group.

<groupAttributes> **syntax.**

```
{"_id": "$<key>"}
```

for a single key grouping.

```
{"_id": {"newKey1":"$<key1>, ..., "newKeyK":"$<keyK>"}}
```

for a multiple key grouping.

For example, consider a set of documents in the following format:

```
{
 "_id": <id>,
 "state": <State>,
 "age": <Age>,
 "score": <Score>,
}
```

Consider the following aggregations run against a collection `exams` of the documents with this format:

```
db.exams.aggregate({"$group": {"_id":"$state"}})
db.exams.aggregate({"$group": {"_id": "$age"}})
db.exams.aggregate({"$group": {"_id": {"state": "$state", "howOld":"$age"}}})
```

The first command groups all the documents by state. The second does the same by age. The third column creates groups based on `state` and `age` pairs, and renames the `"age"` attribute to `"howOld"`.

A simple grouping operation like the ones above simply create one record (with a unique `"_id"` value) per every unique combination of values of the grouping keys. No other information is passed into the new documents however. Aggregation operations allow one to add more data to the output documents.

Aggragation operations. MongoDB supports the following aggreagation operations inside the `"$group"` command.

Keyword	Operation
`"$sum": "$<key>"`	sum of all values in `<key>`
`"$avg": "$<key>"`	average of all values in `<key>`
`"$max": <expr>`	largest computed value of the expression
`"$min": <expr>`	smallest computed value of the expression
`"$first": <expr>`	first computed value of the expression
`"$last": <expr>`	last computed value of the expression
`"$addToSet": <expr>`	collects a set of values
`"$push": <expr>`	collects a list of values

Examples.

1. To find the sum of all scores for each state:

```
db.exams.aggregate({"$group": {"_id":"$state", "total": {"$sum": "$score"}}})
```

2. To find the average of all scores for each state:

```
db.exams.aggregate({"$group": {"_id":"$state", "avg": {"$avg": "$score"}}})
```

3. To find the largest and the smallest score of each age:

```
db.exams.aggregate({"$group": {"_id":"$age", "Best": {"$max": "$score"}, "Worst":{"$min": "$score"}}})
```

4. To find the age of the person with the best score in each state:

```
db.exams.aggregate({"$sort": {"score": -1}},
                   {"$group": {"_id":"$state", "bestAge":{"$first": "$age"}}}})
```

or

```
db.exams.aggregate({"$sort": {"score": 1}},
                   {"$group": {"_id":"$state", "bestAge":{"$last": "$age"}}}})
```

5. To find all scores seen in each state:

```
db.exams.aggregate({"$group": {"_id":"$state", "scores": {"$addToSet": "$score"}}})
```

6. To find all scores (for every person who took the exam) by state:

```
db.exams.aggregate({"$group": {"_id":"$state", "scores": {"$push": "$score"}}})
```

Unwinding

Syntax.

```
{"$unwind": "$<key>"}
```

Here, `"$<key>"` is the name of the key whose values, if they are arrays will be "unwound".

Semantics. The unwinding of a JSON document

```
key1: value1,
...
keyN: valueN,
key: [v1, v2,..., vK]
```

on key `key` is defined as a collection of JSON objects:

```
{key1: value1,  ..., keyN: valueN, key: v1}
{key1: value1,  ..., keyN: valueN, key: v2}
...
{key1: value1,  ..., keyN: valueN, key: vK}
```

The `"$unwind": "$<key>"` operation replaces every document in the collection which has an array as a value of `<key>` with its *unwinding*. Other objects in the collections are preserved as-is.

Sorting

Syntax. The sorting operation is expressed as follows

```
{"$sort": <SortDocument>}
```

Here, `<SortDocument>` has the same format at the sort document used in `db.<collection>.find().sort(<SortDocument>)` command.

Semantics. As a result of this operation in the pipeline, the incoming collection of objects is sorted according to the instructions contained in `<SortDocument>` document.

Limiting the output

Syntax. The limiting operation is expressed as follows

```
{"$limit": <Number>}
```

Here, `<Number>` is a number of objects to retutrn.

Semantics. As a result of this operation, only the first `<Number>` of documents from the collection will be returned/passed to the next pipeline operation.

Skipping top results

Syntax. The skipping operation is expressed as follows

```
{"$skip": <Number>}
```

Here, `<Number>` is a number of objects to skip.

Semantics. As a result of this operation, the first `<Number>` of documents from the incoming collection will be skipped, all other objects will be returned/passed to the next pipeline operation.