Aggregation in MongoDB: Additional Operations

Additional Aggregation Pipeline Operators

Faceted Filter

The `$facet` aggregation operation allows for a split of aggregation pipelines.

That is, `$facet` allows the user to provide a list of independent pipelines that can be run on a given document collection.

The `$facet` syntax is as follows:

```
{ $facet:
   {
      <outputField1>: [ <stage1>, <stage2>, ... ],
      <outputField2>: [ <stage1>, <stage2>, ... ],
      ...
      <outputFieldK>: [ <stage1>, <stage2>, ... ],

   }
}
```

The result of this operation is a **single** object that consists of fileds: `<outputField1>`,...`<outputFieldK>` each with the result of the aggregation pipeline specified as its input.

Note. This is useful when you want to compute a variety of aggregation operations based on different groupings, or to see different slices of the dataset at the same time.

Example. This is a very simplified example based on a small collection of student grades in two courses:

```
> db.grades.find()
{ "_id" : 1, "name" : "Bob", "class" : 365, "grade" : 88 }
```

```
{ "_id" : 2, "name" : "Bob", "class" : 453, "grade" : 92 }
{ "_id" : 3, "name" : "May", "class" : 365, "grade" : 76 }
{ "_id" : 4, "name" : "May", "class" : 453, "grade" : 90 }
{ "_id" : 5, "name" : "Chris", "class" : 365, "grade" : 93 }
{ "_id" : 6, "name" : "Chris", "class" : 453, "grade" : 74 }
```

Consider a scenario where you want to return back an object that contains the following three things:

1. List of all students with a grade of 85 or above in CSC 365.
2. Average scores for students in each of the classes (as an array).
3. Highest score in CSC 453.

Each of the three requests can be executed in isolation as an aggregation pipeline:

1. To find CSC 365 students with a score of 85 or higher:

   ```
   db.grades.aggregate({$match: {class:365, grade: {$gte: 85}}})
   ```

2. To find average scores in each class and format the output appropriately:

   ```
   db.grades.aggregate({$group: {_id: "$class", avgScore:{$avg: "$grade"}}},
                        {$group: {_id:1, scores:{$push: {class:"$_id", avgScore:"$avgScore"}}}},
                        {$project: {_id:0}})
   ```

3. To find the highest score in CSC 453:

   ```
   db.grades.aggregate({$group: {_id:"$class", maxGrade:{$max: "$grade"}}},
                       {$match: {_id: 453}},
                       {$project: {_id:0}} )
   ```

Using the `$facet` aggregation operation, we can now combine these three pipelines into as single step:

```
> db.grades.aggregate({$facet: {
                      csc365_hi: [{$match: {class:365, grade: {$gte: 85}}}],
                      averages: [{$group: {_id: "$class", avgScore:{$avg: "$grade"}}},
                                 {$group: {_id:1, scores:{$push: {class:"$_id", avgScore:"$avgScore"}}}},
                                 {$project: {_id:0}}
                                ],
                      max453: [{$group: {_id:"$class", maxGrade:{$max: "$grade"}}},
                               {$match: {_id: 453}},
                               {$project: {_id:0}}
                              ]
                         }}
                    ).pretty()

{
        "csc365_hi" : [
                {
```

```
                "_id" : 1,
                "name" : "Bob",
                "class" : 365,
                "grade" : 88
        },
        {
                "_id" : 5,
                "name" : "Chris",
                "class" : 365,
                "grade" : 93
        }
    ],
    "averages" : [
        {
                "scores" : [
                        {
                                "class" : 453,
                                "avgScore" : 85.33333333333333
                        },
                        {
                                "class" : 365,
                                "avgScore" : 85.66666666666667
                        }
                ]
        }
    ],
    "max453" : [
        {
                "maxGrade" : 92
        }
    ]
}
```

Left Join

The $lookup operation performs a left join of the current collection with the collection listed in the parameters of the $lookup on the conditions specified.

The syntax of the $lookup operation is

```
{
   $lookup:
     {
       from: <collection>,
       localField: <field1>,
       foreignField: <field2>,
       as: <arrayField>
     }
}
```

Here, <collection> specifies the collection with which to join. <field1> and <field2> complete the condition that will be checked. The condition is:

```
<coll>.<field1> = <collection>.<field2>
```

where `<coll>` is the collection on which the aggregation pipeline is run. `<arrayField>` stores the results.

This operation works as follows. For each object in the collection on which the aggregation pipeline is run, the collection `<collection>` is scanned, object-by-object. Any object from this collection that contains `<field2>` whose value is **exactly equal** to the value of the `<field1>` from the current object from the host collection is added into the `<arrayField>` as another array element.

The output object preserves all its original fields, and adds `<arrayField>` constructed as described above.

Example. In addition to the `grades` collection shown in the previous example, consider the following collection showing instructors of different classes:

```
> db.prof.find()
{ "_id" : 1, "course" : 365, "name" : "Alex" }
{ "_id" : 2, "course" : 369, "name" : "Alex" }
{ "_id" : 3, "course" : 453, "name" : "Phil" }
{ "_id" : 4, "course" : 307, "name" : "Davide" }
{ "_id" : 5, "course" : 466, "name" : "Foaad" }
{ "_id" : 6, "course" : 357, "name" : "Clint" }
```

The following `$lookup` operation adds the list of students taking each course to each object from the `profs` collection for which there are students in the `grades` collection.

```
> db.prof.aggregate({$lookup: {
...                             from: "grades",
...                             localField: "course",
...                             foreignField: "class",
...                             as: "roster"
...                            }
...                    })
{ "_id" : 1, "course" : 365, "name" : "Alex",
             "roster" : [ { "_id" : 1, "name" : "Bob", "class" : 365, "grade" : 88 },
                          { "_id" : 3, "name" : "May", "class" : 365, "grade" : 76 },
                          { "_id" : 5, "name" : "Chris", "class" : 365, "grade" : 93 } ] }
{ "_id" : 2, "course" : 369, "name" : "Alex", "roster" : [ ] }
{ "_id" : 3, "course" : 453, "name" : "Phil",
             "roster" : [ { "_id" : 2, "name" : "Bob", "class" : 453, "grade" : 92 },
                          { "_id" : 4, "name" : "May", "class" : 453, "grade" : 90 },
                          { "_id" : 6, "name" : "Chris", "class" : 453, "grade" : 74 } ] }
{ "_id" : 4, "course" : 307, "name" : "Davide", "roster" : [ ] }
{ "_id" : 5, "course" : 466, "name" : "Foaad", "roster" : [ ] }
{ "_id" : 6, "course" : 357, "name" : "Clint", "roster" : [ ] }
```

Note. `$lookup` performs, what is known as **left outer join**: the objects from the host collection that are not joined with any objects from the foreign collection are retained in the output with an empty array for the "join" key-value pair.

Bucketing

The bucketing operation is the version of grouping that uses a continuous variable to break the data into separate groups/buckets. Essentially, rather than combining the objects based on the same value of a specific key, bucketing combines them based on the value of a key falling in a specific range (bucket).

In MongoDB aggregation pipelines bucketing is performed using the `$bucket` operator. The syntax of a `$bucket` step is show below:

```
{
  $bucket: {
      groupBy: <expression>,
      boundaries: [ <lowerbound1>, <lowerbound2>, ... ],
      default: <literal>,
      output: {
         <output1>: { <$accumulator expression> },
         ...
         <outputN>: { <$accumulator expression> }
      }
   }
}
```

Here:

- `<expression>` value of the `groupBy` key is the value that will be bucketed. This is usually a referenece to a key storing the value, but it can also be any other numeric expression (see `$project` operation documentation for numeric expressions).

- The array of numeric values supplied for the `boundaries` key stores the breakdown of the space into buckets. Given the K values

  ```
  [ n1, n2, ..., nK]
  ```

  there will be $K-1$ buckets of the form: $[n1, n2), [n2, n3), \ldots, [n_{K-1}, nK]$. Each bucket $[ni, n_{i+1}]$ will be represented in the output by a single object with the key `_id: ni`. Values that are smaller than $n1$ or greater than nK fall outside of the bucket ranges...

- ...and are handled by the `default` key. Its value is a literal that will be used as the unique identifier of the default bucket, which is reserved for accumulating information about objects that do not fall into any of the other buckets.

- Finally, the `output` key describes how the output objects look like. The key-value pairs inside this key are formed the same way as all key-value pairs except for `_id` are formed in `$group` operation.

Example. Let's count how many grades in ranges 75-85, 85-95 and 95-100 are in the `grades` collection.

```
> db.grades.aggregate({$bucket: { groupBy:"$grade",
...                                boundaries: [75,85,95,100],
...                                default: "less than 75",
...                                output: {num: {$sum: 1}}
...                               }
...                       })
{ "_id" : 75, "num" : 1 }
{ "_id" : 85, "num" : 4 }
{ "_id" : "less than 75", "num" : 1 }
```