Aggregation in MongoDB: Part II New and Additional Operators

Overview of Aggregation Pipeline Operators

This handout describes the MongoDB Verision 4.2. aggregation pipeline.

In MongoDB Version 4.2. the following aggregation pipeline operators (also known as *stages* are defined. Note the tables below shows only the list of operators that actually facilitate work with data. There are several aggregation pipeline steps we do not include on the list, that provide MongoDB debugging information.

We break the list of the steps by the basic designation of the operator. Note, some operators show up in multiple lists - whenever they do, it's the same operator, which combines features of multiple atomic operations.

Selection/Filtering Stages. Selection/Filtering operations check each document *in turn* against a supplied condition. If the document does not match the condition, the document is filtered out (not passed to the next processing stage in the pipeline). If the document matches the condition, it is passed to the next stage. The *pure* selection operation either filters the document out, or passes the document in its original form. MongoDB now has a variant that can also modify the contents of the output document (as the projection operation does). The following MongoDB aggregation pipeline stages perform selection/filtering functions. We include here *any and all* types of stages that perform a filtering function, including filters such as `$limit` and `$skip`.

Operator	Explanation
`$match`	takes as input a match condition and outputs documents that match the condition
`$redact`	combines the functionality of `$match` and `$project`: can filter out documents, and reshape them
`$sample`	create a random sample of documents from the collection
`$limit`	output only the first n documents passed to the stage
`$skip`	skip the first n document passed to the stage, output all the remaining ones

Projection/Transformation Stages. On a Projection/Transformation stage, changes are made to the shape of the document: the keys and the values they have. The changes are made to each document one at a time, and only the information contained in the document itself is used in the operation. MongoDB has a projection operation that captures a wide range of behaviors, as well as some restricted versions that have simplified syntax, but provide better readability for certain simple types of data projection/transformation, such as additions of new keys, or removals of existing keys. Pure projection/transformation operations *do not filter* documents - each documeent in the collection is transformed and passed to the next stage of the aggregation pipeline.

Operator	Explanation
`$project`	transform input documents according to the specification given as the argument
`$redact`	combines the functionality of `$match` and `$project`: can filter out documents, and reshape them
`$addFields`	a restricted projection operation with simplified syntax to add new fields to the document
`$set`	alias for `$addFields`
`$unset`	a restricted projection operation with simplified syntax to remove fields from the documents
`$replaceRoot`	a replaces the root document with the contents of a document embedded into it
`$replaceWith`	alias for `$replaceRoot`

Grouping and Aggregation Stages. A grouping operation splits the document collection into a set of groups, based on a specified condition (which may be a value or a set of values for the documents in the same group to share). The output document collection will contain one document per group. With semi-structured data, grouping operations can be done by themselves in order to reshape the documents and collect information about an entire group (stored in multiple original documents). They can also be accompanied by aggregations - operations that compute an aggregate result – one value, rather than a document – from a set of values. Aggregation operations in MongoDB are *not* separate stages (except for `$count`) – rather, they are part of expressions defining the contents of the output documents

in several types of stages – mainly grouping and projection stages. Aggregations can be performed over contents of arrays, or over contents of multiple documents as they are being grouped.

MongoDB offers the following grouping stages:

Operator	Type	Explanation
`$group`	grouping	group (and if necessary aggregate) documents in the collection by a specific key
`$bucket`	grouping	groups documents into "buckets" based on the value of specified expression falling into speficied ranges
`$bucketAuto`	grouping	like `$bucket` only bucket boundaries are automatically identified
`$sortByCount`	grouping	a shorthand for a grouping operation followed by a sorting operation
`$count`	aggregation	returns the number of documents

Joins. A Join is an operation on data that combines together documents from different collections, typically based on a matching condition that compares the contents of the two documents. Relational DBMS are good at performing joins, but one of the major "sacrifices" made in design of distributed key-value stores (and later - distributed document databases) is lack of convenient built-in facilities for join operations. MongoDB has a very limited slate of join operations.

Operator	Explanation
`$lookup`	performs a *left outer join operation* based on key match
`$graphLookup`	performs a transitive closure join operation on graph-like data

Sort. Sorting operations change the order in which the documents in the collection are reported. MongoDB has one key sorting stage and one stage that incorporates sorting in it.

Operator	Explanation
`$sort`	sorts the input collection according to the provided sorting criteria
`$sortByCount`	a shorthand for a grouping operation followed by sorting

Unwind. Unwinding is unique operation to the algebra of data manipulation over semi-structured documents. This is a partial reverasl of a grouping operation (or simply an unrolling of any array value in a document) that replaces an object containing an array of values with a set of objects - one per array element, where individual array elements replace the full array as the value of the array key. Existence of this stage in the aggragation pipeline allows for a lot of powerful manipulations of data based on grouping (to combine certain aspects of data) followed by unwinding (to restore the data back to the original format).

Operator	Explanation
`$unwind`	unwinds an array value in the documents

Faceted Filter. The faceted filter stage does not perform any operations on the data by itself. Rather, it enables one aggregation pipeline to have *several diferent independent* sequences of stages performed in parallel on the data. Faceted filter "splits" the aggregation pipeline into a given number of sub-pipelines, and combines in a signle output document the results of all the pipelines.

Operator	Explanation
`$facet`	splits (facets) the aggregation pipeline into multiple pipelines combines results in a single document

Output. Technically not part of the data manipulation algebra, these operations are nevertheless important, as they allow to persist of the results of an aggragation pipeline in a MongoDB collection.

Operator	Explanation
`$out`	save the results of an operation to a new collection (old)
`$merge`	save the results of an operation to an existing or new collection (new)

Details of Additional Stages

In what follows we discuss most of the stages not covered in the first handout, but included in the tables above.

Adding Projection Expressions to $match

One of the biggest original limitations of `db.collection.find()` and the `$match` aggregation stage is their inability to properly compare the values of two fields/keys/attributes in a document. This is due to the fact that the expressions in the `db.collection.find()` have traditionally had **different syntax** than the expressions used in the `$project` and `$group` stages of the aggregation pipeline. This required separate parsing mechanisms that were incompatible.

In **Version 3.6.** MongoDB developers took steps to somewhat rectify this issue by introducing a new expression construct

```
{$expr: <expression>}
```

Here, `<expression>` is any expression that can be found on the right-hand-side of a key-value pair in a `$project` aggregation stage. This can be leveraged to run comparisons between different parts of a document in the `$match` stage.

Consider the following simple collection:

```
csc369Replicas:PRIMARY> db.ee.find()
```

```
{ "_id" : 1, "x" : 1, "y" : 2 }
{ "_id" : 2, "x" : 2, "y" : null }
{ "_id" : 3, "x" : 5 }
{ "_id" : 4, "x" : 6, "y" : [ 2, 3, 3 ] }
```

Let us find all documents where `x` is greater than the `_id`. We can now do it using `$expr`:

```
db.ee.aggregate({$match: {$expr: {$gt: ["$x", "$_id"]}}})
{ "_id" : 3, "x" : 5 }
{ "_id" : 4, "x" : 6, "y" : [ 2, 3, 3 ] }
```

More complex expressions involving arithmetics and aggregates over arrays are allowed too.

Find all documents where `_id` is less than `x` by 2.

```
db.ee.aggregate({$match: {$expr: {$eq: ["$x",
                                        {$add: ["$_id",
                                                2]
                                         }
                                        ]
                                   }
                           }
                  })

{ "_id" : 3, "x" : 5 }
{ "_id" : 4, "x" : 6, "y" : [ 2, 3, 3 ] }
```

Find all documents where the total value of `y` exceeds `x`

```
db.ee.aggregate({$match: {$expr: {$gt: [{$sum: "$y"},
                                          "$x"
                                         ] }
                           }})
{ "_id" : 1, "x" : 1, "y" : 2 }
{ "_id" : 4, "x" : 6, "y" : [ 2, 3, 3 ] }
```

Note, how `"$sum"` handles non-array elements.

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 original syntax of the `$lookup` operation was:

```
{
   $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. Consider the following two collections:

```
> 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 }
```

```
> 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.

Additional $lookup functionality. The original `$lookup` operation is of **very limited use** by itself, primarily because

- the join condition is ALWAYS the equality of a pair of fields: one local, one foreign.
- it disallows any manipulations with the *foreign collection*. This means that in a lot of situations the foreign collection needs to be prepared in advance for the query (which is difficult with, for example, only read access to the data), or that tricks have to be implemented (esssentially using `$lookup` as a cartesian product operation). The latter deprives `$lookup` of any semblence of efficiency.

In **Version 3.6.** MongoDB developers finally resolved this problem, by offering an alternate syntax for the `$lookup` document. This syntax fixes both issues in the following ways:

- the new `pipeline` field describes a series of aggregation steps to be applied to the foreign collection.
- the new `let` step introduces "local variables". Their values are computed in the scope of the local collection, but *the variables can be referenced in the* `pipeline` *expression, making it possible to use their values in the scope of the foreign collection.*

  This can be used to create correlated join subqueries, and match on more than just equality.

The alternate syntax of the `$lookup` command is thus:

```
{
   $lookup:
     {
       from: <collection>,
       let: { value1: <expression1>, ..., valueK: <expressionK>},
       pipeline: [ <stage1>, ..., <stageM>],
       as: <arrayField>
     }
}
```

Here, `from` and `as` have the same meaning as before: they respectively name the foreign collection and the field in the output where to joined documents from the foreign collection will be stored.

`let` is **optional**. When present, it defines a series of *local variables* with names `value1`, ..., `valueK`. The value of each of these local variables is determined by the respective expression - which is *any* valid expression from the `$project` stage.

`pipeline` takes as its value an array that contains a sequence of one or more aggregation pipeline stages. These stages are applied **to the foreign collection**. Any variables defined in the `let` portion of the syntax can be referenced and used for coomputations and comparisons in the `pipeline` stage using the `$$` syntax. For example, if we have the following `let` definition:

```
let: {total: "$total"}
```

the reference to it in the `pipeline` stage will be `"$$total"`.

Example. We demonstrate the new pipeline below. Consider the task of finding all students whose CSC 453 grade was better than May's. Here is a stress-free way of doing it using the only the `pipeline` stage of new syntax.

```
db.grades.aggregate(
{$match: {class:453}},
{$lookup: {from: "grades",
           pipeline: [{$match:{name: "May",
                               class: 453}},
                      {$project: {grade:1, _id:0}}
                     ],
           as: "may"
           }
},
{$unwind: "$may"},
{$match: {$expr: {$gt: ["$grade", "$may.grade"]}}}
)
```

```
{ "_id" : 2, "name" : "Bob", "class" : 453, "grade" : 92, "may" : { "grade" : 90 } }
```

To give some inisight into the work of `$lookup`, here is specifically the result of the `$lookup` stage of the pipeline:

```
{ "_id" : 2, "name" : "Bob", "class" : 453, "grade" : 92, "may" : [ { "grade" : 90 } ] }
{ "_id" : 4, "name" : "May", "class" : 453, "grade" : 90, "may" : [ { "grade" : 90 } ] }
{ "_id" : 6, "name" : "Chris", "class" : 453, "grade" : 74, "may" : [ { "grade" : 90 } ] }
```

This example essentially, selects May's grade from the foreign collection and performs a cartesian product of the selected grade and all other documents that contain student grades in CSC 453.

Here is the same example, using the power of `let` and local variable declarations to perform the join directly inside the `pipeline` description in the `$lookup`.

```
db.grades.aggregate(
{$match: {class:453}},
{$lookup: {from: "grades",
           let: {total: "$grade"},
           pipeline: [{$match:{name: "May",
                               class: 453}},
                      {$project: {grade:1, _id:0}},
                      {$match: {$expr: {$gt: ["$$total", "$grade"]}}}
                     ],
           as: "may"
           }
},
{$unwind: "$may"},
)
```

```
{ "_id" : 2, "name" : "Bob", "class" : 453, "grade" : 92, "may" : { "grade" : 90 } }
```

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 }
```

Special Versions of Projection

$addFields, $set. The `$addFields` (`$set`) operation works like a projection operation which includes *all* attributes from the input document and adds additional attributes to the document. The syntax of the command is

```
{$addFields: { <newField>: <expression>, ...,
               <newField>: <expression>} }
```

or

```
{$set: { <newField>: <expression>, ...,
               <newField>: <expression>} }
```

Here `<newField>` represents the names of the new fields to be added to the output documents, while `<expression>` has the same syntax as the expressions used in the `$project` operation.

Example. Consider the following simple list of courses with the sizes of each section specified.

```
> db.classes.find()
{ "_id" : 1, "class" : "CSC 369", "roster" : 28 }
{ "_id" : 2, "class" : "CSC 445", "roster" : 34 }
{ "_id" : 3, "class" : "CSC 466", "roster" : 17 }
{ "_id" : 4, "class" : "CSC 357", "name" : "Systems Programming", "sections" : [ "01", "03", "05", "07" ], "roster"
{ "_id" : 5, "class" : "CSC 101", "roster" : 100 }
{ "_id" : 6, "class" : "CSC 480", "name" : "AI", "roster" : 28 }
{ "_id" : 7, "roster" : 34, "class" : "CSC 202" }
{ "_id" : 8, "class" : "CSC 202", "sections" : [ "01", "02", "03", "04" ], "roster" : [ 20, 20, 30, 30 ] }
```

The following aggregation pipeline adds a new key to each document, specifying whether the course in question has large sections.

```
> db.classes.aggregate({$addFields:
                            {sectionSize:
                                        {$cond: [{$gte: ["$roster", 30]}, "large", "small"]}}})

{ "_id" : 1, "class" : "CSC 369", "roster" : 28, "sectionSize" : "small" }
{ "_id" : 2, "class" : "CSC 445", "roster" : 34, "sectionSize" : "large" }
{ "_id" : 3, "class" : "CSC 466", "roster" : 17, "sectionSize" : "small" }
{ "_id" : 4, "class" : "CSC 357", "name" : "Systems Programming", "sections" : [ "01", "03", "05", "07" ],
             "roster" : [ 34, 20, 32, 25 ], "sectionSize" : "large" }
{ "_id" : 5, "class" : "CSC 101", "roster" : 100, "sectionSize" : "large" }
{ "_id" : 6, "class" : "CSC 480", "name" : "AI", "roster" : 28, "sectionSize" : "small" }
{ "_id" : 7, "roster" : 34, "class" : "CSC 202", "sectionSize" : "large" }
{ "_id" : 8, "class" : "CSC 202", "sections" : [ "01", "02", "03", "04" ], "roster" : [ 20, 20, 30, 30 ],
             "sectionSize" : "large" }
```

$unset. The `$unset` stage is a version of a projection stage in which the only change to the documents is deletion of some of the keys from them. Its syntax is a little be shorter and less confusing than the syntax of the `$project` stage that performs only key removals.

The syntax of the `$unset` stage is as follows:

```
{$unset: [<key1>,...,<keyK>]}
```

Here, `<key1>`,...,`keyK` are strings representing the names of the keys in the documents that will be removed by the operation.

If `$unset` needs to remove only one key, a simplified syntax

```
{$unset: <key>}
```

can be used instead of making the value an array.

$replaceRoot, $replaceWith. This stage provides a convenient shorthand for situations when the documents in the collection contain embedded documents that need to be "surfaced", i.e., become "roots" of the return documents. This operation can be performed by the `$project` stage, but the use of `$replaceRoot` or `$replaceWith` in its stead improves readability of your code. The syntax of the stages (we use `$replaceRoot`, the other stage name, `$replaceWith` is an alias, and has exactly the same syntax) is as follows:

```
{ $replaceRoot: { newRoot: <replacementDocument> } }
```

Here, `<replacementDocument>` is a description of what the new output document should look like. It follows the same syntax and the rules as the documents used as the arguments of the `$project` stage.

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
                }
        ]
}
```

Additional Operations

$sample. The `$sample` operation returns a random sample of documents from the collection. The syntax of the aggregation pipeline command is:

`{$sample: {size: <positive integer>}}`

Here, `<positive integer>` represents the number of documents to put in the sample. Note, this operation samples with replacement, so a document from the original collection can be placed in the sample multiple times.

Example. In the example below, we ask for a sample of size three twice in a row and observe different documents returned.

```
> db.classes.aggregate({$sample: {size: 3}})
{ "_id" : 4, "class" : "CSC 357", "name" : "Systems Programming",
```

```
             "sections" : [ "01", "03", "05", "07" ], "roster" : [ 34, 20, 32, 25 ] }
{ "_id" : 5, "class" : "CSC 101", "roster" : 100 }
{ "_id" : 2, "class" : "CSC 445", "roster" : 34 }
> db.classes.aggregate({$sample: {size: 3}})
{ "_id" : 7, "roster" : 34, "class" : "CSC 202" }
{ "_id" : 1, "class" : "CSC 369", "roster" : 28 }
{ "_id" : 8, "class" : "CSC 202", "sections" : [ "01", "02", "03", "04" ],
             "roster" : [ 20, 20, 30, 30 ] }
```

$count. This operation returns an object with a single key-value pair. The key name is provided as the input to the operation. The value is the number of documents produced on the previous stage of the aggregation pipeline. The format of the aggregation pipeline document for `$count` is as follows:

```
{$count: <string>}
```

Here, `<string>` is the name of the key in the output document.

example. The example below runs an aggregation pipeline that counts how many courses have sections that have less than 25 students in them.

```
>  db.classes.aggregate({$match: {"roster": {$lt: 25}}})

{ "_id" : 3, "class" : "CSC 466", "roster" : 17 }
{ "_id" : 4, "class" : "CSC 357", "name" : "Systems Programming",
             "sections" : [ "01", "03", "05", "07" ], "roster" : [ 34, 20, 32, 25 ] }
{ "_id" : 8, "class" : "CSC 202", "sections" : [ "01", "02", "03", "04" ],
             "roster" : [ 20, 20, 30, 30 ] }

> db.classes.aggregate({$match: {"roster": {$lt: 25}}},
                                {$count: "numberSmallClasses"})

{ "numberSmallClasses" : 3 }
```

$out. The `$out` pipeline aggregation command can only show up as the last command of the pipeline. It directs the result of the previous stage of the pipeline to be inserted into a given new collection. The format of the command is simple:

```
{$out: "<output-collection>"}
```

Here, `"<output-collection>"` is a string representing the name of the new collection into which the result of the aggregation pipeline will be inserted.

Example. Here is an example of creating a new collection `classNames` consisting of only course names.

```
> db.classes.aggregate({$project: {"_id":0, "class":1}},
                                  {$out: "classNames"})

> db.classNames.find()

{ "_id" : ObjectId("5c4eb2502209cf8793d2fd04"), "class" : "CSC 369" }
{ "_id" : ObjectId("5c4eb2502209cf8793d2fd05"), "class" : "CSC 445" }
{ "_id" : ObjectId("5c4eb2502209cf8793d2fd06"), "class" : "CSC 466" }
{ "_id" : ObjectId("5c4eb2502209cf8793d2fd07"), "class" : "CSC 357" }
{ "_id" : ObjectId("5c4eb2502209cf8793d2fd08"), "class" : "CSC 101" }
{ "_id" : ObjectId("5c4eb2502209cf8793d2fd09"), "class" : "CSC 480" }
{ "_id" : ObjectId("5c4eb2502209cf8793d2fd0a"), "class" : "CSC 202" }
{ "_id" : ObjectId("5c4eb2502209cf8793d2fd0b"), "class" : "CSC 202" }
```

The `$out` operation has several limitations. The new collection **must reside** in the same database as the collection that is being worked on by the aggregation pipeline. This makes it impossible to use `$out` to extract information in databases where the user only has read access. Additionally, `$out` overwrites the output collection with the result – which, in some cases, is the desired behavior, but in some other cases - *is not*. The `$merge` stage was introduced in Version 4.2. to provide more control over saving of the results of aggregation pipelines.

$merge. The `$merge` operation provides fine-tuned control over saving output of an aggregation pipeline. Like `$out`, the `$merge` can only show up exactly once in the aggregation pipeline, and can only be the last stage of such a pipeline (meaning, among other things, that no pipeline can contain both `$merge` and `$out` stages). But `$merge` can handle several different ways of saving the results:

- **Target collection:** the target collection can be created in a different database than the source collection. This makes `$merge` a versatile operation for users who have read permissions for data access in a specific (usually shared) read-only resource, but who have write access to other databases, into which they can put the output of an aggregation pipeline.
- **New vs. existing collection**: the operation supports both placing data into a brand new collection, as well as into a collection that already exists.
- **Merging the results**: when results are placed into an existing collection, the user can choose one of four available conflict resolution/data integration policies for the situation when a document with the same identity exists both in the collection and in the aggregation pipeline document:
    1. `"replace"`: the results of the aggregation pipeline replace any documents in the same existing collection that have the same itendity.

2. `"keepExisting"`: existing objects from the collection take precedence over objects with the same identity supplied by the aggregation pipeline output.
3. `"merge"`(default): the results of the aggregation pipeline are combined in a single document with the existing objects in the collection that share the same identity.
4. `"fail"`: if the results of the aggregation pipeline contains objects with the same identity but with different content than the objects already in the collection, stop the operation, and fail.
5. **custom handling**: the user can also specify an aggregation pipeline as the data integration policy – the result of the pipeline applied to the document from the aggregation pipeline will be placed into the collection.

Additionally, when a document contained in the output of the aggregation pipeline is not found in the collection, `$merge` operation supports a choice of three different ways of proceeding:

1. `"insert"` (default): the document is inserted into the target collection.
2. `"discard"`: the document is not inserted into the target collection.
3. `"fail"`: fails the operation

The full syntax of the operation is:

```
{$merge: {
          into: <collectionSpecification>,
          on: <idSpecification>,
          whenMatched: <replace|keepExisting|merge|fail|<pipeline> >,
          whenNotMatched: <insert|discard|fail>
          }
}
```

In this stage description, the `on`, `let`, `whenMatched`, and `whenNotMatched` keys are optional. `whenMatched` and `whenNotMatched` have default values speficied above: `"merge"` for the former, and `"insert"` for the latter.

The values of the keys in the `$merge` document have the following syntax and meaning:

`collectionSpecification` is either a string literal identifying the name of a collection in the current database where the output will be placed, or, a document of the form:

```
{
  db: <databaseName>,
coll: <collectionName>
}
```

In the long form, the document provides the name of the database (presumably a different one from the database in which the input collection resides), and the name of the collection into which to place the output.

`on`: specifies how the identity (and matches) between the documents from the aggregation pipeline and those in the output collection is established. The default value of this key is `"_id"` (match on the internal id – remember, you can set it up/manipulate it in your our work).

The syntax of this key is:

```
on: <keyName>
```

or:

```
on: [<keyName1>,...,<keyNameL>]
```

The second format establishes a multi-attribute key to be used for object identification and match.