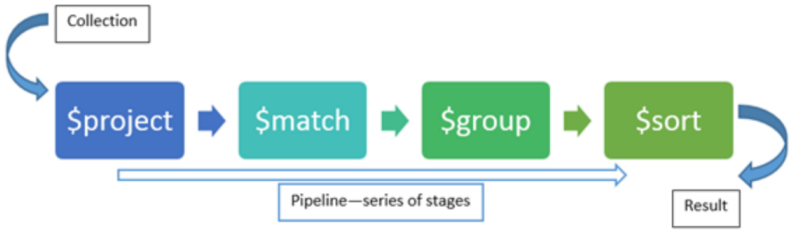
# L18 - NoSQL - Aggregation Framework

In RDBMS we have learnt what are aggregation functions, are there equivalent in MongoDB? MongoDB does have an Aggregation Framework where we can use to perform analytics on documents in one or more collections using MongoDB's set of analytics tools.

Being a framework, it means that it had to be built on top of some underlying structure and this structure is modeled upon the concept of data processing pipelines. A pipeline is a set of data processing stages connected in a series where the output data of each stage is the input data of the next stage.




With the aggregation pipeline, the data is the documents from one or more collections and this data is then passed through one or more stages, where at each stage, a different operation is performed. The output produced at the end of each stage resembles documents returned from a `find()` query function. However, it is not a `cursor` object that is returned but a stream of documents which will be fed as inputs to the next stage in the pipeline.

The operation at each stage are able to transform or filter the documents or even generate new documents therefore it is not always right to say that each stage produces one output document for every input document. Occasionally, some stages are executed multiple times as this helps to filter out unnecessary data being passed along the pipeline.

## Getting Started

Let's start by working with a simple database. It contains two collections. One is called `universities` and made up of two documents. (Data is not real.)

```
{
  country : 'Singapore',
  city : 'Singapore',
  name : 'SIT',
  location : {
    type : 'Point',
    coordinates : [ 1.3005, 103.7803 ]
  },
  students : [
    { year : 2014, number : 24774 },
    { year : 2015, number : 23166 },
    { year : 2016, number : 21913 },
    { year : 2017, number : 21715 }
  ]
}
{
  country : 'Singapore',
  city : 'Singapore',
```

```
  name : 'DigiPen',
  location : {
    type : 'Point',
    coordinates : [ 1.3075, 103.7782 ]
  },
  students : [
    { year : 2014, number : 4788 },
    { year : 2015, number : 4821 },
    { year : 2016, number : 6550 },
    { year : 2017, number : 6125 }
  ]
}
```

The other collection is called `courses` :

```
{
  university : 'DigiPen',
  name : 'RTIS',
  level : 'Excellent'
}
{
  university : 'SIT',
  name : 'Electronics',
  level : 'Excellent'
}
{
  university : 'SIT',
  name : 'Communication',
  level : 'Intermediate'
}
```

The aggregate syntax begins with the aggregate command that operates on a collection like the other CRUD commands.

```
db.universities.aggregate(pipeline, options)
```

where

- `pipeline` parameter is an array of "stages" that the data will go through. Each stage in the pipeline is completed independently, one after another, until none are remaining. The resulting data after the final stage is stored in a cursor and returned to you.
- `options` parameter is an optional document consisting of details such as how the aggregation should execute or some flags that are required during debugging and building your pipelines.

Let's start with a filter that returns all universities located in country `Singapore` :

```
db.universities.aggregate([{$match: {country: "Singapore"}}])
```

Some of you may notice that this operation is the same as the `find()` function on the collection. Next we are going to add a project stage to the pipeline.

```
db.universities.aggregate([{$match: {country: "Singapore"}},
                           {$project: {_id: 0, name: 1, city: 1}}])
```

Running this command you'll get two documents. If we'd like to limit that output to one document.

```
db.universities.aggregate([{$match: {country: "Singapore"}},
                           {$limit: 1},
                           {$project: {_id: 0, name: 1, city: 1}}])
```

Notice how the `$limit` stage is placed before the `$project` stage. The reason for that is because we want to cut down the number of documents being passed to the `$project` stage. If the stages were switched, in this case it would not make a difference on the results. But think of a collection having hundreds of documents, the `$project` stage will have to go through hundreds of documents before limiting it to 1 for display. When it comes to aggregation pipelines in MongoDB, efficiency of your pipeline should be your top priority.

Let's use another one stages `$sort` to sort the result in ascending order by name.

```
db.universities.aggregate([{$match: {country: "Singapore"}},
                           {$sort: {name:1}},
                           {$project: {_id: 0, name: 1, city: 1}}])
```

Output:

```
[ { city: "Singapore", name: "DigiPen"},
  { city: "Singapore", name: "SIT" }
]
```

# Pipeline In-Depth

The aggregation pipeline is not just limited to those stages. A full list of aggregation pipeline stages can be found here. The most commonly used stages are:

- `$project` – select, reshape data
- `$match` – filter data
- `$group` – aggregate data
- `$sort` – sorts data
- `$skip` – skips data
- `$limit` – limit data
- `$unwind` – normalizes data

## `$project` stage

More often than not, when working with big datasets, we want to reshape the document into a usable document for us to work with. We want to find the university `DigiPen` and display its `name`, `city` and `students.number`, omit the `students.year`.

```
db.universities.aggregate([{$match: {name: "DigiPen"}},
                           {$project: {_id: 0, name: 1, city: 1,
"students.number":1}}])
```

The output:

```
[
  {
    city : 'Singapore',
    name : 'DigiPen',
    students : [
    { number : 4788 },
    { number : 4821 },
    { number : 6550 },
    { number : 6125 }
    ]
  }
]
```

Now we use **field paths** to reshape the structured filed `students`.

```
db.universities.aggregate([{$match: {name: "DigiPen"}},
                           {$project: {_id: 0, name: 1, city: 1,
                                       students:"$students.number"}}])
```

Notice the field `students` is reshaped:

```
[
  {
    city : 'Singapore',
    name : 'DigiPen',
    students : [4788, 4821, 6550, 6125]
  }
]
```

## `$group` stage

As the name suggests, `$group` groups documents based on some key. Let's say we want to group `courses` on their `level` , and we want to find the number of courses in the group for each `level`.

```
db.courses.aggregate([{$group:{_id:'$level', no_of_courses:{$sum:1}}}])
```

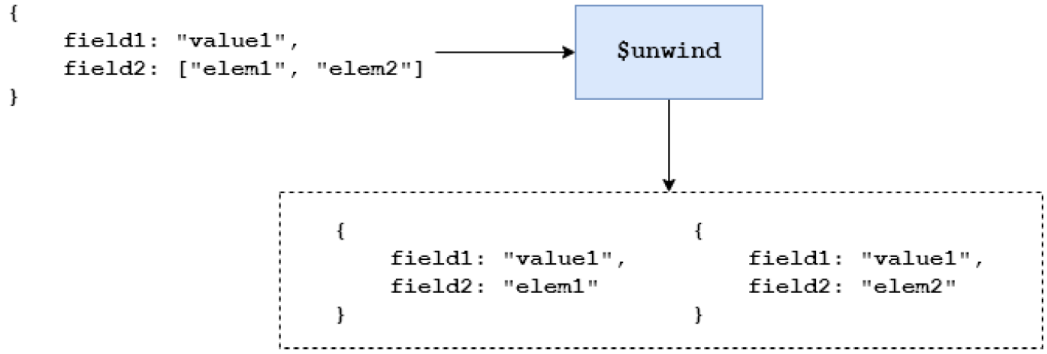
Here, `_id` is the key for grouping, a new key named `no_of_courses` is created.

`$sum:1` is used to find the total record in each group. Expression `1` to a document will return 1, hence, it will aggregate a value of one for each document in the group, thus yielding the total number of documents per group. The output is:

```
{ _id: 'Excellent', no_of_courses: 2}
{ _id: 'Intermediate', no_of_courses: 1}
```

## `$unwind` stage

There will be a lot of instances where you will need to work with array fields in your aggregation pipeline and you may need to "unwind" the array fields to produce a document for each element in a specified array field.




```
db.universities.aggregate([{$match: {name: "DigiPen"}},
                           {$project: {_id: 0, name: 1, students:1}},
                           {$unwind : '$students'}])
```

The output before adding `$unwind`.

```
{
  name : 'DigiPen',
  students : [
    { year : 2014, number : 4788 },
    { year : 2015, number : 4821 },
    { year : 2016, number : 6550 },
    { year : 2017, number : 6125 }
  ]
}
```

After adding `$unwind`, the output above will be broken up into 4 documents each with duplicated field values except that for the elements in the arrays (that are within the `students`), they will be separated into one document each.

```
{  name : 'DigiPen',  students : { year : 2014, number : 4788 } },
{  name : 'DigiPen',  students : { year : 2015, number : 4821 } },
{  name : 'DigiPen',  students : { year : 2016, number : 6550 } },
{  name : 'DigiPen',  students : { year : 2017, number : 6125 } }
```

## `$lookup` stage

Because MongoDB is document-based, we can shape our documents the way we need. However, there is often a requirement to use information from more than one collection.

```
db.universities.aggregate([
  { $match : { name : 'SIT' } },
  { $project : { _id : 0, name : 1 } },
  { $lookup : {
      from : 'courses',
      localField : 'name',
      foreignField : 'university',
      as : 'courses'
  } }
])
```

The output:

```
{
    name: 'SIT',
    courses: [
        {
            _id: ObjectId("605c91205b5702e72ab13643"),
            university: 'SIT',
            name: 'Communication',
            level: 'Intermediate'
        },
        {
            _id: ObjectId("605c91205b5702e72ab13644"),
            university: 'SIT',
            name: 'Electronics',
            level: 'Excellent'
        }
    ]
}
```

## `$sortByCount` stage

This stage is a shortcut for grouping, counting and then sorting in descending order the number of different values in a field.

Suppose you want to know the number of courses per level, sorted in descending order. The following is the query you would need to build:

```
db.courses.aggregate([ { $sortByCount : '$level' }])
```

This is the output:

```
{ _id: 'Excellent', count: 2 },
{ _id: 'Intermediate', count: 1 }
```

## `$facet` stage

Sometimes when creating a report on data, you find that you need to do the same preliminary processing for a number of reports, and you are faced with having to create and maintain an intermediate collection.

You may, for example, do a weekly summary of trading that is used by all subsequent reports. You might have wished it were possible to run more than one pipeline simultaneously over the output of a single aggregation pipeline. We can now do it within a single pipeline thanks to the `$facet` stage.

```
db.universities.aggregate([
  { $match : { name : 'SIT' } },
  { $lookup : {
      from : 'courses',
      localField : 'name',
      foreignField : 'university',
      as : 'courses'
  } },
  { $facet : {
      'countingLevels' :
      [
```

```
         { $unwind : '$courses' },
         { $sortByCount : '$courses.level' }
      ],
      'yearWithLessStudents' :
      [
         { $unwind : '$students' },
         { $project : { _id : 0, students : 1 } },
         { $sort : { 'students.number' : 1 } },
         { $limit : 1 }
      ]
  } }
])
```

What we have done is to create two reports from our database of university courses. **CountingLevels** and **YearWithLessStudents.** They both used the output from the first two stages, the `$match` and the `$lookup`.

With a large collection, this can save a great deal of processing time by avoiding repetition, and we no longer need to write an intermediate temporary collection.

```
{
  countingLevels: [ { _id: "Excellent", count: 1},{ _id: "Intermediate", count:
1} ],
  yearWithLessStudents: [ { students: { year: 2017, number : 21715 } } ]
}
```

## `$out` stage

This stage allows you to carry the results of your aggregation over into a new collection, or into an existing one after dropping it, or even adding them to the existing documents.

The `$out()` operator must be the last stage in the pipeline.

```
db.courses.aggregate([ { $sortByCount : '$level' },
                       { $out : 'aggResults' }
])
```

Now, we check the content of the new `aggResults` collection:

```
db.aggResults.find().
{ "_id" : "UPSA", "totaldocs" : 1 }
{ "_id" : "USAL", "totaldocs" : 1 }
>
```