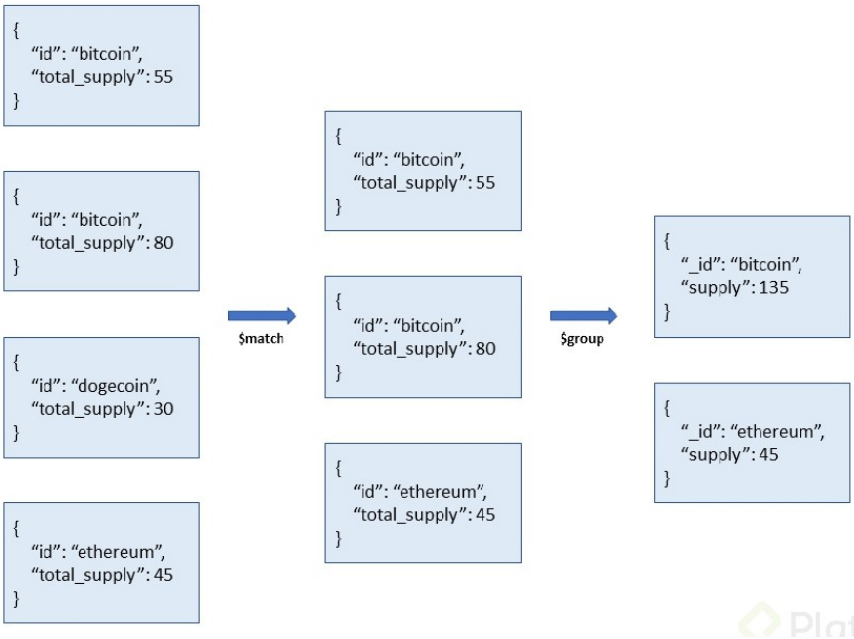
AGREGACIÓN

MongoDB cuenta con tres maneras de realizar operaciones de agregación:

- El pipeline de agregación
- La función map-reduce
- Métodos de agregación de un solo propósito.

Pipeline de Agregación.-

Primero que nada, el **pipeline** (tubería en español) consiste en ir pasando nuestros datos por una serie de etapas en las cuales se modifica, agrupa u opera sobre los mismos; los datos resultantes pueden pasar a una nueva etapa y repetir este proceso tantas veces se quiera hasta obtener los resultados finales que se necesiten.



Las etapas a lo largo del pipeline tienen forma de documentos, y trabajan con operadores nativos de Mongo para hacer la transformación de los datos de manera eficiente.

Al hacer uso del método aggregate, se pasa toda la colección (es decir todos los documentos que están almacenados) al pipeline, por lo cual es altamente recomendado hacer uso de la etapa $match para filtrar los documentos en caso de que no se necesite operar con cada uno de ellos. Así mismo, otra de las ventajas que obtenemos al usar $match o $sort al principio del pipeline es que podemos aprovechar los índices de la colección, lo cual mejora el rendimiento general de la operación.

Para usar el pipeline de agregación, haremos uso del método **aggregate**, el cual tiene la siguiente forma:

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

En **pipeline se definen las operaciones o etapas por las cuales pasaran los datos**, **pueden ser elementos separados o un arreglo que contenga cada una de ellas;** en caso de optar por definir cada operación o etapa en elementos separados, no se podrán definir las opciones. A su vez, e**n el documento options se definen las opciones adicionales que tratará el método de agregación.**

Etapas pipeline.-

Si prestamos atención, nos daremos cuenta de que la operación aggregate recibe un array de operaciones, esto se debe a que el framework de agregación funciona como un pipe de Unix, es decir, se admite la construcción de transformaciones y agrupaciones en distintas etapas que se ejecutarán de forma serializada.

Estas son las etapas admitidas:

Etapa	Descripción	Multiplicidad
$project	Cambia la forma del documento	1:1
$match	Filtra los resultados	n:1
$group	agregación (agrupación)	n:1
$sort	ordenación	1:1
$skip	salta N elementos	n:1
$limit	elige N elementos	n:1
$unwind	normaliza arrays	1:n
$lookup	Desnormaliza arrays	n:1
$geonear	ordenación	01:01:00
$out	cambia nombres, etc	1:1

La multiplicidad se refiere a cuántos documentos obtenemos como resultado después de aplicar la etapa a N documentos. Por ejemplo, en una ordenación, como tan solo cambia el orden de N documentos, el resultado siguen siendo N documentos pero ordenados de otro modo.

Proyección ($project 1:1)
La proyección permite modificar la representación de los datos, por lo que en general se emplea para darles una nueva forma con la que nos resulte más cómodo trabajar. Por ejemplo, **permite eliminar claves, añadir nuevas claves, cambiar su forma, o usar algunas funciones simples de transformación** como $toUpper (convertir en mayusculas), $toLower (convertir en minúsculas), $add (sumar) y $multiply (multiplicar).

Se utiliza también en las primeras etapas para filtrar atributos y liberar memoria evitando procesar aquella información que no se va a necesitar.

Ejemplo:

```
db.zips.aggregate([{
    $project: {
        _id: 0,
        'ciudad':{$toLower: "$city"},
        'poblacion':{
            'unidades': "$pop",
            'miles':{"$multiply":["$pop",0.001]
            }
        }
    }
}])
```

Resultado:

```
{
    "ciudad": "cushman",
    "poblacion": {
        "unidades": 36963,
        "miles": 36.963
    }
}
{
    "ciudad": "barre",
    "poblacion": {
        "unidades": 4546,
        "miles": 4.546
    }
}
…
```

Nótese que poner un 0 en un valor de campo (como en el caso del _id), hace que el atributo se ignore en el resultado. Si se pone un 1, se utilizará exactamente el mismo valor que tiene en el documento original.

Filtrado ($match n:1)
La etapa match permite filtrar los documentos para que en el resultado de la etapa solo estén aquellos que cumplen ciertos criterios. Permite filtrar antes o después de agregar los resultados, en función del orden en que definamos esta etapa.

Por ejemplo, devolver unicamente los codigos postales del estado de California (CA)

```
db.zips.aggregate([{$match:{state: "CA"}}])
```

Agrupación ($group n:1)
La agrupación permite agrupar distintos documentos según compartan el valor de uno o varios de sus atributos, y realizar operaciones de agregación sobre los elementos de cada uno de los grupos. Esta etapa es rica en operadores, vamos a ver algunos ejemplos.

Operador $sum
Suma el valor de los atributos de los documentos del mismo grupo. Por ejemplo, para obtener, la población total de cada estado, y el número de ciudades que hay en el estado, para cada uno de los

estados, basta por agrupar por estado y aplicar la clausula sum convenientemente.

```
db.zips.aggregate([{
          $group: {
                    _id:"$state",
                    state_pop:{$sum: "$pop"},
                    state_cities: {$sum: 1}
                  }
          }])
```

Resultado:

```
{
    "_id": "WA",
    "state_pop": 4866692,
    "state_cities": 484
}
{
    "_id": "HI",
    "state_pop": 1108229,
    "state_cities": 80
}
…
```

Operador $avg
Obtiene la media de los valores de un atributo en el grupo. Por ejemplo, para calcular la población media de las ciudades de un estado.

```
db.zips.aggregate([{
          $group: {
                    _id: "$state",
                    avg_pop: {$avg: "$pop"}
                  }
                 }])
```

Resultado:

```
{
    "_id": "WA",
    "avg_pop": 10055.148760330578
} {
    "_id": "HI",
    "avg_pop": 13852.8625
} {
    "_id": "CA",
    "avg_pop": 19627.236147757256
}…
```

Operador $addToSet
Su objetivo crear un array con todos los valores distintos que toma un atributo en un grupo de documentos. Por ejemplo, la siguiente operación daria como resultado un documento que tendría un atributo categorías con todas las ciudades de cada estado.

```
db.zips.aggregate([{
                    $group: {
                         _id: "$state",
                         cities: {$addToSet: "$city"}
                    }
                  }])
```

Resultado:

```
{
     "_id" : "CA",
     "cities" : [
          "SOUTH LAKE TAHOE",
          "TAHOE CITY",
          …
          "CENTRAL FALLS",
          "PROVIDENCE"
     ]
}
```

Operador $push
Funciona igual que addToSet pero no garantiza que el elemento no vaya aparecer más de una vez si es que hay dos documentos con el mismo valor en el grupo.

Operadores $max y $min
Permiten obtener el valor máximo o el mínimo del grupo de documentos para alguno de sus atributos. Por ejemplo, para obtener la mayor población para cada estado.

```
db.zips.aggregate([{
                    $group: {
                         _id: "$state",
                         max_pop: {$max: "$pop"}
                    }
                  }])
```

Resultado:

```
{
     "_id" : "CA",
     "max_pop" : 99568
},
…
```

Agrupación compuesta
Nos permite agrupar por más de un atributo, esto es, como si agrupamos por subcategorías. Por ejemplo, para obtener la población de cada una de las ciudades de un estado.

```
db.zips.aggregate([{
               $group: {
                    _id: {state:"$state", city:"$city"},
                    state_pop:{$sum: "$pop"},
                    state_cities:{$sum: 1}
               }
               }])
```

Resultado:

```
{
    "_id": { "state": "AK", "city": "CRAIG" },
    "state_pop": 1398,
    "state_cities": 1
}
{
    "_id": {"state": "AK", "city": "KETCHIKAN"},
    "state_pop": 14308,
    "state_cities": 2
}
…
```

Primero y último ($first y $last)
Nos permiten obtener el primer y el último elemento de un grupo. **Para que esta operación tenga sentido debe combinarse con la ordenación… (de otro modo sería impredecible)**.

Ordenación ($sort 1:1)
Ordena los documentos resultantes. Intentará ordenar en memoria (con un límite de 100mb por etapa del pipeline) y en caso de no tener memoria suficiente, realizará una ordenación en disco.

En el ejemplo se muestra como podríamos obtener los documentos ordenados por población (en orden descendente) y estado (ascendente)

```
db.zips.aggregate([{
                    $sort: {
                         pop: -1,
                         state: 1
                    }
                  }])
```

Paginación ($skip y $limit)
En éste caso, a diferencia de cuando se utilizan en clausulas find, **sí importará el orden en el que aparece la etapa en el pipeline**. El resultado es que podremos ignorar los primero N elementos (skip) y obtener solo los N primero (limit).

Por ejemplo, podemos ignorar los 10 primeros y devolver los 5 siguientes (es decir, obtendríamos los elementos del 11 al 15.

```
db.zips.aggregate([
{
    $skip: 10
},
{
    $limit: 5
}
])
```

Operación Unwind ($unwind)
Permite agrupar sobre elementos de un array, para "aplanarlo". Por ejemplo, para el siguiente documento:

```
{ a:1, b:2, c:[pera, manzana]}
```

Después de aplicar unwind quedarían los siguientes documentos resultantes:

```
{a:1, b:2, c:pera}
{a:1, b:2, c:manzana}
```

Dado que esta etapa es la única cuya salida puede tener un mayor número de documentos que la entrada proporcionada, hay que usarla con cuidado para no sobrecargar la memoria, e intentar aplicar todos los filtros posibles antes a nuestros documentos para reducir los resultados potenciales. La sintaxis es la siguiente:

```
{"$unwind":"$c"}
```

Después es posible recuperar el array empleando $push como se ha visto más arriba. También es posible hacer un doble $unwind (en pipelines consecutivos) para aplanar datos que tengan dos arrays, por ejemplo (haciendo el producto cartesiano de todos ellos)

$lookup
Permite obtener los documentos de otra colección a lo que se les tiene una referencia, los anexa en un campo de tipo array y permite tratarlo en la siguiente etapa. En ocasiones, tendremos referencias a otros documentos y si el driver del lenguaje que usemos no nos brinda una manera de hacer esta clase de join, la etapa $lookup es la manera correcta de hacerlo.

Sintaxis:

```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
}
```

Ejemplo:

Dados las siguientes colecciones:

```
db.orders.insert([
   { "_id" : 1, "item" : "almonds", "price" : 12, "quantity" : 2 },
   { "_id" : 2, "item" : "pecans", "price" : 20, "quantity" : 1 },
   { "_id" : 3  }
])

db.inventory.insert([
   { "_id" : 1, "sku" : "almonds", description: "product 1", "instock" : 120 },
   { "_id" : 2, "sku" : "bread", description: "product 2", "instock" : 80 },
   { "_id" : 3, "sku" : "cashews", description: "product 3", "instock" : 60 },
   { "_id" : 4, "sku" : "pecans", description: "product 4", "instock" : 70 },
   { "_id" : 5, "sku": null, description: "Incomplete" },
   { "_id" : 6 }
])
```

Si realizamos la operación de lookup siguiente:

```
db.orders.aggregate([
    {
      $lookup:
        {
          from: "inventory",
          localField: "item",
          foreignField: "sku",
          as: "inventory_docs"
        }
   }
])
```

Obtenemos el siguiente resultado.

```
{
   "_id" : 1,
   "item" : "almonds",
   "price" : 12,
   "quantity" : 2,
   "inventory_docs" : [
      { "_id" : 1, "sku" : "almonds", "description" : "product 1", "instock" : 120 }
   ]
}
{
   "_id" : 2,
   "item" : "pecans",
   "price" : 20,
   "quantity" : 1,
   "inventory_docs" : [
      { "_id" : 4, "sku" : "pecans", "description" : "product 4", "instock" : 70 }
   ]
}
{
   "_id" : 3,
   "inventory_docs" : [
      { "_id" : 5, "sku" : null, "description" : "Incomplete" },
      { "_id" : 6 }
   ]
}
```

$out

Redirige la salida de una agrupación creando una nueva colección. Es muy fácil de utilizar.

```
db.books.aggregate( [
                      { $group : { _id : "$author", books:{$push: "$title"}}},
                      { $out : "authors" }
                    ] )
```

$geonear

Muestra una lista de los documentos ordenados desde el más próximo hasta el más alejado del punto especificado.

Tiene diferentes opciones entre las que destacan:

- **near**: Para especificar el punto respecto al que queremos medir las distancias en formato GeoJSON
- **distanceField**: Campo de salida en el que queremos almacenar la distancia
- **query**: Limita los resultados a los que encajan con la query

Ejemplo:

```
db.places.aggregate([
   {
     $geoNear: {
        near:{type:"Point",coordinates:[-73.98142,40.71782 ]},
        key: "location",
        distanceField: "dist.calculated",
        query: {"category": "Parks"}
     }
   }
])
```

Resultado:

```
{
   "_id" : 2,
   "name" : "Sara D. Roosevelt Park",
   "location":{"type":"Point","coordinates":[-73.9928,40.7193]},
   "category" : "Parks",
   "dist":{"calculated":974.175764916902}
}
…
```

Vistas

Las vistas se introdujeron en la versión 3.4 de MongoDB. Se puede pensar en una vista como una colección "virtual" que se crea a partir de una consulta.

Características principales:

- Las vistas se definen a través de una consulta de agregación.

  ```
  db.createView(view, source, pipeline, collation)
  ```

  Donde
  - *view*: Un string con el nombre de la vista a crear.
  - *source*: Un string con el nombre de la colección en la que se basa.
  - *pipeline*: la secuencia de agregación que define la vista
  - *collation*: adaptaciones locales.

- Las vistas son de solo lectura; es decir no podemos insertar/borrar/actualizar datos en una vista. En particular las vistas solo se pueden usar en las instrucciones:
  - `db.collection.find()`
  - `db.collection.findOne()`
  - `db.collection.aggregate()`
  - `db.collection.count()`
  - `db.collection.distinct()`

- Si actualizamos la colección base la vista automáticamente cambia.

- Se muestran como una colleción más con "`show collections`".
- La consulta se almacena en System.views.

Ejemplos de agragegación.-

Ejemplo1
Obtener la ciudad con mayor población de cada estado

Lo hacemos por etapas:

1. Obtenemos la población de cada ciudad de cada estado

2. Ordenamos por estado (ascendente) y población (descendente, la ciudad con mayor población quedará arriba en cada bloque de estados)

3. Agrupamos por estado, obteniendo el primero de cada grupo.

```
b.zips.aggregate([
    //obtiene la poblacion de cada ciudad de cada estado
    {
        $group: {
            _id: {state: "$state", city: "$city"},
            population: {$sum: "$pop"}
        }
    },
    //ordenamos por estado, poblacion
    {
        $sort: {
            "_id.state": 1,
            "population": -1
        }
    },
    //agrupamos por estado, obtenemos el primero de cada grupo
    {
        $group: {
            _id: "$_id.state",
            city:{$first: "$_id.city"},
            population: {$first: "$population"}
        }
    }
])
```

Resultado:

```
{
    "_id": "WV",
    "city": "HUNTINGTON",
    "population": 75343
}
{
    "_id": "WA",
    "city": "SEATTLE",
    "population": 520096
}…
```

Ejemplo 2:
Obtener la población total de las ciudades cuyo nombre empieza por A o B

```
db.zips.aggregate([
{
     // Eliminar campos innecesarios
    $project: {
        _id: 1,
        first_char:{$substr: ["$city", 0, 1]},
        pop: 1
    }
},
{
     // filtrar documentos
    $match: {
        "first_char": {
            $in: ["A", "B"]
        }
    }
},
{
     // agrupar toda la colección para sumar
    $group: {
        _id: null, // un grupo que representa toda la colección
        total: {
            $sum: "$pop"
        }
    }
}
])
```

Resultado:

```
{
     "_id" : null,
     "total" : 32119088
}
```

Ejemplo 3:
Obtenemos la media de población de entre todas las ciudades de los estados de Nueva York (NY) y California (CA) cuya población supera los 25000 ciudadanos.

```
db.zips.aggregate([
{ // Filtrar los estados
    $match: {
        "state": {
            $in: ["CA", "NY"]
        }
    }
},
```

```
{// Sacar la población de cada ciudad de cada estado
    $group: {
        _id: {
            "state": "$state",
            "city": "$city"
        },
        sum_pop: {
            $sum: "$pop"
        }
    }
},
{// filtrar los que superan los 25000
    $match: {
        "sum_pop": {
            $gt: 25000
        }
    }
},
{ // calcular la media de los que quedan como un sólo grupo
    $group: {
        _id: null,
        avg: {
            $avg: "$sum_pop"
        }
    }
}
])
```

Resultado:

```
{
     "_id" : null,
     "avg" : 82541.46420824295
}
```