2.3 – Persistencia y Análisis de Datos

Tema 3 – Persistencia No Relacional

Universidad Rey Juan Carlos

José Manuel Colmenar
josemanuel.colmenar@urjc.es
@jmcolmenarv

code URJC

Índice

- Introducción
- Recordatorio MongoDB
- Agregación de datos
- *Aggregation framework*
- Vistas
- *Replica Sets*
- *Sharding*
- *GridFS*
- *SpringData y MongoDB*

Bibliografía

- Pethuru Raj; Ganesh Chandra Deka (2018). A Deep Dive into NoSQL Databases: The Use Cases and Applications. Academic Press.
- Sudarshan Kadambi; Aaron Ploetz; Devram Kandhare; Xun Wu (2018). Seven NoSQL Databases in a Week. Packt Publishing.
- Dan Sullivan (2015). NoSQL for Mere Mortals. Ed.: Addison Wesley Professional.
- Cyrus Dasadia y Amol Nayak (2016). MongoDB Cookbook. 2ª Edición Ed.: Packt Publishing.
- David Hows, Peter Membrey, Eelco Plugge, Tim Hawkins (2015). The Definitive Guide to MongoDB: A complete guide to dealing with Big Data using MongoDB, 3ª Edición. Ed.: Apress.
- Douglas Garrett, Peter Bakkum, Kyle Banker, Shaun Verch and Tim Hawkins (2016) MongoDB in Action, Second Edition. Manning Publications.
- MongoDB. The MongoDB Manual https://docs.mongodb.org/manual/
- Webinar Exploring the Aggregation Framework

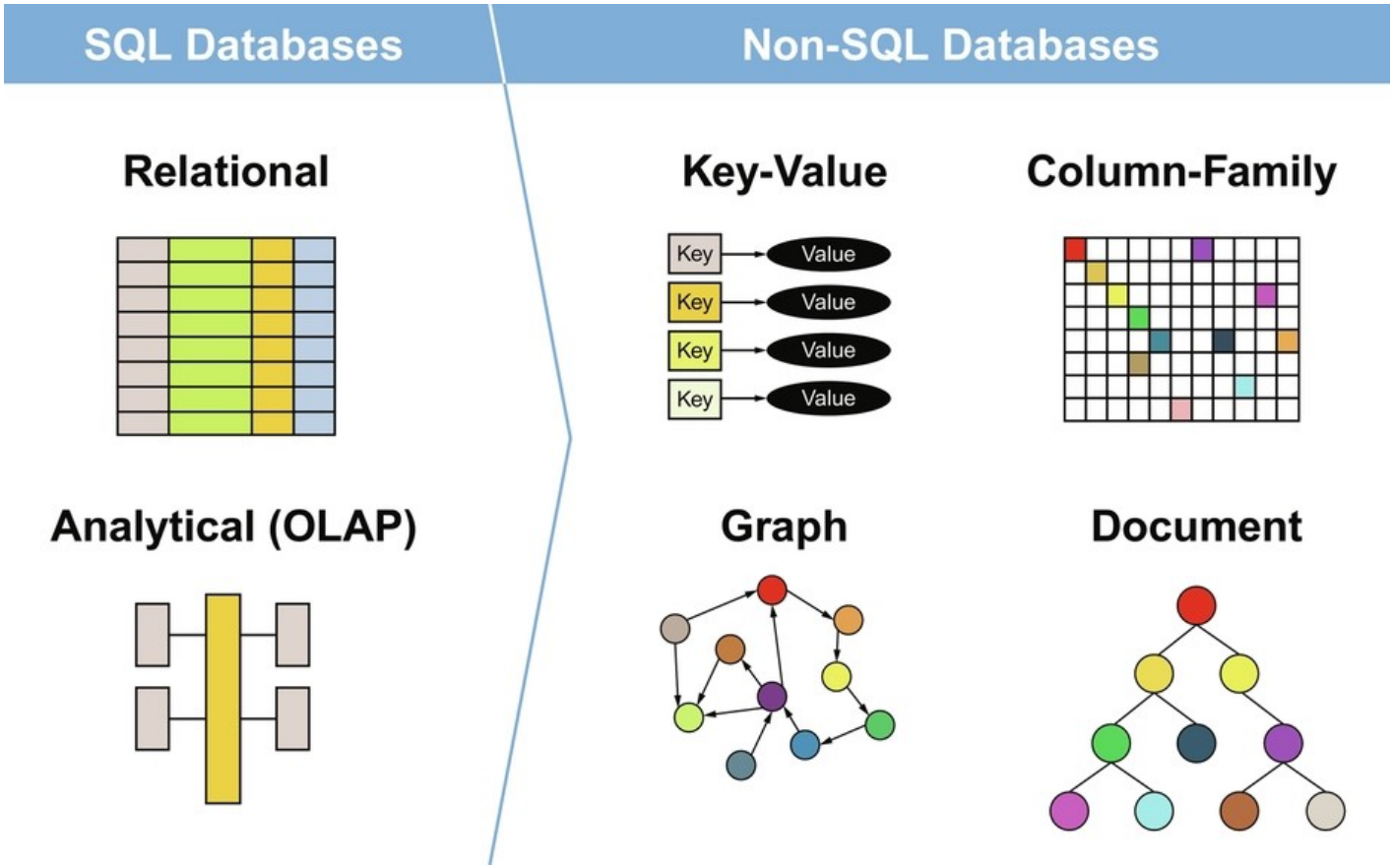
Introducción

- La persistencia no relacional se basa en el uso de modelos de datos diferentes al relacional (tablas).
- El modelo relacional no escala bien.
  - Existen implementaciones en forma de clusters: Oracle Cluster, MySQL NDB Cluster, …
- El modelo relacional es rígido porque la definición de las tablas (esquema) no se cambia.
  - Algunos RDBMS incorporan hibridaciones con JSON: Oracle, MySQL, PostgreSQL, …

Introducción

- Existen diferentes **modelos de datos no relacionales**:
  - **Clave-valor**: utiliza diccionarios (mapas) como estructura de almacenamiento. Amacena datos simples. Ej.: Redis, Oracle NoSQL.
  - **Columnas**: los datos se almacenan por columnas. Son muy adecuadas para *SQL analytics*. Ej.: Apache HBase.
  - **Documentos**: cada dato es un documento (XML, JSON, BSON, …). Son un tipo de BD clave-valor donde el valor es el propio documento. Ej.: MongoDB.
  - **Grafos**: los nodos representan entidades (persona, empleado, producto, …) y las aristas representan relaciones entre entidades. Los grafos están dirigidos. Ej.: Neo4j.

Introducción



Fuente: A Deep Dive into NoSQL Databases: The Use Cases and Applications.

Recordatorio MongoDB

- MongoDB es una base de datos no relacional basada en documentos tipo JSON.
  - Open source (GNU Affero General Public License combinada con licencia Apache).
  - Multi-plataforma, escrita en C++.
- Características más importantes:
  - Soporte de consultas complejas.
  - Indexación.
  - Replicación y balanceo de carga (*replica sets*).
  - Soporte para almacenamiento de ficheros.

Recordatorio MongoDB

- Dentro de una **instancia MongoDB** puede haber cero o más bases de datos.
  - Una **base de datos** es como un contenedor de alto nivel.
- Una base de datos puede contener cero o más colecciones.
  - Una **colección** es similar a la tabla tradicional.
- Cada colección consta de cero o más documentos.
  - Un **documento** es similar a una fila de una tabla tradicional.
- Un documento está formado por cero o más campos.
  - Los **campos** son similares a las columnas de una tabla tradicional.
- Los **índices** en MongoDB y en SGBD relacionales tienen una funcionalidad similar.
- Cuando se pide un dato a MongoDB este devuelve un puntero al resultado denominado **cursor**, de manera que se puede actuar sobre los datos sin necesidad de descargarlos.

Recordatorio MongoDB

- Creación de colecciones (y, a la vez, bases de datos):

```
db.colEj.insert({nombre:'Carmen', mes:'Enero',
                 cantidad: 15, esPremium:true})
db.colEj.insert({nombre:'David', mes:'Enero',
                 cantidad: 32, esPremium:false,
                 gustos:['Acción','Superheroes']})
```

- Lista de documentos de una colección

```
db.colEj.find()
```

- Obtener los índices

```
db.colEj.getIndexes()
```

- Vaciar una colección

```
db.colEj.remove({})
```

Recordatorio MongoDB

- Un **selector** es un objeto JSON donde se especifican los campos deseados. Además puede admitir operadores.

```
db.colEj.find({mes:'Enero'})
db.colEj.find({mes:'Enero', cantidad:{$gte:10}})
db.colEj.find({mes:'Enero', gustos:{$exists:true}})
db.colEj.find({mes:'Enero', gustos:{$in:['Acción','Espías']}})
db.colEj.find({mes:'Enero', $or:[{cantidad:{$lt:20},esPremium:true]})
```

- **Update** cambia un documento por el nuevo que se especifica:

```
db.colEj.update({nombre:'David'}, {esPremium:true})
```

- **Set** actualiza campos sin reemplazar completamente el documento.

```
db.colEj.update({nombre:'David'}, {$set:{esPremium:true}})
db.colEj.update({nombre:'David'}, {$set:{esPremium:true, mes:'Abril'}})
// Actualizar todos los documentos:
db.colEj.update({}, {$set:{esPremium:false}}, {multi:true})
```

- Otros operadores:

```
db.colEj.update({nombre:'David', {$inc:{cantidad:1}})
db.colEj.update({nombre:'David', {$push:{gustos:'Comedia'}})
// Inserta si no existe: upsert = update + insert
db.colEj.update({nombre:'Akiyama'}, {$set:{mes:'Abril'}}, {upsert:true})
```

Agregación de datos en MongoDB

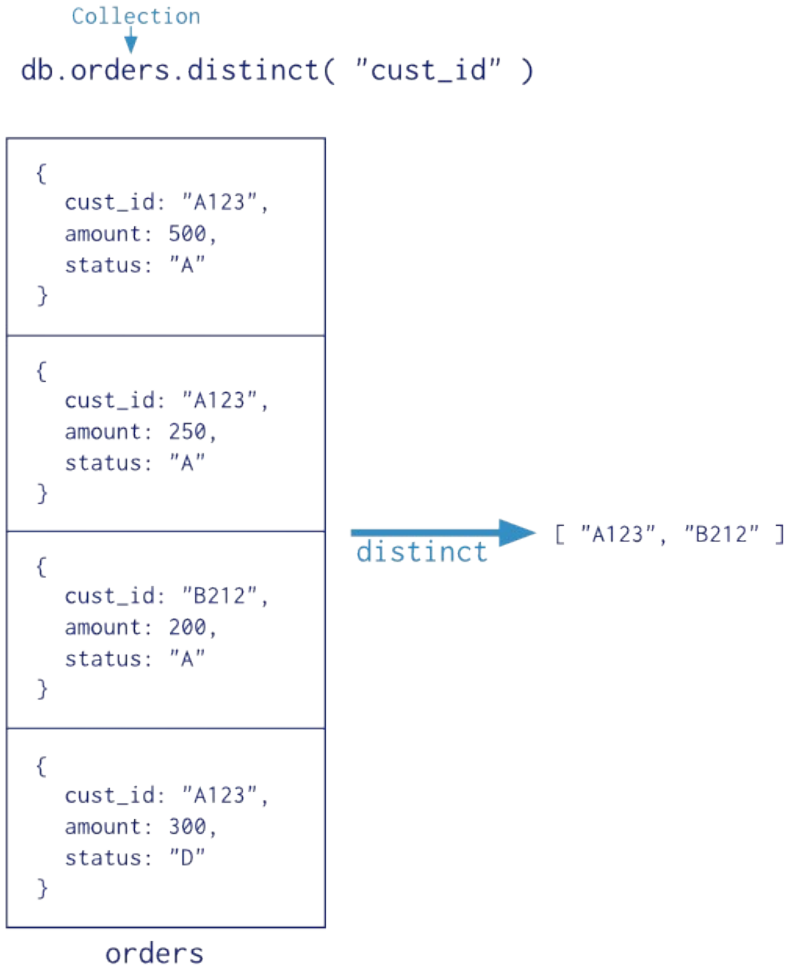
- Las agregaciones procesan registros y devuelven resultados calculados.
- Las operaciones de agregación agrupan valores de varios documentos, y realizan un conjunto de operaciones en los datos agrupados para devolver un resultado único.
- MongoDB proporciona tres formas de realizar agregaciones:
  - Métodos de agregación sencillos (*single purpose aggregation operations*).
  - La función map-reduce.
  - El *Aggregation Framework*.

Agregación de datos en MongoDB

code URJC

- Las *Single Purpose Aggregation Operations* de MongoDB son las siguientes funciones de agregación:
  - `db.collection.count()`
  - `db.collection.distinct()`
  - `db.collection.group()`
- Todas ellas agregan documentos a partir de una colección.
- Son mucho menos flexibles que las otras dos.

Agregación de datos en MongoDB

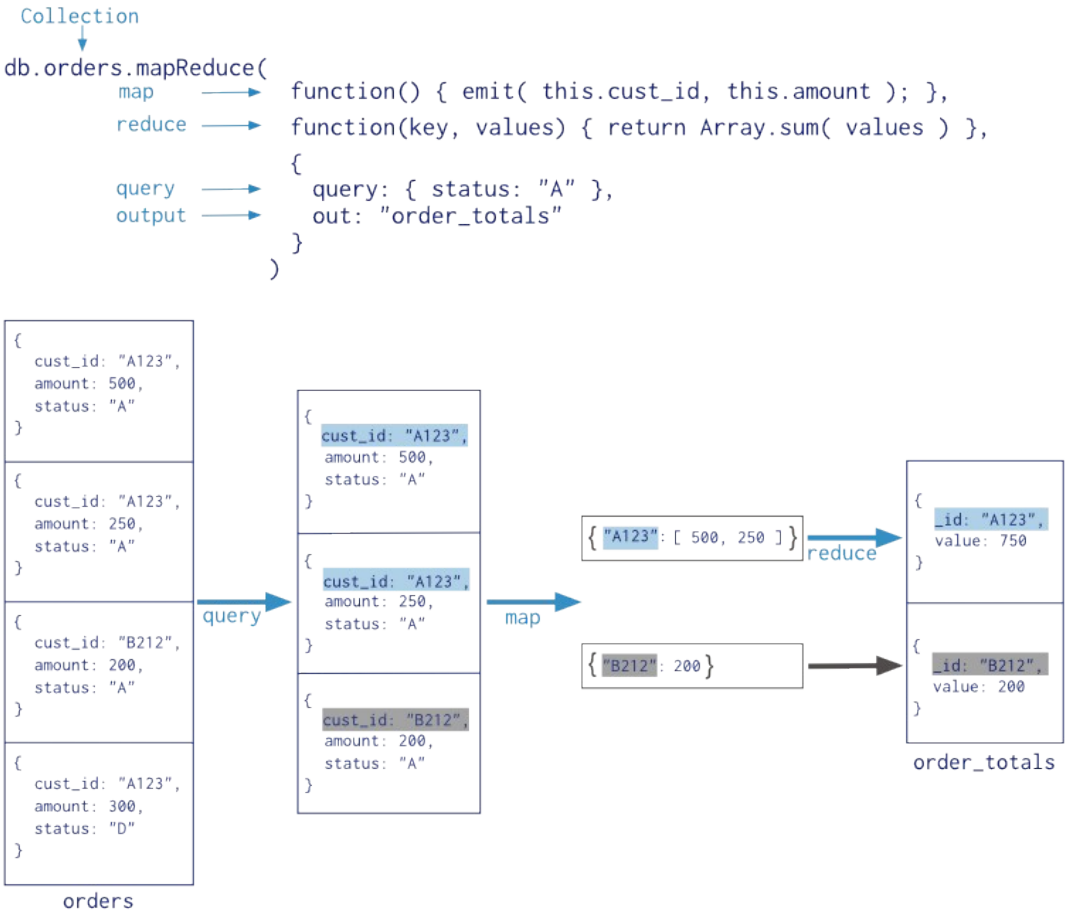


Fuente: Manual de MongoDB

Agregación de datos en MongoDB

- MongoDB proporciona también operaciones **map-reduce** para realizar agregaciones.
  - La etapa *map* procesa cada documento y emite uno o más objetos por cada documento de entrada.
  - La etapa *reduce* combina la salida de map.
- Utilizan funciones JavaScript para realizar las operaciones *map* y *reduce*, así como las operaciones finales opcionales. Es más flexible pero menos eficiente que los *pipelines*.
- Pueden operar también sobre colecciones sharded.

Agregación de datos en MongoDB



Fuente: Manual de MongoDB

Aggregation Framework

code URJC

- Es un framework para agregación de datos basado en el concepto de **pipeline**.
- Los documentos entran en un pipeline de varias etapas que transforma los documentos en resultados agregados.
- Tiene algunas limitaciones.

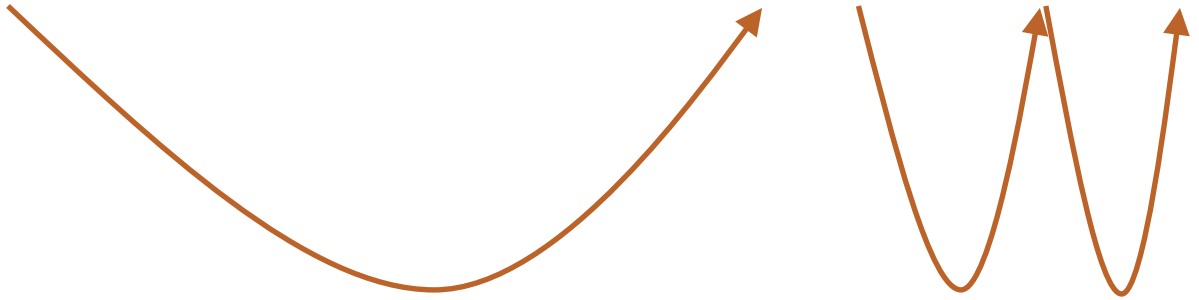
Aggregation Framework

- Los pipelines del Aggregation Framework funcionan como un conjunto de etapas que transforman los documentos.
- Las etapas pueden funcionar como filtros (consultas), agrupar y ordenar documentos, resumir contenidos de arrays, etc...
- Es el método preferido para agregar datos en MongoDB.

Aggregation Framework

- El pipeline consta de etapas.
- Cada etapa transforma los documentos a medida que pasan por las etapas.
- Las etapas pueden repetirse en el pipeline.

```
db.collection.aggregate([{<etapa>},...])
```



Aggregation Framework

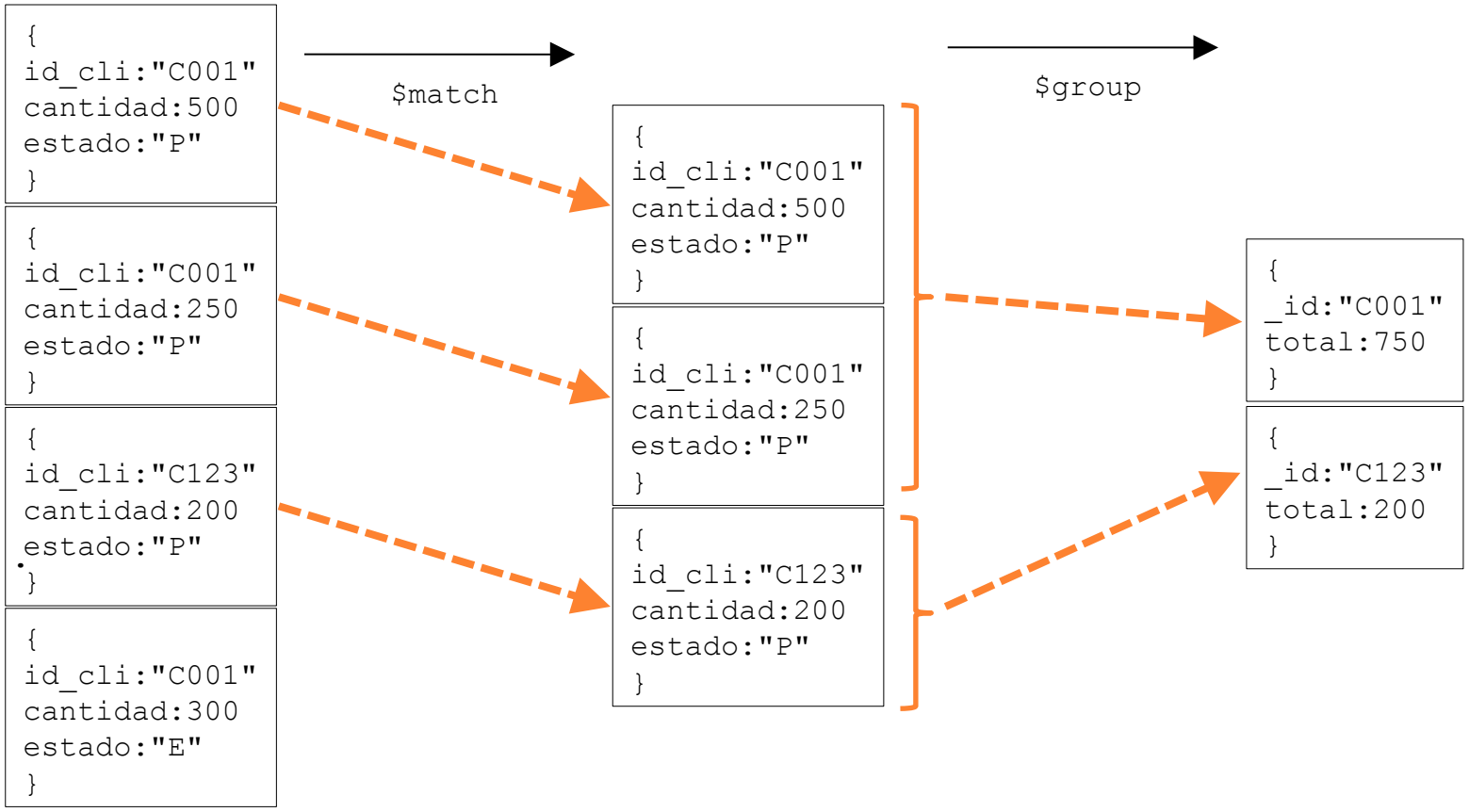
- Las etapas pueden incluir expresiones con operadores que reciban un argumento o un array de argumentos:

  `{<operador>:[<argumento1>, <argumento2>...]}`

  - $and, $or, $not, $setEquals, $setIntersection, $setUnion, $setDifference, $setIsSubset, $cmp, $eq, $gt, $abs, $add, $substract, $concat, $toUpper, $size…
  - Acumuladores: en la etapa $group, algunos también en $project: $sum, $avg, $first, $last, $max, $min, $push, $addToSet, …
- Cada operador puede ser 1:1, 1:0..1, 1:0..n, n:1,…

Aggregation Framework

```
db.ventas.aggregate([
{$match:{estado:"P"}},
{$group:{_id:"$id_cli",total:{$sum:"$cantidad"}}} ])
```



Datos de Ejemplo

```
mongoimport –host <host><:puerto>
   --db <base de datos>
   --collection <colección>
   --file <fichero>

mongoimport
   --db bdnc
   --collection provincias
   --file "provincias.json"
```

Aggregation Framework

code URJC

- Las expresiones utilizadas en un pipeline no pueden hacer referencia a datos en otros documentos.
- Las expresiones no tienen estado. Los acumuladores mantienen su estado.
- Evolución de operadores según las versiones de MongoDB:

2.4	2.6	3.2	3.4	3.6
$project	$out	$sample	$collStats	$currentOp
$match	$redact	$lookup	$facet	$listLocalSessions
$limit		$indexStats	$bucket	$listSessions
$skip			$bucketAuto	
$unwind			$sortByCount	
$group			$addFields	
$sort			$replaceRoot	
$geoNear			$count	
			$graphLookup	

Operadores

- $sample
  - Devuelve un conjunto de documentos al azar

```
db.libros.aggregate(
        [{$sample: { size: <número>}}])
```

- Ejemplo:
  - Obtener una provincia al azar.

```
db.provincias.aggregate([{$sample:{size:1}}]).pretty()
```

Operadores

- $match
  - Filtra los documentos que cumplen las condiciones.

    ```
    { $match: { <consulta> } }
    ```
  - La sintaxis de la consulta es como una lectura.
  - Conviene ponerla al principio.
  - Ejemplos

    ```
    db.libros.aggregate([
       {$match : {autor : "Cervantes"}} ])
    db.provincias.aggregate ([
       {$match:{Superficie:{$lt:100}}} ])
    ```
- Ejemplo:
  - Obtener dos provincias de Andalucía al azar.

    ```
    db.provincias.aggregate ([ {$match:{CA:"Andalucía"}},{$sample:{size:2}}])
    ```

Operadores

- $project
  - Es un campo para “proyectar”, es decir, pasan a la siguiente etapa los campos especificados.
  - Los campos pueden ser preexistentes o creados en esta etapa.

    ```
    {$project: { <especificaciones de campos> }}
    ```
  - Se pueden añadir o eliminar campos (incluido el _id).
  - Se pueden incluir expresiones calculadas utilizando los operadores $add, $subtract, $multiply, $divide, $mod.

Operadores

- $project
  - Sintaxis:
    - <campo>: <1 o true> -- se incluye (por defecto, no).
    - <campo>: <0 o false> -- se quita (e impide cualquier otro tipo de especificación, salvo excluir campos, en este caso por defecto sí aparecen).
    - _id: <0 o false> -- se suprime el _id (por defecto, sí).
    - <campo>: <expresión> -- añade o resetea el valor.
    - En embebidos, “a.b.c” : <1/0/exp> o a:{b:{c: <1/0/exp>}}.
    - También se pueden quitar campos de manera opcional con una sentencia $if.

Operadores

- $project
  - Ejemplos

```
db.provincias.aggregate ([
   {$project:{"Nombre":1}} ])
db.provincias.aggregate ([
   {$project:{"Nombre":1,"_id":0}} ])
db.provincias.aggregate ([
   {$project:{"Provincia":"$Nombre","_id":0}} ])
```

- Ejemplo:
  - Nombre (identificador) y superficie de 3 provincias de Andalucía.

```
db.provincias.aggregate([ {$match:{CA:"Andalucía"}},
    {$sample:{size:3}},
    {$project:{_id:"$Nombre", Superficie:1}}     ])
```

Operadores

Operaciones en $project

Datos:

```
{
  empleado:"Ana",
  ventas:1000,
  compras:500
},
{
  empleado:"Pedro",
  ventas:650,
  Compras:350
},
{
  empleado:"María",
  ventas:1100,
  compras:900
}
```

- Comando:

```
db.datos.aggregate([
  {$project:
    {"_id":0,
     "empleado":1,
     "beneficio":
      {"$subtract":["$ventas","$compras"]}}
  }])
```

- Resultado:

```
{ "empleado" : "Ana", "beneficio" : 500 }
{ "empleado" : "Pedro", "beneficio" : 300 }
{ "empleado" : "María", "beneficio" : 200 }
```

Operadores

- $sort
  - Ordena los documentos

```
{$sort: { <campo1>: <ordenación>, <campo2>: <ordenación> ... } }
```

  - Ejemplo

```
db.provincias.aggregate ([
  {$sort:{Superficie:1}},
  {$project:{"Nombre":1}} ])
```

- Ejemplo:
  - Nombre (identificador) y superficie de 5 provincias de Andalucía al azar ordenadas por superficie.

```
db.provincias.aggregate([{$match:{CA:"Andalucía"}},
    {$sample:{size:5}},{$project:{_id:"$Nombre", Superficie:1}},
    {$sort:{Superficie:1}}
    ])
```

Operadores

- $limit
  - Limita el número de documentos que pasan a la siguiente etapa, sin afectar al contenido.

    ```
    { $limit: <entero positivo> }
    ```
  - Ejemplo

    ```
    db.libros.aggregate( { $limit : 5 });
    ```
- Ejemplo (optimización $sort + $limit):
  - Nombre (identificador) y superficie de las 5 provincias de Andalucía más extensas ordenadas por superficie.

    ```
    db.provincias.aggregate([{$match:{CA:"Andalucía"}},
     {$project:  {_id:"$Nombre",Superficie:1}},{$sort:{Superficie:-1}},{$limit:5}])
    ```
- $skip
  - Ignora los primeros n documentos.

    ```
    { $skip: <entero positivo> }
    ```
  - Ejemplo

    ```
    db.libros.aggregate({ $skip : 5 });
    ```

Operadores

- $out
  - Escribe la salida a una colección.
  - Debe ser la última del pipeline.

```
{ $out: "<colección de salida>" }
```

  - Si la operación falla, la colección no se crea.
  - Si la colección ya existe, se sustituye por la nueva.
- Ejemplo:
  - Colección que contenga documentos con el nombre y la superficie de las provincias de Castilla-La Mancha.

```
db.provincias.aggregate([{$match:{CA:"Castilla-La Mancha"}},
{$project:{Nombre:1,Superficie:1,_id:0}},
{$out:"CastillaLM"}])
```

Operadores

- $group
  - Agrupa los documentos de acuerdo a alguna expresión. El documento de salida contiene un _id que sirve como campo de agrupación. El resultado puede incorporar campos calculados mediante algún acumulador.

```
{$group: {_id: <expresión>, <campo1>:
{ <acumulador1> : <expresión1> }, ... }}
```

  - El campo _id es obligatorio. Si se pone null, agrupa todos los documentos.
  - Acumuladores: $sum, $avg, $first , $last, $max, $min, $push, $addToSet…

Operadores

Ejemplo de $group

Datos:

```
{
 estado: "New York",
 areaM: 218,
 region: "North East”}
{
 estado: "New Jersey",
 areaM: 90,
 region: "North East”}
{
 estado: "California",
 areaM: 300,
 region: "West”}
```

- Comando:

```
db.coleccion.aggregate([
  {$group:
     {_id: "$region",
      avgAreaM: {$avg: "$areaM"},
      estados:{$push:"$estado"}}}])
```

- Resultado:

```
{ _id: "Noth East",
  avgAreaM: 154,
  estados:["New York","New Jersey"]
}
{ _id: "West",
  avgAreaM: 300,
  estados:["California"]
}
```

Operadores

- El modificador $first de $group devuelve el primer elemento del grupo; $last el último.
  - Ejemplo: primera provincia en orden alfabético de cada comunidad.

```
db.provincias.aggregate([
    {$sort:{Nombre:1}},
    {$group:{_id:"$CA","Primera":{$first:"$Nombre"}}}])
```

  - Ejemplo: última provincia en orden alfabético de cada comunidad.

```
db.provincias.aggregate([
    {$sort:{Nombre:1}},
    {$group:{_id:"$CA","Última":{$last:"$Nombre"}}}])
```

Operadores

- $group
  - Ejemplo de función de agregación:

```
db.provincias.aggregate ([
  {$group:{"_id":"$CA",
           "media": {"$avg":"$Superficie"}}}])
```

- Ejemplos:
  - Superficie y número de provincias total.

```
db.provincias.aggregate([{$group:{"_id":null, "totalProv":
{$sum:1}, "totalSup":{$sum:"$Superficie"}}}])
```

  - Nueva colección con un documento por cada Comunidad, que contenga el número de provincias, el total de superficie y un array con las provincias.

```
db.provincias.aggregate([{$group:{_id:"$CA", numProvs:{$sum:1},
totalSup:{$sum:"$Superficie"}, provs:{$push:"$Nombre"}}},
{$out:"CAconProvincias"}])
```

Operadores

- $unwind
  - Descompone un array en tantos documentos como elementos tenga el array.

```
{$unwind: <campo>}
```

  - Si no es un array, lo trata como un array de 1.
  - Si está vacío o no existe, lo ignora.
  - Ejemplo:

```
{"_id": 1, "item": "ABC1", tallas: ["S","M","L"]}

db.inventory.aggregate([{$unwind : "$tallas" }])

{ "_id" : 1, "item" : "ABC1", "tallas" : "S" }
{ "_id" : 1, "item" : "ABC1", "tallas" : "M" }
{ "_id" : 1, "item" : "ABC1", "tallas" : "L" }
```

Operadores

- $unwind

```
db.provincias.aggregate([{$unwind: "$Datos"}])

// Población en 2015 de todas las provincias (con CA)
db.provincias.aggregate([
   {$unwind: "$Datos"},
   {$match: {"Datos.Anyo":2015}},
   {$project:
     {"Nombre":1,
      "CA":1,
      "Población a 2015":"$Datos.Valor"}}])
```

- Ejemplo:
  - Una colección con un documento por provincia que contenga la población del último año guardado (sin asumir que es el 2015).

```
db.provincias.aggregate([ {$unwind: "$Datos"},{$sort:{"Datos.Anyo":1}},
     {$group:{_id:"$Nombre", "Poblacion":{$last:"$Datos.Valor"},
              "Anyo":{$last:"$Datos.Anyo"}}} ])
```

Operadores

- $unwind
  - Sintaxis alternativa (a partir de versión 3.2):

```
{$unwind:{path: <campo>, includeArrayIndex: <string>,
  preserveNullAndEmptyArrays: <boolean>}}
```

  - Donde:
    - campo: cadena que identifica un array: “$Datos”
    - includeArrayIndex : nombre del campo que contendrá el índice del valor que desagrega. Si no es un array, tendrá un null.
    - preserveNullAndEmptyArrays Por defecto, es falso, y no incluye los valores de la colección en el caso de que <campo> no contenga nada. Si vale true, los incluye. Permite simular un “EXTERNAL JOIN”.
  - Ejemplos:

```
db.provincias.aggregate([
    {$unwind:{path:"$Datos",includeArrayIndex:"indice"}},
    {$match:{indice:2}}])
db.provincias.aggregate([
    {$unwind:{path:"$CA",includeArrayIndex:"arrayIndex",
              preserveNullAndEmptyArrays:true}},
    {$project:{Datos:0}}])
```

Operadores

- $sortByCount
  - Agrupa los documentos y los cuenta, devolviendo en el resultado el _id correspondiente a la agrupación y un valor con el número de documentos.

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

  - Es equivalente a:

```
{ $group: { _id: <expression>, count:
{ $sum: 1 } } },
{ $sort: { count: -1 } }
```

Operadores

- $sortByCount
  - Ejemplo. Dados los siguientes documentos:

```
{ "_id" : 1, "title" : "The Pillars of Society", "artist" :
"Grosz", "year" : 1926, "tags" : [ "painting", "Expressionism" ] }
{ "_id" : 2, "title" : "Melancholy III", "artist" : "Munch",
"year" : 1902, "tags" : [ "woodcut", "Expressionism" ] }
{ "_id" : 3, "title" : "Dancer", "artist" : "Miro", "year" : 1925,
"tags" : [ "oil", "Surrealism", "painting" ] }
{ "_id" : 4, "title" : "The Persistence of Memory", "artist" :
"Dali", "year" : 1931, "tags" : [ "Surrealism", "painting",
"oil" ] }
{ "_id" : 5, "title" : "Composition VII", "artist" : "Kandinsky",
"year" : 1913, "tags" : [ "oil", "painting", "abstract" ] }
{ "_id" : 6, "title" : "The Scream", "artist" : "Munch", "year" :
1893, "tags" : [ "Expressionism", "painting", "oil" ] }
```

Operadores

- $sortByCount
  - La siguiente consulta

```
db.coleccion.aggregate([
   {$unwind: "$tags"},{$sortByCount: "$tags"} ])
```

  - Devuelve el siguiente resultado:

```
{ "_id" : "painting", "count" : 5 }
{ "_id" : "oil", "count" : 4 }
{ "_id" : "Expressionism", "count" : 3 }
{ "_id" : "Surrealism", "count" : 2 }
{ "_id" : "abstract", "count" : 1 }
{ "_id" : "woodcut", "count" : 1 }
```

- Ejemplo:
  - Número de provincias de cada CA.

```
db.provincias.aggregate([{$sortByCount:"$CA"}])
```

Operadores

- $addFields
  - Añade nuevos campos a documentos, manteniendo lo que existen.
  - Es equivalente a un $project que mantenga todos los campos y añada otros nuevos.
  - Formato:

```
{ $addFields: { campo: <expresión>, ... } }
```

  - Si el campo ya existe (incluyendo _id), se sobreescribe.

Operadores

- $addFields
  - Ejemplo, colección alumnos

```
{_id: 1,
 estudiante: "Juan",
 deberes: [ 10, 5, 10 ],
 pruebas: [ 10, 8 ]}
{_id: 2,
 estudiante: "María",
 deberes: [ 5, 6, 5 ],
 pruebas: [ 8, 8 ]}
```

  - Si ejecutamos:

```
db.alumnos.aggregate([
   {$addFields: {totaldeberes: { $sum: "$deberes" },
                 totalpruebas: { $sum: "$pruebas" }}},
   {$addFields: {notaFinal: {$add:["$totaldeberes",
                                   "$totalpruebas"]}}}])
```

Operadores

- $addFields
  - Resultado

```
{"_id" : 1,
 "estudiante" : "Juan",
 "deberes" : [ 10, 5, 10 ],
 "pruebas" : [ 10, 8 ],
 "totaldeberes" : 25,
 "totalpruebas" : 18,
 "notaFinal" : 43}
{ "_id" : 2,
  "estudiante" : "María",
  "deberes" : [ 5, 6, 5 ],
  "pruebas" : [ 8, 8 ],
  "totaldeberes" : 16,
  "totalpruebas" : 16,
  "notaFinal" : 40}
```

Operadores

- $addFields
  - Se pueden añadir campos a un documento embebido:

```
{ _id: 1, tipo: "coche", especificaciones: { puertas: 4, ruedas: 4 } }
{ _id: 2, tipo: "motocicleta", especificaciones: { puertas: 0, ruedas: 2 } }
{ _id: 3, tipo: "moto de agua" }
```

  - Ejemplo:

```
db.vehículos.aggregate([
    {$addFields: {"especificaciones.combustible": "sin plomo"}} ] )
```

  - Resultado:

```
{ _id: 1, tipo: "coche",
   especificaciones: { puertas: 4, ruedas: 4, combustible: "sin plomo" } }
{ _id: 2, tipo: "motocicleta",
   especificaciones: { puertas: 0, ruedas: 2, combustible: "sin plomo" } }
{ _id: 3, tipo: "moto de agua",
   especificaciones: { combustible: "sin plomo" } }
```

  - Se puede reescribir un campo, incluido el _id

```
db.vehículos.aggregate([
  {$addFields: {_id : "$tipo",
                tipo: "vehículo"}} ])
```

- Ejemplo:
  - Campo que contenga el nombre de la provincia y el de la CA ($concat).

```
db.provincias.aggregate([{$project:{_id:"$Nombre",CA:1,Superficie:1}},{$addFields:
{nombreCompleto:{$concat:["$_id",", ","$CA"]}}}]).pretty()
```

Operadores

- $count
  - Devuelve un documento que contiene el número de documentos que entran en la etapa.

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

  - <string> es el nombre del campo que contendrá la salida.
  - Ejemplo: colección "notas" con los siguientes documentos:

```
{"_id" : 1, "asignatura" : "Historia", "nota" : 45 }
{"_id" : 2, "asignatura" : "Historia", "nota" : 79 }
{"_id" : 3, "asignatura" : "Historia", "nota" : 97 }
```

  - Ejemplo:

```
db.notas.aggregate(
  [{ $match: {nota: {$gt: 50}}},
   { $count: "aprobados"}
  ])
```

  - Resultado

```
{"aprobados" : 2}
```

- Ejemplo:
  - Número total de provincias utilizando $count.

```
db.provincias.aggregate([{$count:"TotalProvincias"}])
```

Operadores

- $bucket
  - Clasifica los documentos en grupos (buckets) en función de los rangos especificados.
    - [lim1,lim2), [lim2,lim3), …
  - Si se omite el output, se incluye un campo count.

```
db.colección.aggregate ([
  {$bucket:
     {groupBy: <expresión>,
      boundaries: [ lim1, lim2 …],
      default: <literal para fuera de rango>,
      output:{campo1: {<expresión con acum>},
              campo2: {<expresión con acum>}…
  }}}])
```

Operadores

- $bucket
  - Ejemplo: agrupación de provincias en función de su superficie.

```
db.provincias.aggregate ([
    {$bucket:
       {groupBy: "$Superficie",
        boundaries: [ 0, 1000, 10000],
        default: "Otros",
        output:{"numProvincias": {$sum: 1 },
                "provincias": {$push:"$Nombre"}
     }}}])
```

Operadores

- $bucketAuto
  - Agrupa por rangos de valores automáticos según el número de buckets.
  - Devuelve los valores mínimos y máximos de cada bucket.
  - Ejemplo:

```
db.provincias.aggregate([
  {$bucketAuto:
     {groupBy: "$Superficie",
      buckets:4,
      output:{"numProvincias":{$sum: 1 },
              "provincias":{$push:"$Nombre"}
}}}])
```

Operadores

- $lookup
  - Permite hacer joins.
  - Añade un array a cada documento para aquellos elementos que coinciden con la colección enlazada.

```
{$lookup:
 {from: <colección2 a enlazar>,
  localField: <campo de colección origen>,
  foreignField: <campo de colección "from" >,
  as: <array de salida>
 }}
```

Operadores

- $lookup
  - Ejemplo

```
db.asignatura.insertMany([
    {num : 1, nombre: "mongodb"},
    {num : 2, nombre: "relacional"}])
db.temario.insertMany([
    {asig : 1, tema: "CRUD"},
    {asig : 1, tema : "Aggregation Framework"}])
```

  - Se mezclan las asignaturas con los temarios:

```
db.asignatura.aggregate([
  {$lookup:{from : "temario",
            localField : "num",
            foreignField : "asig",
            as : "temasdeasignatura"}} ])
```

Operadores

- $lookup
  - Resultado:

```
{
    "_id" : ObjectId("5a992b0086f7389d38d0dabb"),
    "num" : 1.0,
    "nombre" : "mongodb",
    "temasdeasignatura" : [
        {   "_id" : ObjectId("5a992b0086f7389d38d0dabd"),
            "asig" : 1.0,
            "tema" : "CRUD"},
        {   "_id" : ObjectId("5a992b0086f7389d38d0dabe"),
            "asig" : 1.0,
            "tema" : "Aggregation Framework"}
]}
{
    "_id" : ObjectId("5a992b0086f7389d38d0dabc"),
    "num" : 2.0,
    "nombre" : "relacional",
    "temasdeasignatura" : []
}
```

Operadores

- $lookup
  - Ejemplo:

```
db.provincias.aggregate ([
    {$group:{"_id":"$CA",
             "numProvincias":{$sum:1}}},
    {$out: "Comunidades"}])

db.provincias.aggregate([
    {$project:{Nombre:1,CA:1}},
    {$out:"soloProvincias"}])
```

Operadores

- $graphLookup
  - Sirve para hacer búsquedas recursivas, incluyendo filtros y profundidad de la recursión.

```
{
   $graphLookup: {
      from: <collection>,
      startWith: <expression>,
      connectFromField: <string>,
      connectToField: <string>,
      as: <string>,
      maxDepth: <number>,
      depthField: <string>,
      restrictSearchWithMatch: <document>
   }
}
```

Operadores

- $graphLookup
  - from: colección con la que se hará el join (puede ser la misma que a la que aplica la función).
  - startWith: valor o array de valores pertenecientes a la colección a la que se aplica, que se igualará con connectToField para iniciar la recursividad.
  - connectFromField: nombre del campo o array de campos de la colección From que se igualará con el campo connectToField. En relacional sería la clave ajena.
  - connectToField: nombre del campo de la colección From que deberá coincidir con el especificado en connectFromField. En relacional sería el destino de la clave ajena.
  - as: nombre del campo array que se añadirá a cada documento de salida y que contiene los documentos alcanzados (no necesariamente ordenados).
  - maxDepth: opcionalmente, el máximo nivel de recursividad.
  - depthField: opcionalmente, nombre del campo que contendrá la profundidad alcanzada en cada caso.
  - restrictSearchWithMatch: opcionalmente, un documento con algunas opciones adicionales (similares a la sintaxis de una consulta).

Operadores

- $graphLookup
  - Dada la colección empleados

```
{ "_id" : 1, "nombre" : "Mariano" }
{ "_id" : 2, "nombre" : "María", "jefe" : "Mariano" }
{ "_id" : 3, "nombre" : "Luis", "jefe" : "María" }
{ "_id" : 4, "nombre" : "Andrés", "jefe" : "María" }
{ "_id" : 5, "nombre" : "Ana", "jefe" : "Andrés" }
```

  - Ejemplo con la misma colección: jerarquía de jefes de cada empleado.

```
db.empleados.aggregate([
   {$graphLookup: {
         from: "empleados",
         startWith: "$jefe",
         connectFromField: "jefe",
         connectToField: "nombre",
         as: "jerarquia"
   }}])
```

Operadores

- $graphLookup
  - Resultado

```
{    "_id" : 1.0,
     "nombre" : "Mariano",
     "jerarquia" : []}
{    "_id" : 2.0,
     "nombre" : "María",
     "jefe" : "Mariano",
     "jerarquia" : [{"_id" : 1.0,
          "nombre" : "Mariano"}]}
{    "_id" : 3.0,
     "nombre" : "Luis",
     "jefe" : "María",
     "jerarquia" : [{ "_id" : 1.0,
           "nombre" : "Mariano"},
         { "_id" : 2.0,
           "nombre" : "María",
           "jefe" : "Mariano"}]}
{    "_id" : 4.0,
     "nombre" : "Andrés",
     "jefe" : "María",
     "jerarquia" : [{ "_id" : 1.0,
           "nombre" : "Mariano"},
         { "_id" : 2.0,
           "nombre" : "María",
           "jefe" : "Mariano"}]}
...
```

Operadores

- $graphLookup
  - aeropuertos:

```
{ "_id" : 0, "nomAeropuerto" : "JFK", "conecta" : [ "BOS", "ORD" ] }
{ "_id" : 1, "nomAeropuerto" : "BOS", "conecta" : [ "JFK", "PWM" ] }
{ "_id" : 2, "nomAeropuerto" : "ORD", "conecta" : [ "JFK" ] }
{ "_id" : 3, "nomAeropuerto" : "PWM", "conecta" : [ "BOS", "LHR" ] }
{ "_id" : 4, "nomAeropuerto" : "LHR", "conecta" : [ "PWM" ] }
```

  - viajeros:

```
{ "_id" : 1, "nombre" : "Mariano", "aeropuertoCercano" : "JFK" }
{ "_id" : 2, "nombre" : "María", "aeropuertoCercano" : "JFK" }
{ "_id" : 3, "nombre" : "Luis", "aeropuertoCercano" : "BOS" }
```

  - Ejemplo con varias colecciones → Destinos posibles de cada viajero con una escala.

```
db.viajeros.aggregate([
   {$graphLookup: {
         from: "aeropuertos",
         startWith: "$aeropuertoCercano",
         connectFromField: "conecta",
         connectToField: "nomAeropuerto",
         maxDepth: 1,
         depthField: "numConexiones",
         as: "destinos"} }])
```

Operadores

- $graphLookup
  - Resultado:

```
{ "_id" : 1, "nombre" : "Mariano", "aeropuertoCercano" : "JFK",  "destinos" : [
      { "_id" : 0, "nomAeropuerto" : "JFK", "conecta" : [ "BOS", "ORD" ],
        "numConexiones" : NumberLong(0) },
      { "_id" : 2, "nomAeropuerto" : "ORD", "conecta" : [ "JFK" ],
        "numConexiones" : NumberLong(1) },
      { "_id" : 1, "nomAeropuerto" : "BOS", "conecta" : [ "JFK", "PWM" ],
        "numConexiones" : NumberLong(1) } ] }
{ "_id" : 2, "nombre" : "María", "aeropuertoCercano" : "JFK", "destinos" : [
      { "_id" : 0, "nomAeropuerto" : "JFK", "conecta" : [ "BOS", "ORD" ],
        "numConexiones" : NumberLong(0) },
      { "_id" : 2, "nomAeropuerto" : "ORD", "conecta" : [ "JFK" ],
        "numConexiones" : NumberLong(1) },
      { "_id" : 1, "nomAeropuerto" : "BOS", "conecta" : [ "JFK", "PWM" ],
        "numConexiones" : NumberLong(1) } ] }
{ "_id" : 3, "nombre" : "Luis", "aeropuertoCercano" : "BOS", "destinos" : [
      { "_id" : 0, "nomAeropuerto" : "JFK", "conecta" : [ "BOS", "ORD" ],
        "numConexiones" : NumberLong(1) },
      { "_id" : 3, "nomAeropuerto" : "PWM", "conecta" : [ "BOS", "LHR" ],
        "numConexiones" : NumberLong(1) },
      { "_id" : 1, "nomAeropuerto" : "BOS", "conecta" : [ "JFK", "PWM" ],
        "numConexiones" : NumberLong(0) } ] }
```

Limitaciones del Aggregation Pipeline

- Restricciones en el tamaño del resultado
  - Cada documento del resultado está limitado a 16 Mb (BSON document size).
  - Los resultados intermedios pueden exceder ese límite.
- Restricciones de memoria
  - Cada etapa del pipeline tiene un límite de 100 Mb de RAM. Para evitarlo, se puede usar la opción *allowDiskUse* para permitir la escritura en ficheros temporales.
- https://docs.mongodb.com/manual/core/aggregation-pipeline-limits/

Mejoras en Rendimiento

- Otros aspectos:
  - Los operadores $match y $sort pueden aprovechar la existencia de índices cuando aparecen al principio del pipeline.
    - Los índices no se utilizan en los resultados cargados en memoria.
  - Si la operación de agregación requiere solo un subconjunto de los datos de la colección, conviene utilizar $match, $limit, y $skip para restringir los documentos al principio del pipeline, para optimizar el uso de índices.
  - El pipeline tiene una fase interna de optimización que proporciona un rendimiento mejorado para ciertas secuencias de operadores. Ej: $sort + $match, $project+$match, ...
    - https://docs.mongodb.com/manual/core/aggregation-pipeline-optimization/

SQL vs Aggregation Framework

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM()	$sum
COUNT()	$sum $sortByCount
join	$lookup
SELECT INTO NEW_TABLE	$out
MERGE INTO TABLE	$merge (Available starting in MongoDB 4.2)

- Fuente: https://docs.mongodb.com/manual/reference/sql-aggregation-comparison/

Vistas

- Mongodb soporta desde su versión 3.4 la creación de vistas de sólo lectura.

```
db.createView(<view>, <source>, <pipeline>, <options>)
```

  - Donde
    - <view>: el nombre de la vista
    - <source>: colección o vista en la que se basa
    - <pipeline>: array con las etapas de un pipeline
    - <options> documento con opciones adicionales
- La definición de la vista es pública:

```
db.getCollectionInfos()
```

Vistas

- Ejemplo

```
db.createView(
"pruebaVistaCA",
"provincias",
[  {$match:{CA:"Castilla-La Mancha"}},
   {$project:{_id:"$Nombre",Superficie:1}}  ])
```

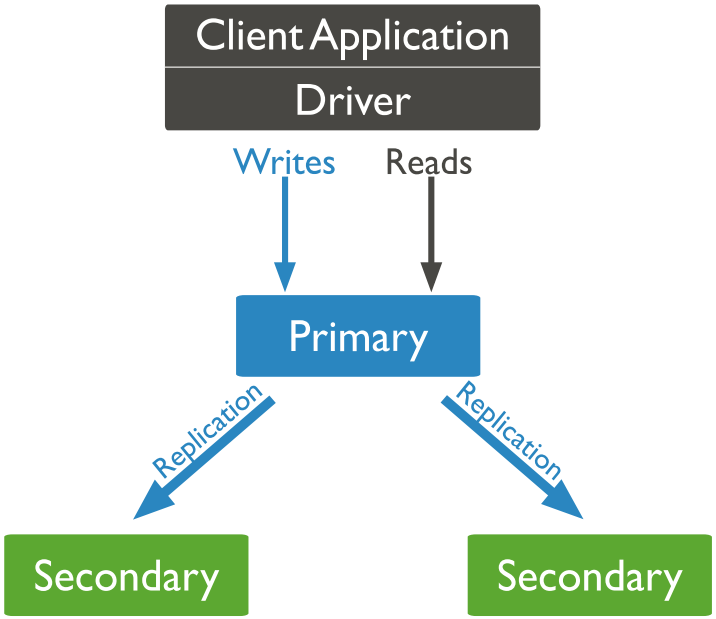
  - Son de sólo lectura y se ejecutan cuando se ejecuta una lectura sobre ellas.

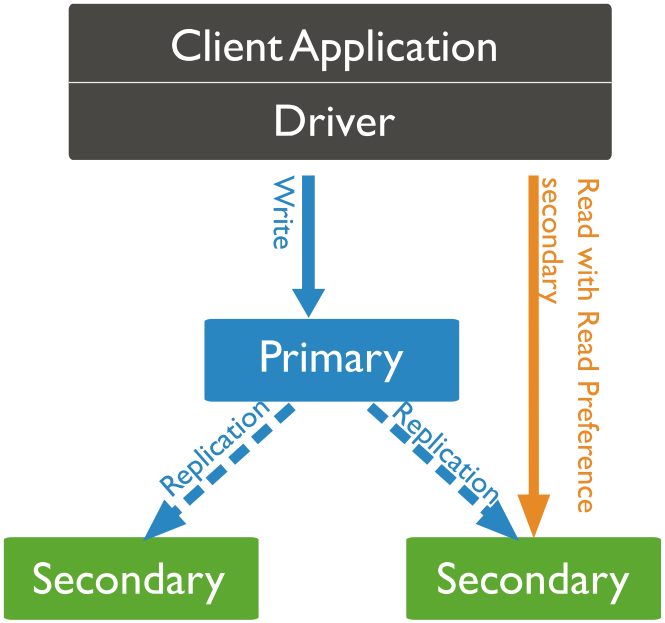
```
// Consulta
db.pruebaVistaCA.find()
// Info sobre las colecciones:
db.getCollectionInfos()
```

Replica Sets

- Grupo de instancias de MongoDB que contienen los mismos datos (son réplicas unas de otras).
- Todo conjunto de réplicas tiene un (y sólo un) primario. El resto de bases de datos de la réplica funcionan como secundarios.
  - La elección del primario se hace de forma automática.
  - Número impar de miembros (puede incluir un árbitro).
  - Todas las escrituras van al primario.
- Proporcionan redundancia y alta disponibilidad.
  - Dado que los datos se replican asíncronamente, se pueden leer datos antiguos de las réplicas.

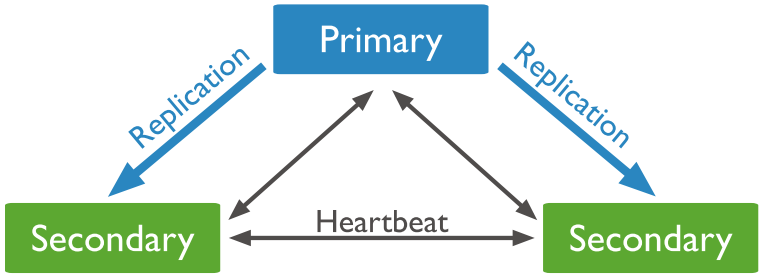
Replica Sets

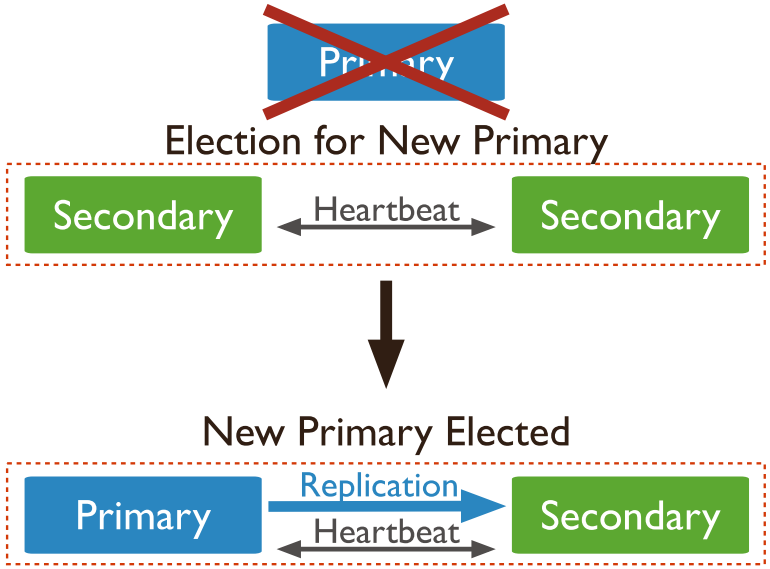




- Por defecto, todas las lecturas van al set primario, aunque se pueden establecer preferencias para acceso a sets secundarios.
  - Acceso a sets más cercanos geográficamente.
  - Mantener disponibilidad en caso de fallos.
- Fuente imágenes: https://docs.mongodb.com/manual/replication/

Replica Sets





- Las instancias de un *replica set* intercambian mensajes de estado continuamente (*heartbeat*).
- Si el primario deja de responder durante más de 10 segundos, se elige un nuevo primario entre las instancias restantes (secundarios).
- Si el primario volviera a estar disponible, se unirá de nuevo al *replica set* como secundario.
- Fuente imágenes: https://docs.mongodb.com/manual/replication/

Sharding

- *Shard* = fragmento.
- Reparto de los datos entre diferentes máquinas. Escalado horizontal:
  - Nodo A: usuarios de la A-H
  - Nodo B: usuarios de la I-O
  - Nodo C: usuarios de la P-Z
- Útil para manejo de grandes conjuntos de datos y/o aplicaciones de alto rendimiento.
  - Mejoran rendimiento de escrituras y lecturas distribuyendo los accesos siempre que no haya que leer de varios shards.
- Esfuerzo del programador
- Difícil hacer re-sharding

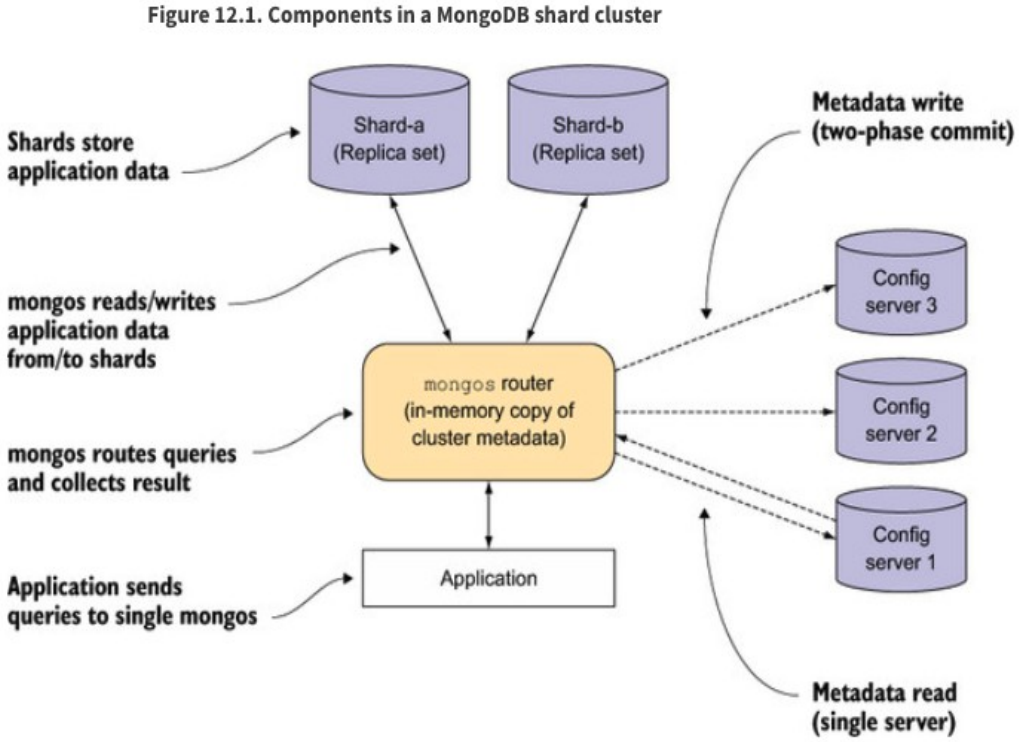
Sharding

- Escalado horizontal: reparto de los datos entre diferentes máquinas que forman un *sharded cluster*.
  - Nodo A: usuarios de la A-H
  - Nodo B: usuarios de la I-O
  - Nodo C: usuarios de la P-Z
- Útil para manejo de grandes conjuntos de datos y/o aplicaciones de alto rendimiento.
- Ventajas:
  - Distribuyen lecturas y escrituras.
  - Incrementan capacidad de almacenamiento vs replica sets.
  - Proporcionan alta disponibilidad.

Sharding

- "*It's a complex system that adds administrative and performance overhead, so make absolutely sure it's what your application needs*" (MongoDB in Action, Second Edition, Manning Publications 2016, Capítulo 12 -disponible en Safari-)
- Desventajas:
  - La *shard key* (campo utilizado para distribuir los documentos en el *sharded cluster*) no se puede cambiar.
  - La elección de la *shard key* debería generar un reparto uniforme.
  - No se puede hacer *unsharding*.
  - Si la consulta no incluye la *shard key*, se hace *broadcasting* a todos los *shards*. Puede ser una operación lenta.
- A su vez, cada *shard* puede actuar como un replica set con las colecciones no troceadas.

Sharding



- Shards: almacenan los datos.
- Mongos router: cachea los metadatos del cluster para dirigir las operaciones.
- Config servers: almacenan los metadatos del cluster, incluyendo la distrubución de datos en los shards.
- Chunk: agrupación lógica de documentos. Un shard puede tener varios chunks.
- Fuente imagen: MongoDB in Action, Second Edition, Manning Publications 2016, Capítulo 12

GridFS

- Forma de almacenar grandes ficheros en MongoDB (> 16 MB = BSON-document size limit).
- Divide el fichero en partes (chunks), almacenándolos por separado.
- GridFS almacena la información en dos colecciones:
  - Chunks (almacena los chuncks): fs.chunks
  - Files (almacena los metadatos del fichero): fs.files
- Se puede elegir un nombre de bucket diferente de fs, así como crear múltiples buckets en la misma base de datos.
- Se puede hacer sharding de GridFS con { files_id : 1, n : 1 } o { files_id : 1 } como shard key index.

GridFS

- Chunks:

```
{ "_id" : <ObjectId>,
  "files_id" : <ObjectId>,
  "n" : <num>,
  "data" : <binary> }
```

- Donde:
  - “_id”: identificador
  - “files_id”: identificador del padre (en files)
  - ”n”: número de orden
  - “data”: chunk

GridFS

- Files:

```
{
  "_id" : <ObjectId>,
  "length" : <num>,
  "chunkSize" : <num>,
  "uploadDate" : <timestamp>,
  "md5" : <hash>,
  "filename" : <string>,
  "contentType" : <string>,
  "aliases" : <string array>,
  "metadata" : <dataObject>,
}
```

GridFS

```
mongofiles <options> <commands> <filename>
```

- Donde:
  - <options>: host, puesto, usuario, bd, nombre local del fichero (sobreescribe a <filename>,…
  - <commands>: cargar un fichero, leerlo, borrarlo, buscar, consultar…
  - <filename>: nombre del fichero o identificador
  - Versión reducida:

```
mongofiles --local "ruta del fichero en local" -d
basededatos put "nombre del fichero en GridFS"

mongofiles --local "ruta del fichero en local" -d
basededatos get "nombre del fichero en GridFS"
```

GridFS

- Ejemplo

```
# En Bash
mongofiles --local "/tmp/ClaseERPs.mp4" -d bdprueba put "ClaseERPs"

# En Mongo
show databases
use bdprueba

db.fs.files.find().pretty()

db.fs.chunks.find({},{_id:1,files_id:1,n:1})

# En Bash
mongofiles --local "/tmp/ClaseERPs-copia.mp4" -d bdprueba get "ClaseERPs"
md5sum /tmp/ClaseERPs.mp4
md5sum /tmp/ClaseERPs-copia.mp4
```

Spring Data y MongoDB

- Spring Data soporta MongoDB con repositorios similares a los del modelo relacional.
- Dependencia de pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

- Las entidades a almacenar NO se anotan con la etiqueta `@Entity`.
- La clave primaria SÍ se debe anotar como `@Id`, y debe ser de tipo *String*.
- Los repositorios son interfaces que heredan de

```
MongoRepository<ObjetoGuardado, String>
```

Código en GitHub:

https://github.com/MasterCloudApps/2.3.Persistencia-y-analisis-de-datos/tree/master/tema3/ejemplo1_springdata_mongodb