



## ***NoSQL***

### **Sesión 4 - Agregaciones y Escalabilidad en *MongoDB***



## Índice

- Agregaciones
- Pipeline de Agregación
  - Operaciones
  - Agregaciones con Java
- Replicación
  - Replicación en Java
- Participando de datos (*Sharding*)



## 4.1 Agregaciones

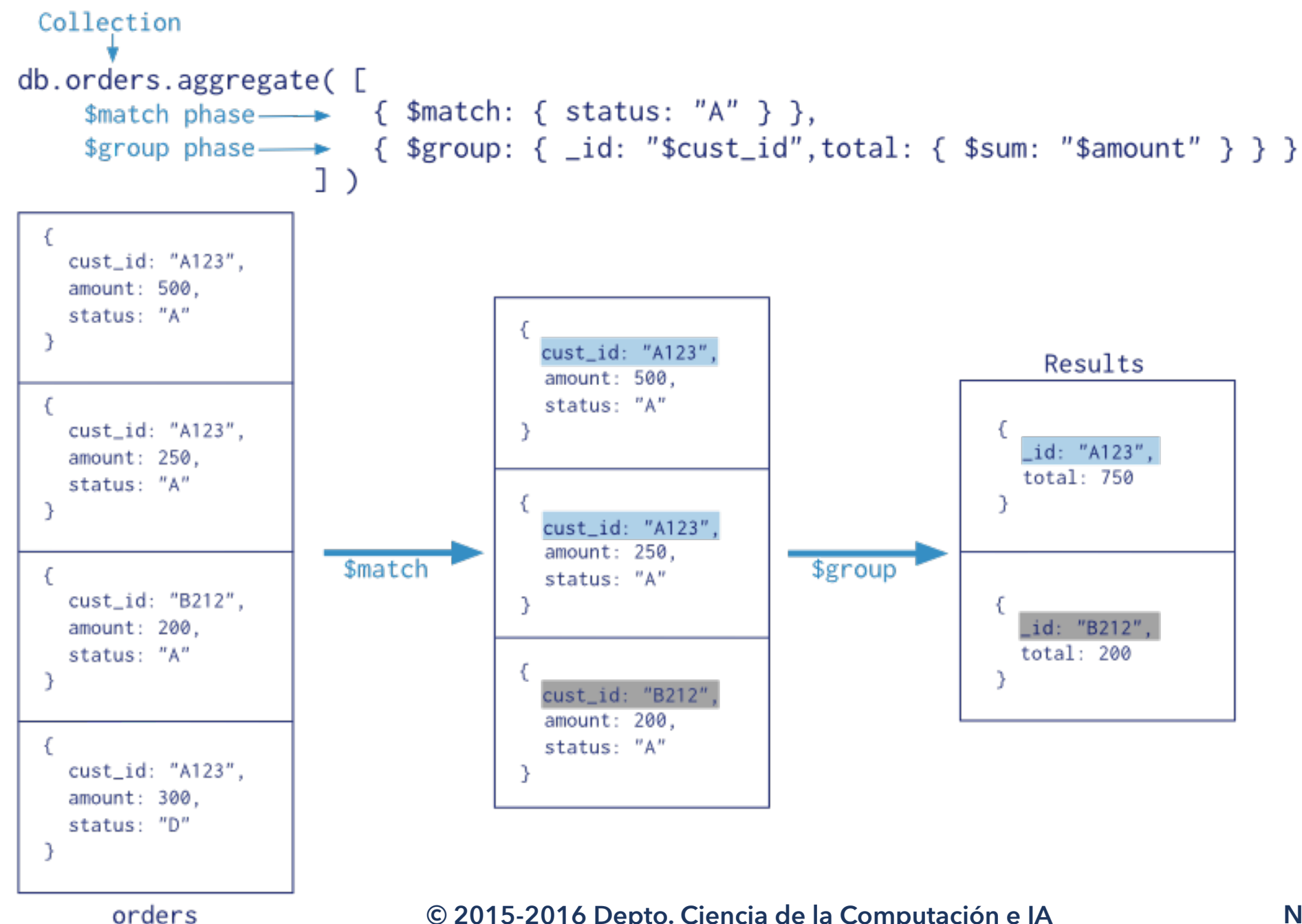
- Operaciones *map-reduce* mediante `mapreduce()`
  - Planteamiento paralelo → *Hadoop*
- ( $\geq 2.2$ ) Pipeline de agregación mediante `aggregation()`
- Operaciones de agrupación sencilla mediante `group()`, `count()` o `distinct()`
  - **group**(`key`, `reduce`, `initial`)
    - `key`: atributo por el que se van a agrupar los datos
    - `initial`: define un valor base para cada grupo de resultados.
    - `reduce`: función que agrupa los elementos similar. Recibe dos parámetros, el documento actual (`item`) sobre el cual se itera, y el objeto contador agregado (`prev`).

```
db.people.group( {
  key: { hijos: true },
  reduce: function ( item, prev ) {
    prev.total += 1;
  },
  initial: { total : 0 }
} )
```



## 4.2 Pipeline de Agregación

- `db.coleccion.aggregate([op1, op2, ... opN])`
  - Array donde cada elemento es una fase del *pipeline* → la salida de una fase es la entrada de la siguiente
- El resultado del *pipeline* es BSON → `size(BSON) < 16MB`





## Operadores del Pipeline

- Las fases se pueden repetir

| Operado          | Descripción  | Cardinalida |
|------------------|--|-------------|
| <b>\$project</b> | Proyección de campos, es decir, propiedades en las que estamos interesados. También nos permite modificar un documento, o crear un subdocumento ( <i>reshape</i> ) | 1:1         |
| <b>\$match</b>   | Filtrado de campos, similar a <i>where</i>   | N:1         |
| <b>\$group</b>   | Para agrupar los datos, similar a <i>group by</i>  | N:1         |
| <b>\$sort</b>    | Ordenar  | 1:1         |
| <b>\$skip</b>    | Saltar   | N:1         |
| <b>\$limit</b>   | Limitar los resultados   | N:1         |
| <b>\$unwind</b>  | Separa los datos que hay dentro de un array  | 1:N         |



## Preparando los ejemplos

- Colección de productos

```
> db.productos.findOne()  
{  
  "_id" : ObjectId("5345afc1176f38ea4eda4787"),  
  "nombre" : "iPad 16GB Wifi",  
  "fabricante" : "Apple",  
  "categoria" : "Tablets",  
  "precio" : 499  
}
```



## 4.2.2 \$group

- Agrupa los documentos para calcular valores agregados de una colección.
- La salida está desordenada
- N:1
- Recibe un objeto compuesto de:
  - 1er campo → `_id` → elementos de agrupación
    - Para referenciar a un campo, entre comillas poner `$` delante el nombre del campo
    - Permite un campo de la fase anterior, un documento/subdocumento o una proyección
  - Resto de campos → cálculos agregados mediante los operadores de agregación



## Ejemplos \$group

```
> db.productos.aggregate([{$group:
  {
    _id: "$fabricante",
    total: { $sum:1 }
  }
}])
{ "_id" : "Sony", "total" : 1 }
{ "_id" : "Amazon", "total" : 2 }
{ "_id" : "Google", "total" : 1 }
{ "_id" :
```

```
> db.productos.aggregate([{$group:
  {
    _id: { "empresa": "$fabricante" },
    total: { $sum:1 }
  }
}])
{ "_id" : { "empresa" : "Amazon" }, "total" : 2 }
{ "_id" : { "empresa" : "Sony" }, "total" : 1 }
{ "_id" : { "empresa" : "Samsung" }, "total" : 2 }
{ "_id" : { "empresa" : "Google" }, "total" : 1 }
{ "_id" : { "empresa" : "Apple" }, "total" : 4 }
```

```
> db.productos.aggregate([{$group:
  {
    _id: {
      "empresa": "$fabricante",
      "tipo": "$categoria" },
    total: {$sum:1}
  }
}])
{ "_id" : { "empresa" : "Amazon", "tipo" : "Tablets" }, "total" : 2 }
{ "_id" : { "empresa" : "Google", "tipo" : "Tablets" }, "total" : 1 }
{ "_id" : { "empresa" : "Apple", "tipo" : "Portátiles" }, "total" : 1 }
...
```





## Operadores de Agrupación

| Nombre            | Descripción  |
|-------------------|--|
| <b>\$addToSet</b> | Devuelve un array con todos los valores únicos para los campos seleccionados entre cada documento del grupo (sin repeticiones) |
| <b>\$first</b>    | Devuelve el primer valor del grupo. Se suele usar después de ordenar.  |
| <b>\$last</b>     | Devuelve el último valor del grupo. Se suele usar después de ordenar.  |
| <b>\$max</b>      | Devuelve el mayor valor de un grupo  |
| <b>\$min</b>      | Devuelve el menor valor de un grupo.   |
| <b>\$avg</b>      | Devuelve el promedio de todos los valores de un grupo  |
| <b>\$push</b>     | Devuelve un array con todos los valores del campo seleccionado entre cada documento del grupo (puede haber repeticiones)       |
| <b>\$sum</b>      | Devuelve la suma de todos los valores del grupo  |



## \$sum y \$avg

- Acumula los valores y devuelve la suma (`$sum`) o el promedio (`$avg`)
  - Para contar, `{ $sum: 1 }`

```
> db.productos.aggregate([ {
  $group: {
    _id: {
      "categoria": "$categoria"
    },
    precioMedio: { $avg: "$precio" }
  }
}] )
{ "_id" : { "categoria" : "Portátiles" }, "precioMedio" : 499 }
{ "_id" : { "categoria" : "Smartphones" }, "precioMedio" : 563.99 }
{ "_id" : { "categoria" : "Tablets" }, "precioMedio" : 396.4271428571428 }
```

```
> db.productos.aggregate([ {
  $group: {
    _id: {
      "empresa": "$fabricante"
    },
    totalPrecio: { $sum: "$precio" }
  }
}] )
{ "_id" : { "empresa" : "Amazon" }, "totalPrecio" : 328 }
{ "_id" : { "empresa" : "Sony" }, "totalPrecio" : 499 }
{ "_id" : { "empresa" : "Samsung" }, "totalPrecio" : 1014.98 }
{ "_id" : { "empresa" : "Google" }, "totalPrecio" : 199 }
{ "_id" : { "empresa" : "Apple" }, "totalPrecio" : 2296 }
```



## \$addToSet y \$push

- Crean un array con los valores de los campos
  - \$addToSet → sin repeticiones
  - \$push → con repeticiones

```
> db.productos.aggregate([ {
  $group: {
    _id: {
      "empresa": "$fabricante"
    },
    categorias: {$push: "$categoria"}
  }
}])
{ "_id": {"empresa": "Amazon"}, "categorias": ["Tablets", "Tablets"] }
{ "_id": {"empresa": "Sony"}, "categorias": ["Portátiles"] }
{ "_id": {"empresa": "Samsung"}, "categorias": ["Smartphones", "Tablets"] }
{ "_id": {"empresa": "Google"}, "categorias": ["Tablets"] }
{ "_id": {"empresa": "Apple"}, "categorias": ["Tablets", "Tablets", "Tablets", "Portátiles"] }
```

```
> db.productos.aggregate([ {
  $group: {
    _id: {
      "fabricante": "$fabricante"
    },
    categorias: {$addToSet: "$categoria"}
  }
}])
{ "_id": { "fabricante": "Amazon" }, "categorias": [ "Tablets" ] }
{ "_id": { "fabricante": "Sony" }, "categorias": [ "Portátiles" ] }
{ "_id": { "fabricante": "Samsung" }, "categorias": [ "Tablets", "Smartphones" ] }
{ "_id": { "fabricante": "Google" }, "categorias": [ "Tablets" ] }
{ "_id": { "fabricante": "Apple" }, "categorias": [ "Portátiles", "Tablets" ] }
```



## \$max y \$min

- Mayor y menor valor del campo por el que se agrupan

```
> db.productos.aggregate([ {
  $group: {
    _id: {
      "empresa": "$fabricante"
    },
    precioMaximo: { $max: "$precio" },
    precioMinimo: { $min: "$precio" },
  }
}] )
{ "_id" : { "empresa" : "Amazon" }, "precioMaximo" : 199, "precioMinimo" : 129 }
{ "_id" : { "empresa" : "Sony" }, "precioMaximo" : 499, "precioMinimo" : 499 }
{ "_id" : { "empresa" : "Samsung" }, "precioMaximo" : 563.99, "precioMinimo" : 450.99 }
{ "_id" : { "empresa" : "Google" }, "precioMaximo" : 199, "precioMinimo" : 199 }
{ "_id" : { "empresa" : "Apple" }, "precioMaximo" : 699, "precioMinimo" : 499 }
```



## Doble agrupación

- Se puede agrupar los datos que provienen de una agrupación
- La segunda agrupación referencia a un campo de la primera agrupación

```
> db.productos.aggregate([
  {$group: {
    _id: {
      "empresa": "$fabricante",
      "categoria": "$categoria"
    },
    precioMedio: {$avg: "$precio"}
  }},
  {$group: {
    _id: "$_id.empresa",
    precioMedio: {$avg: "$precioMedio"}
  }
}]
{ "_id" : "Samsung", "precioMedio" : 507.49 }
{ "_id" : "Sony", "precioMedio" : 499 }
{ "_id" : "Apple", "precioMedio" : 549 }
{ "_id" : "Google", "precioMedio" : 199 }
{ "_id" : "Amazon", "precioMedio" : 164 }
```

precio medio de los precios medios de los tipos de producto por empresa



## `$first` y `$last`

- Devuelven el valor resultante de aplicar la expresión al primer/ último elemento de un grupo de elementos que comparten el mismo grupo por clave

```
> db.productos.aggregate([
  {$group: {
    _id: {
      "empresa": "$fabricante",
      "tipo" : "$categoria" },
    total: {$sum:1}
  }},
  {$sort: {"total":-1}},
  {$group: {
    _id:"$_id.empresa",
    producto: {$first: "$_id.tipo"},
    cantidad: {$first: "$total"}
  }
}]
{ "_id" : "Samsung", "producto" : "Tablets", "cantidad" : 1 }
{ "_id" : "Sony", "producto" : "Portátiles", "cantidad" : 1 }
{ "_id" : "Amazon", "producto" : "Tablets", "cantidad" : 2 }
{ "_id" : "Google", "producto" : "Tablets", "cantidad" : 1 }
{ "_id" : "Apple", "producto" : "Tablets", "cantidad" : 3 }
```

para cada empresa, cual es el tipo de producto que más tiene y la cantidad de dicho tipo





## 4.2.3 \$project

- Proyección sobre el conjunto de resultados → subconjunto de los campos
- 1:1
- Mantiene el orden
- Permite:
  - Renombrar campos.
  - Introducir campos calculados en el documento resultante, mediante `$add`, `$subtract`, `$multiply`, `$divide` o `$mod`
  - Transformar campos a mayúsculas `$toUpper` o minúsculas `$toLower`, concatenar campos mediante `$concat` u obtener subcadenas con `$substr`.
  - Transformar campos en base a valores obtenidos a partir de una condición mediante expresiones lógicas con los operadores de comparación vistos en las consultas.



## Ejemplo \$project

```
> db.productos.aggregate([
  {$project:
    {
      _id:0,
      'empresa': {$toUpper: "$fabricante"},
      'detalles': {
        'categoria': "$categoria",
        'precio': {"$multiply": ["$precio", 1.1]}
      },
      'elemento': '$nombre'
    }
  }
])
{ "empresa" : "APPLE", "detalles" : { "categoria" : "Tablets", "precio" : 548.90000000000001 },
"elemento" : "iPad 16GB Wifi" }
{ "empresa" : "APPLE", "detalles" : { "categoria" : "Tablets", "precio" : 658.90000000000001 },
"elemento" : "iPad 32GB Wifi" }
```





## 4.2.4 \$match

- Filtra documentos → N:1 `db.productos.aggregate([{$match:{categoria:"Tablets"}}])`
- Similar a WHERE en SQL
- Permite utilizar los operadores de consulta: `$gt`, `$lt`, `$in`, ...
- Se recomienda utilizar en las primeras fases del *pipeline* para reducir el conjunto de resultados
- Si lo usamos en la primera fase, emplea los índices de la colección

```
> db.productos.aggregate([
  {$match:
    {categoria:"Tablets",
     precio: {$lt: 500}}},
  {$group:
    {_id: {"empresa": "$fabricante"},
     cantidad: {$sum:1}}
  }]
)
{ "_id" : { "empresa" : "Amazon" }, "cantidad" : 2 }
{ "_id" : { "empresa" : "Samsung" }, "cantidad" : 1 }
{ "_id" : { "empresa" : "Google" }, "cantidad" : 1 }
{ "_id" : { "empresa" : "Apple" }, "cantidad" : 1 }
```



## 4.2.5 \$sort

- Ordena los documentos por el campo y el orden indicado

```
db.productos.aggregate( { $sort: { precio: -1 } } )
```

- Ordena los datos en memoria → cuidado con el tamaño de los datos.
  - Se emplea en las últimas fases del *pipeline*, cuando el conjunto de resultados es el menor posible.

```
> db.productos.aggregate([
  { $match: { categoria: "Tablets" } },
  { $group:
    { _id: { "empresa": "$fabricante" },
      totalPrecio: { $sum: "$precio" } }
  },
  { $sort: { totalPrecio: -1 } }
])
{ "_id" : { "empresa" : "Apple" }, "totalPrecio" : 1797 }
{ "_id" : { "empresa" : "Samsung" }, "totalPrecio" : 450.99 }
{ "_id" : { "empresa" : "Amazon" }, "totalPrecio" : 328 }
{ "_id" : { "empresa" : "Google" }, "totalPrecio" : 199 }
```



## 4.2.6 \$skip y \$limit

- Permiten saltar o restringir un número determinado de documentos

```
db.productos.aggregate([{$limit:3}])
```

```
db.productos.aggregate([{$skip:3}])
```

- El orden importa

```
> db.productos.aggregate([{$skip:2},{$limit:4}])
{ "_id" : ObjectId("54ffff889836d613eee9a6e7"), "nombre" : "iPad 64GB Wifi",
"categoria" : "Tablets", "fabricante" : "Apple", "precio" : 699 }
{ "_id" : ObjectId("54ffff889836d613eee9a6e8"), "nombre" : "Galaxy S3", "categoria" :
"Smartphones", "fabricante" : "Samsung", "precio" : 563.99 }
{ "_id" : ObjectId("54ffff889836d613eee9a6e9"), "nombre" : "Galaxy Tab 10", "categoria"
: "Tablets", "fabricante" : "Samsung", "precio" : 450.99 }
{ "_id" : ObjectId("54ffff889836d613eee9a6ea"), "nombre" : "Vaio", "categoria" :
"Portátiles", "fabricante" : "Sony", "precio" : 499 }
```

```
> db.productos.aggregate([{$limit:4},{$skip:2}])
{ "_id" : ObjectId("54ffff889836d613eee9a6e7"), "nombre" : "iPad 64GB Wifi",
"categoria" : "Tablets", "fabricante" : "Apple", "precio" : 699 }
{ "_id" : ObjectId("54ffff889836d613eee9a6e8"), "nombre" : "Galaxy S3", "categoria" :
"Smartphones", "fabricante" : "Samsung", "precio" : 563.99 }
```



## 4.2.7 \$unwind

- Al usarlo con un campo **array** de tamaño N en un documento, lo transforma en N documentos con el campo tomando el valor individual de cada uno de los elementos del array
- Desenrolla el array → 1:N

```
> db.enlaces.findOne()  
{  
  "_id" : ObjectId("54f9769212b1897ae84190cf"),  
  "titulo" : "www.google.es",  
  "tags" : [  
    "mapas",  
    "videos",  
    "blog",  
    "calendario",  
    "email",  
    "mapas"  
  ]  
}
```

```
> db.enlaces.aggregate(  
  {$match: {titulo: "www.google.es"}},  
  {$unwind: "$tags"})  
{ "_id" : ObjectId("54f9769212b1897ae84190cf"), "titulo" : "www.google.es", "tags" : "mapas" }  
{ "_id" : ObjectId("54f9769212b1897ae84190cf"), "titulo" : "www.google.es", "tags" : "videos" }  
{ "_id" : ObjectId("54f9769212b1897ae84190cf"), "titulo" : "www.google.es", "tags" : "blog" }  
{ "_id" : ObjectId("54f9769212b1897ae84190cf"), "titulo" : "www.google.es", "tags" : "calendario" }  
{ "_id" : ObjectId("54f9769212b1897ae84190cf"), "titulo" : "www.google.es", "tags" : "email" }  
{ "_id" : ObjectId("54f9769212b1897ae84190cf"), "titulo" : "www.google.es", "tags" : "mapas" }
```



## Ejemplo \$unwind

- Permite sumar/contar elementos del array
- Tras desenrollar, volver a agrupar

```
> db.enlaces.aggregate([
  { "$unwind" : "$tags" },
  { "$group" :
    { "_id" : "$tags",
      "total" : { $sum : 1 }
    }
  },
  { "$sort" : { "total" : -1 } },
  { "$limit" : 3 }
])
{ "_id" : "mapas", "total" : 3 }
{ "_id" : "email", "total" : 2 }
{ "_id" : "calendario", "total" : 1 }
```



## 4.2.8 De SQL al *Pipeline* de Agregación

| SQL             | <i>Pipeline</i> de Agregaciones |
|-----------------|---------------------------------|
| <b>WHERE</b>    | <code>\$match</code>            |
| <b>GROUP BY</b> | <code>\$group</code>            |
| <b>HAVING</b>   | <code>\$match</code>            |
| <b>SELECT</b>   | <code>\$project</code>          |
| <b>ORDER BY</b> | <code>\$sort</code>             |
| <b>LIMIT</b>    | <code>\$limit</code>            |
| <b>SUM()</b>    | <code>\$sum</code>              |
| <b>COUNT()</b>  | <code>\$sum</code>              |



## 4.2.9 Limitaciones

- En versiones anteriores a la 2.6, el *pipeline* devolvía en cada fase un objeto BSON, y por tanto, el resultado estaba limitado a 16MB.
- Las fases tienen un límite de 100MB en memoria. Si una fase excede dicho límite, se producirá un error.
  - Solución → Habilitar el uso de disco mediante `allowDiskUse` en las opciones de la agregación (segundo parámetro, tras el *pipeline*)






## 4.3 Agregaciones en Java

- `coleccion.aggregate (List <DBObject>)`
  - Cada fase es un `DBObject`
  - Una vez tenemos las fases → `Arrays.asList()`

```
> db.productos.aggregate([
  {$match:{categoria:"Tablets"}},
  {$group:
    {_id: {"empresa":"$fabricante"},
     totalPrecio: {$sum:"$precio"}}
  },
  {$sort:{totalPrecio:-1}}
])
```



```
DBObject match = new BasicDBObject("$match", new BasicDBObject("categoria", "Tablets"));
DBObject group = new BasicDBObject("$group",
  new BasicDBObject("_id", new BasicDBObject("empresa", "$fabricante"))
  .append("totalPrecio", new BasicDBObject("$sum", "$precio")));
DBObject sort = new BasicDBObject("$sort", new BasicDBObject("totalPrecio", -1));

AggregationOutput output = coleccion.aggregate(Arrays.asList(match, group, sort));

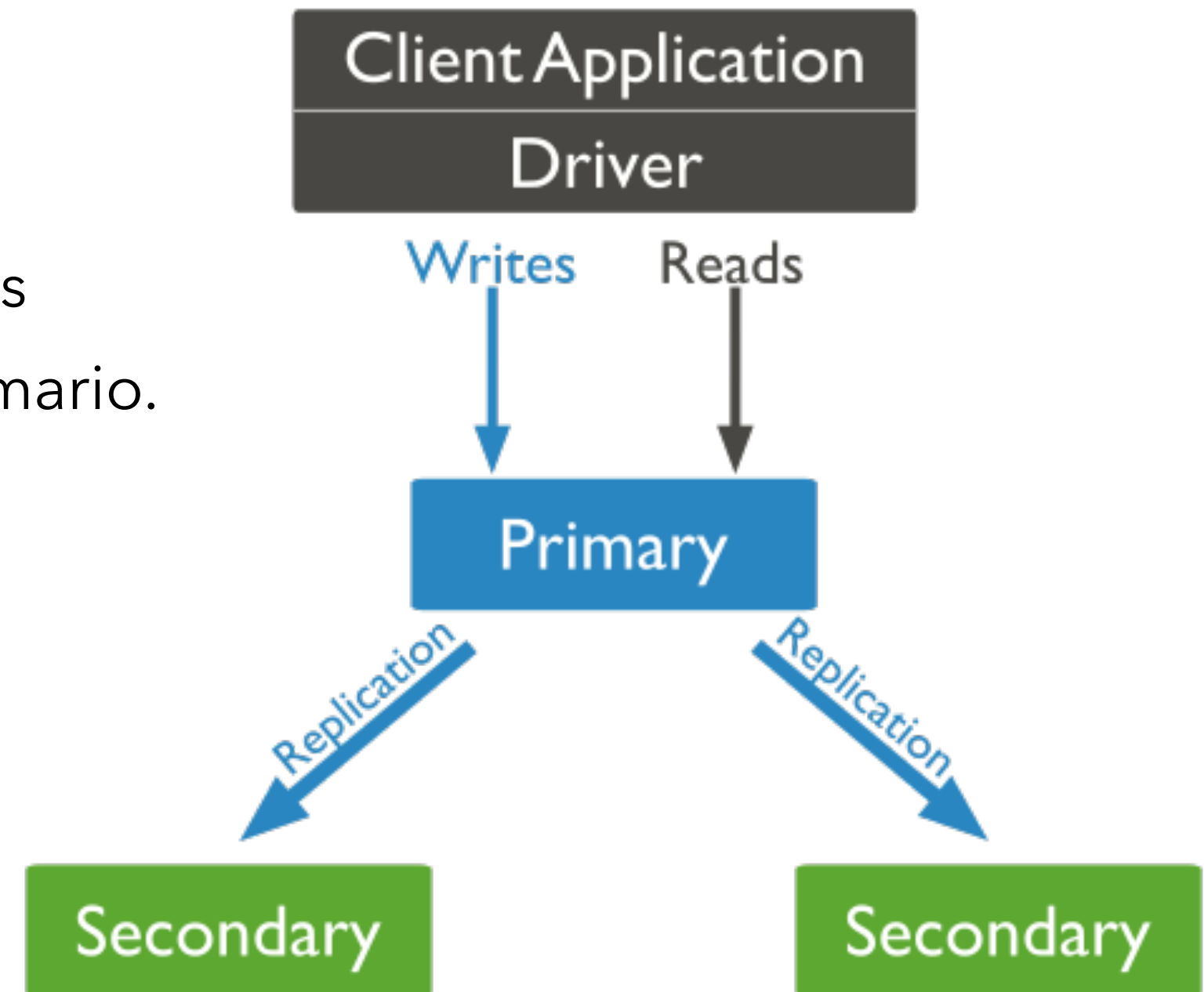
Iterable<DBObject> datos = output.results();
for (DBObject doc : datos) {
  System.out.println(doc);
}
```





## 4.4 Replicación - Conjunto de Réplicas

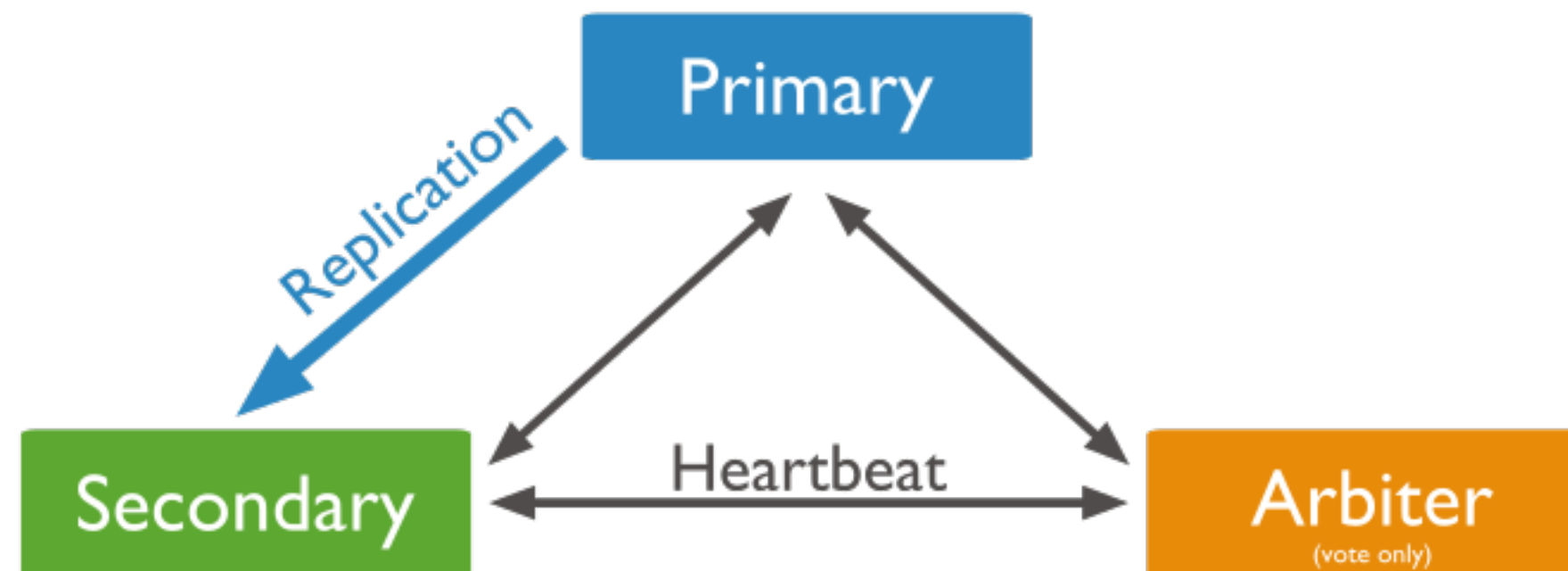
- Soportado por *MongoDB* de forma nativa
- **Conjunto de Réplicas** (*Replica Set*)
  - Grupo de servidores (nodos mongod) :
  - 1 **primario** → recibe las peticiones de los clientes
    - N **secundarios** → copias de los datos del primario.





## Funcionamiento del Conjunto de Réplicas

- Si el nodo primario se cae, los secundarios eligen un nuevo primario entre ellos mismos →
- **votación.**
  - La aplicación se conectará al nuevo primario de manera transparente.
  - Cuando el antiguo nodo primario vuelva en sí, será un nuevo nodo secundario.
- Si los datos de un servidor se dañan o son inaccesibles, podemos crear una nueva copia desde uno de los miembros del conjunto.





## Nodos Regulares

- **Primario:** acepta todas las operaciones de escritura de los clientes.
  - Sólo hay uno, y como sólo un miembro acepta operaciones de escritura, ofrece consistencia estricta para todas las lecturas realizadas desde él.
- **Secundario:** Replican el *oplog* primario y aplican las operaciones a sus conjuntos de datos. Son un espejo del primario.
  - Por defecto, los clientes realizan las lecturas desde el nodo primario. Sin embargo, los clientes pueden indicar que quieren realizar lecturas desde los nodos secundarios.
    - Posibles datos obsoletos



## 4.4.3 Lanzando un Conjunto de Réplicas

- Al lanzar una instancia `mongod` podemos indicar parámetros opcionales:
  - `--dbpath`: ruta de la base de datos
  - `--port`: puerto de la base de datos
  - `--replSet`: nombre del conjunto de réplicas
  - `--fork`: indica que se tiene que crear en un hilo
  - `--logpath`: ruta para almacenar los archivos de log.
- En un conjunto de réplicas, cada instancia `mongod` se coloca en un servidor físico y todos en el puerto estándar.



## Ejemplo de Creación de Conjunto de Réplicas

```
$ mkdir -p /data/rs1 /data/rs2 /data/rs3

$ mongod --replSet replicaExperto --logpath /data/logs/rs1.log --dbpath /data/db/rs1 --port 27017 --oplogSize 64 --smallfiles --fork

$ mongod --replSet replicaExperto --logpath /data/logs/rs2.log --dbpath /data/db/rs2 --port 27018 --oplogSize 64 --smallfiles --fork

$ mongod --replSet replicaExperto --logpath /data/logs/rs3.log --dbpath /data/db/rs3 --port 27019 --oplogSize 64 --smallfiles --fork
```

- Tras conectarnos a una de las réplicas, si intentamos obtener su estado veremos que no se ha inicializado todavía

```
> rs.status()
{
  "startupStatus" : 3,
  "info" : "run rs.initiate(...) if not yet done for the set",
  "ok" : 0,
  "errmsg" : "can't get local.system.replset config from self or any seed (EMPTYCONFIG)"
}
```



## Inicializando el Conjunto de Réplicas

- Dentro del *shell*, crear un documento compuesto de:
  - **\_id**: identificador del conjunto. Tiene que coincidir con el indicado al lanzar `mongod`
  - **members**: array que contiene los elementos del conjunto
    - **\_id**: identificador del elemento
    - **host**: URL del nodo
    - `slaveDelay` (opcional): permite configurar el nodo con retardo
    - `priority` (opcional): prioridad del nodo. Si `priority:0`, el nodo no será principal
- Inicializar el conjunto mediante `rs.initiate` (doc)

```
config = { _id: "replicaExperto", members:[
  { _id : 0, host : "localhost:27017" },
  { _id : 1, host : "localhost:27018" },
  { _id : 2, host : "localhost:27019" }
]};
```



```
> rs.initiate(config)
{
  "info" : "Config now saved locally.
Should come online in about a minute.",
  "ok" : 1
}
```



## Estado del Conjunto de Réplicas

```
> rs.status()
{
  "set" : "replicaExperto",
  "date" : ISODate("2015-05-02T15:52:20Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "localhost:27017",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 572,
      "optime" : Timestamp(1430581402, 1),
      "optimeDate" : ISODate("2015-05-02T15:43:22Z"),
      "electionTime" : Timestamp(1430581412, 1),
      "electionDate" : ISODate("2015-05-02T15:43:32Z"),
      "self" : true
    },
```

```
    {
      "_id" : 1,
      "name" : "localhost:27018",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 536,
      "optime" : Timestamp(1430581402, 1),
      "optimeDate" : ISODate("2015-05-02T15:43:22Z"),
      "lastHeartbeat" : ISODate("2015-05-02T15:52:19Z"),
      "lastHeartbeatRecv" : ISODate("2015-05-02T15:52:18Z"),
      "pingMs" : 0,
      "syncingTo" : "localhost:27017"
    },
    {
      "_id" : 2,
      "name" : "localhost:27019",
      ...
    }
  ],
  "ok" : 1
}
```





## 4.4.4 Trabajando con el Conjunto de Réplicas

- Al conectarnos a un nodo del conjunto, el *shell* nos indica tanto el nombre del conjunto como si es el nodo primario (PRIMARY) o secundario (SECONDARY):

```
$ mongo --port 27017
```

```
replicaExperto:PRIMARY>
```

- Podemos consultar el tipo de nodo y su configuración mediante `rs.isMaster()`

```
replicaExperto:PRIMARY> rs.isMaster()  
{  
  "setName" : "replicaExperto",  
  "setVersion" : 1,  
  "ismaster" : true,  
  "secondary" : false,  
  "hosts" : [  
    "localhost:27017",  
    "localhost:27019",  
    "localhost:27018"  
  ],  
  "primary" : "localhost:27017",  
  "me" : "localhost:27017",  
  "maxBsonObjectSize" : 16777216,  
  "maxMessageSizeBytes" : 48000000,  
  "maxWriteBatchSize" : 1000,  
  "localTime" : ISODate("2015-05-02T16:02:42.291Z"),  
  "maxWireVersion" : 2,  
  "minWireVersion" : 0,  
  "ok" : 1  
}
```





## Trabajando con el Conjunto de Réplicas II

- Insertamos 1000 documentos en el nodo primario

```
for (i=0; i<1000; i++) {  
    db.pruebas.insert({num: i})  
}
```

- Conexión a un nodo secundario

```
$ mongo --port 27018  
replicaExperto:SECONDARY>
```

- Realizar una consulta en un secundario

```
replicaExperto:SECONDARY> db.pruebas.count()  
count failed: { "note" : "from execComand", "ok" : 0, "errmsg" : "not master" }
```

- Habilitar consultas en nodos secundarios → `rs.slaveOk()`

```
replicaExperto:SECONDARY> rs.slaveOk()  
replicaExperto:SECONDARY> db.pruebas.count()  
1000
```

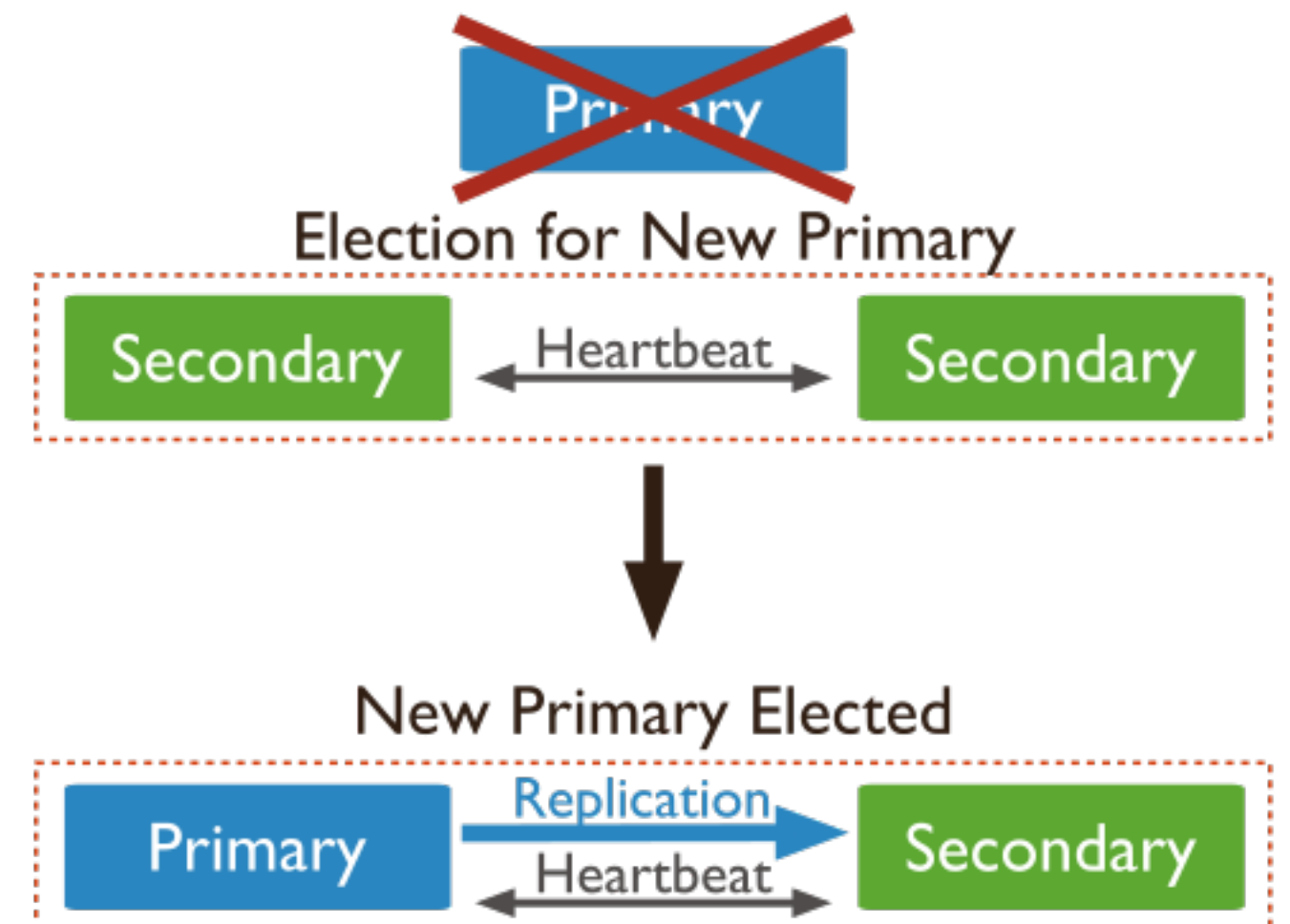
- Escribir en un nodo secundario

```
replicaExperto:SECONDARY> db.pruebas.insert({num : 1001})  
{ "writeError" : { "code" : undefined, "errmsg" : "not master" } }
```



## 4.4.5 Tolerancia a Fallos

- Cuando un nodo primario no se comunica con otros nodos del conjunto durante más de 10 segundos, el conjunto de réplicas intentará, de entre los secundarios, que un miembro se convierta en el nuevo primario.
- Se realiza un proceso de votación.
- El nodo que obtenga el mayor número de votos se erigirá en primario.
- Duración: de 10 a 30 segundos.
- Durante la votación no existe ningún nodo primario y por tanto el conjunto no acepta escrituras y todos los miembros se convierten en nodos de sólo-lectura.





## Proceso de Votación

- Cuando un secundario no puede contactar con su primario, lo hará con el resto de miembros y les indicará que quiere ser elegido como primario.
  - **Cada nodo que no encuentre un primario se auto-nomina como posible primario.**
  - **Un nodo no nomina a otro a ser primario, sólo vota sobre una nominación ya existente.**
- Antes de dar su voto, el resto de nodos comprobarán:
  - Si tienen conectividad con el primario
  - Si el nodo que solicita ser primario tienen una replica actualizada de los datos → oplog
  - Si existe algún nodo con una prioridad mayor que debería ser elegido.
- Si algún miembro que quiere ser primario recibe una mayoría de "sís" se convertirá en el nuevo primario, siempre y cuando no haya un servidor que vete la votación.
  - Si un miembro la veta es porque ha conseguido contactar con el antiguo primario.
- Una vez un candidato recibe una mayoría de "sís", su estado pasará a ser primario.



## Consistencia en el Escritura

- Las aplicaciones pueden decidir que las escrituras vayan al nodo primario pero las lecturas al secundario.
- *MongoDB* permite configurar el nivel de consistencia mediante las operaciones de escritura:
  - Al reducir el nivel de consistencia, el rendimiento será mejor, a costa de poder obtener datos obsoletos u perder datos que no se han terminado de serializar en disco.
  - Con un nivel de consistencia más alto, los clientes esperan tras enviar una operación de escritura a que el primario les confirme la operación.
- Los valores que podemos configurar se realizan mediante las siguientes opciones:
  - `w`: número de servidores que se han de replicar para que la inserción devuelva un ACK.
  - `j`: indica si las escrituras se tienen que trasladar a un diario de bitácora (journal)
  - `wtimeout`: límite de tiempo a esperar como máximo, para prevenir que una escritura se bloquee indefinidamente.



## Niveles de Consistencia

- Con estas opciones, podemos configurar diferentes niveles de consistencia son:
  - Sin confirmación: `w:0`
  - Con confirmación: `w:1`
  - Con diario: `w:1, j:true`. Cada inserción primero se escribe en el diario y posteriormente en el directorio de datos.
  - Con confirmación de réplica: `w:2`
  - Con confirmación de la mayoría: `w:"majority"`
- Se indica como último parámetro en las operaciones de inserción/modificación:

```
db.pruebas.insert(  
  {num : 1002},  
  {writeConcern: {w:"majority", wtimeout: 5000}}  
)
```



## 4.5 Replicación en Java

- En el constructor de `MongoClient`, se le pasa una lista (*seed list*) de `ServerAddress` con los nodos del conjunto de réplicas

```
MongoClient client = new MongoClient(Arrays.asList(  
    new ServerAddress("localhost", 27017),  
    new ServerAddress("localhost", 27018),  
    new ServerAddress("localhost", 27019)));  
  
// Resto de código similar a si nos conectamos a un sólo servidor  
DBCollection prueba = client.getDB("expertojava").getCollection("replica.test");  
prueba.drop();  
  
for (int i = 0; i < Integer.MAX_VALUE; i++) {  
    prueba.insert(new BasicDBObject("_id", i));  
    System.out.println("Insertado documento: " + i);  
    Thread.sleep(500);  
}
```





## Reintento de Operaciones

```
MongoClient client = new MongoClient(Arrays.asList(
    new ServerAddress("localhost", 27017),
    new ServerAddress("localhost", 27018),
    new ServerAddress("localhost", 27019)));

DBCollection coleccion = client.getDB("expertojava").getCollection("pruebas");
coleccion.drop();

for (int i = 0; i < Integer.MAX_VALUE; i++) {
    for (int intentos = 0; intentos <= 2; intentos++) {
        try {
            coleccion.insert(new BasicDBObject("_id", i));
            System.out.println("Documento insertado: " + i);
            break;
        } catch (MongoException e) {
            System.out.println(e.getMessage());
            System.out.println("Reintentando");
            Thread.sleep(5000);
        }
    }
}
```

```
...
Documento insertado: 39855
Operation on server localhost:27018 failed
Reintentando
Documento insertado: 39856
...
```

```
replicaExperto:PRIMARY> rs.stepDown()
2015-05-03T09:41:52.619+0200 DBClientCursor::init call() failed
2015-05-03T09:41:52.621+0200 Error: error doing query: failed at src/mongo/shell/query.js:81
2015-05-03T09:41:52.623+0200 trying reconnect to 127.0.0.1:27018 (127.0.0.1) failed
2015-05-03T09:41:52.626+0200 reconnect 127.0.0.1:27018 (127.0.0.1) ok
replicaExperto:SECONDARY>
```



## Consistencia en la Escritura

```
MongoClient cliente = new MongoClient(Arrays.asList(
    new ServerAddress("localhost", 27017), new ServerAddress("localhost", 27018),
    new ServerAddress("localhost", 27019)));

cliente.setWriteConcern(WriteConcern.JOURNALED);

DB db = cliente.getDB("expertojava");
db.setWriteConcern(WriteConcern.ACKNOWLEDGED);
DBCollection coleccion = db.getCollection("pruebas");
coleccion.setWriteConcern(WriteConcern.MAJORITY);

coleccion.drop();

DBObject doc = new BasicDBObject("_id", 1);

coleccion.insert(doc);

try {
    coleccion.insert(doc, WriteConcern.UNACKNOWLEDGED);
} catch (MongoException e) {
    System.out.println(e.getMessage());
}
```





## Preferencias de Lectura

- Las aplicaciones vía el *driver* pueden elegir en que nodos realizar las lecturas.
- Posibilidades:
  - `primary`: valor por defecto. Todas las lecturas se realizan en el nodo primario. Si se cae este nodo, el sistema no acepta lecturas. Es el único modo que garantiza los datos más recientes, ya que todas las escrituras pasan por él.
  - `primaryPreferred`: lectura de nodos secundarios sólo si el primario ha caído.
  - `nearest`: acceder a los datos con la menor latencia posible, sin tener en cuenta si accedemos a un primario o a un secundario.
  - `secondary`: lecturas en nodos secundarios. Si no hubiese ningún nodo secundario disponible, la lectura fallará.
  - `secondaryPreferred`: envía las lecturas a nodos secundarios, pero si no hubiese ninguno, la enviaría al primario.



## Preferencias de Lectura en Java

```
MongoClient cliente = new MongoClient(Arrays.asList(
    new ServerAddress("localhost", 27017),
    new ServerAddress("localhost", 27018),
    new ServerAddress("localhost", 27019)));
cliente.setReadPreference(ReadPreference.primary());

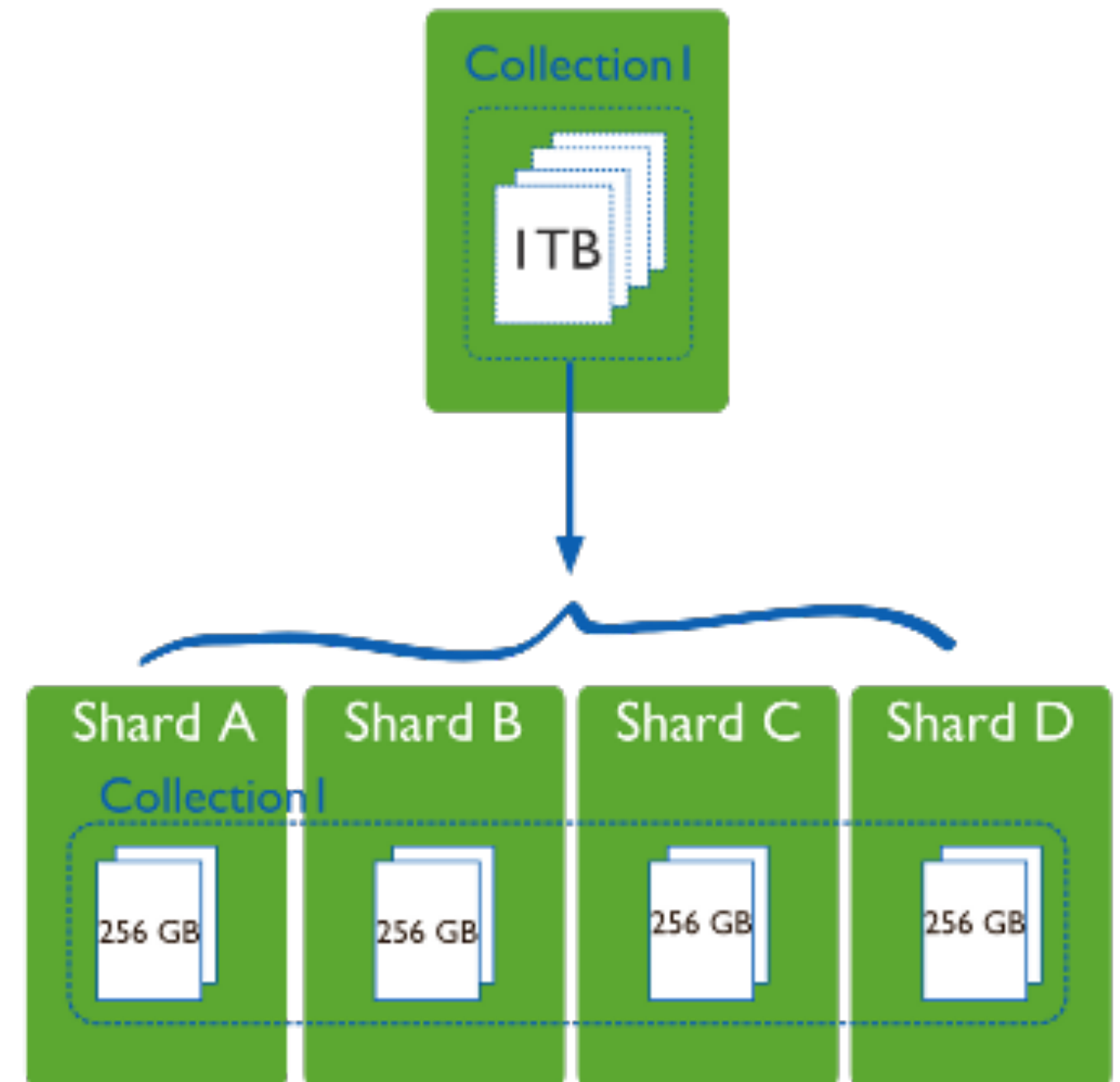
DB db = cliente.getDB("expertojava");
db.setReadPreference(ReadPreference.primary());
DBCollection coleccion = db.getCollection("pruebas");
coleccion.setReadPreference(ReadPreference.primaryPreferred());

DBCursor cursor = coleccion.find().setReadPreference(ReadPreference.nearest());
try {
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
} finally {
    cursor.close();
}
```



## 4.6 Sharding

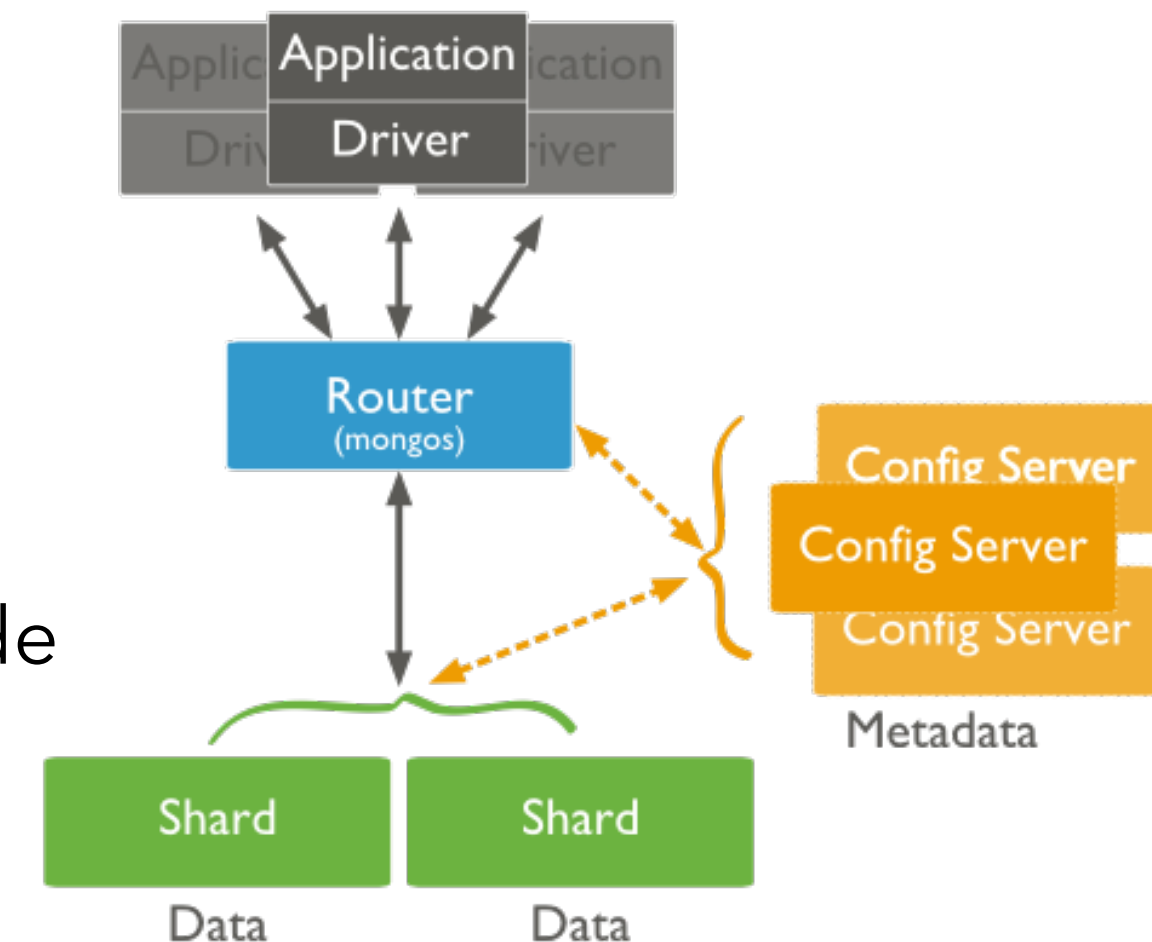
- Particionado: dividir los datos entre múltiples máquinas.
- Permite almacenar más información y soportar más carga sin necesidad de máquinas más potentes, sino una mayor cantidad de máquinas más modestas (y mucho más baratas).
- Fragmenta los datos de la base de datos horizontalmente agrupándolos de algún modo que tenga sentido y que permita un direccionamiento más rápido.
- Los *shards* (fragmentos) pueden estar localizados en diferentes bases de datos y localizaciones físicas





## Elementos del *Sharded Cluster*

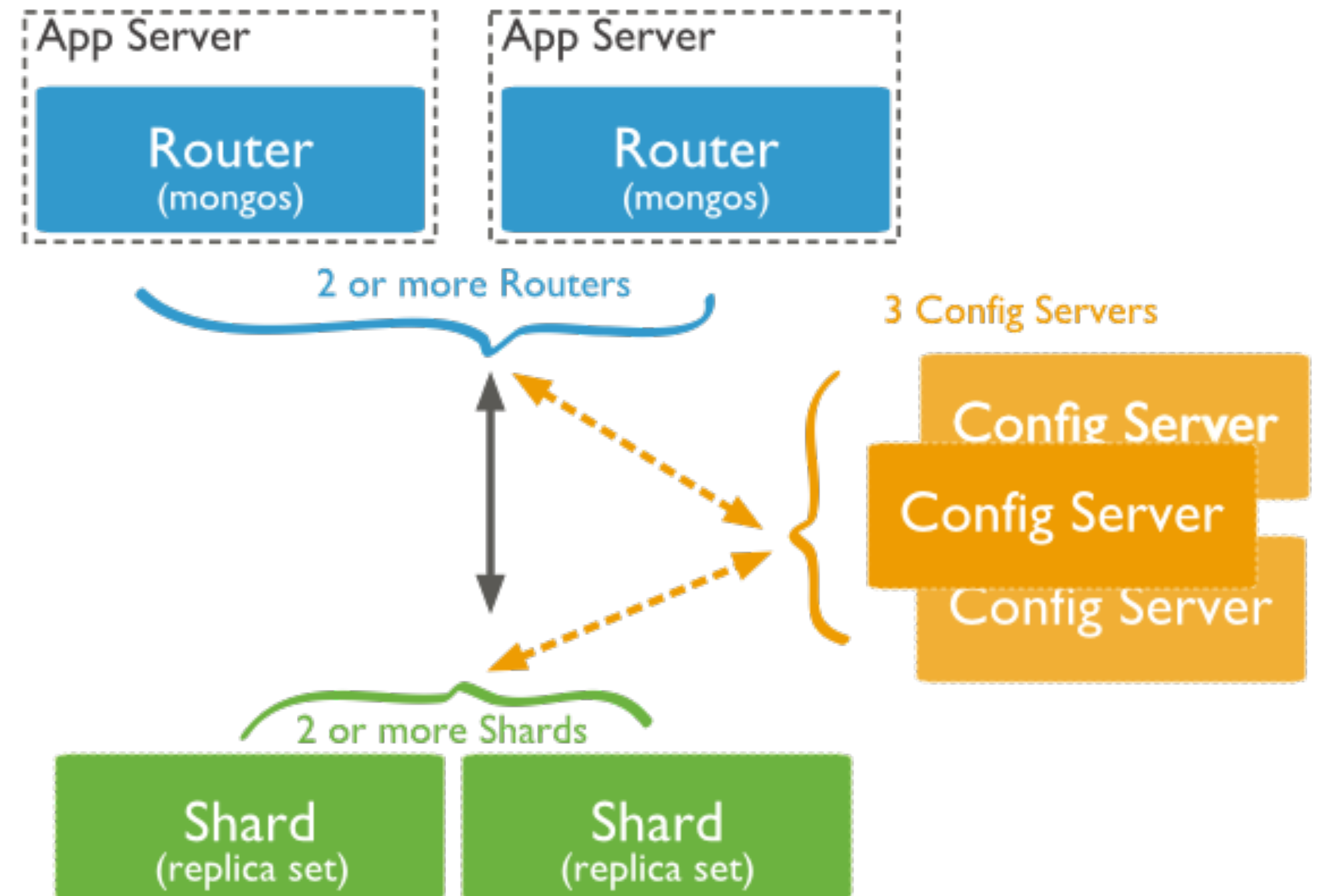
- **Shards:** Cada una de las máquinas del *cluster*
  - Almacena un subconjunto de los datos de la colección.
  - Cada shard es una instancia de `mongod` o un conjunto de réplicas.
- **Servidor de configuración:** Instancia de `mongod` que almacena metadatos sobre el *cluster*.
  - Mapean los trozos con los *shards*, definiendo qué rangos de datos definen un trozo (*chunk*) de la colección, y qué trozos se encuentran en un determinado *shard*.
- **Enrutador:** Instancia `mongos` que enruta las lecturas y escrituras de las aplicaciones a los shards.
  - Las aplicaciones no acceden directamente a los shards, sino al router.
  - Una vez recopilados los datos de los diferentes shards, se fusionan y se encarga de devolverlos a la aplicación.





## Sharding en Producción

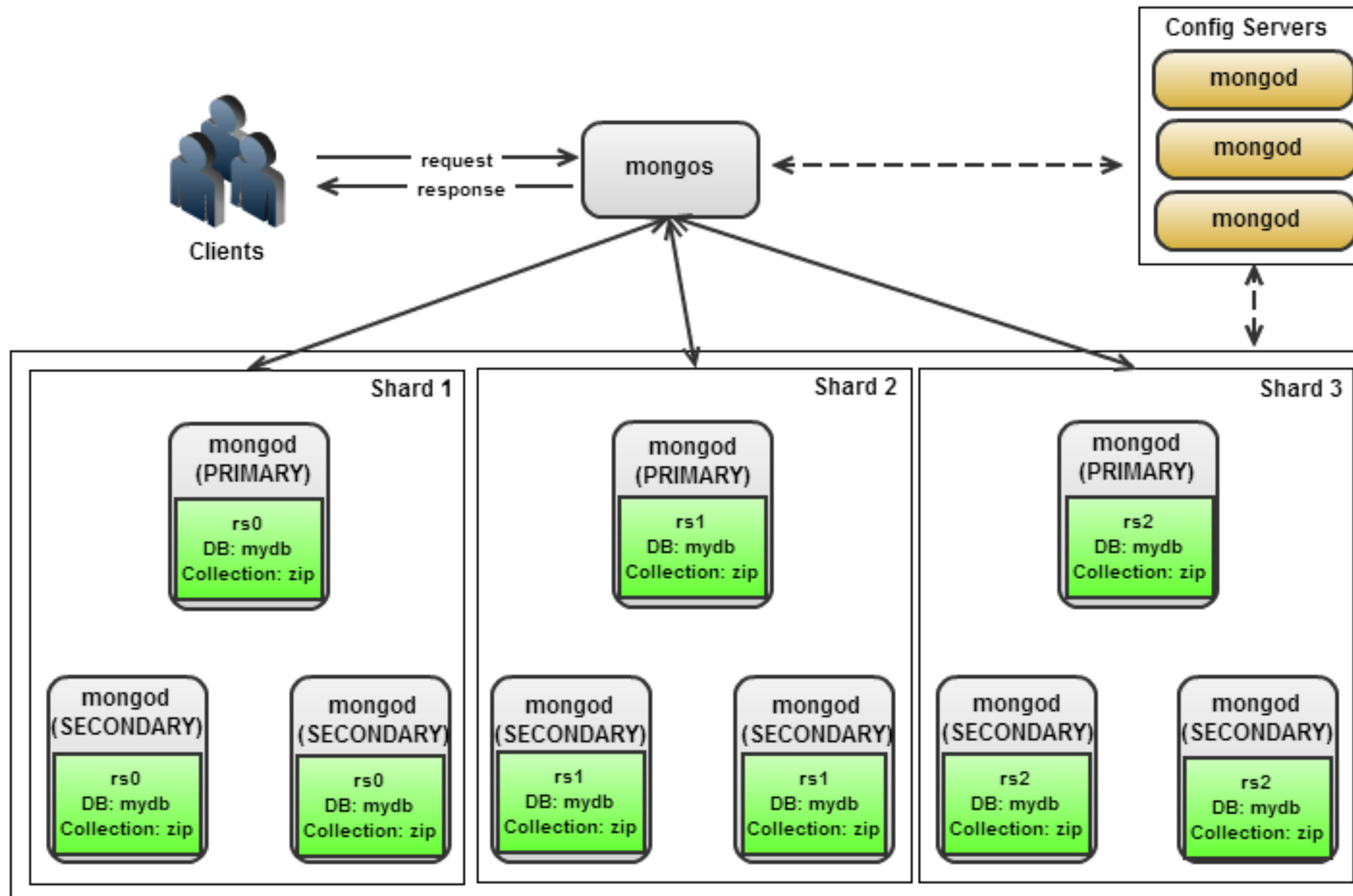
- Varios routers
- 3 Servidores de configuración
  - Elemento crítico
- Cada *shard* es un conjunto de réplicas de al menos 3 nodos







## Infraestructura completa





## Clave de Fragmentación (*shard key*)

- Permite que *MongoDB* sepa cómo dividir una colección en trozos.
- Normalmente el identificador del documento → `student_id`.
  - Este identificador es la clave del *chunk* (misma función que una clave primaria).
- Para las búsquedas, borrados y actualizaciones, al emplear la *shard key*, `mongos` sabe a que *shard* enviar la petición.
  - Si la operación no la indica, se hará un *broadcast* a todas los *shards* para averiguar donde se encuentra.
- Toda inserción debe incluir la clave de fragmentación. Si la clave es compuesta, la inserción debe contener la clave completa.





## Elección de la Clave de Fragmentación

- Alta **cardinalidad** → asegura que los documentos se dividan en los distintos fragmentos.
  - Por ejemplo, si elegimos un *shard key* con solo 3 posibles valores y tenemos 5 fragmentos, no podríamos separar los documentos en los 5 fragmentos al solo tener 3 valores posibles para separar.
  - Cuantos más valores posibles pueda tener la clave de fragmentación, más eficiente será la división de los trozos entre los fragmentos disponibles.
- Alto nivel de **aleatoriedad** → Si utilizamos una clave que sigue un patrón incremental como una fecha o un ID, al insertar documentos, el mismo fragmento estará siendo utilizado constantemente durante el rango de valores definido para él.
  - Provoca que los datos estén separados de una manera óptima
  - Estresa a un fragmento en períodos de tiempo mientras que los otros posiblemente queden con muy poca actividad → *hotspotting*.



## Preparando el *Sharding*

- Crear carpetas
- Arrancar *shards* y *router*

```
mkdir -p /data/s1/db /data/s2/db /data/logs /data/conf1/db  
chown `id -u` /data/s1/db /data/s2/db /data/logs /data/conf1/db
```

```
mongod --shardsvr --dbpath /data/s1/db --port 27000 --logpath /data/logs/sh1.log --smallfiles --  
oplogSize 128 --fork  
mongod --shardsvr --dbpath /data/s2/db --port 27001 --logpath /data/logs/sh2.log --smallfiles --  
oplogSize 128 --fork  
mongod --configsvr --dbpath /data/conf1/db --port 25000 --logpath /data/logs/config.log --fork  
mongos --configdb localhost:25000 --logpath /data/logs/mongos.log --fork
```

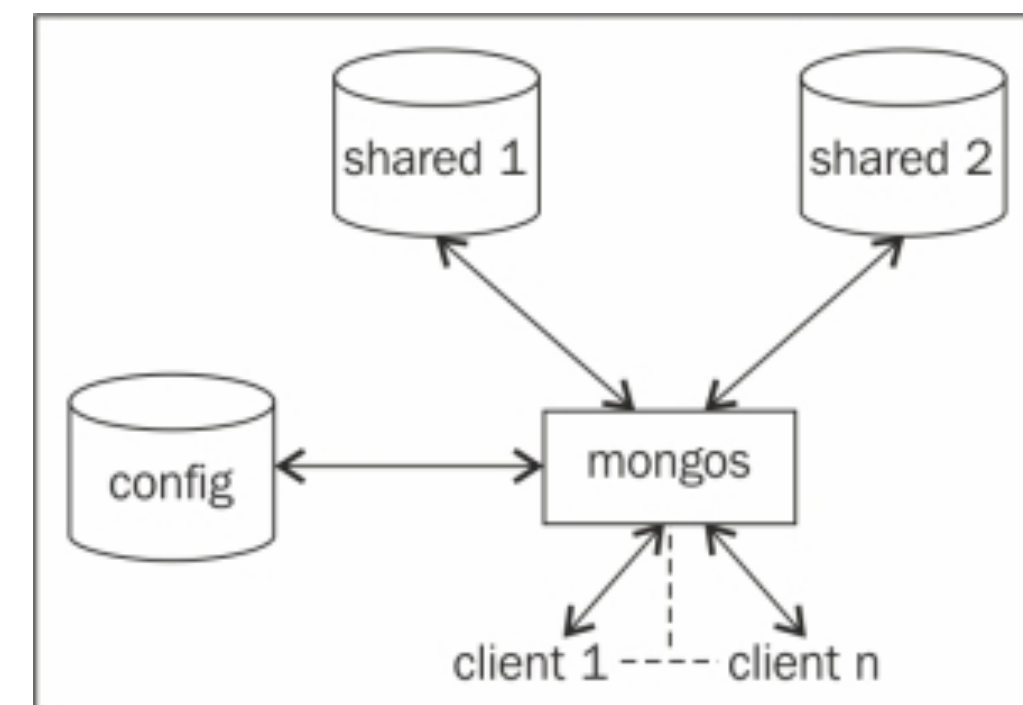
- Conexión al *router*

```
$ mongo  
MongoDB shell version: 3.0.8  
connecting to: test  
mongos>
```

- Configurar el *shard*

- `sh.addShard(URI)`

```
sh.addShard("localhost:27000")  
{ "shardAdded" : "shard0000", "ok" : 1 }  
mongos> sh.addShard("localhost:27001")  
{ "shardAdded" : "shard0001", "ok" : 1 }
```





## Estado del Shard

- `sh.status()`

```
mongos> sh.status()
--- Sharding Status ---
sharding version: {
  "_id" : 1,
  "minCompatibleVersion" : 5,
  "currentVersion" : 6,
  "clusterId" : ObjectId("56bc7054ba6728d2673a1755")
}
shards:
  { "_id" : "shard0000", "host" : "localhost:27000" }
  { "_id" : "shard0001", "host" : "localhost:27001" }
active mongoses:
  "3.2.1" : 1
balancer:
  Currently enabled:  yes
  Currently running:  no
  Failed balancer rounds in last 5 attempts:  0
  Migration Results for the last 24 hours:
    No recent migrations
databases:
```



## Habilitando el *Sharding*

```
mongos> use expertojava
switched to db expertojava
mongos> for (var i=0; i<100000; i++) {
  db.usuarios.insert({"login":"usu" + i, "nombre":"nom" + i*2, "fcreacion": new Date()});
}
mongos> db.usuarios.count()
100000
```

- Habilitar el sharding a nivel de base de datos

- `sh.enableSharding(nombreDB)`

- Crear índice sobre la clave de fragmentación

```
mongos> sh.enableSharding("expertojava")
```

```
mongos> db.usuarios.createIndex({"login": 1})
{
  "raw" : {
    "localhost:27000" : {
      "createdCollectionAutomatically" : false,
      "numIndexesBefore" : 1,
      "numIndexesAfter" : 2,
      "ok" : 1
    }
  },
  "ok" : 1
}
```



## Fragmentar la colección

- `sh.shardCollection(db.coleccion, claveFragmentacion, usaIndiceUnique)`

```
mongos> sh.shardCollection("expertojava.usuarios", {"login": 1}, false)

mongos> sh.status()
--- Sharding Status ---
  sharding version: {
    "_id" : 1,
    ...
  }
  shards:
    { "_id" : "shard0000", "host" : "localhost:27000" }
    { "_id" : "shard0001", "host" : "localhost:27001" }
  active mongoses:
    "3.2.1" : 1
  balancer:
    ...
  databases:
    { "_id" : "expertojava", "primary" : "shard0000", "partitioned" : true }
      expertojava.usuarios
        shard key: { "login" : 1 }
        unique: false
        balancing: true
        chunks:
          shard0000 1
          { "login" : { "$minKey" : 1 } } -->> { "login" : { "$maxKey" : 1 } } on : shard0000 ...
```



## Trabajando con el *Sharding*

```
mongos> sh.status()
--- Sharding Status ---
sharding version: {
  ...
}
shards:
  { "_id" : "shard0000", "host" : "localhost:27000" }
  { "_id" : "shard0001", "host" : "localhost:27001" }
active mongoses:
  "3.2.1" : 1
balancer:
  ...
databases:
  { "_id" : "expertojava", "primary" : "shard0000", "partitioned" : true }
  expertojava.usuarios
    shard key: { "login" : 1 }
    unique: false
    balancing: true
    chunks:
      shard0000 32
      shard0001 31
    too many chunks to print, use verbose if you want to force print
```

```
mongos> for (var i=100000; i<200000; i++) {
  db.usuarios.insert({"login":"usu" + i, ...});
}
mongos> db.usuarios.count()
200000
```





## Consultas con *Sharding* con clave de fragmentación

```
mongos> db.usuarios.find({"login":"usu12345"}).explain()
{
  "queryPlanner" : {
    "mongosPlannerVersion" : 1,
    "winningPlan" : {
      "stage" : "SINGLE_SHARD",
      "shards" : [
        {
          "shardName" : "shard0001",
          "connectionString" : "localhost:27001",
          "serverInfo" : {
            "host" : "MacBook-Air-de-Aitor.local",
            "port" : 27001,
            "version" : "3.2.1",
            "gitVersion" : "a14d55980c2cdc565d4704a7e3ad37e4e535c1b2"
          },
          "plannerVersion" : 1,
          "namespace" : "expertojava.usuarios",
          "indexFilterSet" : false,
          "parsedQuery" : {
            "login" : {
              "$eq" : "usu12345"
            }
          },
          "rejectedPlans" : [ ]
        }
      ]
    },
    "rejectedPlans" : [ ]
  },
  "ok" : 1
}
```

```
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "SHARDING_FILTER",
        "inputStage" : {
          "stage" : "IXSCAN",
          "keyPattern" : {
            "login" : 1
          },
          "indexName" : "login_1",
          "isMultiKey" : false,
          "isUnique" : false,
          "isSparse" : false,
          "isPartial" : false,
          "indexVersion" : 1,
          "direction" : "forward",
          "indexBounds" : {
            "login" : [
              ["usu12345", "usu12345"]
            ]
          }
        },
        "rejectedPlans" : [ ]
      },
      "rejectedPlans" : [ ]
    },
    "ok" : 1
  }
}
```





## Consultas con *Sharding* sin clave de fragmentación

```
mongos> db.usuarios.find().explain()
{
  "queryPlanner" : {
    "mongosPlannerVersion" : 1,
    "winningPlan" : {
      "stage" : "SHARD_MERGE",
      "shards" : [
        {
          "shardName" : "shard0000",
          "connectionString" : "localhost:27000",
          "serverInfo" : {
            ...
          },
          "plannerVersion" : 1,
          "namespace" : "expertojava.usuarios",
          "indexFilterSet" : false,
          "parsedQuery" : {
            "$and" : [ ]
          },
          "winningPlan" : {
            "stage" : "SHARDING_FILTER",
            "inputStage" : {
              "stage" : "COLLSCAN",
              "filter" : {
                "$and" : [ ]
              },
              "direction" : "forward"
            }
          },
          "rejectedPlans" : [ ]
        }
      ],
      "rejectedPlans" : [ ]
    }
  },
  "rejectedPlans" : [ ]
},
```

```
{
  "shardName" : "shard0001",
  "connectionString" : "localhost:27001",
  "serverInfo" : {
    ...
  },
  "plannerVersion" : 1,
  "namespace" : "expertojava.usuarios",
  "indexFilterSet" : false,
  "parsedQuery" : {
    "$and" : [ ]
  },
  "winningPlan" : {
    "stage" : "SHARDING_FILTER",
    "inputStage" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "$and" : [ ]
      },
      "direction" : "forward"
    }
  },
  "rejectedPlans" : [ ]
}
},
"ok" : 1
}
```



**¿Preguntas?**