

# Gestores NoSQL - MongoDB

Marta Zorrilla - Diego García-Saiz

Enero 2017



Este material se ofrece con licencia: [Creative Commons BY-NC-SA 4.0](https://creativecommons.org/licenses/by-nc-sa/4.0/)



# Tabla de contenidos

- Introducción
- Arquitectura
- Tareas administrativas
- Modelo de datos
- BSON y operaciones CRUD
- Ventajas y desventajas de MongoDB

# Bibliografía y documentación complementaria

- Bibliografía:
  - Rick Copeland: **MongoDB Applied Design Patterns**. O'Reilly Media (2013)
  - Kristina Chodorow: **MongoDB: The Definitive Guide**. O'Reilly Media (2013)
- Tutoriales:
  - <https://www.tutorialspoint.com/mongodb/>
  - <https://docs.mongodb.com/manual/tutorial/> (oficial)
- Manuales:
  - <https://docs.mongodb.com/manual/> (oficial)
  - <http://bsonspec.org/> (lenguaje BSON)

# Introducción: Revisión histórica de MongoDB

- BD NoSQL orientada a documentos.
  - MongoDB: derivado de la palabra inglesa “humongous” (enorme, extraordinariamente largo).
- Primera versión publicada en marzo de 2009: 0.9
  - Publicada en código abierto bajo licencia AGPL.
- Primera versión estable publicada en agosto de 2009: 1.0
  - En 2011, se lanzó la versión 1.4, la primera considerada como apta para su producción y distribución.
- Última versión publicada el 29 de noviembre de 2016: 3.4
  - Las únicas versiones que actualmente se mantienen actualizadas son de la 2.0 en adelante.

# Introducción: ¿Pará que se usa MongoDB?

- Internet de las cosas, grupo industrial Bosch.
- Visualización geospacial de elementos de una ciudad en tiempo real (Boston city).
- Gestión de contenidos, caso de Sourceforge.
- Aplicaciones móviles, como compra de viajes por Expedia.
- Videojuegos como FIFA online 3.
- Log de eventos, caso de Facebook para recoger anuncios accedidos
- Algunos usuarios conocidos de MongoDB son:  
Ebay, Expedia, Orange, Barclays, Adobe, Telefónica...

*Ver más en:*

<https://www.mongodb.com/use-cases>

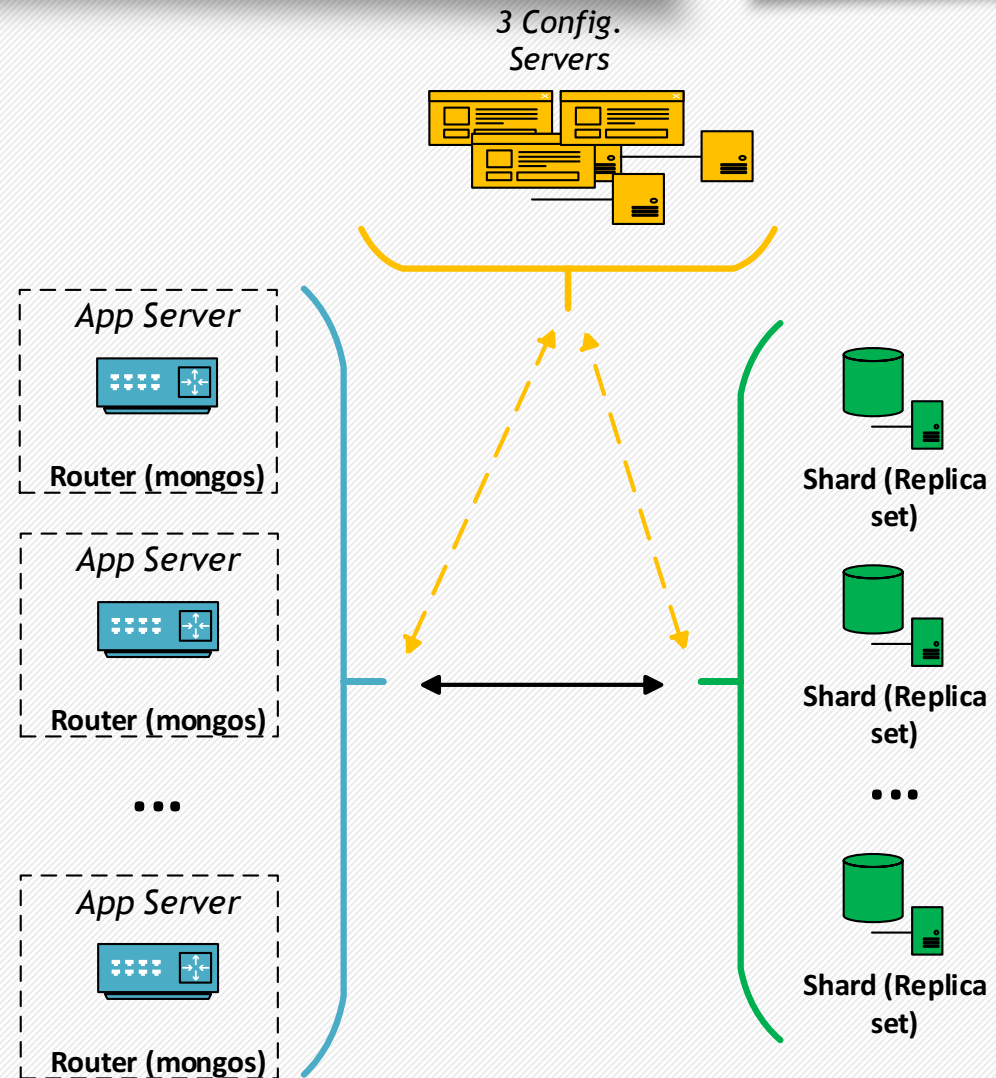
<https://www.mongodb.com/who-uses-mongodb>

# Arquitectura: conceptos

- ***mongod***: proceso primario (demonio) instanciable de MongoDB para la gestión del acceso a los datos.
- ***mongos***: servicio de enrutado entre la aplicación y la Base de Datos.
- ***Config. server***: servidor que almacena los metadatos para localizar los datos de las operaciones requeridas por el cliente.
- ***Replica set***: grupo de procesos *mongod* que almacenan las mismas copias de los datos.
  - ***Primary***: guarda las copias principales.
  - ***Secondary***: guarda las copias secundarias de *Primary*.
  - ***Arbitrer***: Si el *Primary* se cae, vota para decidir que *Secondary* pasa a ser *Primary*.
- ***Shard***: *replica set* que almacena una parte de los datos de la BD.
  - Se basa en el concepto de ***Sharding***, que se define como “un método para distribuir los datos en varias máquinas” (definición en <https://docs.mongodb.com/manual/sharding/>)

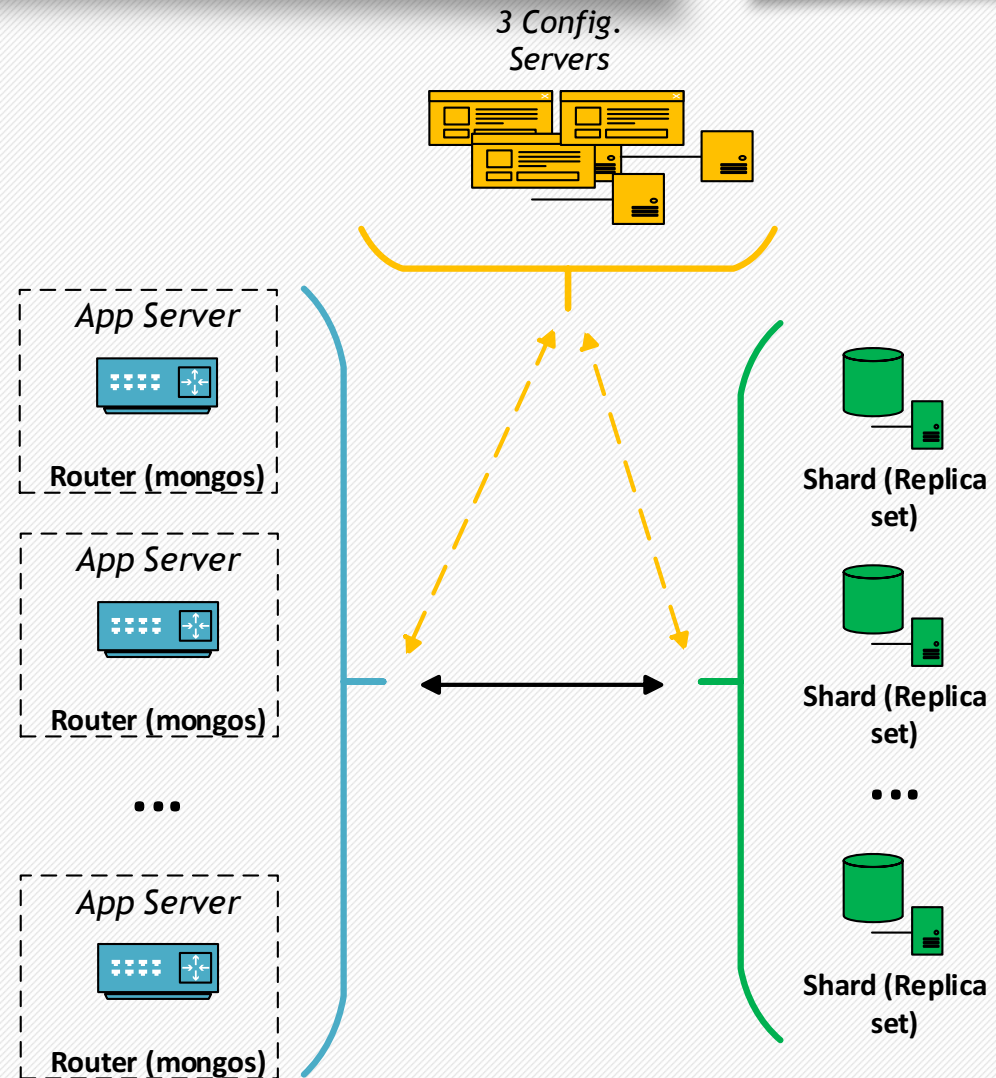
# Arquitectura: *Sharding*, repartiendo los documentos

- Los clientes (*Apps*) se conectan a un *router* (*mongos*) de la Base de Datos.
- El *mongos* se comunica con los *Config. Server* para determinar en donde están los datos requeridos, o en dónde escribir los nuevos.
- A través de un *mongos*, la *App* se conecta con el *Shard* (conjunto de réplicas). Cada *Shard* tendrá un *mongod* principal y cero o varios *mongod* secundarios (ver más adelante).



# Arquitectura: *Sharding*, repartiendo los documentos

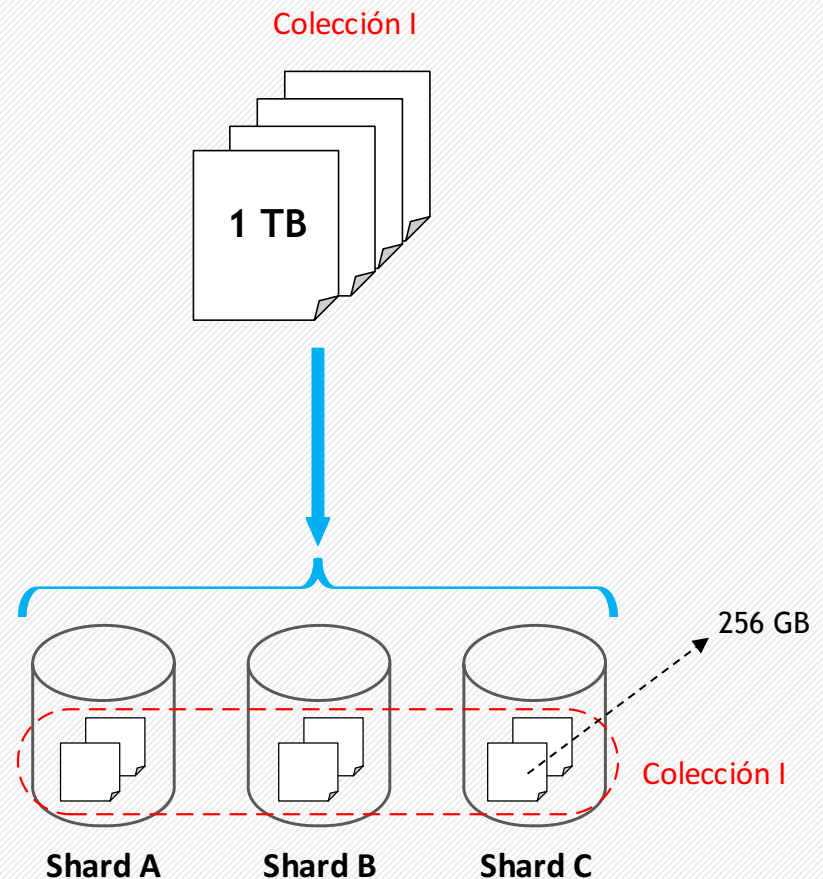
- Tres *Config. Server*: garantizan acceso aunque 1 ó 2 de ellos “se caigan”.
  - También reparten la carga de trabajo.
  - Sólo se escribe en ellos si los metadatos cambian (p.e. si los *chunks* de una colección cambian de ubicación).
  - Si se caen los tres, aún se puede acceder a los *shards* (datos) siempre que los *mongos* no se reinicien.
- Dos o más *mongos*: reparten la carga de peticiones por parte de las aplicaciones.
- Dos o más *shards*: para repartir los documentos en varios nodos del clúster.





# Sharding: chunks y Shard key

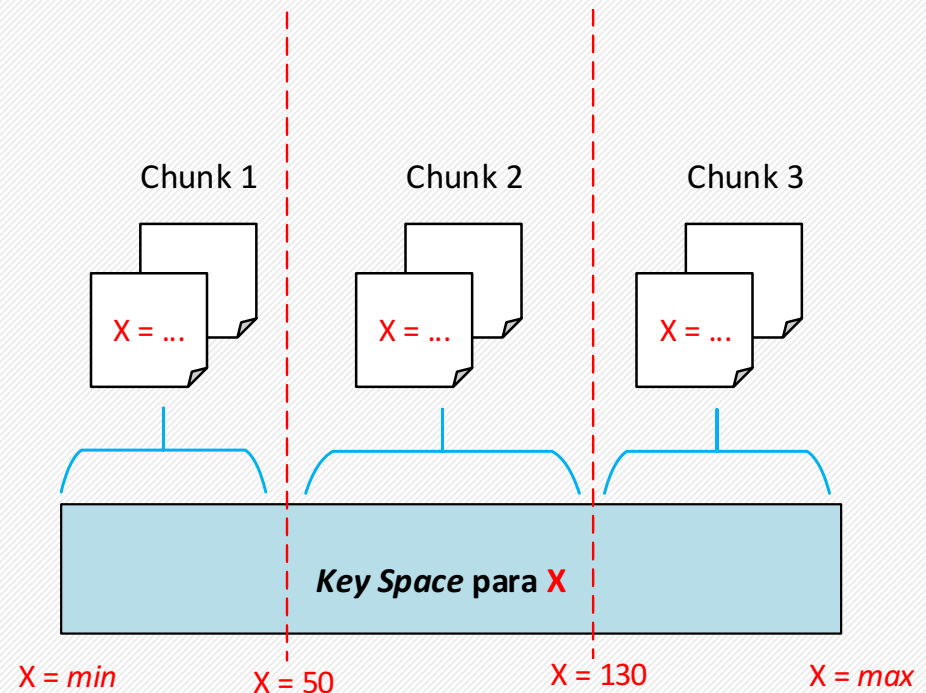
- **Chunks:** “trozos” en los que se reparten los documentos de una colección en varios *shards*.
- **Shard key:** clave que determina cómo se divide una colección en diferentes *chunks*.
  - La *Shard key* siempre ha de ser un campo del documento que esté indexado (más adelante veremos los tipos de índices en mongoDB).



# Sharding: chunks y Shard key

- **Range Based Sharding:** los *chunks* se crean en base a rangos de valores de la *Shard key*.

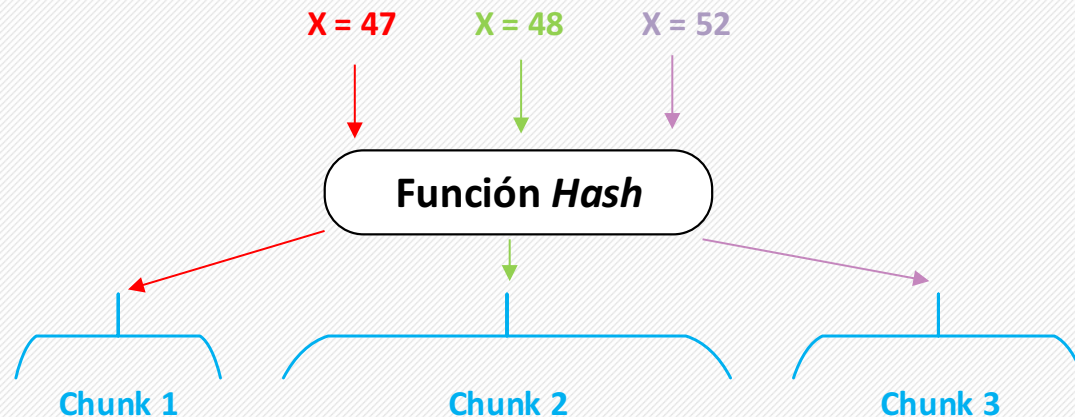
- Documentos con un valor de *Shard key* similar estarán probablemente en un mismo *shard*.
  - Bueno para consultas.
- Si hay muchos documentos con un *Shard key* en el mismo rango, habrá *shards* “sobrecargados” de datos.



# Sharding: chunks y Shard key

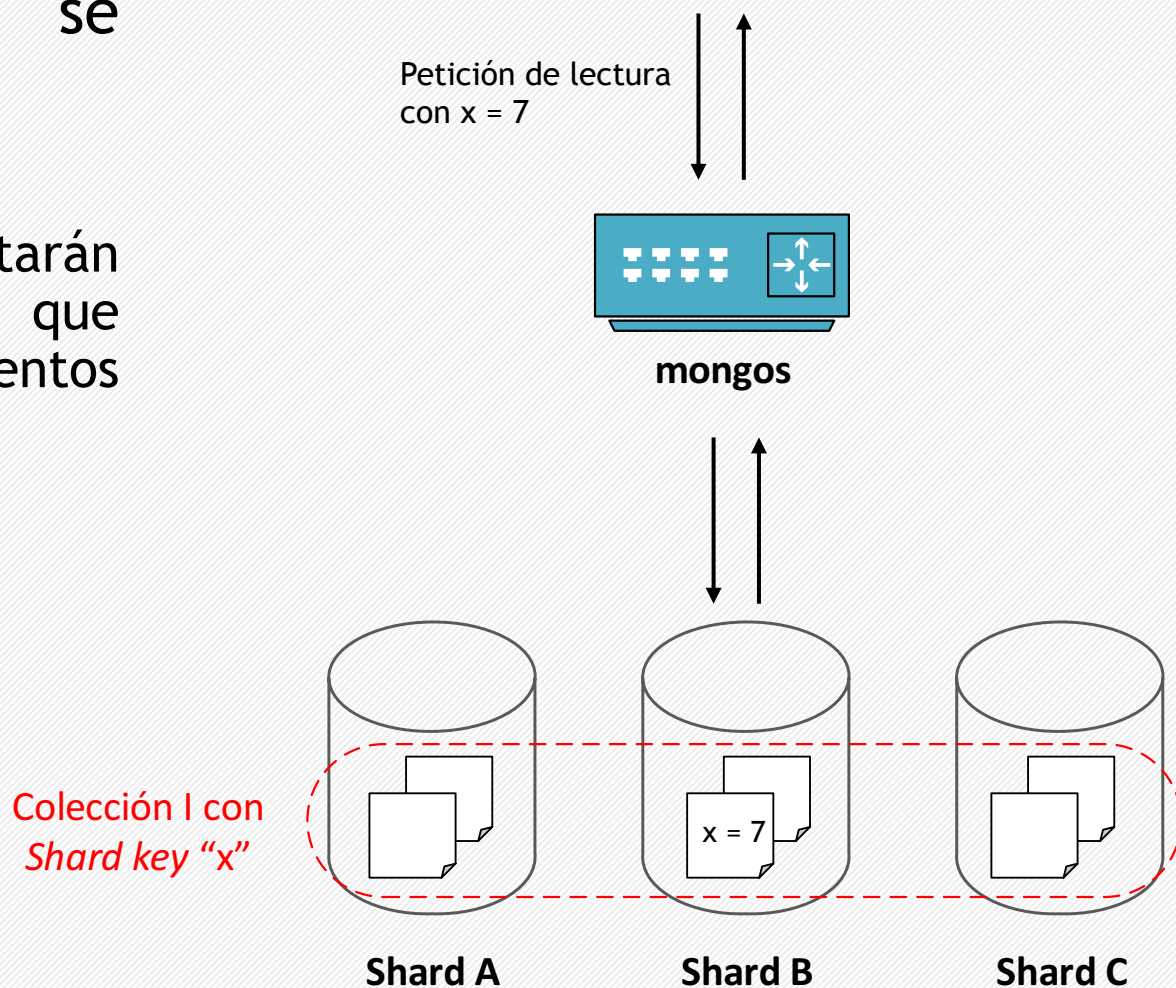
- **Hash Based Sharding:**  
cálculo de valor *hash* en  
base a la *Shard Key*.

- No hay rangos.
- Los documentos se reparten de forma “aleatoria” entre los *shards*.
- Buen equilibrio.
- Las consultas por rangos sobre la *Shard key* requerirán acceder a varios *shards* (más lentas).



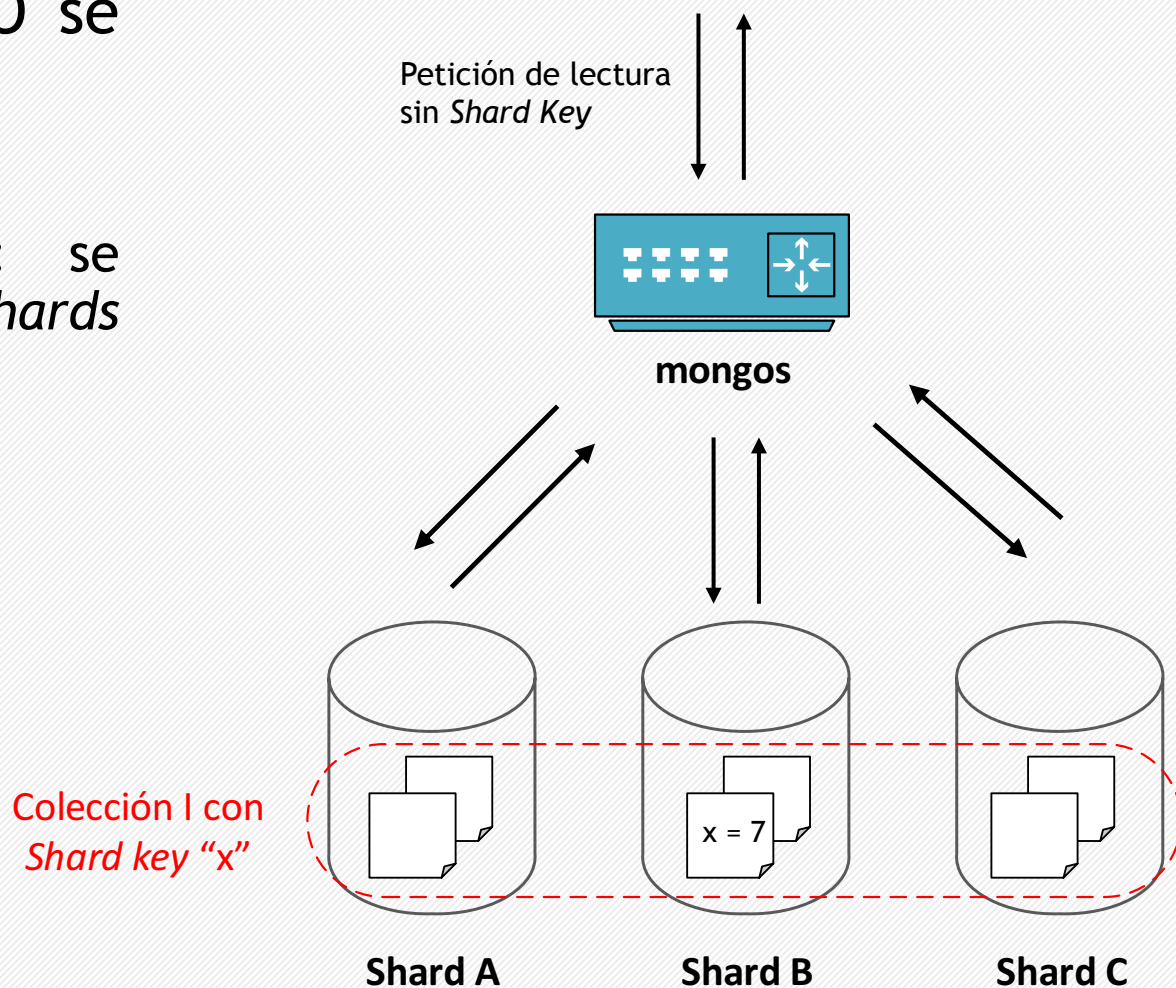
# La importancia de la *Shard key*

- Si en una consulta se incluye la *Shard key*:
- Sólo se consultarán aquellos *shards* que contengan los documentos requeridos.



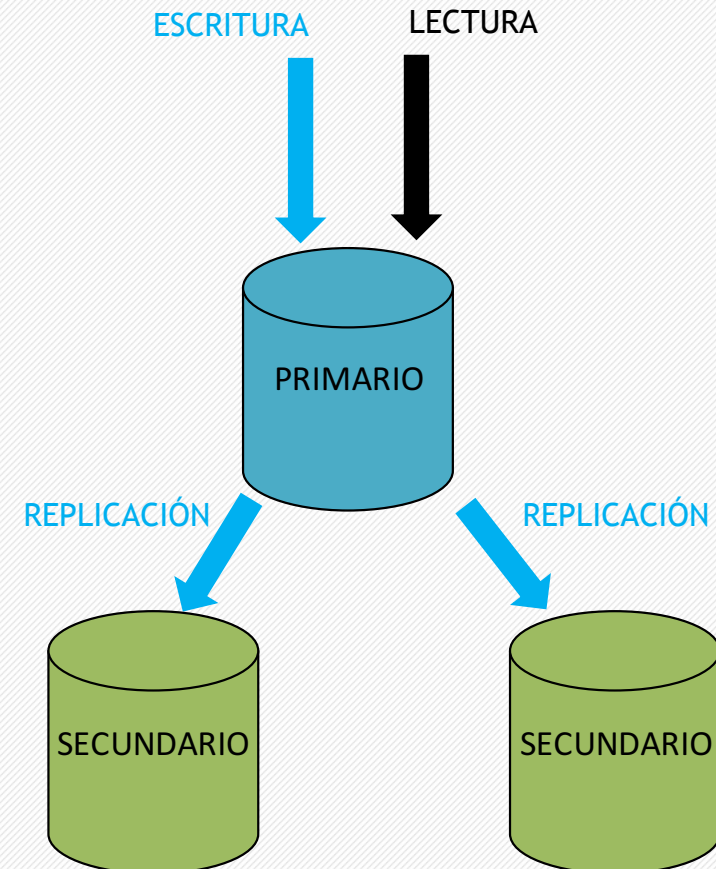
# La importancia de la *Shard key*

- Si en una consulta NO se incluye la *Shard key*:
  - Búsqueda secuencial: se busca en todos los *shards* (consulta ineficiente).



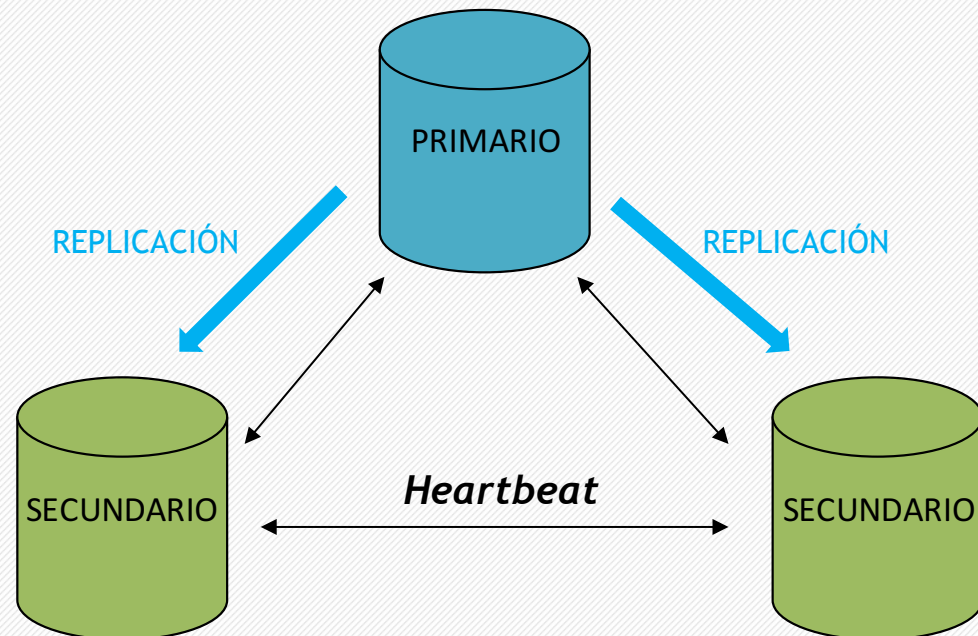
# Replicación de los datos

- En MongoDB, existen copias o réplicas de los datos principales y secundarias. Cada réplica es controlada por una instancia del proceso *mongod*.
  - Las operaciones de escritura del cliente se realizan exclusivamente sobre el primario.
    - Las de lectura, por defecto también. Si bien, se puede reconfigurar la BD para que se hagan sobre los secundarios.
  - El primario almacena en una estructura de log los cambios en los datos, que luego son propagados a los secundarios.



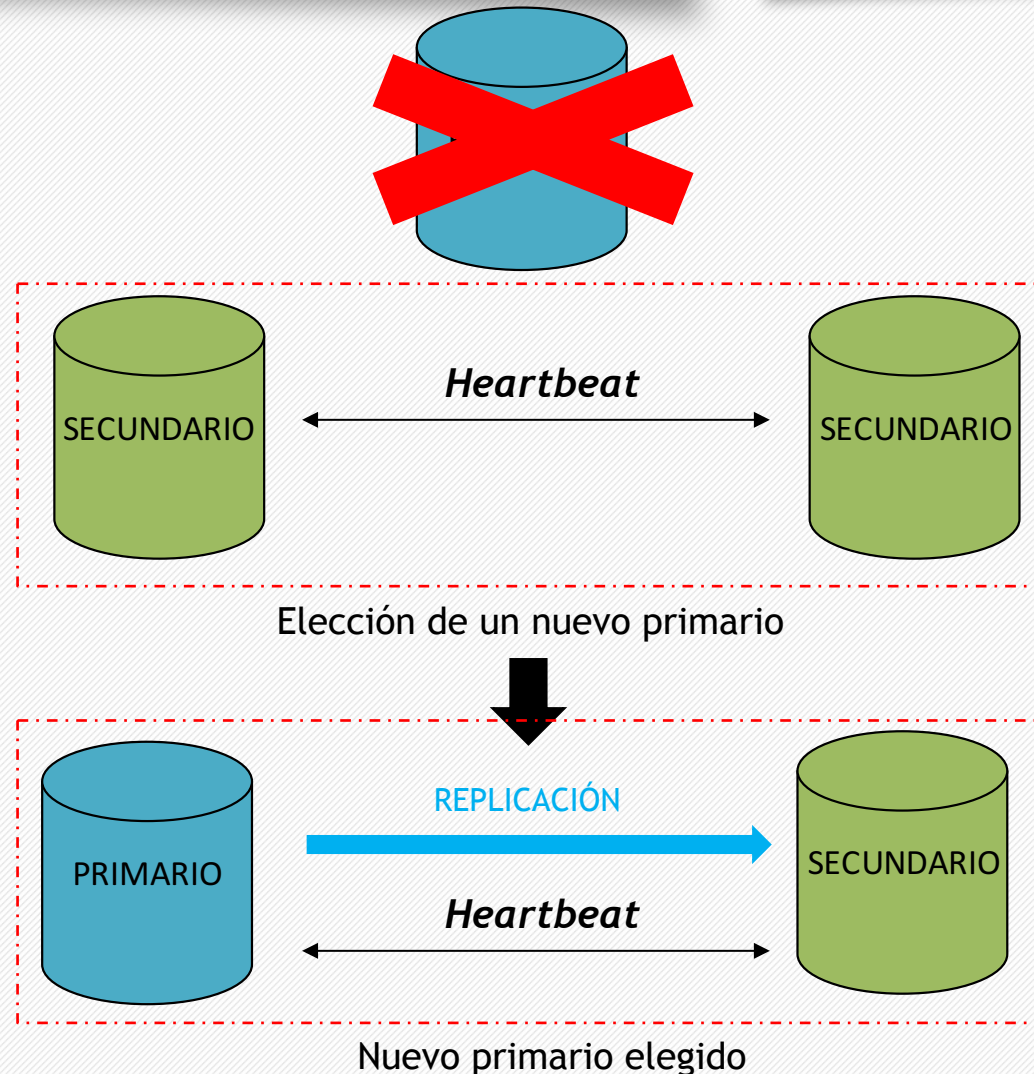
# Replicación de los datos

- *Heartbeat*: comunicación entre nodos del mismo *replica set* indicando que “siguen vivos”. Se envía cada 2 segundos.
- Si el primario cae, los secundarios tienen que elegir un nuevo primario de entre ellos.
  - Para considerar que el primario ha caído, por defecto, ha de estar 10 segundos sin responder.



# Replicación de los datos

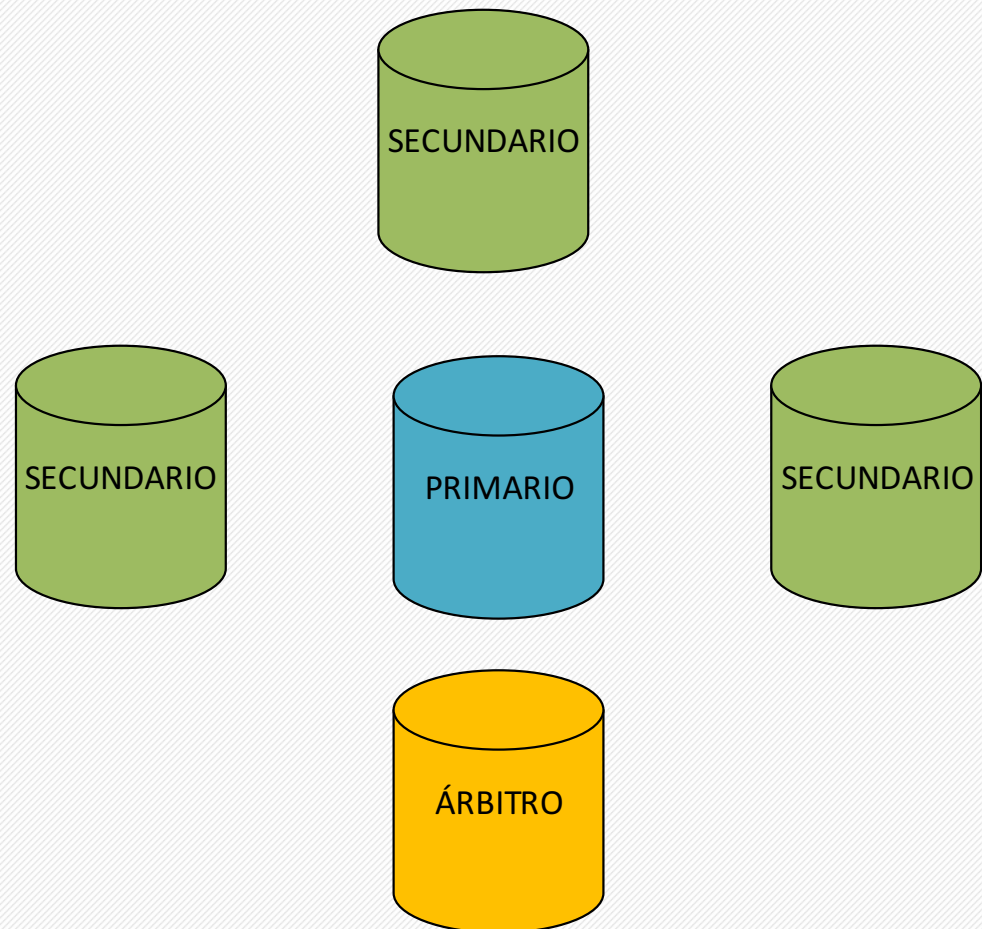
- Los secundarios votan para decidir quién de ellos se convierte en el nuevo primario.





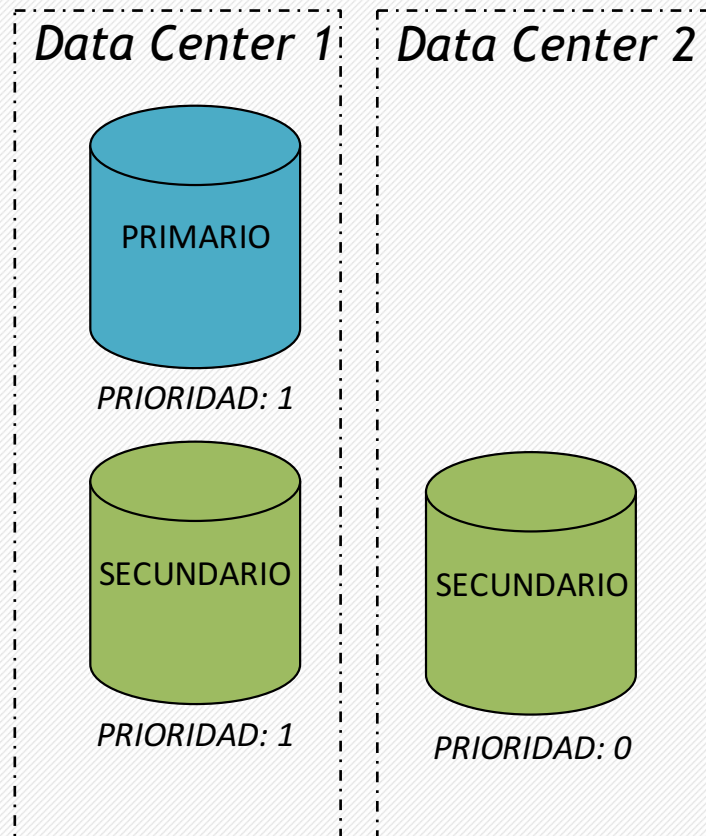
# Replicación de los datos

- ¿Y si hay un empate en las votaciones?.
- *Arbitrer* (Árbitro): instancia *mongod* que se encarga de desempatar las votaciones.
- No almacena réplicas.
- Se recomienda sólo cuando en los *replica sets* se prevean votaciones con empates.



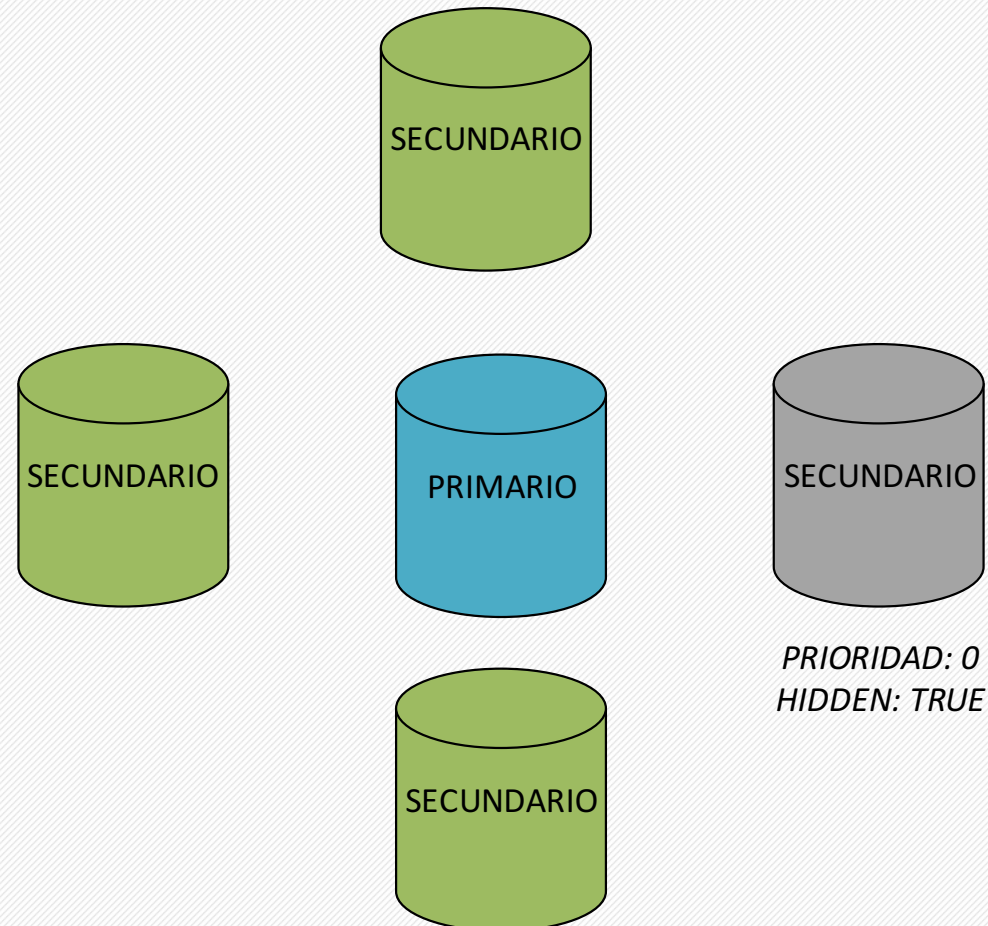
# Casos especiales de secundarios

- *Priority 0* (Prioridad 0): secundarios que no pueden convertirse en primarios.
  - Almacenan las réplicas de los primarios.
  - Votan para elegir primarios al igual que el resto.
  - Pueden ser leídos por el cliente, si la configuración de la BD lo permite.
  - Útil en BDs repartidas en múltiples Data Centers:
    - Se puede “forzar” a que los primarios sólo sean de un *Data Center*.



# Casos especiales de secundarios

- *Hidden replica set*: invisibles a la aplicación cliente.
  - Siguen guardando todas las copias del primario.
  - Participan en las votaciones.
  - No pueden ser escogidos como primarios (Prioridad 0).
  - No pueden ser leídos por la aplicación cliente.
  - Útiles para realizar backups.



# Casos especiales de secundarios

- *Delayed replica set*: guardan las copias del principal “con retraso”.

- No puede convertirse en primario (Prioridad 0).
- No puede ser accesible al cliente (*Hidden*).
- Puede votar en la elección del primario.
- Útil para guardar *backups* y *snapshots*.



PRIORIDAD: 0  
HIDDEN: TRUE  
DELAYED: 3600

# Lectura de datos: configuración

- En la propia consulta, se define la configuración de la lectura.
  - Por defecto, la lectura se hace siempre sobre el primario.
  - Otras posibilidades:
    - *PrimaryPreferred*: si el primario no está disponible, no se espera a que lo esté, se va a un secundario.
    - *Secondary*: siempre sobre secundarios.
    - *SecondaryPreferred*: preferencia sobre secundarios, y si no hay ninguno disponible, sobre el primario.
    - *Nearest: replica set* “más cercano” (menor tiempo de latencia).

# Lectura de datos: consistencia

- En la propia consulta, se define la configuración de lectura.
  - *Primary*: garantiza consistencia. Si el primario no está disponible, se espera a que otro secundario sea elegido como primario.
  - *PrimaryPreferred*: equilibrio entre consistencia y accesibilidad inmediata. Si el primario está caído, no se espera a la votación y se accede a uno secundario.
  - *Secondary*: útil si se quiere repartir la carga de trabajo del primario llevando algunas consultas al secundario. No asegura consistencia. Si no hay secundarios disponibles, no se puede ejecutar.
  - *SecondaryPreferred*: igual que con *Secondary*, pero se garantiza acceso a los datos si no hay secundarios disponibles y sí lo está el primario.
  - *Nearest*: puede ser primario o secundario, prioriza el tiempo de respuesta.

# Almacenamiento de los datos

- Estrategia de almacenamiento de los datos:
  - *Snapshot* en memoria de los últimos datos escritos/modificados.
  - *Checkpoint*: cada 60 segundos o al alcanzar los 2 GB, se escribe el *Snapshot* en disco, siendo la nueva copia perdurable de los datos.
    - Durante la escritura de un nuevo *CheckPoint*, el antiguo sigue siendo válido hasta que este último se escriba por completo.
    - Si ocurre cualquier error, puede recuperarse desde el último *CheckPoint* escrito.
  - *Journal*: MongoDB escribe todas las transacciones del *Snapshot* en un fichero de *log* (*Journal*) para poder recuperar datos escritos entre dos *CheckPoint*.

# Escritura de datos: configuración

- Se pueden definir diferentes niveles de *Write Concern*:
  - Puede no requerirse confirmación alguna: la aplicación cliente supone que se ha escrito el dato.
  - Confirmación de escritura únicamente en primario.
  - Confirmación de escritura en primario y uno o varios secundarios.
  - Confirmación de que se haya escrito o no en el *Journal*.



# Administración: backups

- MongoDB ofrece diferentes estrategias para realizar *backups*:
  - *MongoDB Atlas*: servicio de base de datos (*database-as-a-service*) para administrar *backups*:  
<https://www.mongodb.com/cloud/atlas>
  - *MongoDB Cloud Manager*: soporta operaciones de *backup* y recuperación de bases de datos en MongoDB:  
<https://www.mongodb.com/cloud/cloud-manager>
  - *Ops Manager*: con funciones similares a Cloud Manager, sólo disponible para usuarios de la versión Enterprise:  
<https://docs.opsmanager.mongodb.com/current/tutorial/nav/backup-use/>
  - Otras opciones: *Snapshots*, *cp*, *rsync*, *mongodump*:  
<https://docs.mongodb.com/manual/core/backups/>

# Administración: configuración, mantenimiento y monitorización

- MongoDB ofrece diferentes posibilidades de configuración y mantenimiento:
  - Parámetros de configuración de la base de datos.
  - Consideraciones de seguridad.
  - Configuración de la replicación y *sharding*.
  - Configuración para diagnóstico de la base de datos.

<https://docs.mongodb.com/manual/administration/configuration/>

- También ofrece diferentes herramientas de monitorización.

<https://docs.mongodb.com/manual/administration/monitoring>

# Modelo de datos

- MongoDB es una Base de Datos NoSQL orientada a documentos en formato BSON.
  - Similar a JSON, con algunas particularidades:
    - Tipos de datos adicionales
    - Diseñado para ser más eficiente en espacio.
      - Aunque en ocasiones un JSON puede ocupar menos que un BSON.
    - Diseñado para que las búsquedas sean más eficientes.

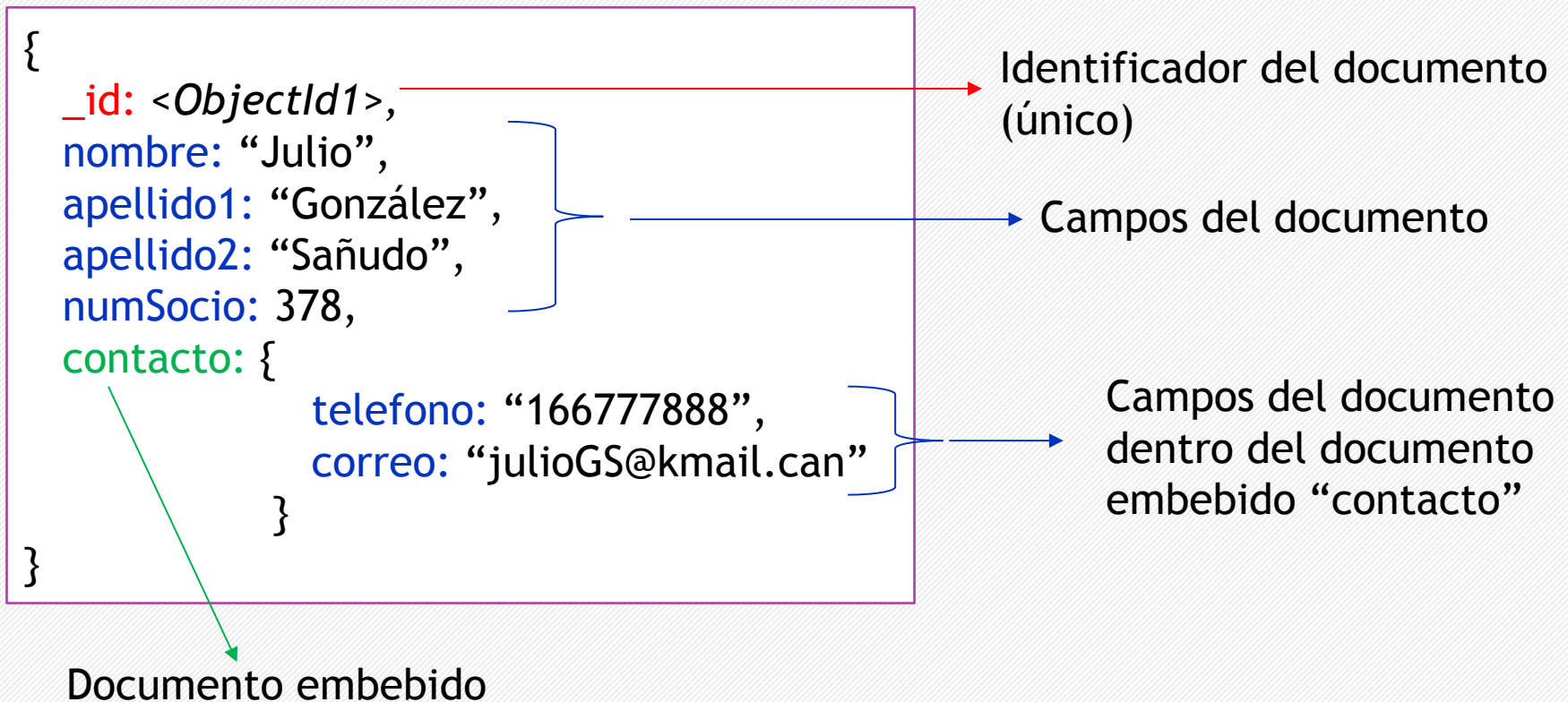
Más información sobre el formato BSON en el siguiente enlace: <http://bsonspec.org/>

# Modelo de datos: conceptos

- **Colección:** conjunto de documentos. Similar con las tablas del modelo relacional, pero...
  - ...no tiene esquema, por lo que cada documento de una colección puede tener diferentes campos.
  - ...cada documento dentro de una colección tiene un identificador único SIEMPRE.
- **Documento:** conjunto de pares campo-valor. Similar a las filas en relacional, pero...
  - ...no siguen un esquema de tabla pre-definido: pueden tener tantos campos como se desee.
  - ...se almacena el nombre del campo junto a su valor.
  - ...no hay valores nulos: o el campo está, o no está (aunque existe un tipo de dato denominado nulo).
  - ...pueden contener otros documentos (jerarquía de documentos).
- **Campos:** campos de un documento a los que se les asigna valor y sobre los que se pueden crear índices. Similar a las columnas en relacional.
  - **\_id:** campo especial que identifica a cada documento en una colección. Si no se le da valor en el insertado, se genera automáticamente. **¡SIEMPRE ESTÁ PRESENTE EN LOS DOCUMENTOS!**

# Modelo de datos: documento

- Ejemplo de documento:



# Modelo de datos: colección

- Ejemplo de colección:

```
{  
  _id: <ObjectId1>,  
  nombre: {  
    apellido1:  
    apellido2:  
    numSocio:  
    contacto:  
  }  
}
```

```
  {  
    _id: <ObjectId2>,  
    nombre: {  
      apellido1:  
      apellido2:  
      numSocio:  
      contacto:  
    }  
  }
```

```
    {  
      _id: <ObjectId3>,  
      nombre: "Julio",  
      apellido1: "González",  
      apellido2: "Sañudo",  
      numSocio: 378,  
      contacto: {  
        telefono: "166777888",  
        correo: "julioGS@kmail.can"  
      }  
    }  
  }
```

# Modelo de datos: colección

- Las colecciones pueden crearse de la siguiente forma:

```
db.createCollection(name, options)
```

- Extraído de:

<https://docs.mongodb.com/manual/reference/method/db.createCollection/>

- Ejemplo de creación de colección:

Nombre de la colección

Opciones: tamaño, autoíndice...

```
db.createCollection("socios", {size: 2145678899,  
autoIndexId: true, ...})
```

- En mongoDB, las colecciones se crean automáticamente la primera vez que se hace referencia a ellas (por ejemplo, al insertar un primer documento), por lo que no es necesario utilizar "createCollection". Este es sólo necesario si se quieren modificar los valores por defecto de las opciones.

# Modelo de datos: tipos de dato

- Numéricos:
  - *Double*: coma-flotante de 64 bits.
  - *Decimal*: coma-flotante de 128 bits.
  - *Int*: enteros de hasta 32 bits.
  - *Long*: enteros de hasta 64 bits.
- Texto:
  - *String*: strings UTF-8.
  - *Regex*: almacena expresiones regulares.
- De fecha:
  - *Date*: entero de 64 bits que representa el número de milisegundos desde el 1 de enero de 1970.
  - *Timestamp*: entero de 64 bits, en el que los primeros 32 bits representan los segundos pasados desde el 1 de enero de 1970, y los otros 32 bits son ordinales incrementales. En una instancia mongod, cada valor de timestamp es único.



# Modelo de datos: tipos de dato

- Especiales:
  - *Array*: almacena un conjunto de elementos de cualquier tipo.
  - *ObjectId*: tipo de dato único, principalmente utilizado para dar valor al campo `_id` de los documentos.
  - *Javascript*: código javascript.
  - *Null*: valor nulo.
  - *Boolean*: valor booleano.

Más información sobre los tipos de datos BSON en los siguientes enlaces:

<https://docs.mongodb.com/manual/reference/bson-types/>  
<http://bsonspec.org/spec.html>

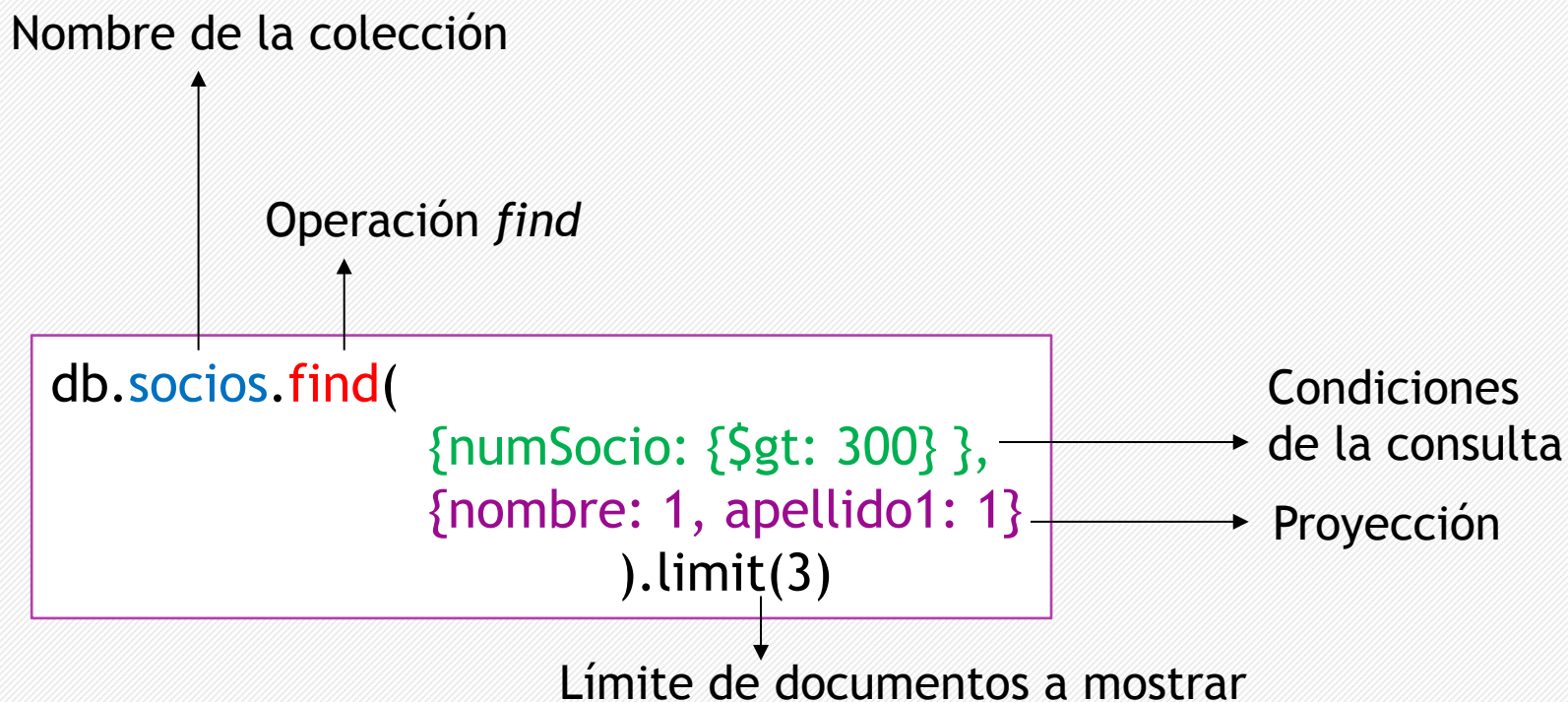
# Consulta y modificación de datos: operaciones CRUD

- Los datos son accedidos mediante operaciones *CRUD* (*Create, Read, Update, Delete*):
  - Find: lectura (Read) de datos. Símil con el SELECT en SQL.
  - Insert: escritura (Create) de datos.
  - Update: actualización de valores en los datos.
  - Remove: borrado (Delete) de datos.
- La ejecución de cualquiera de estas instrucciones sigue el siguiente esquema:

db.<nombre\_colección>.<nombre\_operación>(<condiciones de la operación>)

# Consultas: operación *find*

- Ejemplo de uso de *find*:



# Consultas: operación *find*

- Ejemplo de uso de *find*: los campos consultados pueden ir opcionalmente entre comillas simples o dobles. Las tres consultas que se muestran a continuación funcionan exactamente igual:

```
db.socios.find({numSocio: {$gt: 300} })
```

```
db.socios.find({'numSocio': {$gt: 300} })
```

```
db.socios.find({"numSocio": {$gt: 300} })
```

# Consultas: operación *find*

- Para acceder a campos de documentos embebidos, se utiliza la notación con punto. En este caso, el uso de las comillas es necesario:

```
db.socios.find(  
    {numSocio: {$gt: 300} },  
    {nombre: 1, apellido1: 1, "contacto.teléfono": 1}  
).limit(3)
```

# Consultas: condiciones

- Existen diferentes tipos de operaciones:

De igualdad (\$eq):

```
db.socios.find({numSocio: {$eq: 300}})
```

=

```
db.socios.find({numSocio: 300})
```

*Forma implícita de la operación \$eq*

Mayor que (\$gt):

```
db.socios.find({numSocio: {$gt: 300}})
```

Mayor o igual que (\$gte):

```
db.socios.find({numSocio: {$gte: 300}})
```

Menor que (\$lt):

```
db.socios.find({numSocio: {$lt: 300}})
```

Menor o igual que (\$lte):

```
db.socios.find({numSocio: {$lte: 300}})
```

Diferente a (\$ne):

```
db.socios.find({numSocio: {$ne: 300}})
```

Pertenece (\$in) o no (\$nin) a un conjunto

```
db.socios.find({numSocio: {$in: [300, 301]}})  
db.socios.find({numSocio: {$nin: [300, 301]}})
```

# Consultas: condiciones

- Operadores lógicos:

Ambas condiciones han de cumplirse (\$and):

```
db.socios.find({$and: [{nombre: "Juan"}, {apellido1: "Galindo"} ]})
```

Alguna de las condiciones ha de cumplirse (\$or):

```
db.socios.find({$or: [{nombre: "Juan"}, {apellido1: "Galindo"} ]})
```

No debe de cumplir la condición (\$not):

```
db.socios.find({numSocio: { $not: { $eq: 300 } }})
```

Que no se cumpla ninguna condición (\$nor):

```
db.socios.find({$nor: [{nombre: "Juan"}, {apellido1: "Galindo"} ]})
```

# Consultas: condiciones

- Los operadores lógicos puede anidarse:

```
db.socios.find({
  $or: [
    {$and: [{nombre: "Juan"}, {apellido1: "Galindo" } ]},
    {numSocio: 300}
  ]
})
```



# Consultas: condiciones

- Operadores lógicos sobre campos:

Existe el campo (\$exists):

```
db.socios.find({ apellido2: { $exists: true } })
```

El campo es de un tipo concreto (\$type):

```
db.socios.find({ apellido2: { $type: "string" } })
```

- Operadores lógicos sobre arrays:

El array contiene todos los valores (\$all), algún valor del array cumple las condiciones (\$elemMatch), el array es de un tamaño concreto (\$size):

```
db.serie.find({ tags: { $all: ["historia", "aventura"] } })  
db.serie.find({ puntuacion: { $elemMatch: { $gte: 2, $lt: 5 } } })  
db.serie.find({ tags: { $size : 3 } })
```

# Consultas: condiciones

- Proyecciones:
  - Valor 1 si queremos mostrar el campo.
  - Valor 0 si no queremos mostrarlo.
  - Por defecto, si no se declara nada en la proyección, se muestran todos los campos:
    - Al poner un campo a 1, se muestra solamente ese campo.
      - ¡OJO!: el campo `_id` se muestra siempre por defecto aunque haya otros campos en la proyección con valor 1.
      - Si no se quiere mostrar el campo `_id`, hay que ponerlo a 0 explícitamente en la proyección.
- Ejemplos de proyecciones:

```
db.socios.find({numSocio: {$eq: 300}}, { nombre: 1, apellido1: 1 })
```

→ Muestra nombre, apellido1 y `_id`

```
db.socios.find({numSocio: {$eq: 300}}, { nombre: 1, apellido1: 1, _id: 0 })
```

→ Muestra nombre y apellido1

```
db.socios.find({numSocio: {$eq: 300}}, { numSocio: 0 })
```

→ Muestra todos los campos excepto `numSocio`

# Consultas: modificadores

- *Sort*: ordena los resultados según los campos dados. Un valor de 1 indica ordenamiento ascendente, y un valor de -1 ordenamiento descendente:

```
db.socios.find({numSocio: {$eq: 300}}, { nombre: 1, apellido1: 1 }).sort({nombre: 1})
```

- *Limit*: limita el número de colecciones retornadas. El 0 indica que no hay límite:

```
db.socios.find({nombre: {$eq: "Juan"}}, { nombre: 1, apellido1: 1 }).limit(3)
```

# Consultas: agrupamiento

- *Count*: cuenta el número de veces que aparece en los documentos de una colección un valor en un campo.
- Ejemplo: suponiendo que en una colección llamada “series” tenemos un campo puntuación, y los siguientes documentos:

{puntuacion: 5}, {puntuacion: 3}, {puntuacion: 5}, {puntuacion: 5}, {puntuacion: 2},

La siguiente instrucción retornaría 3 (3 veces que se repite el valor 5 en el campo puntuación:

```
db.series.count({puntuacion: 5})
```

# Consultas: agrupamiento

- *Distinct*: Retorna un array con los diferentes valores que tiene un campo concreto en los documentos de una colección.
- Ejemplo: suponiendo que en una colección llamada “series” tenemos un campo puntuación, y los siguientes documentos:

{puntuacion: 5}, {puntuacion: 3}, {puntuacion: 5}, {puntuacion: 5}, {puntuacion: 2},

La siguiente instrucción retornaría el array [5, 3, 2] (los tres valores que toma el campo puntuación):

```
db.series.distinct("puntuacion")
```

# Consultas: agrupamiento

- *Group*: agrupa los elementos que cumplan una condición concreta (*cond*) bajo los distintos valores de un campo (*key*).
- Ejemplo: suponiendo que en una colección llamada “series” tenemos un campo puntuación y un campo numSerie, y los siguientes documentos:

```
{numSerie: 1, puntuacion: 5}, {numSerie: 1, puntuacion: 3}, {numSerie: 2, puntuacion: 5},  
{numSerie: 1, puntuacion: 5}, {numSerie: 2, puntuacion: 2}, {numSerie: 3, puntuacion: 5}
```

- Si se ejecutase la siguiente instrucción:

```
db.series.group({ key: { numSerie: 1 }, cond: { numSerie: { $lte: 2 } }, reduce: function(cur, result) {  
  result.puntuacion += cur.puntuacion }, initial: { puntuacion: 0 } })
```

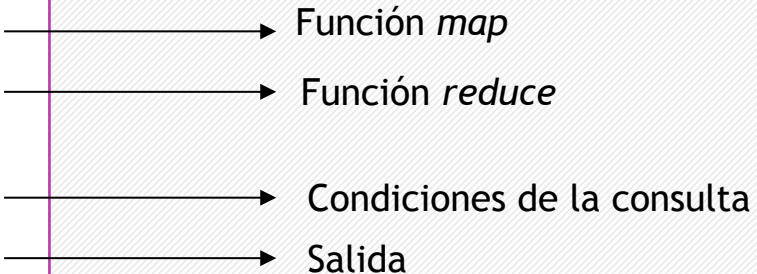
- El resultado sería la suma de las puntuaciones agrupando por numSerie, siempre que el numSerie sea igual o menor que 2:

```
{numSerie: 1, puntuacion: 13}, {numSerie: 2, puntuacion: 7}
```

# Consultas: agrupamiento

- *Mapreduce*: Utiliza el método *map/reduce* para hacer consultas y agrupamientos. La instrucción *group* utiliza *map/reduce* implícitamente. Sintaxis:

```
db.<nombre_colección>.mapreduce(  
    function() ...,  
    function(key, values)...,  
    {  
        query: {...},  
        out:...  
    }  
)
```



Función *map*

Función *reduce*

Condiciones de la consulta

Salida

Para más información sobre las funciones de agrupamiento, se recomienda visitar el siguiente enlace: <https://docs.mongodb.com/manual/aggregation/>

# Insertado: operación *insert*

- Ejemplo de uso de *insert*:

Nombre de la colección

Operación *insert*

```
db.socios.insert({
  numSocio: 300,
  nombre: "Pablo"
  ...
})
```

Valores  
asignados a  
cada campo

- Si no se inserta un valor para el campo `_id`, se genera automáticamente:
  - `_id` único, no hay peligro de *UPSERTS*: si se inserta un valor existente en la colección, devuelve error.

Más sobre la operación *insert* en <https://docs.mongodb.com/manual/tutorial/insert-documents/>



# Actualizado: operación *update*

- Ejemplo de uso de *update*:

Nombre de la colección

Operación *update*

```
db.socios.update(
```

```
  {numSocio: 300},
```

```
  {$set: {nombre: "Pablo"}}
```

```
  {multi: true}
```

```
)
```

Condición

Valor nuevo

Opción.

- La operación *update* admite opciones: `multi:true` indica que se pueden modificar varios documentos simultáneamente.
- UPSERT: si se establece `{upsert: true}` y no hay documentos que cumplan las condiciones, se crea uno nuevo con los valores proporcionados.

Más sobre la operación *update* en <https://docs.mongodb.com/manual/tutorial/update-documents/>

# Actualizado: operación *remove*

- Ejemplo de uso de *remove*:

Nombre de la colección

Operación *remove*

```
db.socios.remove(
```

```
    {numSocio: 300}  
)
```

Condición

# Optimización de consultas: índices

- Los índices sobre los campos ayudan a optimizar las consultas.
- Sintaxis:

```
db.<nombre_colección>.createIndex({<campo>: <orden>, ...})
```

Más información sobre índices en <https://docs.mongodb.com/manual/indexes/>

- Tipos de índices en mongoDB:
  - *Default\_id*: existe por defecto sobre el campo `_id`. Garantiza unicidad.
  - *Single field*: índice creado sobre un solo campo de la colección. Puede ser ascendente (1) o descendente (-1):

```
db.socios.createIndex({"nombre": 1})
```

- *Compound*: índice creado sobre varios campos de la colección:

```
db.socios.createIndex({"nombre": 1, "apellido1": -1})
```

- Multikey Index: si se declara un índice sobre un campo *array*, se crea sobre cada elemento del *array*.
- Otros índices: geoespaciales, sobre texto, etc.

# Optimización de consultas: índices

- Algunas propiedades de los índices:
  - TTL: índices que “caducan” al cabo de un tiempo.

```
db.socios.createIndex({"nombre": 1}, {expireAfterSeconds: 7200})
```

- Unique: garantizan unicidad.

```
db.socios.createIndex({"numSocio": 1}, {unique: true})
```

- Partial: se crean exclusivamente sobre documentos con valores concretos en el campo (índice condicionado)

```
db.socios.createIndex({"nombre": 1}, {partialFilterExpression:  
{numSocio: { $gt: 300}}})
```

- Sparse: sólo se crea el índice en documentos que contienen el campo.

```
db.socios.createIndex({"nombre": 1}, {sparse: true})
```

# Optimización de consultas: índices

- MongoDB tiene internamente un optimizador de consultas que decide cuál es el plan de consulta más eficiente en base a los índices disponibles.
- Se guarda en memoria: *Query Plan Cache Methods* (<https://docs.mongodb.com/manual/reference/method/js-plan-cache/>).
- El usuario puede indicar que índices desea que el optimizador tenga en consideración.
- El método `explain()` nos permite ver el plan llevado a cabo para una consulta:

```
db.socios.find().explain()
```

- Es un proceso empírico que se reevalúa constantemente:
  - Cada 1000 operaciones de escritura sobre una colección.
  - Cuando se añaden o eliminan índices.
  - Cuando se reconstruye un índice.
  - Cuando el proceso *mongod* se reinicia.

# Optimización de consultas: índices

- Tras cada *Insert*, *Update* y *Delete*, MongoDB tiene que actualizar todos los índices de la colección. Esto añade sobrecarga a este tipo de operaciones.
- ¡Hay que ser cuidadosos a la hora de crear índices!. Solamente se han de crear aquellos con los que el beneficio en las consultas supere al perjuicio causado en las escrituras.
  - Si las consultas de tu aplicación no utilizan determinados campos, o lo hacen muy poco, no crees índices sobre ellos.

# Modelado de datos

- En mongoDB, el esquema de los datos es flexible, correspondiendo, por tanto, al desarrollador decidir como implementarlo según los requisitos que se tengan.
- Se pueden definir 3 tipos de relaciones:
  - One-to-one (uno a uno): en este tipo de relación, uno de los documentos suele embeberse dentro de otro.
  - One-to-many embebido (uno-a-varios): los documentos de la relación se embeben dentro de otro en una estructura de tipo *array*.
  - One-to-many con referencias (uno-a-varios): se utilizan referencias al `_id` de los documentos relacionados, en vez de embeberlos por completo.

# Modelado de datos: *One-to-one*

- Las relaciones uno-a-uno pueden modelarse embebiendo uno de los documentos dentro de otro.
- Supongamos que almacenamos los datos de los socios de un gimnasio junto con su dirección, que se compone de sus propios campos. La solución pasaría por embeber la dirección como un subdocumento del documento principal:

```
{ _id: <ObjectId3>,  
  nombre: "Julio",  
  apellido1: "González",  
  ...  
  dirección: {  
    calle: "Avenida de la República",  
    numero: 678  
    ...  
  }  
}
```



# Modelado de datos: *One-to-many* embebido

- Las relaciones uno-a-varios pueden modelarse embebiendo documentos dentro de un campo *array*.
- Siguiendo con el ejemplo anterior, imaginemos que los socios pueden tener varias direcciones:

```
{ _id: <ObjectId>,
  nombre: "Julio",
  apellido1: "González",
  ...
  direcciones: [{
    calle: "Avenida de la República",
    numero: 678
    ...},
    {calle: "Avenida de la República",
    numero: 678
    ... }
  ] }
```

# Modelado de datos: *One-to-many* con documentos embebidos en arrays

- En ocasiones, puede ser muy pesado almacenar arrays con todos los datos de los subdocumentos.
- Imaginemos que queremos almacenar libros junto con la referencia a los editores\*. Siguiendo los ejemplos anteriores, podríamos almacenar los libros en un array como subdocumentos de los documentos de los editores:

```
{ _id: "edicionesSotileza",  
  ciudad: "Santander",  
  ...  
  libros: [{  
    nombre: "Historia de Cantabria",  
    ISBN: "678789789"  
    ...},  
    nombre: "El pleito de los nueve valles",  
    ISBN: "2671982093"  
    ...  
  ]  
}
```

\*Ejemplo extraído, traducido y adaptado de <https://docs.mongodb.com/manual/tutorial/model-referenced-one-to-many-relationships-between-documents/>

# Modelado de datos: *One-to-many* con documentos embebidos

- El problema de la anterior solución es que el array podría crecer en exceso, con lo que las búsquedas sería muy ineficientes.
- Podríamos pensar como solución el embeber a los editores en los libros:

```
{ _id: "HdC1",  
  nombre: "Historia de Cantabria",  
  ISBN: "678789789"  
  ...  
  editor: { nombre: "edicionesSotileza",  
            ciudad: "Santander",  
            ...}  
}
```

# Modelado de datos: *One-to-many* con referencias

- El problema de la anterior solución es que se repetirían con frecuencia los datos de los editores, ocupando una gran cantidad de espacio.
- Podríamos pensar como solución el referenciar a los editores en los libros:

```
{ _id: "edicionesSotileza",  
  ciudad: "Santander",  
  ... }
```

```
{ _id: "HdC1",  
  nombre: "Historia de Cantabria",  
  ISBN: "678789789"  
  ...  
  editor: "edicionesSotileza" }
```

# Distribuyendo las colecciones: *Shard Key*

- Si se quisiese hacer uso del potencial del *Sharding* y distribuir así una colección en varios *Shards*, habría de crearse una *Shard Key* sobre uno o varios campos. La sintaxis para la creación de la *Shard Key* es la siguiente:

```
sh.shardCollection(<namespace>,  
                  <key>)
```

Nombre completo de la colección sobre la que se va a hacer *Sharding*, en formato <database>.<collection>

Campo o campos indexados que van a formar parte de la *Shard Key*, en formato {<field>:<direction> | “hashed”}

- Características de la *Shard Key*:
  - Puede ser *hashed* o *ranged* (ver transparencias 10 y 11).
  - Una colección sobre la que se haga *Sharding* ha de tener un índice que de soporte a la *Shard Key*. Puede ser un índice simple creado sobre el campo de la *Shard Key* o un índice compuesto en el que la *Shard Key* sea prefijo (primer campo) del mismo.
    - Si la colección está vacía, la anterior instrucción crea automáticamente el índice si no existe.
    - Si la colección no está vacía, el índice ha de ser creado previamente.
  - En una colección que haga *Sharding*, únicamente el campo `_id` y el índice definido en la *Shard Key* pueden ser únicos. También se permite unicidad en un índice compuesto en el que la *Shard Key* sea el prefijo.
    - Una *Hashed Shard Key* no puede ser única.

# Distribuyendo las colecciones: *Shard Key*

- Para crear una *Hashed Shard Key*, se utiliza la siguiente sintaxis:

```
sh.shardCollection("<database>.<collection>", {<field> : "hashed"})
```

- Ejemplo:

```
sh.shardCollection("db.socios", {numSocio : "hashed"})
```

- Para crear una *Ranged Shard Key*, ha de indicarse el ordenamiento de la misma (1 ascendente, -1 descendente):

```
sh.shardCollection("<database>.<collection>", {<field> : <ordering>})
```

- Ejemplo:

```
sh.shardCollection("db.socios", {numSocio : 1})
```

# Ventajas de MongoDB

- Esquema muy flexible: cada documento de la colección puede almacenar campos diferentes.
- Lenguaje de consulta y manipulación sencillos (operaciones CRUD).
- Facilidad de integración con aplicaciones gracias al uso del lenguaje BSON, fácilmente traducible a JSON.
- Accesibilidad a los datos.
  - Posibilidad de realizar lecturas en instancias secundarias, repartiendo la carga de trabajo.

# Desventajas de MongoDB

- Aplicaciones clientes más complejas de desarrollar al trabajar con esquemas flexibles, desnormalizados y dinámicos.
- No garantiza ACID:
  - Consistencia eventual, podrían leerse datos de nodos secundarios que aún no estén actualizados.
  - Sin soporte transaccional.