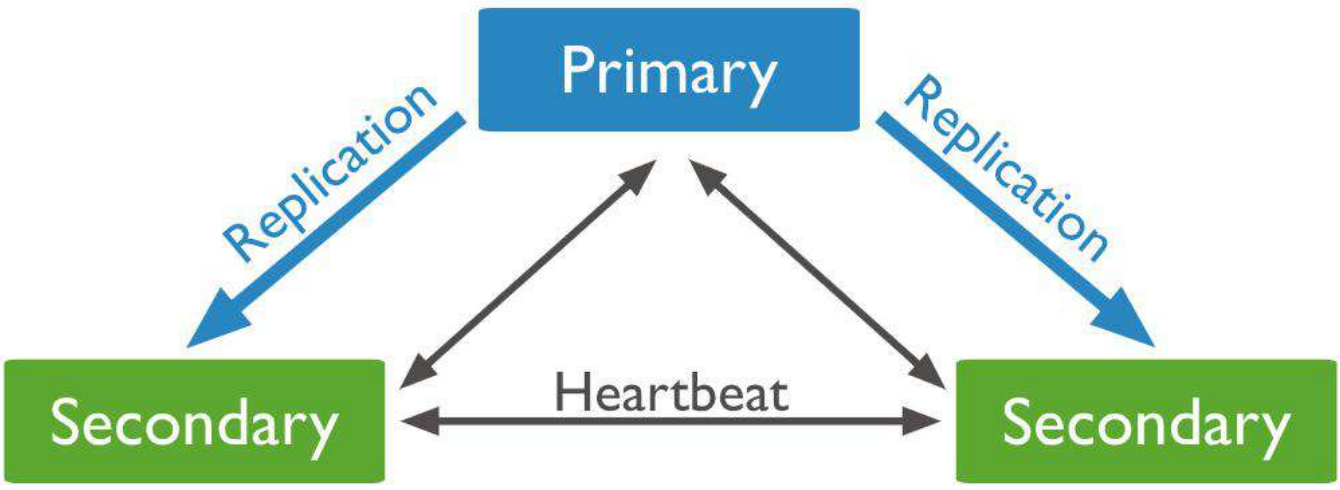
# Replicación

## Índice



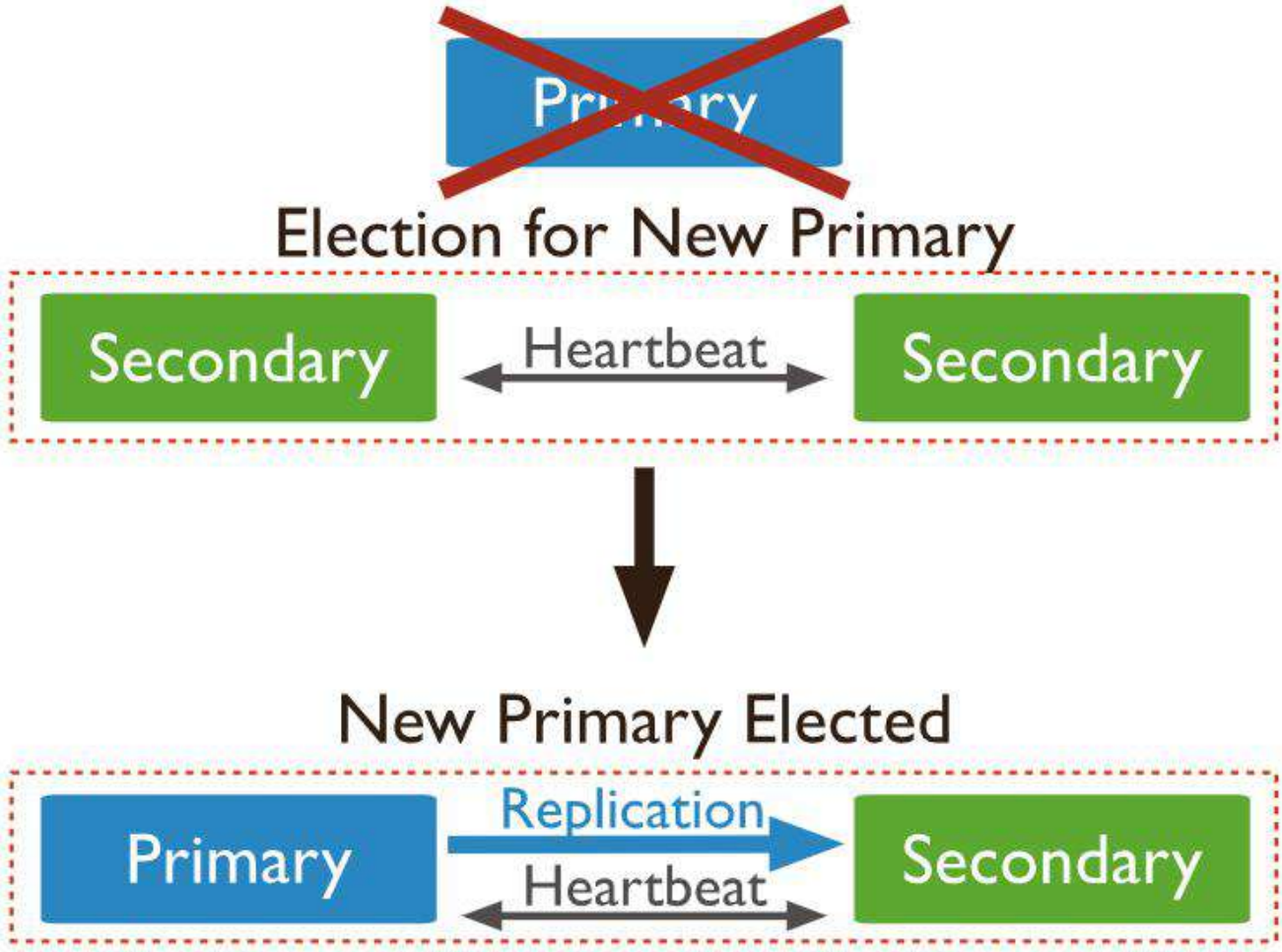
## Introducción

- El objetivo de la replicación es el de mantener múltiples copias de los datos para asegurar alta disponibilidad (high availability) y automática tolerancia a fallos (automatic failover).
- En MongoDB, un replica set o conjunto de réplicas es una serie de nodos que mantienen copias de los mismos datos.
- Los datos son manejados por un nodo principal o nodo primario que sincroniza los datos de forma asíncrona con el resto de nodos secundarios (el cliente podría tener aceptación de que la escritura de los datos se realizó con éxito, pero por defecto no se garantiza la propagación hacia el resto de nodos).




- Para facilitar la replicación, todos los miembros del replica set envían latidos (pings o heartbeats) al resto de miembros.

- Si el nodo primario cae (sucede cuando transcurren 10 segundos sin que los nodos secundarios hayan recibido el heartbeat del nodo primario), uno de los nodos secundarios toma el rol de nodo primario.
- Los nodos secundarios deciden quién será el nodo primario mediante una elección. Si el nodo que ha caído se recupera, se une de nuevo al replica set como nodo secundario y descarga la nueva información actualizada del resto de nodos.



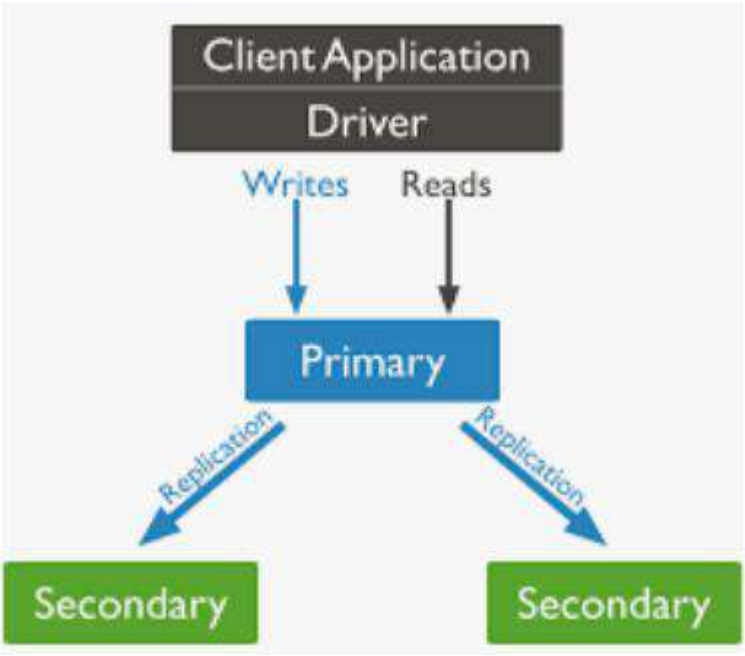

- El únido nodo que permite escrituras en la base de datos es el nodo primario. En los nodos secundarios no se pueden escribir datos, ni tampoco leer (salvo casos excepcionales). La aplicación (a través del driver) realiza Las lecturas y escrituras se realizan en el nodo primario (la conexión se realiza a través de una sucesión de URL de conexión).
- Un replica set puede tener hasta 50 nodos, pero solo un máximo de 7 nodos pueden votar y solamente uno puede convertirse en el nodo primario.
- En el mundo de las bases de datos la replicación de los datos puede realizarse de dos formas:
    - Mediante replicación binaria: el nodo primario examina los bytes exactos que cambiaron en los archivos de datos y los transmite a los nodos secundarios, que escriben los cambios en las mismas localizaciones de bytes. Este tipo de replicación es muy rápida (pocos datos son transferidos) y simple, aunque los nodos secundarios desconocen qué instrucciones se están ejecutando. La gran desventaja es que habitualmente se requiere de las mismas versiones de sistemas operativos y de bases de datos.

    - Mediante replicación basada en sentencias (statements): las instrucciones de escritura son almacenadas en el nodo primario y transmitidas a los nodos secundarios, donde son ejecutadas. Este tipo de replicación es agnóstica de tecnología y de versiones. La replicación basada en sentencias (statements) es más precisa que la replicación binaria.

- MongoDB utiliza replicación basada en statements.
- Un nodo que no utiliza replicación se llama nodo standalone.
- MongoDB aplica operaciones de escritura en lotes usando múltiples hilos para mejorar la concurrencia. MongoDB agrupa los lotes por espacio de nombres (MMAPv1) o por documento id (WiredTiger) y simultáneamente aplica cada grupo de operaciones usando un hilo diferente.

## Nodos

- El protocolo que utilizan los nodos secundarios para sincronizar los datos es asíncrono y puede ser pv1 (protocol version 1) y pv0 (protocol version 0).
- pv1 es el valor predeterminado para todos los replica set creados con MongoDB 3.2 o posterior. A partir de la versión 4.0, MongoDB solamente admite el protocolo pv1.
- pv está basado en el protocolo RAFT.
- pv1 está basado en la existencia del operation log (oplog), que mantiene todas las operaciones de escrituras del replica set.
- Cuando una operación de escritura se ha ejecutado exitosamente en el nodo primario, se registra en el oplog en un formato idempotente (las operaciones de oplog producen los mismos resultados aunque se apliquen una o varias veces al mismo conjunto de datos).
- Para replicar datos, un nodo secundario aplica las operaciones del oplog primario a su propio conjunto de datos en un proceso asíncrono.
- Un replica set posee al menos tres nodos. No es posible tener solamente dos. Alternativamente, puede desplegarse un replica set de tres miembros con dos miembros que contienen datos: uno primario, uno secundario y un árbitro, pero no es buena práctica porque provee peor redundancia de los datos.



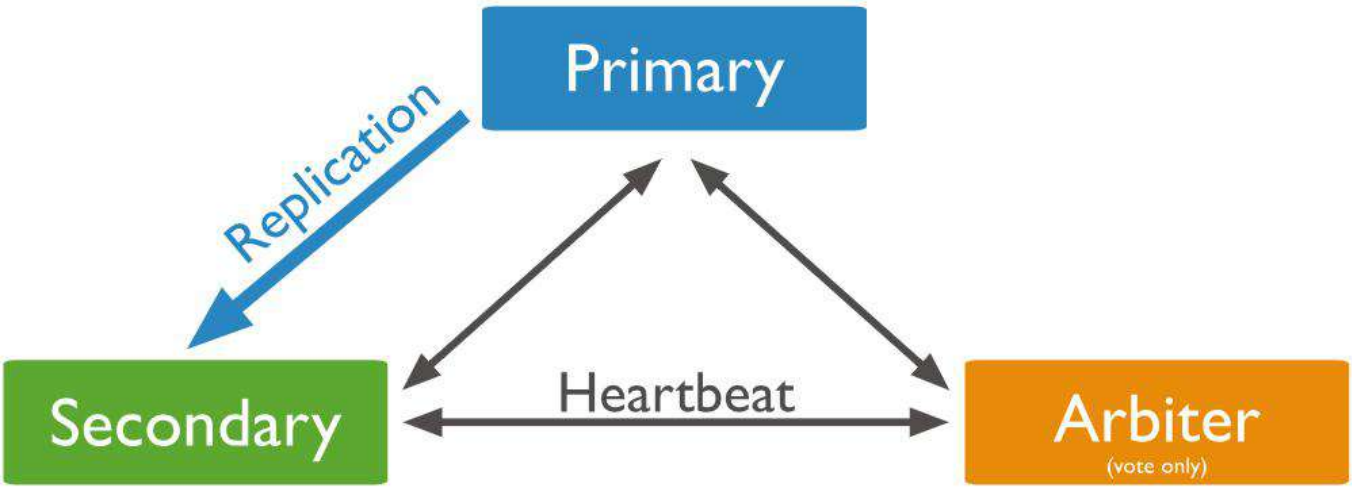

- Existen diferentes tipos de nodos secundarios: normal/regular (primario o secundario), árbitro, oculto, retrasado (que es un tipo de nodo oculto) y no votante. A nivel de configuración es necesario conocer los parámetros priority, slaveDelay, hidden

### Nodo normal/regular

- Tienen un funcionamiento normal con respecto a la replicación de los datos. El nodo primario es siempre un nodo normal/regular.

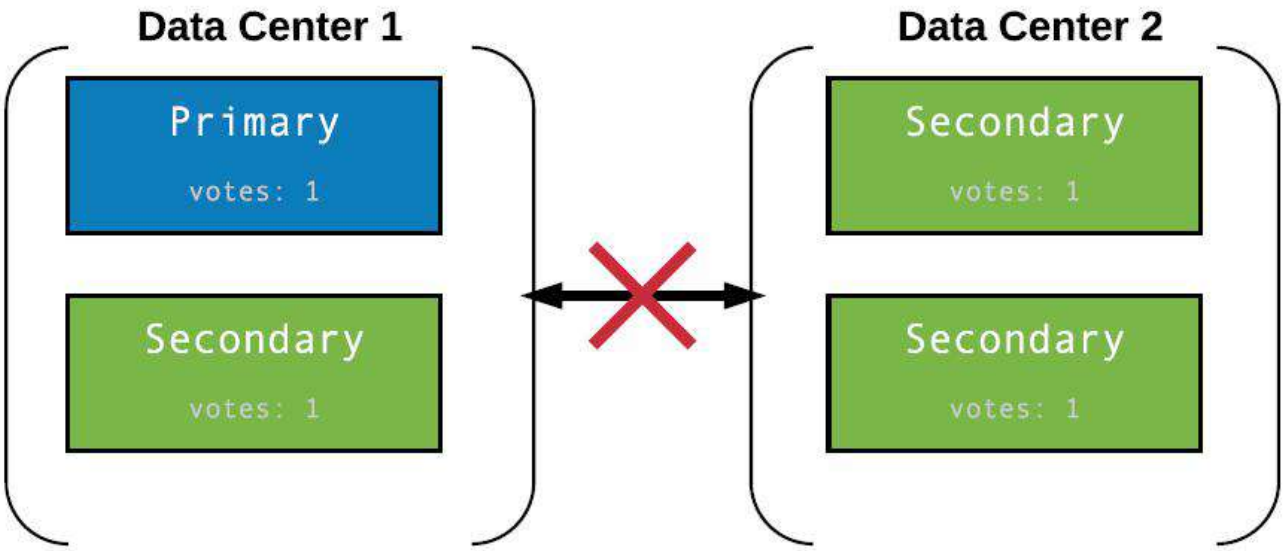
### Nodo árbitro

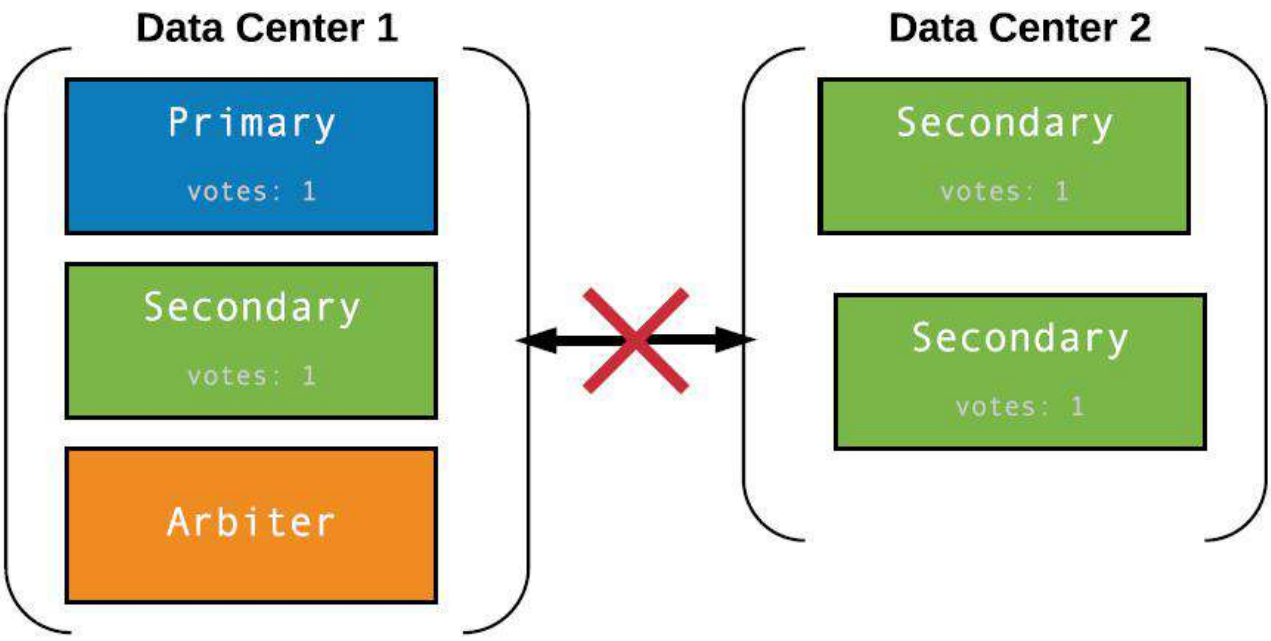
- Se trata de un nodo que no mantiene los datos replicados y no puede convertirse en nodo primario, pero puede votar en una elección. Es recomendable que no haya más de un árbitro por replica set y no deberían ser alojados en los mismos sistemas que alojan los nodos primarios o secundarios.
- Los árbitros siempre tienen exactamente 1 voto de elección y, por lo tanto, permiten que los replica set tengan un número desigual de miembros votantes sin la sobrecarga de un miembro adicional que replique datos.




- Para añadir un nodo árbitro es necesario utilizar el método rs.addArb.

- El uso de un árbitro permite garantizar fácilmente un número impar de votantes en los conjuntos de réplicas. ¿Porque es esto importante? Para protegerse contra particiones de red.
    - Considérese que un replica set tiene un número par de nodos (4, por ejemplo). Si ocurre una desafortunada partición de red que divide el conjunto en dos (2 + 2). ¿Qué partición debe aceptar escrituras? ¿Primero? ¿Segundo? ¿Ambos? Con un árbitro corriendo en una de las particiones de red no existe ese problema porque tendría mayoría esa partición de red y podría elegirse un miembro primario.
    - La mayoría es calculada a partir de --> floor(n/2) + 1, donde n el número de miembros del replica set.

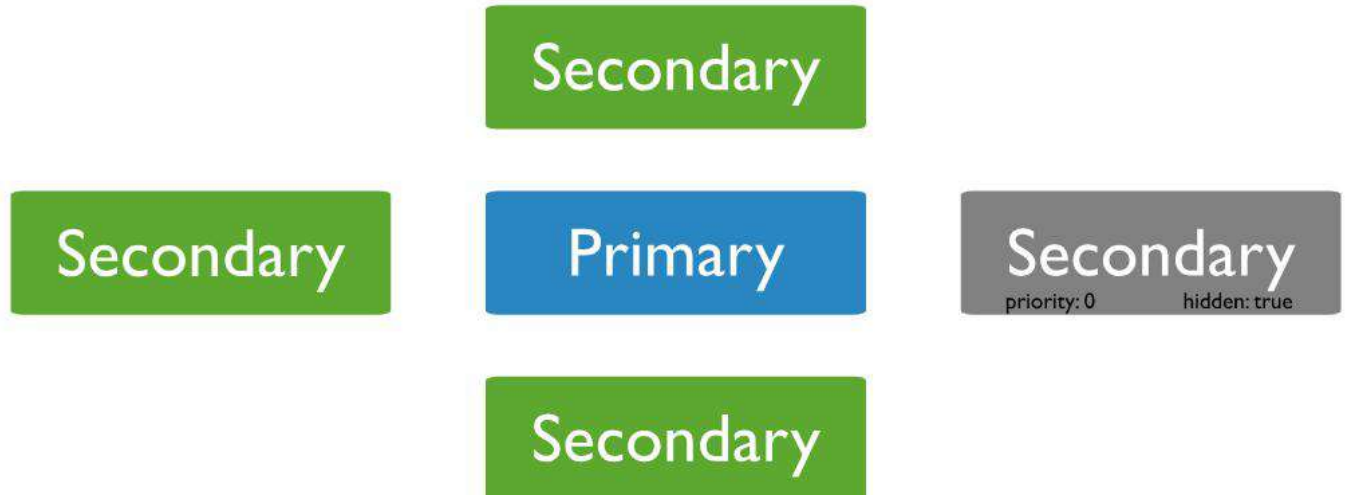



- A partir de MongoDB 3.6, los árbitros tienen prioridad 0. Cuando se actualiza un replica set a MongoDB 3.6, si la configuración existente tiene un árbitro con prioridad 1, MongoDB 3.6 reconfigura al árbitro para que tenga prioridad 0.

## Nodo oculto

- Estos nodos son invisibles para la aplicación, replican los datos desde el nodo primario y votan en las elecciones, pero no pueden ser elegidos como nodos primarios.
- Estos nodos son recomendados para tareas específicas como generación de informes y copias de seguridad.
- Son invisibles para la aplicación (no reciben nunca operaciones de lectura). El único tráfico que reciben es el de la replicación.

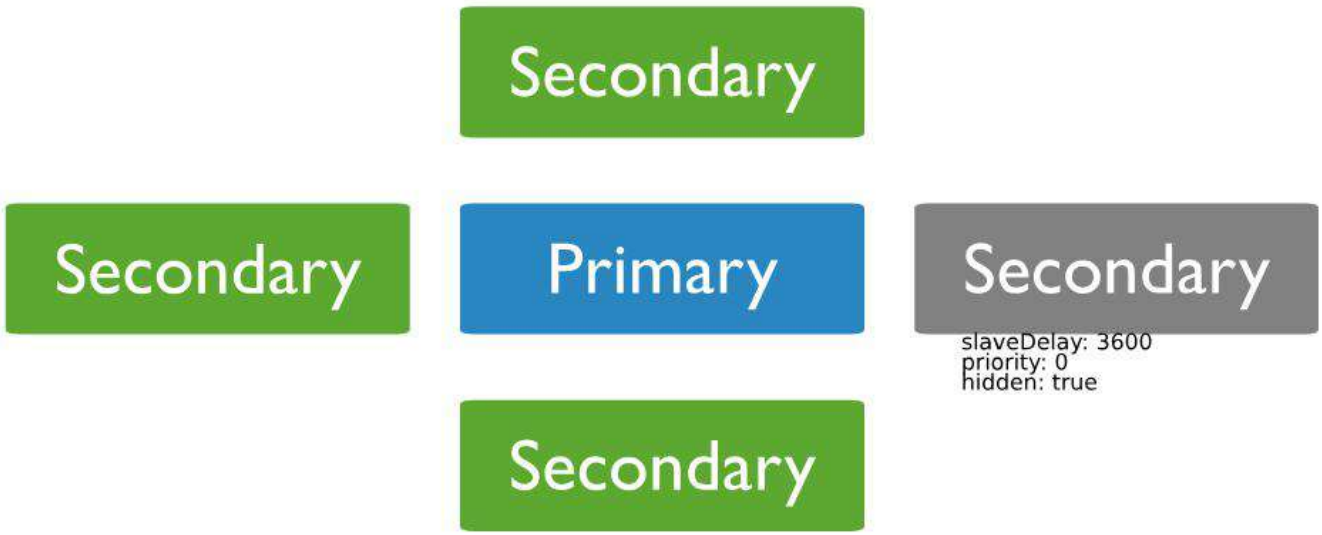



```
{
  "_id" : <num>,
  "host" : <hostname:port>,
  "priority" : 0,
  "hidden" : true
}
```

- En un cluster sharding, mongos no interactúa con miembros ocultos.
- Si se detiene un miembro oculto que vota, es conveniente asegurarse de que el conjunto tenga una mayoría activa o el primario reunciará (step down).
- Para propósitos de backup, el método db.fsyncLock sobre el nodo oculto, puede garantizar que los archivos de datos no cambien para las instancias de MongoDB utilizando los motores de almacenamiento MMAPv1 o WiredTiger, lo que brinda consistencia de los datos durante la creación de la copia de seguridad.
- En versiones anteriores de MongoDB 3.2, db.fsyncLock podía garantizar un conjunto consistente de archivos para copias de seguridad de bajo nivel (por ejemplo, a través de copia de archivo cp, scp, tar) para WiredTiger.
- db.fsyncLock() obliga al mongod a vaciar todas las operaciones de escritura pendientes en el disco y bloquea toda la instancia de mongod para evitar escrituras adicionales hasta que el usuario libere el bloqueo con el comando db.fsyncUnlock() cuando corresponda.

### Nodo retrasado

- Los nodos también pueden ser configurados con un determinado retraso de tiempo en el proceso de replicación y además son también nodos ocultos al mismo tiempo. Son los llamados nodos delayed o retrasados.
- Estos nodos permiten resistencia a corrupción de datos a nivel de aplicación sin necesidad de confiar en copias de backup en frío si sucede un imprevisto. Por tanto, permite una especie de backups retrasados o snapshot.
- Estos nodos retrasados pueden votar también en las elecciones, están ocultos y no pueden ser elegidos como nodos primarios (priority = 0).




```
{
   "_id" : <num>,
   "host" : <hostname:port>,
   "priority" : 0,
   "slaveDelay" : <seconds>,
   "hidden" : true
}
```

- El retraso especificado debe ser menor que la capacidad del oplog.
- En los shards, los miembros retrasados tienen una utilidad limitada cuando el equilibrador está habilitado. Debido a que los miembros retrasados replican las migraciones de chunks con un retraso, el estado de los miembros retrasados en un grupo fragmentado no es útil para recuperar un estado anterior del grupo fragmentado si se producen migraciones durante la ventana de retraso.

### Nodo no votante

- Los miembros sin derecho a voto permiten agregar miembros adicionales para la distribución de lectura más allá del máximo de siete miembros con derecho a voto.
- Para configurar un miembro como no votante, hay que establecer sus votos y valores de prioridad en 0.

```
cfg = rs.conf();
/* convertir el nodo cuarto en un nodo no votante */
cfg.members[3].votes = 0;
cfg.members[3].priority = 0;
rs.reconfig(cfg);
```

- El número de votos que tiene cada miembro es 1 o 0.
  - Los árbitros siempre tienen 1 voto.
  - Los miembros con prioridad mayor a 0 no pueden tener 0 votos.
  - Los miembros no pueden tener un número de votos mayor de 1.
  - Los miembros sin voto deben tener prioridad 0 (si no se vota, tampoco se puede ser elegido nodo primario).
  - Los miembros con 0 votos no pueden asentir las operaciones de escritura emitidas con un write concern de la mayoría.

## Iniciar un replica set

- A continuación se detallan los pasos para desplegar un replica set con tres nodos.

| Dato | Valor |
|---|---|
| usuario | vagrant |
| ip | 10.0.2.15 |
| nombre del replica set | replica |
| directorio de trabajo | /home/vagrant/ |
| directorio de los datos | ./data/ |
| directorio de los logs | ./log/ |
| directorio de la clave | ./pki/ |

```
cd /home/vagrant
vim node1.conf
```

```
# node1.conf
storage:
  dbPath: data/node1/
net:
  bindIp: 10.0.2.15,localhost
  port: 27011
security:
  authorization: enabled
  # keyFile obliga a que los miembros del replica set se autentiquen unos con
otros
  keyFile: pki/keyfile
```

```
# esta sección no es necesaria. Si se omite, el log es mostrado por consola
systemLog:
  destination: file
  path: log/node1/mongod.log
  logAppend: true
# esta sección no es necesaria. Si se omite, el servicio mongod bloquea la consola
processManagement:
  fork: true
replication:
  # nombre del replica set
  replSetName: replica
```

```
mkdir -p ./data/node1
mkdir -p ./pki/
mkdir -p ./log/node1
openssl rand -base64 741 > ./pki/keyfile
chmod 400 ./pki/keyfile
```

```
mongod -f node1.conf
```

- Es necesario crear los otros dos archivos de configuración de los nodos, cambiando el dbPath, el puerto y el path.

```
#node2.conf
storage:
  dbPath: data/node2/
net:
  bindIp: 10.0.2.15,localhost
  port: 27012
security:
  authorization: enabled
  keyFile: pki/keyfile
systemLog:
  destination: file
  path: log/node2/mongod.log
  logAppend: true
processManagement:
  fork: true
replication:
  replSetName: replica
```

```
#node3.conf
storage:
  dbPath: data/node3/
net:
```

```
  bindIp: 10.0.2.15,localhost
  port: 27013
security:
  authorization: enabled
  keyFile: pki/keyfile
systemLog:
  destination: file
  path: log/node3/mongod.log
  logAppend: true
processManagement:
  fork: true
replication:
  replSetName: replica
```

```
mkdir -p ./data/{node2,node3}
mkdir -p ./log/{node2,node3}
mongod -f node2.conf
mongod -f node3.conf

## Es importante conectarse a la base de datos que tenga los datos iniciales.
Desde allí se inicia el replica set
mongo --port 27011
```

```
// este comando solamente funciona desde localhost e inicia el replica set usando
la configuración por defecto del replica set. Opcionalmente, el método puede tomar
un argumento en forma de documento que contiene la configuración
rs.initiate()
rs.status()
use admin
db.createUser({
  user: "admin",
  pwd: "pass",
  roles: [
    {role: "root", db: "admin"}
  ]
})
exit
```

- A continuación es necesario conectarse al replica set, concretamente al nodo primario porque en la cadena de host se indica el nombre del replica set (independientemente del puerto). Si no se especifica el nombre del replica set, entonces se conecta directamente al nodo específico.

```
# Conexión con el nodo primario
mongo --host "replica/10.0.2.15:27011" -u "admin" -p "pass" --
authenticationDatabase "admin"

# Conexión con el nodo secundario
```

```
# mongo --host "10.0.2.15:27012" -u "admin" -p "pass" --authenticationDatabase
"admin"
```

```
rs.status()
rs.add("10.0.2.15:27012")
rs.add("10.0.2.15:27013")
rs.isMaster()

// Forzar la elección del nodo primario (convirtiendo a éste en nodo secundario y
no podrá ser nodo primario en la siguiente elección)
rs.stepDown()

// comprobar quién es el nodo primario
rs.isMaster()
```

- Para mantener copias actualizadas del conjunto de datos compartidos, los miembros secundarios de un conjunto de réplicas sincronizan o replican datos de otros miembros. MongoDB utiliza dos formas de sincronización de datos: sincronización inicial y replicación.
- MongoDB utiliza dos formas de sincronización de datos:
    - Sincronización inicial para incluir nuevos miembros con el conjunto completo de datos:
        - Clona todas las bases de datos excepto la base de datos local.
            - Construye todos los índices de colección.
            - Inicialmente aplica los posibles registros oplog recién creados durante la copia de datos.
    - Replicación para aplicar cambios continuos a todo el conjunto de datos.
        - Los miembros secundarios replican datos continuamente después de la sincronización inicial.
        - Se copia el oplog desde su sincronización desde el origen y se aplican estas operaciones en un proceso asíncrono
        - Los miembros del replica set con 1 voto no pueden sincronizarse con los miembros con 0 votos.
        - Los secundarios evitan la sincronización de miembros retrasados y miembros ocultos.
        - Si un miembro secundario tiene members[n].buildIndexes a true (por defecto es true), solo se puede sincronizar desde otros miembros donde buildIndexes es verdadero.
- Cuando se agregue el nuevo miembro, se someterá a lo que se llama una sincronización inicial. Durante esa fase, el miembro secundario comenzará a extraer todos los documentos del miembro primario
    - Paralelamente a la extracción de los documentos, el miembro secundario también extraerá las entradas de oplog, que reflejan las modificaciones realizadas en esos documentos o la inserción de nuevos documentos.
    - La aplicación de esas entradas de oplog después de que haya obtenido todos los documentos, garantizará un estado coherente de los documentos. Esta garantía se basa en la omnipotencia de las operaciones insertadas en el oplog.

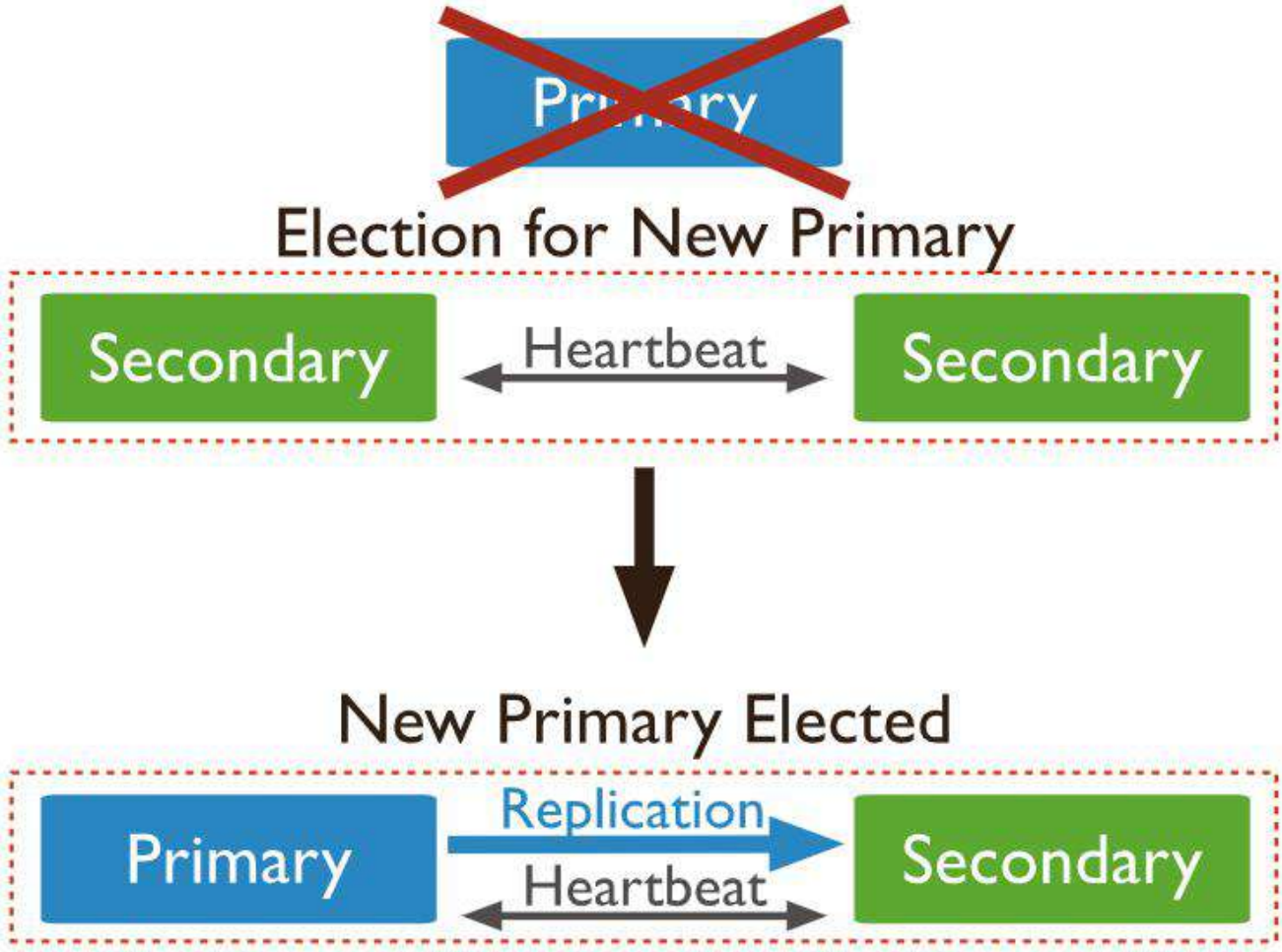
- Algunos comandos para interactuar con el replica set: rs.add y rs.remove.
- rs.add permite agregar miembros a un replica set y posee la sintaxis rs.add(host, arbiterOnly), donde arbiterOnly es un campo opcional booleano para indicar si el nodo será árbitro o no, mientras que host puede ser de tipo texto o de tipo documento.

```
{
   _id: <int>,
   /* requerido */
   host: <string>,
   arbiterOnly: <boolean>,
   buildIndexes: <boolean>,
   hidden: <boolean>,
   priority: <number>,
   tags: <document>,
   slaveDelay: <int>,
   votes: <number>
}
```

- rs.remove elimina un miembro de un replica set y posee la sintaxis rs.remove(hostname) donde hostname es un string con el hostname de un sistema en el replica set. Esta función desconectará el shell brevemente y forzará a reconectar mientras que el replica set renegocia qué miembro se convertirá en primario.

## Elecciones

- Los replica set pueden desencadenar una elección en respuesta a una variedad de eventos como:
  - Añadir un nuevo nodo al replica set.
  - Iniciar replica set.
  - Realizar el mantenimiento del replica set utilizando métodos como rs.stepDown() o rs.reconfig().
  - Cuando los miembros secundarios pierden conectividad con el primario por más del tiempo de espera configurado (10 segundos de forma predeterminada).




- El replica set no puede procesar operaciones de escritura hasta que la elección se complete con éxito. El replica set puede continuar sirviendo consultas de lectura si dichas consultas están configuradas para ejecutarse en secundarias (se ha ejecutado el comando rs.slaveOk() en esos miembros).
- El tiempo medio antes de que un grupo elija un nuevo primario no debe exceder los 12 segundos, asumiendo las configuraciones de configuración de réplica predeterminadas.
    - Esto incluye el tiempo requerido para marcar el primario como no disponible y llamar y completar una elección. Puede ajustarse este período de tiempo modificando la opción de configuración de la replicación settings.electionTimeoutMillis.
- Factores que condicionan el proceso de elección:
    - La latencia de la red pueden extender el tiempo requerido para que se completen las elecciones de conjuntos de réplicas, lo que a su vez afecta a la cantidad de tiempo que el grupo puede operar sin un primario.
    - Con el protocolo PV1 se reduce el tiempo de conmutación por error del replica set y se acelera la detección de múltiples primarios simultáneos.
    - Los miembros del replica set se envían latidos (heartbeats o ping) entre sí cada dos segundos. Si un latido no vuelve dentro de 10 segundos, los otros miembros marcan al miembro como inaccesible.

    - Después de que un replica set tenga un primario estable, el algoritmo de elección intenta que el secundario con la mayor prioridad disponible llame a una elección.
        - Los secundarios con mayor prioridad convocan elecciones relativamente antes que los secundarios con menor prioridad, y también tienen más probabilidades de ganar.
        - Sin embargo, un secundario de prioridad más baja puede ser elegido como primario por períodos breves, incluso si hay una secundario de mayor prioridad disponible.
        - Los miembros del replica set continúan convocando elecciones hasta que el miembro de mayor prioridad disponible se convierte en primario.

- Los miembros con un valor de prioridad 0 no pueden convertirse en primarios y no buscan elección.

- Una partición de red (fallo de red que separa un sistema distribuido en particiones de modo que los nodos en una partición no pueden comunicarse con los nodos en la otra partición.) puede segregar una primaria en una partición con una minoría de nodos.

- Miembros que votan:

    - Todos los miembros del replica set que tienen sus members[n].votes con un valor igual a 1 voto en las elecciones pueden votar. Para excluir a un miembro de la votación en una elección hay que cambiar el valor de la configuración de los members[n].votes a 0.
    - Los miembros sin voto deben tener prioridad 0. El valor de prioridad puede ser cualquier número de punto flotante (es decir, decimal) entre 0 y 1000. El valor predeterminado para el campo de prioridad es 1.
    - Los miembros con prioridad mayor a 0 no pueden tener 0 votos.
    - Miembros que no votan deben tener prioridad 0.
    - Pueden existir miembros con prioridad igual 0 y votos 1 (miembros ocultos o retrasados pueden votar).
    - Solo los miembros votantes en los siguientes estados son elegibles para votar:
        - PRIMARY
        - SECONDARY
        - STARTUP2
        - RECOVERING
        - ARBITER
        - ROLLBACK

- Miembros que no votan: los miembros que no votan en las elecciones pueden tener copias de los datos del replica set y pueden aceptar operaciones de lectura desde las aplicaciones de los clientes. Un miembro sin voto tiene votos y prioridad igual a 0.

- No es recomendable modificar el número de votos para controlar qué miembros se convertirán en primarios (de 0 a 1 o de 1 a 0, que son los dos posibles valores para los votos). En su lugar, es mejor modificar la opción de members[n].priority. Solo alterar el número de votos en casos excepcionales. Por ejemplo, para permitir más de siete miembros.

- Es importante asegurarse que los nodos secundarios con índices distintos al nodo primario no sean elegibles como primarios.

## Failover

- Si un replica set tiene un número par de miembros, es conveniente un árbitro para obtener la mayoría de los votos en una elección para elegir el nodo primario. Los árbitros no requieren hardware dedicado.
- El mecanismo de failover de un replica set consiste en poder elegir un nodo primario cuando éste cae.
- El failover para un replica set es la cantidad de miembros que pueden dejar de estar disponibles y aún dejar suficientes miembros en el conjunto para elegir un primario.
    - En otras palabras, es la diferencia entre el número de miembros en el conjunto y la mayoría de los miembros con derecho a voto necesarios para elegir un primario.
    - Sin un primario, un replica set no puede aceptar operaciones de escritura.

| Nodos de un replica set | Mayoría requerida para elegir un nodo primario | Tolerancia a fallos |
|---|---|---|
| 3 | 2 | 1 |
| 4 | 3 | 1 |
| 5 | 3 | 2 |
| 6 | 4 | 2 |

- Cuando no hay mayoría de nodos en el replica set, todos los nodos pasan a ser secundarios y no es posible escribir. Por ejemplo, en un replica set de tres nodos, si dos de ellos caen, el tercero pasa a ser secundario (aunque sea el nodo primario) y no es posible escribir datos en la base de datos.
- Es importante que el número de nodos en el replica set sea impar. En caso de que sea un número par es recomendable desplegar un árbitro para que el conjunto tenga un número impar de miembros con derecho a voto.
- Cualquier campo en la topología (añadiendo, eliminando o fallando nodos) puede cambiar la elección del nodo primario.
- El cliente de la aplicación realmente se conecta al nodo primario hasta que éste se caiga por alguna razón.
- Si el nodo primario va a sufrir algún tipo de mantenimiento, es conveniente forzar la elección de un nuevo nodo primario mediante el comando rs.stepDown().
- En la elección, el primer elemento fundamental que se analiza es la prioridad (por defecto, todos poseen prioridad 1), así como el nodo que almacena la última copia de los datos.
- Si todos tienen la misma prioridad, normalmente los nodos que poseen la copia más reciente de los datos se votarán a sí mismos. Por ejemplo, suponiendo un replica set de tres nodos.
    - Si un nodo solo posee la última copia de los datos: ese es elegido primario.
    - Si dos nodos poseen la última copia de los datos: el tercer nodo es el que deshace el empate.
- En caso de empate, se volverá realizar la elección hasta poder elegir un nodo primario. Mientras tanto, los datos no están accesibles.
- Un miembro sin voto debe tener 0 votos (no puede votar) y prioridad igual a 0:

```
{
   "_id" : <num>,
   "host" : <hostname:port>,
   "arbiterOnly" : false,
   "buildIndexes" : true,
   "hidden" : false,
   "priority" : 0,
   "votes" : 0
}
```

## Rollback

- Una rollback revierte las operaciones de escritura en un nodo primario antiguo cuando éste vuelve a unirse a su replica set después de una failover.
- Un rollback es necesario solamente si el nodo primario ha aceptado operaciones de escritura que los secundarios no han replicado con éxito antes de que el primario cayera.
- Cuando el primario se vuelve a unir al conjunto como secundario, revierte o "retrocede", sus operaciones de escritura para mantener la coherencia de la base de datos con la de otros miembros.
- MongoDB intenta evitar los rollbacks. Cuando se produce un rollback, a menudo es el resultado de una partición de red.
- No se produce un rollback si las operaciones de escritura se replican en otro miembro del replica set antes de que el nodo primario caiga y si ese miembro permanece disponible y accesible para la mayoría del replica set.
- Cuando ocurre un rollback, MongoDB, de forma predeterminada, escribe los datos del rollback en los archivos BSON en la carpeta /rollback dentro del directorio dbPath de la base de datos. Los nombres de los archivos de rollback tienen la siguiente forma:

```
<database>.<collection>.<timestamp>.bson
```

- Para leer el contenido de los archivos de rollback es necesario utilizar bsondump. Según el contenido y el objetivo de la aplicación, los administradores pueden decidir la acción a ejecutar.
- La cantidad máxima de un rollback son 300 MB.
- Para los replica set, la write concern predeterminado es {w: 1} y solo proporciona acuse de recibo de las operaciones de escritura en el primario. Con la write concern predeterminada, los datos se pueden revertir si las operaciones de escritura se han replicado en cualquiera de los secundarios antes de la caída del miembro primario.
- Para evitar el rollback, es importante desplegar todos los miembros con el journaling habilitado y utilizar w: majority para garantizar que las operaciones de escritura se propagan a la mayoría de los nodos del replica set antes de regresar al cliente.

- Si existe mucho movimiento de datos en la replica (muchas escrituras en el primario), será necesario un oplog grande. También si la conexión entre los nodos es lenta. En estos casos es conveniente aumentar el tamaño del oplog (990 megabytes por defecto). Si el secundario no puede replicar con el oplog y ésta llega a límite (roll over), entonces replica la base de datos completa, que es un proceso muy lento.

## Configuración de un replica set

- Las opciones de configuración de un replica set viene definida por un objecto JSON. Puede ser configurado manualmente desde la shell de MongoDB. Existe un conjunto de métodos de replicación que ayudan a gestionar la configuración.

```
rs.add
rs.initiate
rs.remove
rs.reconfig
rs.config
```

- Las opciones de configuración más importantes son:
    - _id: nombre del replica set.
    - version: número entero que se incrementa automáticamente cuando hay algún cambio en la configuración del replica set.
    - members[].id: identificador del miembro del replica set que no puede ser cambiado.
    - members[].host: hostname y puerto.
    - members[].arbiterOnly: booleano que indica si el miembro es un árbitro.
    - members[].hidden: booleano que indica si el miembro se encuentra oculto para la aplicación.
    - members[].priority: permite establecer una jeraquía entre los miembros del replica set. El valor de la prioridad puede oscilar entre 0 (el miembro nunca será nodo primario) y 1000. El valor por defecto es 1. Miembros con prioridad más alta tienden a ser elegidos con mayor probabilidad para ser nodos primarios. Un cambio en la prioridad de un nodo desembocará en una elección de un nuevo nodo primario porque esto supone un cambio en la topología. Cuando el miembro es árbitro o está oculto, la prioridad debe ser 0 porque estos miembros no pueden convertirse en primario.
    - members[].slaveDelay: determina una replicación retrasada en segundos. Si el valor es mayor que 0, entonces la prioridad debe ser 0 también y el nodo debe estar oculto.
    - settings{}: otras opciones de configuración que permite modificar el protocolo de replicación.

- Comandos para obtener información de la replicación.
    - rs.status(): reporta el estado de salud de los nodos del replica set. Esta información procede de los heartbeats. Algunos datos:
        - myState: un entero entre 0 y 10 que representa el estado de la réplica del actual miembro --> 0 (STARTUP), PRIMARY (1), SECONDARY (2), RECOVERING (3), STARTUP2 (5), UNKNOWN (6), ARBITER (7), DOWN (8), ROLLBACK (9), REMOVED (10)
        - heartbeatIntervalMillis: intervalo de tiempo en el que se envían los heartbeats.
        - members[].uptime: número de segundos que ha estado ejecutándose un miembro.

        - members[].optime: última vez que este nodo aplicó una operación de su oplog.
        - members[].optimeDate: versión más legible del optime.
        - members[].self: el campo self solo se incluye en el documento para la instancia mongod actual en el array de miembros. Su valor es true.
        - members[].health: este campo transmite si el miembro está activo (es decir, 1) o inactivo (es decir, 0)
        - members[].state: un entero entre 0 y 10 que representa el estado de la réplica del miembro.
        - members[].stateStr: significado del estado de la réplica.
        - members[].ping: latencia hasta el miembro.

    - rs.isMaster(): describe el rol del nodo y también proporciona información sobre el replica set. La salida de datos es más simplificada que la proporcionada por rs.status e indica dónde está el nodo primario.

    - db.serverStatus()['repl']:

        - Proporciona información sobre el proceso mongod, pero utilizando repl concramente se muestran datos de la replicación
        - Su salida es muy similar a rs.isMaster, aunque el campo rbid no está incluido en el isMaster.
        - El rbid indica el número de rollbacks que se produjeron en el nodo.

    - rs.printReplicationInfo():

        - Muestra información del oplog en el que se ejecuta el comando. Contiene timestamps para el primer y últimos eventos del oplog, longitud del oplog en tiempo y en megabytes. El primer evento del oplog puede cambiar porque oplog es una colección capped. No se muestra información sobre las sentencias.

## Base de datos local en un replica set (oplog)

- La base de datos local no se replica.
- El oplog (registro de operaciones) es una colección capped o con límite que mantiene un registro continuo de todas las operaciones que modifican los datos almacenados en la base de datos.
- El oplog solo contiene entradas para consultas de escritura, es decir, cuando se inserta, modifica o elimina un documento.

```
/*
{ "_id" : ObjectId("57fd59a2d630a0fd9685a148"), "firstName" : "Arthur", "lastName"
: "Aaronson", "state" : "WA", "city" : "Seattle", "likes" : [ "dogs", "cats" ] }
{ "_id" : ObjectId("57fd59a2d630a0fd9685a149"), "firstName" : "Beth", "lastName" :
"Barnes", "state" : "WA", "city" : "Richland", "likes" : [ "forest", "desert" ] }
{ "_id" : ObjectId("57fd59a2d630a0fd9685a14a"), "firstName" : "Charlie",
"lastName" : "Carlson", "state" : "CA", "city" : "San Diego", "likes" : [
"desert", "beach" ] }
{ "_id" : ObjectId("57fd59a2d630a0fd9685a14b"), "firstName" : "Dawn", "lastName" :
"Davis", "state" : "WA", "city" : "Seattle", "likes" : [ "forest", "mountains" ] }
```

```
*/

// no crea entrada en el oplog en la colección no existe ningún documento con
state=NB
db.people.deleteMany( { state : "NB" } )
```

- MongoDB aplica las operaciones de la base de datos en el nodo primario y luego registra las operaciones en el oplog del primario. Los miembros secundarios luego copian y aplican estas operaciones en un proceso asíncrono. Todos los miembros del replica set contienen una copia del oplog, en la colección local.oplog.rs, lo que les permite mantener el estado actual de la base de datos.
- Para facilitar la replicación, todos los miembros del replica set envían heartbeat (pings) a todos los demás miembros. Cualquier miembro secundario puede importar entradas de oplog de cualquier otro miembro.
- Cada operación en el oplog es idempotente. Es decir, las operaciones de oplog producen los mismos resultados si se aplican una o varias veces al conjunto de datos de destino.
    - Cuando una operación de escritura modifica muchos documentos en un miembro primario, este miembro debe insertar una entrada separada en el oplog para cada documento modificado. Esta es la única forma en que el sistema puede garantizar que oplog permanezca omnipotente.
    - Una sola consulta de escritura puede dar como resultado múltiples entradas oplog. No se garantiza que las consultas de escritura, tal como las reciba la Primaria, sean omnipotentes, y es posible que deban transformarse. Por ejemplo, una operación de escritura que modifica muchos documentos no es omnipotente y se transformará en una serie de escrituras, una por documento modificado.
    - Una entrada en el oplog no puede afectar a múltiples documentos.
    - Cada entrada de oplog especifica si un documento se inserta, actualiza o elimina.
- oplog.rs es el punto central del mecanismo de replicación de datos. Mantendrá todas las instrucciones que serán replicadas en el replica set.

```
use local
db.oplog.rs.findOne()
```

```
let stats1 = db.oplog.rs.stats()

// true porque es una colección capped
stats1.capped

// tamaño actual de la colección
stats1.maxSize

// tamaño máximo de la colección capped
stats1.maxSize
```

```
// obtener los datos en MB
let stats2 = db.oplog.rs.stats(1024*1024)
```

- Una operación puede resultar en muchas entradas en el oplog.rs debido a la idempotencia. Por ejemplo, una operación simple updateMany que actualice 100 documentos, puede generar 100 entradas en el oplog.rs para garantizar la idempotencia.
- Por defecto, rs.oplog tomará un 5% como máximo del disco duro (con un mínimo de 990MB). Aunque este tamaño puede modificarse. Cada nodo puede tener diferentes tamaños de oplog, lo que aumenta el tamaño de la ventana de replicación.
- Tamaño de una colección oplog para sistemas Unix y Windows.

| **Motor de almacenamiento** | **Tamaño de oplog por defecto** | **Límite inferior** | **Límite superior** |
|---|---|---|---|
| Motor de almacenamiento en memoria | 5% de la memoría física | 50 MB | 50GB |
| Motor de almacenamiento WiredTiger | 5% del espacio libre en disco | 990MB | 50GB |
| Motor de almacenamiento MMAPv1 | 5% del espacio libre en disco | 990MB | 50GB |

- Tamaño de una colección oplog para sistemas Mac.

| **Motor de almacenamiento** | **Tamaño de oplog por defecto** |
|---|---|
| Motor de almacenamiento en memoria | 192 MB de la memoría física |
| Motor de almacenamiento WiredTiger | 192 MB del espacio libre en disco |
| Motor de almacenamiento MMAPv1 | 192 MB del espacio libre en disco |

- El tamaño del oplog puede cambiarse desde los archivos de configuración.

```
#node1.conf
replication:
  oplogSizeMB: 16000MB
```

- Una vez que ha iniciado un miembro del replica set por primera vez, hay que utilizar el comando administrativo replSetResizeOplog para cambiar el tamaño de oplog. replSetResizeOplog permite cambiar el tamaño del oplog dinámicamente sin reiniciar el proceso mongod.

```
use local

// muestra el tamaño actual del oplog en bytes (1038090240 bytes;
1038090240/1024/1024 = 990 megabytes)
db.oplog.rs.stats().maxSize
```

```
// cambiar el tamaño del oplog. Se suministra un número que representa megabytes y
que debe ser 990 más (en este caso se suministra 16000 megabytes, es decir, 16
gigabytes)
db.adminCommand({replSetResizeOplog: 1, size: 16000})

// opcionalmente puede compactarse el oplog (importante ejecutar esta operación
cuando se realice una disminución del oplog)
db.runCommand({ "compact" : "oplog.rs" } )
```

- Cuando se alcanza el máximo del tamaño del oplog, la siguiente instrucción sobreescribe la operación más antigua.
    - La ventana de tiempo de la replicación es el tiempo que tarda en comenzar llenarse el oplog desde el inicio hasta el final.
    - Este tiempo también es el máximo que una máquina puede estar caída y recuperarse sin problemas. Llegado a ese punto, el nodo entraría en recovery mode.
- Por tanto, la ventana de la replicación es proporcional a la carga del sistema.
- Las siguientes cargas de trabajo pueden requerir un tamaño mayor de oplog:
    - Actualizaciones a varios documentos a la vez: el oplog debe traducir las actualizaciones múltiples en operaciones individuales para mantener la idempotencia.
    - Las eliminaciones equivalen a la misma cantidad de datos que las inserciones: si se elimina aproximadamente la misma cantidad de datos que se inserta, la base de datos no aumentará significativamente en el uso del disco, pero el tamaño del oplog puede ser bastante grande.
    - Cantidad significativa de actualizaciones: si una parte importante de la carga de trabajo son actualizaciones que no aumentan el tamaño de los documentos, la base de datos registra una gran cantidad de operaciones, pero no cambia la cantidad de datos en el disco.
- Con el comando rs.printReplicationInfo() se puede obtener información sobre cuánto tiempo tardará en llenarse esta colección oplog.rs, aunque se necesita cierto movimiento de replicación de datos para que el tiempo arrojado sea coherente. También incluye el tamaño y el intervalo de tiempo de las operaciones.
- Si se pretende no replicar una colección, se debe utilizar la base de datos local.

```
use local
// devolver oplog, excepto los heartbeats (periodic noop). $natural: -1 ordena los
resultados en el orden natural descendentes en el que fueron insertados
db.oplog.rs.find( { "o.msg": { $ne: "periodic noop" } } ).sort( { $natural: -1 }
).limit(1).pretty()

// encontrar todas las operaciones relacionadas con el namespace m103.messages
db.oplog.rs.find({"ns": "m103.messages"}).sort({$natural: -1})
```

- Es cierto que las entradas de oplog provienen generalmente del miembro primario, pero las secundarias pueden sincronizarse desde otra secundarias, siempre que al menos haya una cadena de sincronizaciones oplog que conduzca de regreso a la primaria.

## Colecciones capped o con límite

- Las colecciones con límite o capped son colecciones de tamaño fijo que admiten operaciones de alto rendimiento que insertan y recuperan documentos en función del orden de inserción.
- Las colecciones capped funcionan de manera similar a los búferes circulares: una vez que una colección llena su espacio asignado, deja espacio para nuevos documentos al sobrescribir los documentos más antiguos de la colección.
    - Para hacer espacio para nuevos documentos, las colecciones capped eliminan automáticamente los documentos más antiguos de la colección sin requerir scripts ni operaciones explícitas de eliminación.
    - Por ejemplo, la colección oplog.rs, que almacena un log de las operaciones en un replica set, utiliza una capped. A partir de MongoDB 4.0, a diferencia de otras colecciones capped, el oplog puede crecer más allá de su límite de tamaño configurado para evitar eliminar el majority commit point (operación más reciente que se ha escrito en la mayoría de los miembros del replica set).
- Las colecciones capped garantizan la preservación del orden de inserción. Como resultado, las consultas no necesitan un índice para devolver los documentos en orden de inserción. Sin esta sobrecarga de indexación, las colecciones capped pueden admitir un mayor rendimiento de inserción.
- Para crear una colección capped se puede utilizar varios parámetros:
    - size (requerido): tamaño máximo de la colección en bytes. Si el campo de tamaño es menor o igual a 4096, entonces la colección tendrá un límite de 4096 bytes. De lo contrario, MongoDB aumentará el tamaño provisto para convertirlo en un múltiplo entero de 256.
    - max (opcional): número máximo de documentos que puede almacenar la colección

```
db.createCollection( "log", { capped: true, size: 100000 } )

// devolver en orden inverso
db.log.find().sort( { $natural: -1 } )

// comprobar si la colección es capped
db.log.isCapped()

// convertir una colección capped
db.runCommand({"convertToCapped": "mycoll", size: 100000});
```

- Cuestiones importantes sobre las colecciones capped:
    - Para hacer espacio para nuevos documentos, las colecciones capped eliminan automáticamente los documentos más antiguos de la colección sin requerir scripts ni operaciones explícitas de eliminación.

- Las colecciones capped tienen un campo _id y un índice en el campo _id de forma predeterminada.
- Si se planea actualizar documentos en una colección capped es conveniente crea un índice para que estas operaciones de actualización no requieran un escaneo de colección.
- Si hay una actualización o una operación de reemplazo y se cambia el tamaño del documento, la operación fallará. Concretamente:
  - No es posible añadir nuevos campos.
  - No es posible cambiar el tipo de dato de un campo, ni con $set, ni tampoco reemplazando el documento completo.
  - Es posible cambiar el valor de un campo (siempre que se conserve el tipo) reemplazando el documento completo y también con $set.
  - Es posible cambiar el nombre de un campo reemplazando el documento completo, pero no utilizando $set.
- No se puede eliminar documentos de una colección capped. Para eliminar todos los documentos de una colección hay que utilizar el método drop para eliminar la colección y volver a crear la colección.
- No es posible crear una colección capped fragmentada.
- Los documentos son almacenados de forma ordenada.
- Es posible renombrar la colección
- No es posible cambiar el parámetro size una colección capped una vez creada.
- Es importante utilizar el orden natural para recuperar los elementos insertados más recientemente de la colección de manera eficiente.
- La fase de agregación $out no puede escribir resultados en una colección capped.
- Si se realiza un find en una colección capped sin ordenamiento especificado, MongoDB garantiza que el orden de los resultados es el mismo que el orden de inserción.
- Como alternativa colecciones capped se puede utilizar los índices TTL.

## Reconfigurar un replica set

```
#node4.conf
storage:
  dbPath: /var/mongodb/db/node4
net:
  bindIp: 192.168.103.100,localhost
  port: 27014
systemLog:
  destination: file
  path: /var/mongodb/db/node4/mongod.log
  logAppend: true
```

```
processManagement:
  fork: true
replication:
  replSetName: m103-example
```

```
# arbiter.conf
storage:
  dbPath: /var/mongodb/db/arbiter
net:
  bindIp: 192.168.103.100,localhost
  port: 28000
systemLog:
  destination: file
  path: /var/mongodb/db/arbiter/mongod.log
  logAppend: true
processManagement:
  fork: true
replication:
  replSetName: m103-example
```

```
mongod -f node4.conf
mongod -f arbiter.conf
```

```
mongo --host "m103-example/192.168.103.100:27011" -u "m103-admin" -p "m103-pass" --authenticationDatabase "admin"
```

```
rs.add("m103.mongodb.university:27014")
rs.addArb("m103.mongodb.university:28000")
rs.isMaster()

// eliminar árbitro
rs.remove("m103.mongodb.university:28000")

// editar configuración
cfg = rs.conf()
cfg.members[3].votes = 0
cfg.members[3].hidden = true
cfg.members[3].priority = 0
rs.reconfig(cfg)
```

- El método rs.reconfig reconfigura un replica set existente, sobrescribiendo la actual configuración del replica set. La sintaxis que posee es rs.reconfig(configuration, force), donde configuration es un documento que especifica la configuración de un replica set y force es un documento opcional ({ force:

true } ) que fuerza al replica set a aceptar la nueva configuración incluso si una mayoría de miembros no están accesible. Puede conducir a situaciones de rollback.

## Leer datos en un nodo secundario

- Solamente es posible leer datos de un nodo primario si se ejecuta previamente rs.slaveOk()

```
mongo --host "m103-example/m103.mongodb.university:27011" -u "m103-admin" -p
"m103-pass" --authenticationDatabase "admin"
```

```
rs.isMaster()
use newDB
db.new_collection.insert( { "student": "Matt Javaly", "grade": "A+" } )
exit
```

```
mongo --host "m103.mongodb.university:27012" -u "m103-admin" -p "m103-pass"
--authenticationDatabase "admin"
```

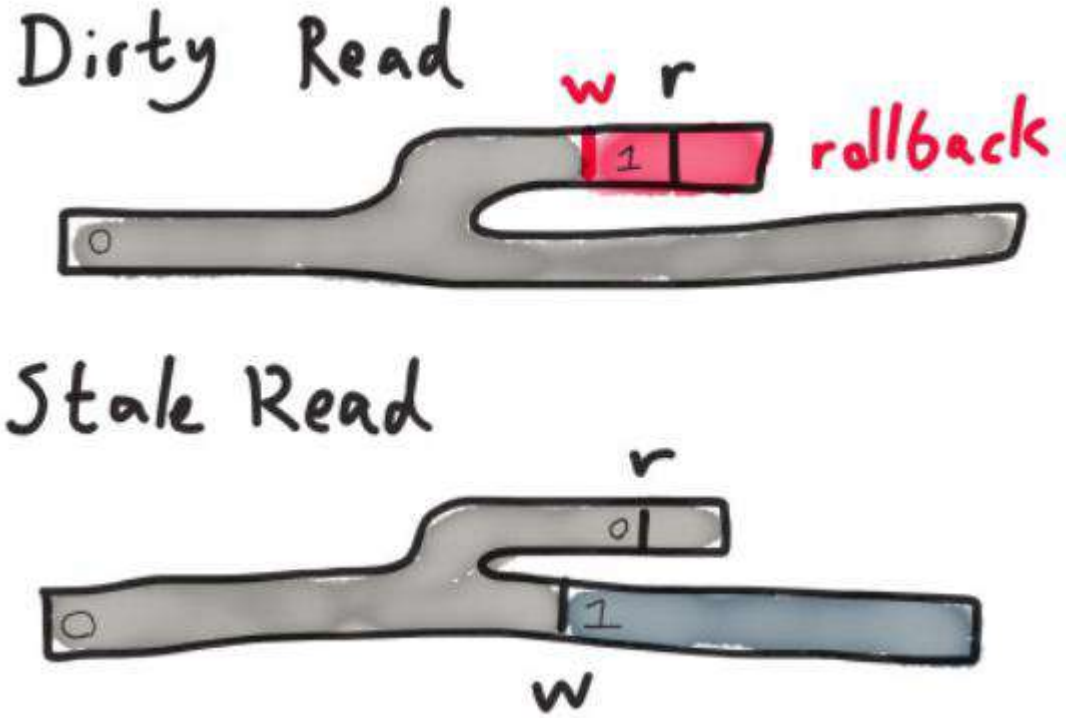
```
show dbs

// permitir comandos de lectura en un nodo secundario
rs.slaveOk()
use newDB
db.new_collection.find()

// error porque no se puede escribir en el nodo secundario
db.new_collection.insert( { "student": "Norberto Leite", "grade": "B+" } )

use admin
db.shutdownServer()
exit
```

```
mongo --host "m103.mongodb.university:27011" -u "m103-admin" -p "m103-pass"
--authenticationDatabase "admin"
```

```
// verificar quién es el nuevo nodo maestro
rs.isMaster()
```

## Read preferences

- Razones por las que leer de los nodos secundarios:
    - Ejecución de operaciones de sistemas que no afectan a la aplicación front-end.
    - Para proporcionar lecturas locales (en una base de datos local).
    - Mantener la disponibilidad durante una failover, aunque debe configurarse el read preference a primaryPreferred.
- No es conveniente leer de los nodos secundarios cuando el nodo primario está bajo una carga de escritura pesada. En esos casos es mejor utilizar sharding.
- El riesgo de leer de un nodo secundario es que los datos pueden estar obsoleto (stale read). En algunos casos puede ser aceptable.
    - Esto únicamente ocurre en nodos secundarios, aunque en circustancias extrañas donde dos miembros pueden considerarse primarios durante un breve periodo de tiempo, también existe riesgo de obtener datos obsoletos.
    - Los sistemas con lecturas obsoletas son considerados "eventualmente consistentes".
    - No es lo mismo datos obsoletos que datos sucios (dirty reads). Las lecturas sucias ocurren cuando se leen datos que no se han confirmado por una mayoría porque esos datos podrían revertirse.




- Los miembros de un replica set pueden quedarse rezagados respecto al primario debido a la congestión de la red, el bajo rendimiento del disco, las operaciones de larga ejecución, etc.
    - La opción de preferencia de lectura maxStalenessSeconds permite especificar un máximo retardo de replicación para lecturas de secundarios. Cuando un secundario excede el maxStalenessSeconds, el cliente para de usar las operaciones de lectura.
    - La opción de preferencia de lectura maxStalenessSeconds está diseñada para aplicaciones que lean desde miembros secundarios y desean evitar la lectura desde un miembro secundario que se ha retrasado demasiado en la reproducción de las escrituras de la primaria.

    - Esta opción maxStalenessSeconds se establece en la URL de conexión de MongoDB

- La configuración de read preferences permite dirigir las operaciones de lectura a miembros específicos de un replica set. Es una configuración exclusiva de la aplicación y del driver.

```
db.products.find({}).readPref("secondaryPreferred")
```

- Existen cinco posibles modos:
    - primary (valor por defecto): redirige las operaciones de lectura al nodo primario
    - primaryPreferred: lee los datos preferentemente del nodo primario, pero si el primario no está disponible, como sucede en un failover, las operaciones serán redirigidas a algún nodo secundario.
    - secondary: dirige las operaciones de lectura a miembros secundarios.
    - secondaryPreferred: dirige las operaciones de lectura a miembros secundarios, pero si ningún nodo secundario está disponible, entonces la aplicación redirige la lectura al nodo primario.
    - nearest: dirige las operaciones de lectura al nodo con menor latencia.

| Scenario | Tradeoff | Read Preference |
|---|---|---|
| Read from the primary only | Secondaries are for availability only. | *primary* (default) |
| If the primary is unavailable, read from a secondary | Possible to read stale data. | *primaryPreferred* |
| Read from the secondary members only | Possible to read stale data. | *secondary* |
| If all secondaries are unavailable, read from the primary | Possible to read stale data. | *secondaryPreferred* |
| Application's read from the geographically closest member | Possible to read stale data. | *nearest* |

- Generalmente para todas las opciones (más info):
    - Cuando se incluye un valor de maxStalenessSeconds, el cliente estima qué tan obsoleto es el miembro secundario comparando la última escritura del nodo secundario con el miembro primario. El cliente luego dirige la operación de lectura a un secundario cuyo retraso estimado es menor o igual a maxStalenessSeconds.
    - Cuando se incluye un conjuntos de tags, el cliente intenta encontrar miembros secundarios que coincidan con los conjuntos de tags especificados y dirige las lecturas a un secundario aleatorio entre el grupo más cercano de secundarios coincidentes. Si no hay etiquetas secundarias que coincidan, la operación de lectura produce un error.
    - Cuando se incluye un valor maxStalenessSeconds y un conjunto de tags, el cliente filtra primero por obsolescencia y luego considera las tags especificadas.

```
// configurar tags para los cinco miembros de un replica set
conf = rs.conf();
conf.members[0].tags = { "dc": "center1"};
conf.members[1].tags = { "dc": "center1"};
conf.members[2].tags = { "dc": "center2"};
```

```
conf.members[3].tags = { "dc": "center2"};
conf.members[4].tags = { "dc": "center2"};
rs.reconfig(conf);
```

````javascript
db.collection.find().readPref('nearest', [ { 'dc': 'center1' } ])
````

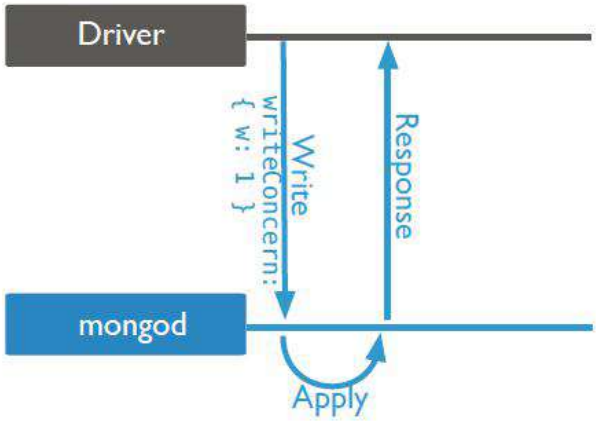
- Posibles casos de uso:
  - primary: preferido para la mayoría de los casos
  - nearest: cuando los datos obsoletos son aceptables y las instancias están repartidas geográficamente.
  - secondary: cargas de trabajo específicas (ej: analíticas).
  - secondaryPreferred: cuando el nodo primario tiene un carga intensiva y se quiere distribuir la carga de lectura por todos los nodos (con preferencia a los secundarios), teniendo en cuenta que los datos obsoletos son aceptables.

## Write concern

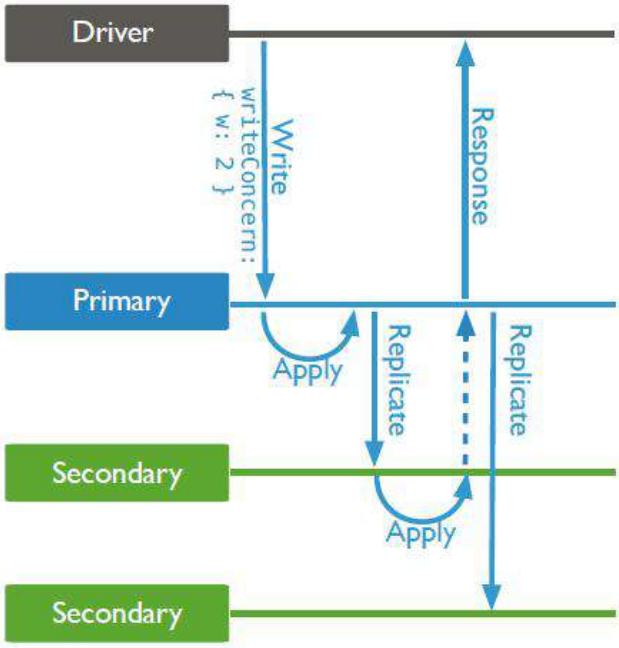
- Write concern es un mecanismo de asentimiento de escritura que los desarrolladores pueden añadir para operaciones de escrituras.
- Altos niveles de asentimiento garaniza una fuerte durabilidad en los datos. La durabilidad significa que la escritura ha sido propagada a los nodos del replica set especificados en el write concern.
  - Cuanto más miembros del replica set han asentido el éxito de una escritura, más probabilidad existe de que la escritura sea durable en caso de failure o error, porque podría suceder que el miembro primario cayera justo cuando se están escribiendo datos y éstos no han sido propagados a los otros miembros (incluso aunque w=1, j=1). Obviamente, alcanzar una alta durabilidad requiere de más tiempo.
- Write concern incluye los siguientes campos.

```
{ w: <value>, j: <boolean>, wtimeout: <number> }
```

**Write Concern: { w : 1 }**




**Write Concern: { w : 2 }**




```
db.customer.updateOne( { user : "mary_p" },
                       { $push : { shoppingCart:
                         { _id : 335443, name : "Brew-a-cup",
                           price : 45.79 } } },
                       { writeConcern : { w : 2 } } )
```

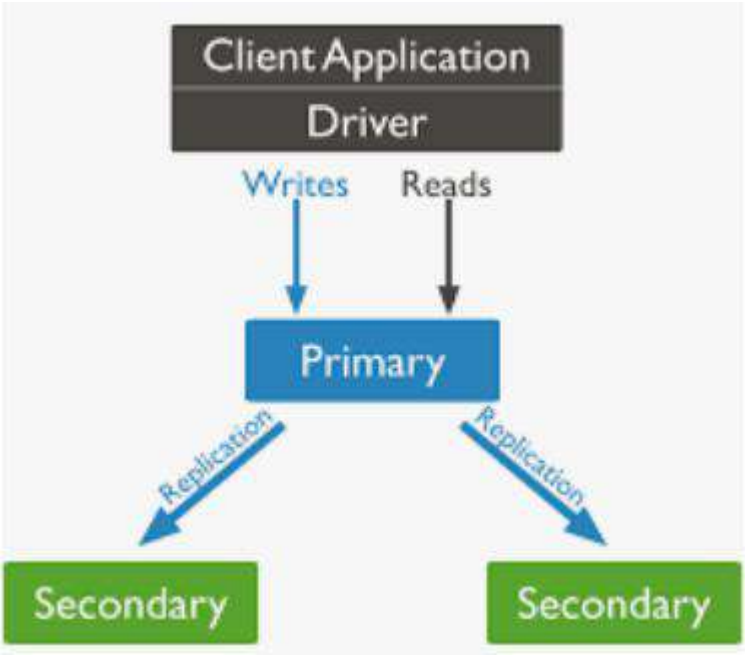
- Opciones del write concern:
    - w: solicitar el reconocimiento de que la operación de escritura se ha propagado a un número específico de instancias mongod o a instancias mongod con etiquetas específicas. Valores posibles:
        - 0: no se espera por asentimientos. Puede devolver información sobre excepciones de socket y errores de red a la aplicación. Opción no recomendada, aunque para tareas de

log puede ser útil. Si se especifica w: 0 pero se incluye j: true, j: true prevalece para solicitar el reconocimiento del mongod independiente o el primario de un replica set.

        - 1 (valor por defecto): solo se espera por el asentimiento del nodo primario.
        - mayor de 1: espera por el nodo primario y los secundarios. Puede ser interesante para bulks de datos o importaciones, aunque las operaciones se retrasan cuanto más asentimientos existan. Por ejemplo, con w=2, se espera confirmación del nodo primario y de uno secundario.
        - "majority": se espera por el asentimiento de una mayoría de miembros del replica set, incluyendo el primario. Con este valor se evita el rollback en la mayoría de los casos (se garantiza que los datos estén siempre almacenados al menos en la mayoría de miembros, de tal forma que si uno de esos miembros cae, el otro puede recuperar los datos).
        - tag set: solicita el reconocimiento de que las operaciones de escritura se hayan propagado a un miembro del conjunto de réplicas con la etiqueta especificada. Recomendado para data centers.
    - j: es un booleano que exige el asentimientos del journaling para todos los nodos especificados por w, incluyendo el primario. El valor por defecto es false.
    - wtimeout: tiempo a esperar por las peticiones write concern antes de considerar que la operación ha fallado.
        - Esto no quiere decir que el proceso de propagación de datos en el replica set haya fallado, sino que si vence este tiempo significa es que el nivel de durabilidad exigido por la aplicación no se ha cumplido. Sólo aplicable con valores de w mayor que 1.
        - Especificar un wtimeout de 0 es equivalente a un write concern sin el wtimeout.

- El rollback debe hacerse manualmente cuando un primario cae y los datos almacenados no están propagados hacia los otros miembros. Cuando el miembro se recupera, los datos perdidos se encuentran en un archivo que es necesario restaurarlo manualmente.
- MongoDB también soporta write concern para instancias standalone (solamente tiene sentido w=0|1 y j=0|1) y shards.
- Tanto w, j y wtimeout pueden establecerse a nivel de conexión del cliente, a nivel de colección y a nivel de replica set.
- Hay que tener cuidado con el valor de w cuando existen nodos retrasados. Si w incluye a nodos retrasados, entonces la respuesta al cliente de la confirmación de la escritura tardará lo que tarde el nodo retrasado en escribir la información.
- Operaciones soportadas por el write concern: insert, update, delete y findAndModify, etc. Es decir, a todas las operaciones de escritura. Las operaciones de lectura utiliza read concerns o read preferences.
- Si se establece un nivel de asentamiento de 3 en un replica set de 3 nodos y uno de ellos cae, la aplicación puede quedarse esperando indefinidamente. Por eso es importante establecer un wtimeout.

```
db.new_data.insert({"m103": "very fun"}, { writeConcern: { w: 3, wtimeout: 1000
}})

// si no se puede garantizar el writeConcern, la espera será indefinida porque no
se ha establecido un wtimeout
db.new_data.insert({"m103": "very fun"}, { writeConcern: { w: 3 }})
```




## Read concern

- Es un mecanismo de asentimiento para operaciones de lectura. Los read concern permite a la aplicación especificar una garantía de durabilidad y los datos solamente son retornados cuando se han escrito en un número concreto de miembros del replica set especificados en el read concern.

```
db.collection.find().readConcern(<level>)
```

- Un documento que no reúne el específico nivel de read concern no es devuelto, pero no está garantizado que se haya perdido. Lo único que significa es que en el momento de la consulta con read concern, el documento no se ha propagado todavía por los suficientes nodos secundarios.

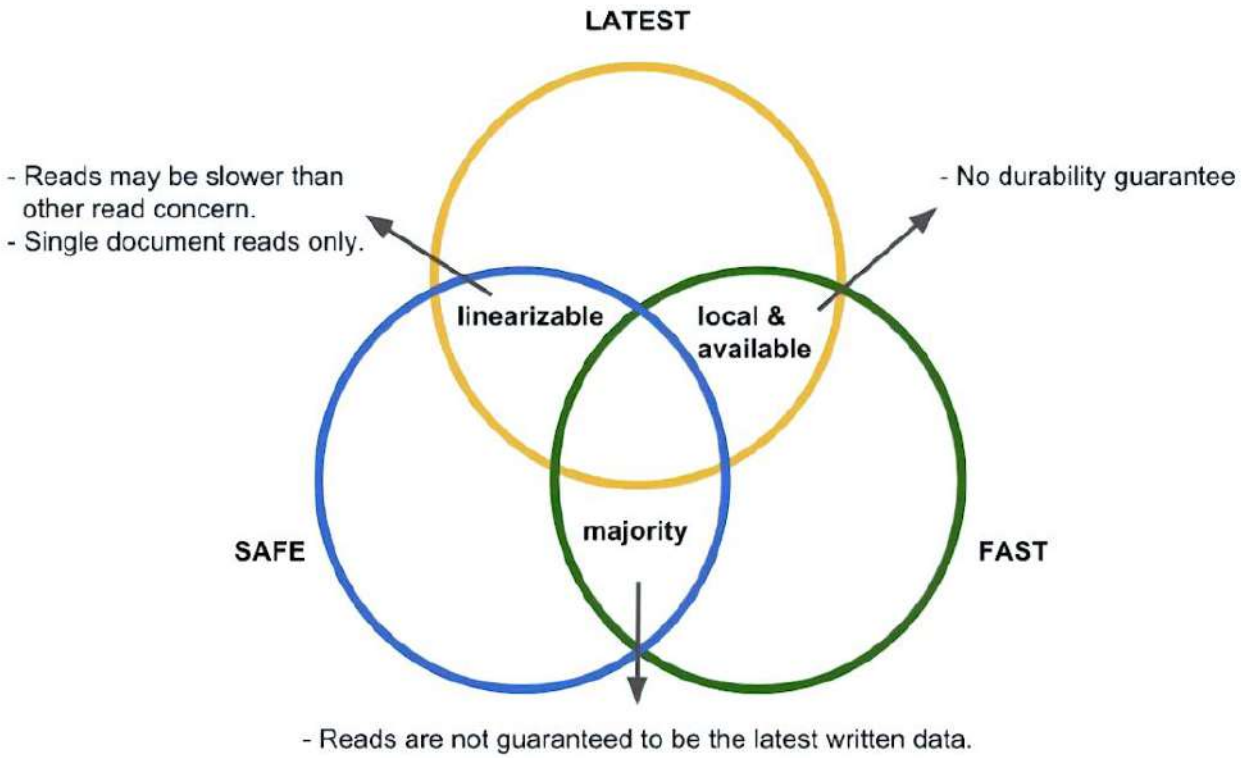
- Niveles de read concern:

    - local (valor por defecto): retorna el dato más reciente en el cluster, que se encuentra en el primario. El dato puede retornarse pero también puede perderse temporalmente si el nodo primario cae y la propagación todavía no ha ocurrido.
    - available (valor por defecto cuando se lee contra un nodo secundario): igual que local, es decir, devuelve datos de la instancia sin garantía de que los datos se hayan escrito en la mayoría de los miembros del replica set. Sin sharding, local es lo mismo que available.
        - Para un clúster fragmentado, available proporciona una mayor tolerancia para las particiones, ya que no espera para garantizar consistencia. Sin embargo, una consulta read

concern available puede devolver documentos huérfanos (que existen en varios shards) si el fragmento está experimentando migraciones de trozos, ya que available, a diferencia de local, no se comunica con los servidores de configuración para obtener los metadatos actualizados.

- majority: solamente devuelve datos que han sido asentidos por la mayoría de miembros del replica set. Provee un nivel de durabilidad mayor que los anteriores, pero es posible que no se pueda obtener el último dato más reciente. El sistema de almacenamiento NMAPv1 no soporta este tipo de read concern.
- linearizable: leerá los últimos datos confirmados con w:majority o bloqueará hasta que el replica set asienta una escritura en progreso con w: majority.
  - La consultas pueden ser muy lentas.
  - Es recomendable establecer un tiempo maxTimeMS con linearizable.
  - La principal diferencia entre majority y linearizable es que linearizable bloquea la operación de lectura hasta que la operación de escritura ha sido confirmada por la mayoría. En majority no hay ningún bloqueo.
  - Es posible encontrar una buena explicación aquí.

- La decisión de un nivel de read concern depende de los requisitos de la aplicación: últimos datos, velocidad o seguridad. Se toman dos.
  - local y available: no garantiza durabilidad, pero toma los datos rápidamente y los últimos.
  - majority: se garantiza mayor durabilidad, pero los datos podrían estar obsoletos.
  - linearizable: últimos datos y garantía de durabilidad, pero de forma más lenta.




- En lecturas sobre nodos secundarios, las configuraciones local y available permiten consultas rápidas pero no necesariamente los datos más recientes.

## Replica set con diferentes índices

- No es muy habitual que los nodos secundarios de un replica set posean diferentes índices, pero puede ser de utilidad para casos de uso específicos:
    - Analíticas para nodos secundarios específicos.
    - Reportar consistencia de los datos.
    - Búsqueda de texto.
- Para estos nodos con índices exclusivos, es conveniente evitar que el nodo se convierta en primario. Para ello:
    - Priority = 0
    - Hidden node
    - Delayed secondary
- Hay que recordar que los índices que se crean en el nodo primario también son propagados a los nodos secundarios de un replica set.
- Para crear un índice en un nodo secundario, primero hay que apagarlo, iniciarlo como standalone, crear el índice correspondiente, apagarlo, y a continuación, relanzar la configuración del nodo secundario en el replica set. El índice no estará disponible en el nodo primario del replica set, sino que únicamente estará en el nodo secundario donde se creó.
- La replicación permite que se pueden mezclar varios motores de almacenamiento en un replica set (MMAPv1 y WiredTiger, por ejemplo).

# Sharding

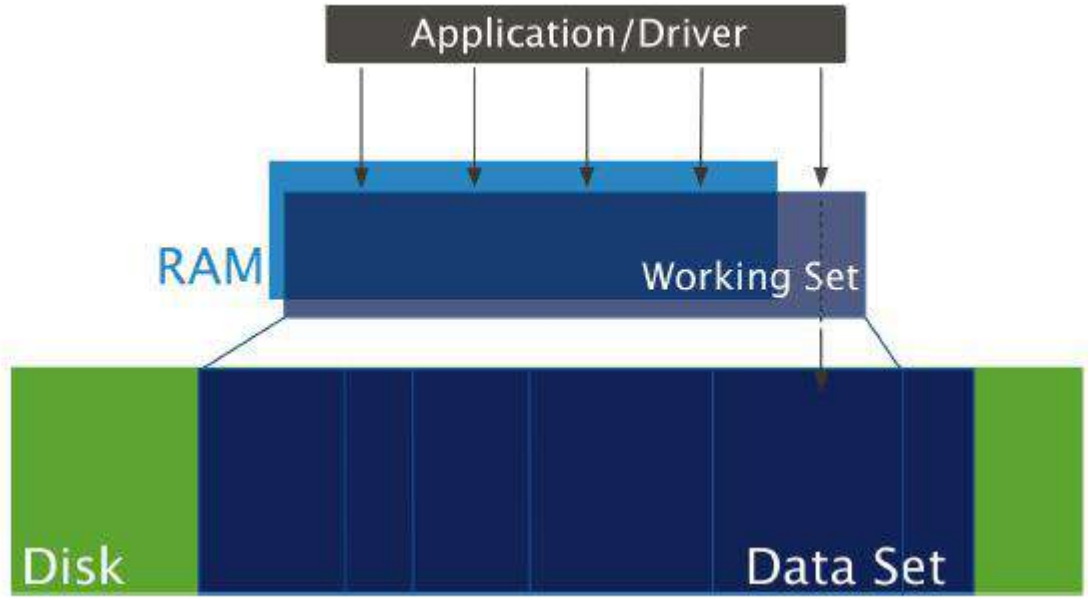
## Índice



## Conceptos básicos de escalabilidad

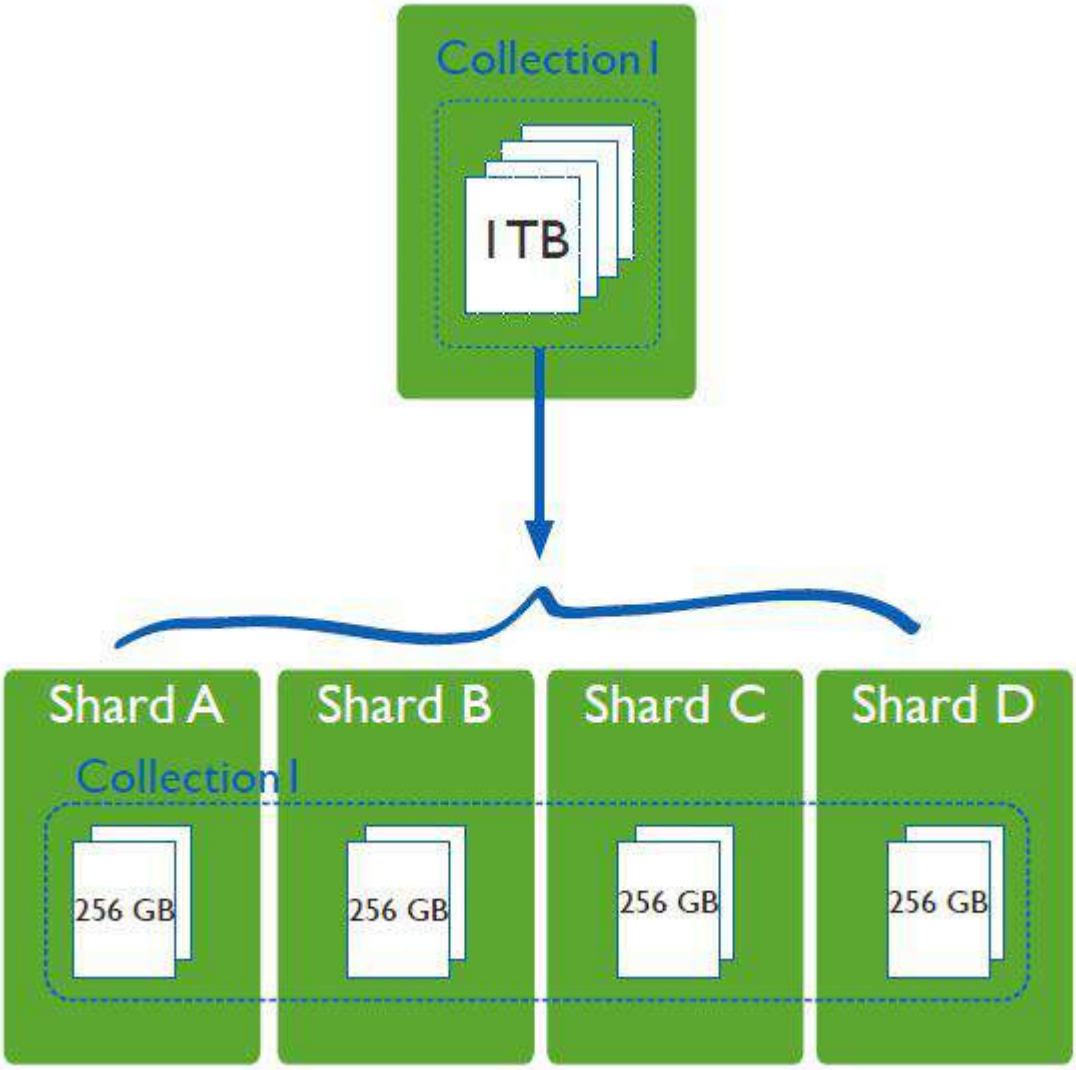
- Antes de recurrir a la escalabilidad es conveniente optimizar al máximo la aplicación, la indexación, el esquema de la base de datos, la configuración del sistema operativo (ulimits, swap, NUMA, NOOP scheduler con hypervisor), tunear el sistema I/O (ext4, RAID10, ...), prestar atención a los logs y comprobar el checklist de producción de MongoDB. El esquema se debe ajustar a los patrones de consulta de la aplicación.
    - De esta forma se consigue minimizar la sobrecarga, se reducen los acceso al disco duro, se reduce la latencia de la aplicación y se consume menos RAM.
- Ejemplos de cuándo utilizar sharding:
    - Una sola instancia de MongoDB no puede mantenerse al día con la carga de escritura y se han agotado otras opciones.
    - El conjunto de datos completo es demasiado grande para una sola instancia de MongoDB.
    - Se pretende mejorar el rendimiento de lectura para la aplicación.
    - El conjunto de datos toma demasiado tiempo en hacer copias de seguridad y restaurar.

- working set: índices más el conjunto de documentos más accedidos. Si el working set se encuentra en RAM, entonces la latencia es menor y el rendimiento es mayor. El working set debería estar siempre en RAM. Si no es posible, es buen momemto para realizar sharding. Working sets más grandes que la memoria RAM del sistema acentúan la cantidad de operaciones E/S de las unidades de disco.




```
// tamaño de los índices para una base de datos
db.stats().indexSize

// tamaño de los índices para una base de datos (en megabytes)
db.stats(1024 * 1024).indexSize

// tamaño aproximado del working set (este comando solo funciona en versiones
antiguas de MongoDB y solamente estimaba un valor adecuado cuando la aplicación
estaba varios días funcionando en producción)
db.serverStatus({workingSet: 1})
```

- Actualmente no hay forma de obtener el tamaño del working set, pero puede obtenerse una aproximación a partir de la suma de todos los índices y de los datos devueltos en las consultas más recurridas por la aplicación. A partir de ahí se puede determinar el número de shards necesarios -> ceil(working set / RAM)
    - Ejemplo: Working Set = 428 GB, Server RAM = 128 GB; 428/128 = 3.34 --> 4 shards requeridos

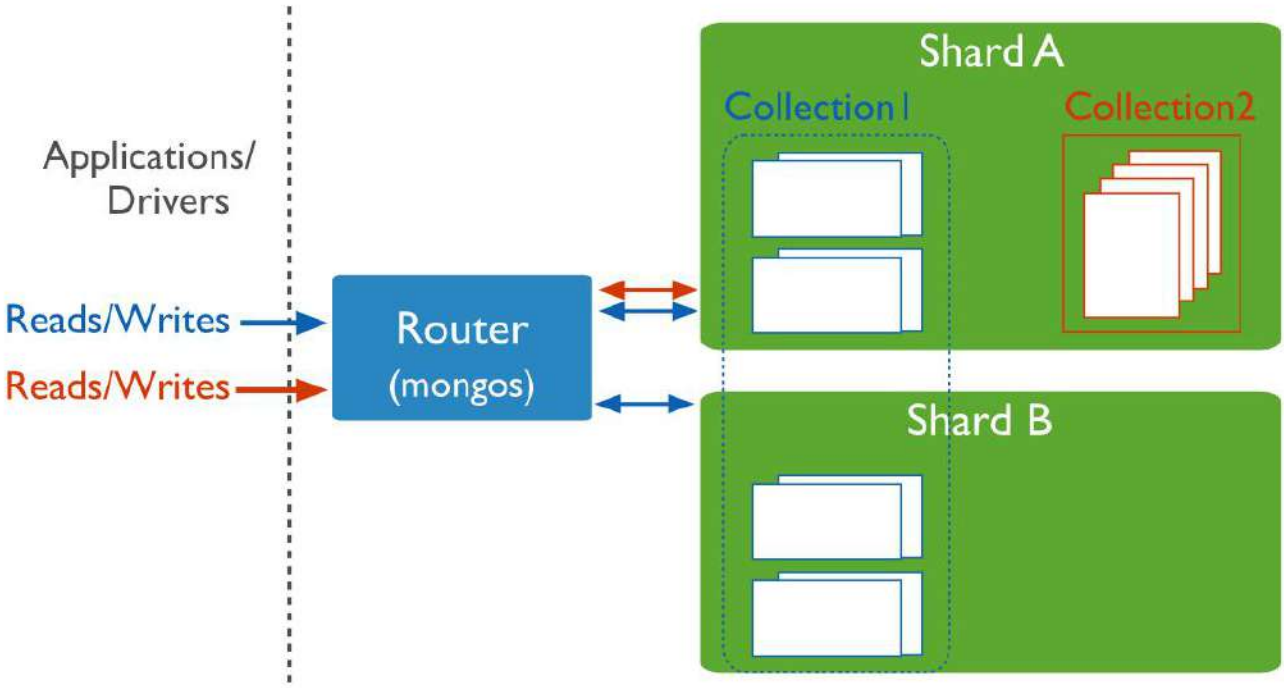



- También se puede calcular el número de shards necesarios a partir del número de operaciones IOPS. Sin embargo, el número de IOPS es difícil de estimar (actualizaciones de documentos, índices, journal, logs, etc.)
- En general -> N = G / (0.7 * S), donde S es el número de operaciones/segundos de un servidor simple (puede calcularse mediante un benchmark), G es el número de operaciones/segundo requerido (puede calcularse mediante el despliegue de la aplicación y simulando el acceso del número de usuarios medio que debe soportar el servicio) y N el número de shards necesario para soportar la carga. Se redondea hacia arriba.
- Será necesario por tanto examinar los requisitos de almacenamiento, latencia y ancho de banda.
- Escalabilidad vertical --> más RAM, más memoria, más disco duro, más velocidad de disco duro o más ancho de banda. Es la opción recomendada cuando es viable económicamente, aunque los precios aumentan considerablemente a medida que crecen los recursos computacionales.
    - DigitalOcean.
    - RamNode
- Escalabilidad horizontal --> más nodos y repartición de la carga.

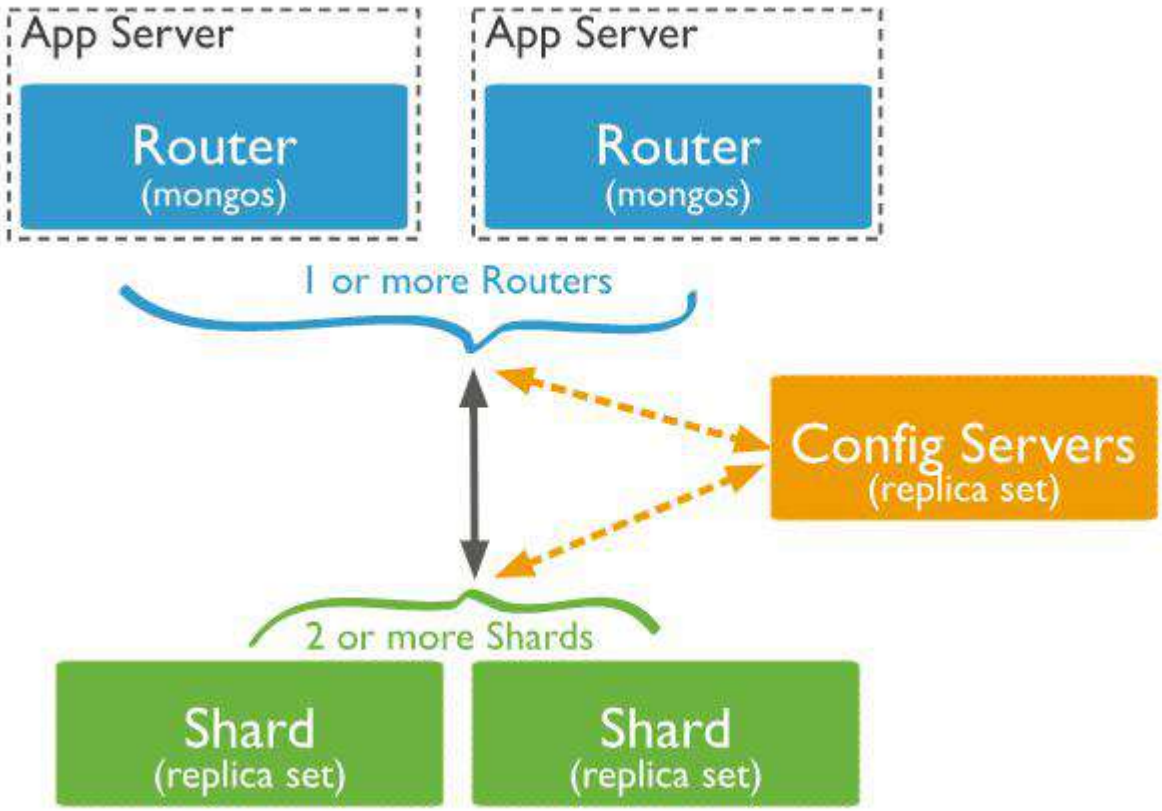
- La escalabilidad vertical a nivel de disco duro puede ser un problema en caso de realizar copias de seguridad y restauración de datos.
    - No es lo mismo hacer copias de seguridad o restauraciones de conjuntos de 5 TB que hacerlo de uno completo de 20TB.
    - La segunda opción es más lenta porque la primera puede ejecutarse de forma paralela. Los índices también serán mayores y por tanto, se necesitará más RAM. No es recomendado que el disco duro tenga una capacidad de más allá de 2-5TB.
- Esta escalabilidad horizontal permite también que las copias de seguridad puedan restaurarse de forma más rápida.
- En MongoDB, la escalabilidad es horizontal.
- Cuando las consultas con el aggregation framework son lentas, también puede ser conveniente realizar sharding. Algunas operaciones del aggregation framework pueden ser paralelizadas, pero no todas.

## Sharded cluster

- Sharding permite escalabilidad mediante shards. Los shards son realmente replica set, es decir, conjunto de réplicas con alta disponibilidad.
- El conjunto de datos se distribuyen en estos shards.
- Los sharded clusters están compuestos por:
    - shard: contiene un subconjunto de los datos fragmentados. El shard debe implementarse como un replica set y deben existir al menos dos shards (para testing y desarrollo se puede utilizar uno).
    - mongos: actúa como un enrutador de consultas y provee la interfaz de acceso para las aplicaciones.
        - Es habitual ubicar mongos en el servidor de aplicación (de esta forma se reduce la latencia).
        - También se puede ubicar un mongos en un host dedicado.
        - No hay límite de instancias de mongos. Sin embargo, mongos se comunica frecuentemente con los config servers y esto puede degradar el servicio mongos.
        - mongos no tiene un estado persistente y consume muy pocos recursos.
        - mongos combinan los datos de cada shard seleccionado y devuelve el documento resultante. Ciertos modificadores de consulta, como sort, se ejecutan en el shard primario antes de que mongos recupere los resultados.
    - config server: almacenan metadatos y políticas de configuración para el cluster. Los config servers deben ser deplegados como un replica set.
- mongos es el responsable de enrutar todas las operaciones de la aplicación a los shards que están afectados y provee servicio a la aplicación. La aplicación no interactuará con los instancias mongod de los shards.



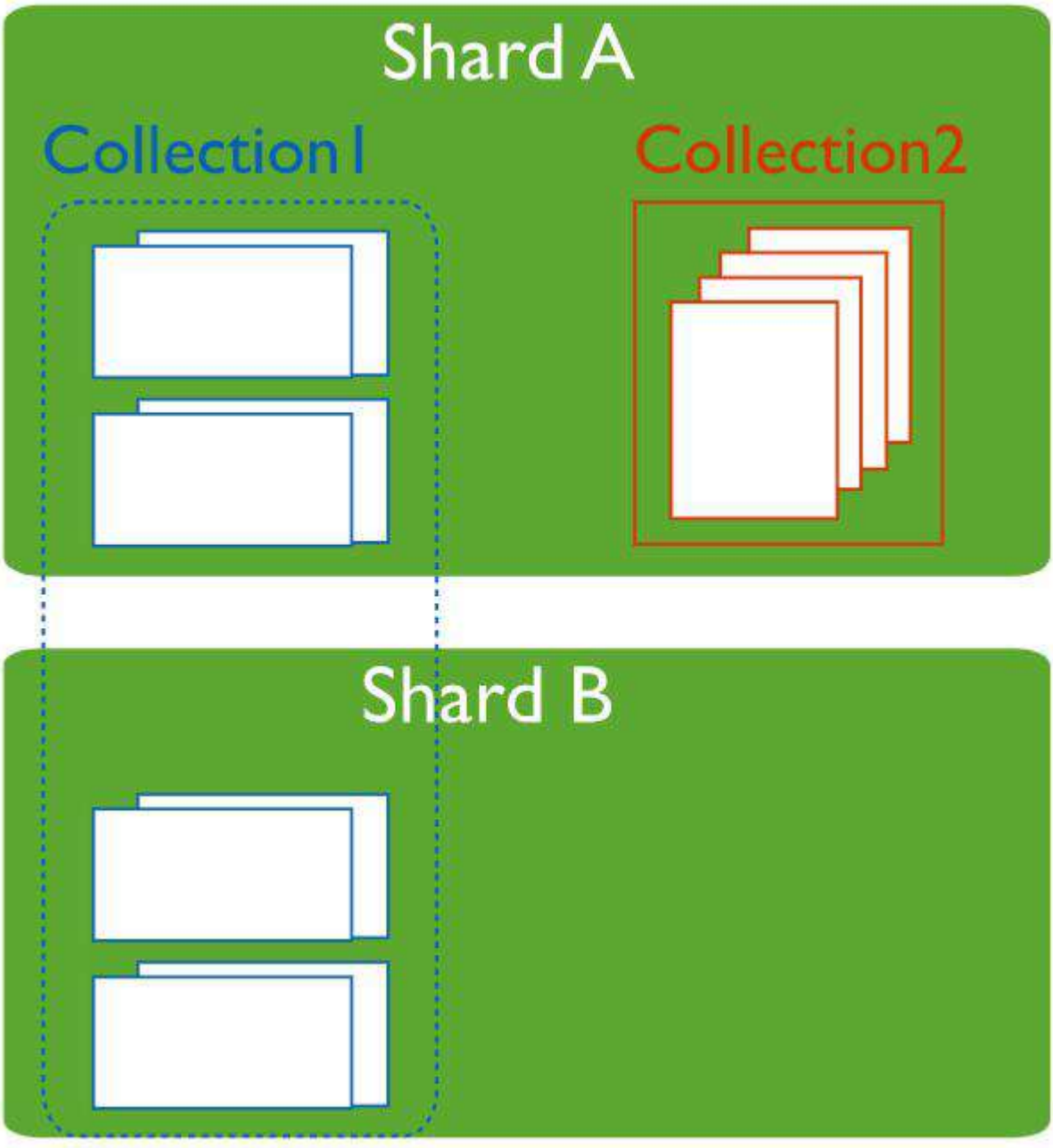

- Pueden existir también múltiples mongos.




- Los metadatos de la colección se encuentran almacenados en config servers, que hacen un seguimiento de dónde se encuentra cada dato que existe en el cluster.
- Los config server también se encargan de distribuir homogéneamente la información entre dos instancias de mongod dentro de una shard cuando una de ellas se encuentra sobrecargada con

respecto al resto

- Mongos consulta con los servidores de configuración a menudo, en caso de que una parte de los datos se haya movido.
- En un sharded cluster, también existe el concepto de shard primario.
- Cada base de datos en un sharded cluster tiene un shard primario que contiene todas las colecciones no fragmentadas para esa base de datos. Cada base de datos tiene su propio shard primario. El shard primario no tiene relación con el primario en un replica set.
- No todas las colecciones de un sharded cluster necesitan ser fragmentadas. Estas colecciones son colocadas en el shard primario.



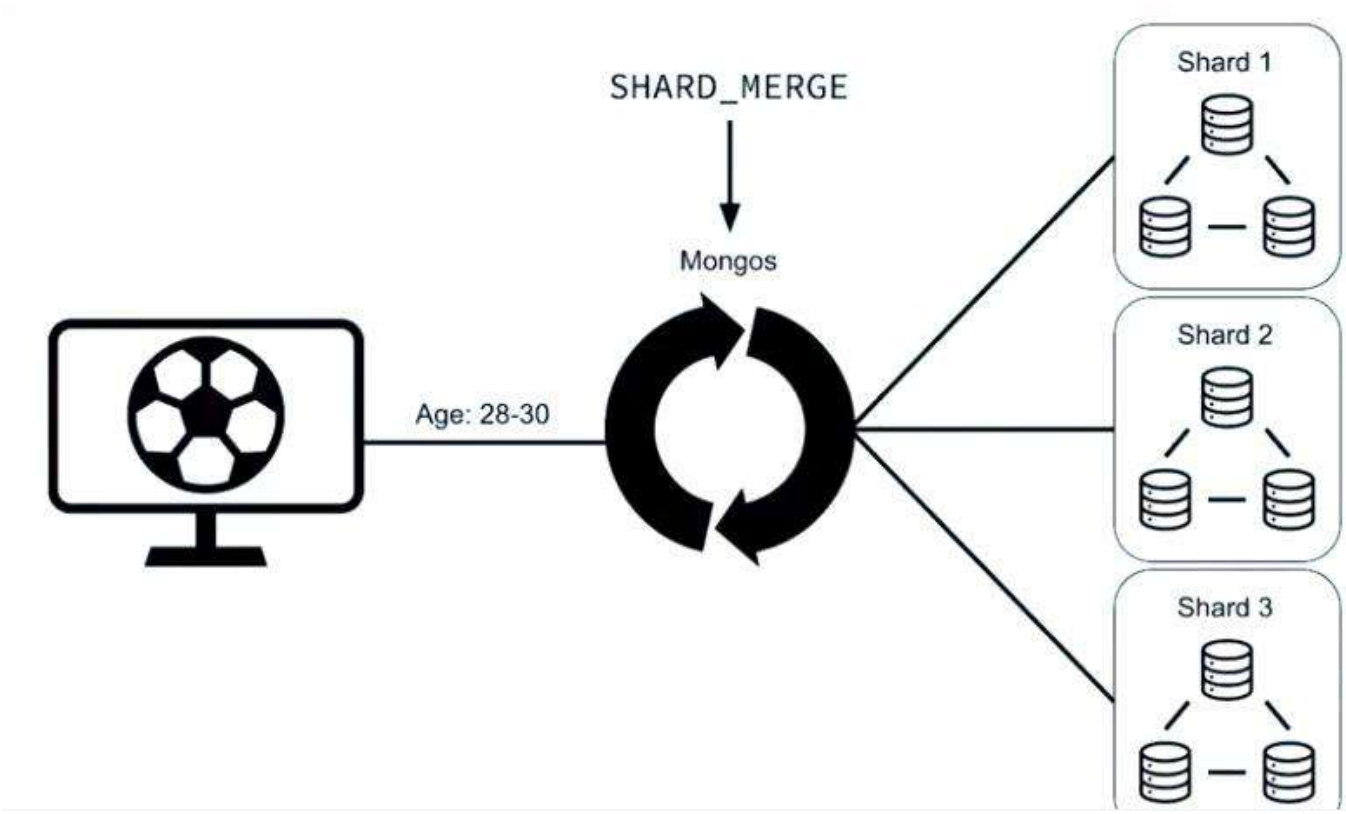

- Es posible cambiar el shard primario de una base de datos.

- Para cambiar el shard primario de una base de datos es necesario utilizar el comando movePrimary. El proceso de migración del shard primario puede tardar un tiempo considerable en completarse, y no debe accederse a las colecciones asociadas a la base de datos hasta que se complete.

```
// mueve la base de datos test (no fragmentada) al shard0001
db.adminCommand( { movePrimary : "test", to : "shard0001" } )
```

- Mongos selecciona el shard primario al crear una nueva base de datos seleccionando el shard en el cluster que tiene la menor cantidad de datos. mongos usa el campo totalSize devuelto por el comando listDatabase como parte de los criterios de selección.

- Las labores de merge es realizada por mongos




- Por tanto, en un sharded cluster mínimo se requiere de una instancia de mongos, un replica set de config server (CSRS) y al menos un shard.

- Si es posible, es conveniente distribuir los miembros de un replica set en al menos tres centros de datos (con un árbitro en el tercer centro de datos).

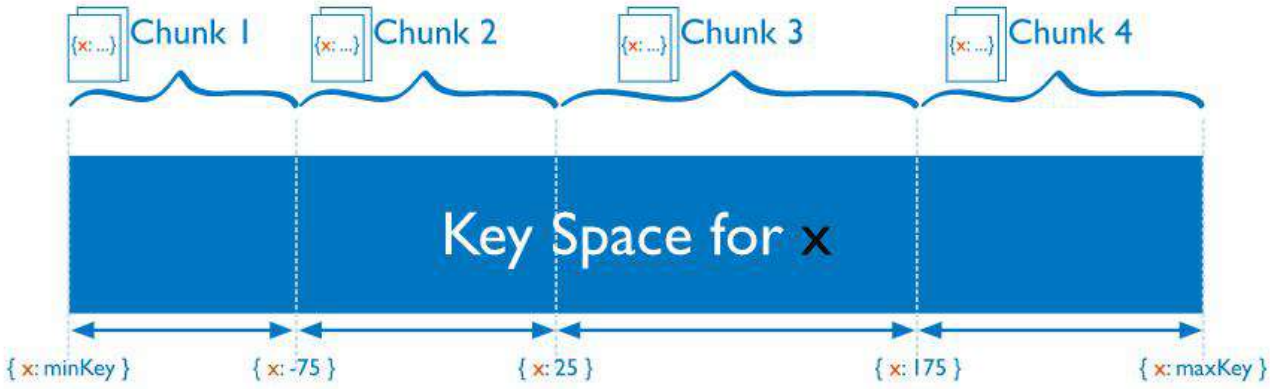
    - Si el centro de datos con minoría de miembros se cae, el replica set puede seguir sirviendo operaciones de escritura y de lectura.
    - Sin embargo, si el centro de datos con mayoría se cae completamente, el replica set se convierte en solo lectura.

- La fragmentación requiere al menos dos shards para distribuir datos fragmentados. Un solo shard puede ser útil si se planea habilitar sharding en un futuro próximo.

- Puede utilizarse autenticación interna para imponer seguridad dentro del cluster y evitar que componentes no autorizados accedan al cluster.
    - Cada shard es compatible con el Control de acceso basado en roles (RBAC) para restringir el acceso no autorizado a datos y operaciones de shard. Para configurar la autenticación, cada mongod en el replica set debe arrancar con la opción --auth para aplicar RBAC.
- El comando sh.status() sirve para obtener información sobre un cluster.

## Shard key

- Para distribuir los documentos en una colección, MongoDB particiona la colección usando la shard key, que consiste en un campo o campos inmutables que existen en cada documento de la colección objetivo.
- Es necesario elegir la shard key al fragmentar una colección.




- La elección de la shard key no se puede cambiar después de fragmentar.
- Para fragmentar una colección no vacía, la colección debe tener un índice que comience con la shard key. Para colecciones vacías, MongoDB crea el índice si la colección todavía no tiene un índice apropiado para la shard key especificada.
- La elección de la shard key afecta el rendimiento, la eficiencia y la escalabilidad de un sharded cluster. Un cluster con el mejor hardware e infraestructura posible puede verse obstaculizado por la elección de un incorrecto shard key. La elección de la shard key y su índice de respaldo también pueden afectar a la estrategia de fragmentación.
- No se pueden especificar índices multiclaves como shard key. Sin embargo, si el índice de la shard key es un prefijo de un índice compuesto, se permite que el índice compuesto se convierta en un índice compuesto multiclave si una de las otras claves (es decir, las claves que no forman parte de la shard key) indexa un array.

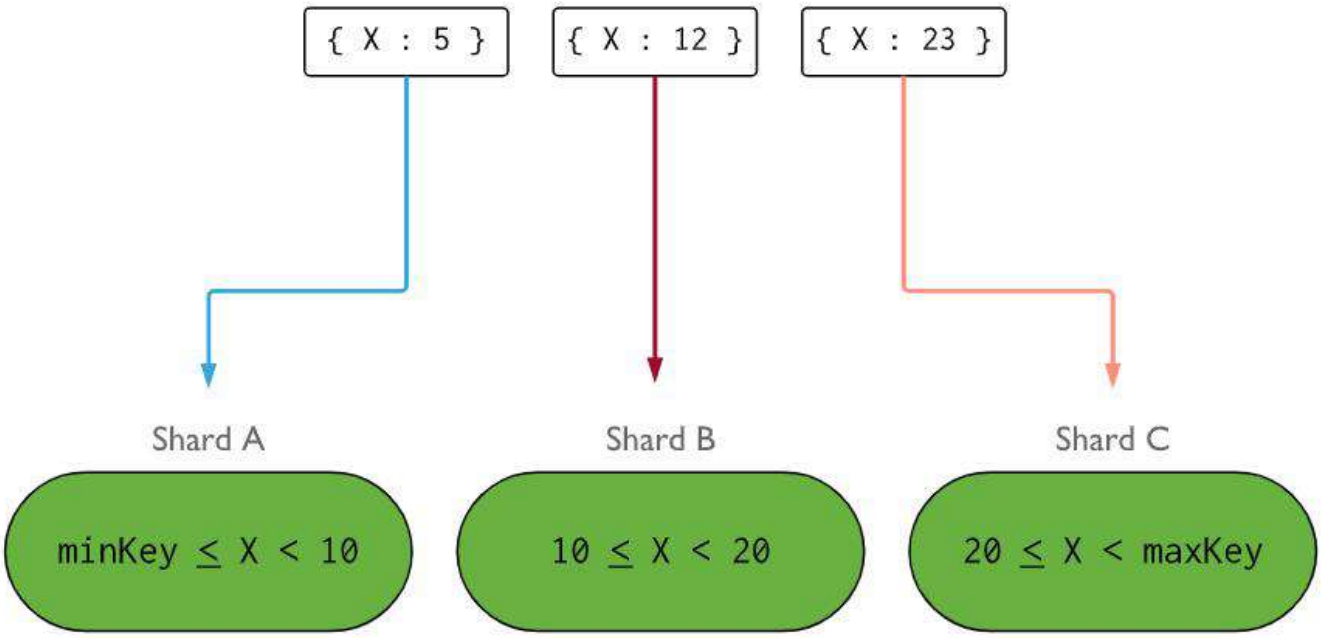
```
// documento ejemplo en una colección a fragmentar
{
  a: 4,
  b: 3,
  c: [1,2,3]
}
```

```
// índice
{ a: 1, b: 1, c: 1}

// shard key válido
{ a: 1, b: 1 }
```

- En un sharded cluster, el _id debe ser único en toda la colección fragmentada porque los documentos pueden migrar a otro shard, y valores idénticos evitan que el documento se inserte en el shard del receptor. Es responsabilidad de la aplicación garantizar la exclusividad en _id para una colección determinada en un sharded cluster si no es la clave de la partición.
    - Si se utiliza _id como shard key, el sistema impondrá automáticamente la unicidad de los valores, ya que los rangos de chunks se asignan a un solo shard, y el shard puede garantizar la unicidad de los valores en ese rango.
    - En cuanto al shard key, si no es el _id, es perfectamente aceptable tener valores idénticos para documentos diferentes. Sin embargo, hay que tener cuidado de tener demasiados documentos con los mismos valores, ya que esto llevará a chunks gigantes.

## Chunks

- En un entorno de sharding, MongoDB divide los datos en chunks (trozo o pedazo). Cada chunk tiene un rango superior inferior inclusivo y superior exclusivo basado en la shard key.




- En un intento de lograr una distribución uniforme de los chunks en todos los shards del cluster, un balanceador se ejecuta en segundo plano para migrar los chunks a través de los shards.
- MongoDB no garantiza que dos chunks contiguos residan en un solo shard.
- El tamaño de chunk predeterminado en MongoDB es de 64 megabytes. Puede aumentarse o reducirse el tamaño del chunks pero hay implicaciones:

    - Los chunks pequeños conducen a una distribución de datos más uniforme a expensas de migraciones más frecuentes. Esto crea gastos en la capa de enrutamiento de consulta (mongos).
    - Los chunks grandes conducen a menos migraciones. Esto es más eficiente tanto desde la perspectiva de la red como en términos de sobrecarga interna en la capa de enrutamiento de consultas. Pero, estas eficiencias se producen a expensas de una distribución de datos potencialmente desigual.

- Cambiar el tamaño del chunk afecta cuando los chunks se dividen, pero sus efectos tienen algunas limitaciones:

    - La división automática solo ocurre durante las inserciones o actualizaciones. Si se reduce el tamaño del chunk, puede tomar tiempo para que todos los chunks se dividan al nuevo tamaño.
    - Las divisiones no pueden ser "deshechas". Si se aumenta el tamaño del chunk, los chunks existentes deben crecer a través de inserciones o actualizaciones hasta que alcancen el nuevo tamaño.

- La migración de chunks es posible realizarla también manualmente.

- Cuando se está migrando un chunk, todavía se puede leer en el shard origen hasta que el chunk sea copiado al shard destino.

- Para determinar si todas las migraciones están completas hay que ejecutar sh.isBalancerRunning() mientras se está conectado a una instancia de mongos.

- Pueden existir chunks que no contienen documentos.

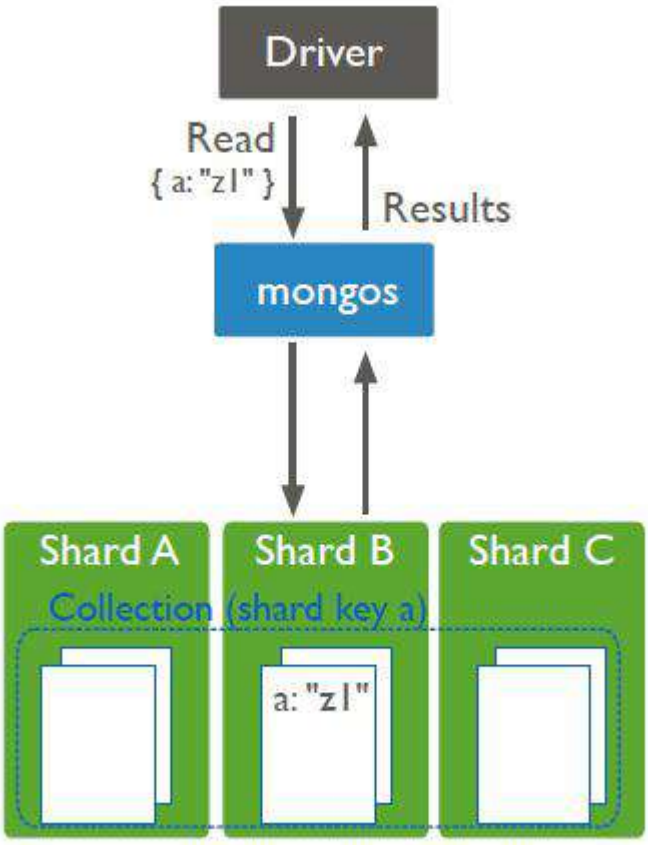
- Los rangos de dos chunks no se pueden superponer.

- Cada chunk es asignado a un shard y solamente a uno.

## Ventajas de sharding

- MongoDB distribuye la carga de trabajo de lectura y escritura entre los shards del cluster sharding, lo que permite que cada shard procese un subconjunto de operaciones de cluster.

- Para las consultas que incluyen el shard key o el prefijo de una shard key compuesta, los mongos pueden orientar la consulta a un shard específico o conjunto de hard. Estas operaciones dirigidas son generalmente más eficientes que la difusión a todos los shards del cluster.

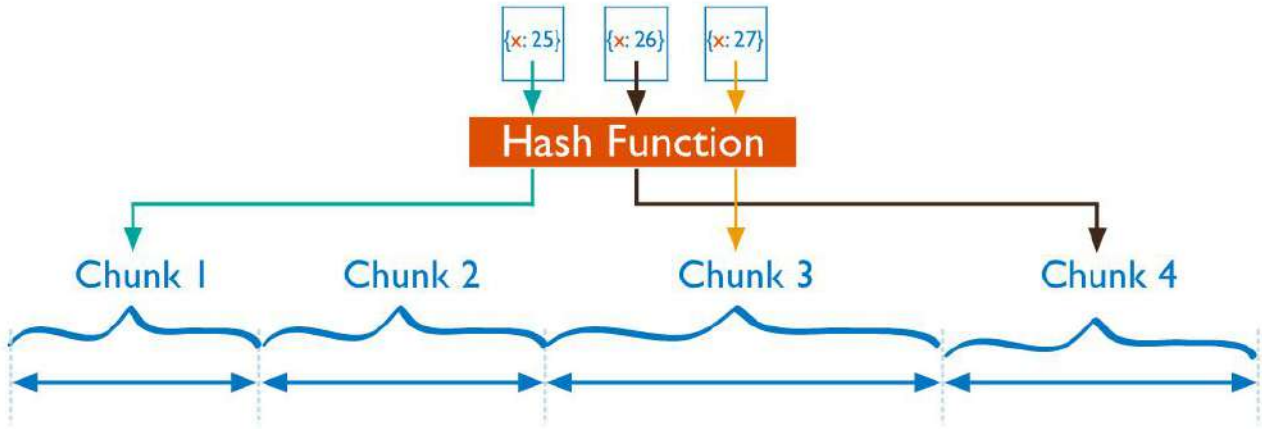



- Sharding distribuye los datos a través de los shards en el cluster, permitiendo que cada shard contenga un subconjunto de los datos totales del cluster. A medida que el conjunto de datos crece, los shard adicionales aumentan la capacidad de almacenamiento del cluster.

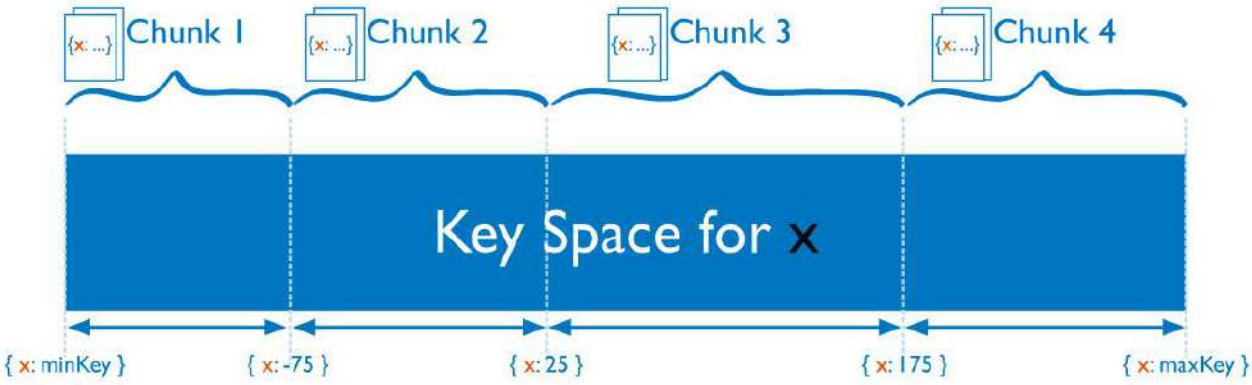
- Un cluster sharding puede continuar realizando operaciones parciales de lectura / escritura, incluso si uno o más shard no están disponibles. Si bien no se puede acceder al subconjunto de datos en los shard no disponibles durante el tiempo de inactividad, las lecturas o escrituras dirigidas a los shards disponibles todavía pueden tener éxito.

## Estrategias de sharding

- Hashed sharding: implica calcular un hash del valor del campo de shard key. A cada shard se le asigna un rango basado en los valores de la shard key hasheada.
    - La distribución de datos basada en valores de hash facilita una distribución de datos más uniforme, especialmente en conjuntos de datos donde la shard key cambia monótonamente.
    - Sin embargo, la distribución de hash significa que las consultas basadas en rangos en la shard key tienen menos probabilidades de apuntar a un solo shard, lo que da como resultado más operaciones de difusión en todo el cluster.
    - Los ObjectId generados automáticamente son valores monótonamente crecientes. El uso de estos como shard keys conduce a que un shard reciba todas las operaciones de inserción. Hasheando esos valores de ObjectId se creará una distribución uniforme de valores, y por lo tanto, se enviará operaciones de inserción en todos los shards de manera distribuida.




- Ranged sharding: implica dividir los datos en intervalos en función de los valores de la shard key. A cada shard se le asigna un rango basado en los valores de la shard key.
    - Es más probable que un rango de shard keys cuyos valores están cercanos, residan en el mismo shard. Esto permite operaciones dirigidas, ya que mongos puede enrutar las operaciones solo a los shards que contienen los datos requeridos.
    - La eficiencia de la fragmentación depende de la shard key elegida. La elección de una shard key incorrecto puede dar como resultado una distribución desigual de los datos, lo que puede anular algunos de los beneficios de la fragmentación o causar cuellos de botella en el rendimiento.




## Despliegue de un shard cluster mínimo

- A continuación se detallan los pasos para desplegar un shard (replica set con tres nodos) y tres config server en replica set.

| Dato | Valor |
|---|---|
| usuario | vagrant |
| ip para todas las instancia | 10.0.2.15 |
| nombre del replica set de los config server | csrs |
| nombre del replica set para el shard | replica1 |
| directorio de trabajo | /home/vagrant/ |
| directorio de los datos | ./data/ |

| Dato | Valor |
|---|---|
| directorio de los logs | ./log/ |
| directorio de la clave | ./pki/ |

- En primer lugar, es conveniente establecer la configuración del replica set del config server.

```
cd /home/vagrant
vim csrs_1.conf
```

```
# csrs_1.conf
sharding:
  clusterRole: configsvr
replication:
  replSetName: csrs
security:
  keyFile: pki/keyfile
net:
  bindIp: localhost,10.0.2.15
  port: 26001
systemLog:
  destination: file
  path: log/csrs1/mongod.log
  logAppend: true
processManagement:
  fork: true
storage:
  dbPath: data/csrs1/
```

```
# csrs_2.conf
sharding:
  clusterRole: configsvr
replication:
  replSetName: csrs
security:
  keyFile: pki/keyfile
net:
  bindIp: localhost,10.0.2.15
  port: 26002
systemLog:
  destination: file
  path: log/csrs2/mongod.log
  logAppend: true
processManagement:
  fork: true
storage:
  dbPath: data/csrs2/
```

```
# csrs_3.conf
sharding:
  clusterRole: configsvr
replication:
  replSetName: csrs
security:
  keyFile: pki/keyfile
net:
  bindIp: localhost,10.0.2.15
  port: 26003
systemLog:
  destination: file
  path: log/csrs3/mongod.log
  logAppend: true
processManagement:
  fork: true
storage:
  dbPath: data/csrs3/
```

```
mkdir -p ./data/{csrs1,csrs2,csrs3}
mkdir -p ./pki/
mkdir -p ./log/{csrs1,csrs2,csrs3}
openssl rand -base64 741 > ./pki/keyfile
chmod 400 ./pki/keyfile
```

- Iniciar los tres config servers.

```
mongod -f csrs_1.conf
mongod -f csrs_2.conf
mongod -f csrs_3.conf
```

- Crear replica set.

```
mongo --port 26001
```

```
rs.initiate()
rs.status()
use admin
db.createUser({
  user: "admin",
  pwd: "pass",
  roles: [
```

```
    {role: "root", db: "admin"}
  ]
})
db.auth("admin", "pass")
rs.add("10.0.2.15:26002")
rs.add("10.0.2.15:26003")
rs.isMaster()
```

- Configuración de mongos para que apunte al CSRS. No requiere de dbPath como en los shard o config server.

```
# mongos.conf
sharding:
  configDB: csrs/10.0.2.15:26001,10.0.2.15:26002,10.0.2.15:26003
security:
  keyFile: pki/keyfile
net:
  bindIp: localhost,10.0.2.15
  port: 26000
systemLog:
  destination: file
  path: log/mongos/mongos.log
  logAppend: true
processManagement:
  fork: true
```

- Arrancar mongos.

```
mkdir -p ./log/mongos
mongos -f mongos.conf

# la autenticación para mongos es la misma que para el nodo primario del replica
set
mongo --port 26000 -u "admin" -p "pass" --authenticationDatabase "admin"
```

- Comprobar el status del sharding. Actualmente no hay ningún shard configurado. Este informe incluye qué shard es primario para la base de datos y la distribución.

```
sh.status()
```

- Actualizar la configuración para los nodos del replica set del cluster. La limitación de cacheSizeGB es necesario en entornos de vagrant.

```
# node1.conf
sharding:
```

```
  clusterRole: shardsvr
storage:
  dbPath: data/node1/
  wiredTiger:
    engineConfig:
      cacheSizeGB: .1
net:
  bindIp: 10.0.2.15,localhost
  port: 27011
security:
  keyFile: pki/keyfile
systemLog:
  destination: file
  path: log/node1/mongod.log
  logAppend: true
processManagement:
  fork: true
replication:
  replSetName: replica1
```

```
# node2.conf
sharding:
  clusterRole: shardsvr
storage:
  dbPath: data/node2
  wiredTiger:
    engineConfig:
      cacheSizeGB: .1
net:
  bindIp: 10.0.2.15,localhost
  port: 27012
security:
  keyFile: pki/keyfile
systemLog:
  destination: file
  path: log/node2/mongod.log
  logAppend: true
processManagement:
  fork: true
replication:
  replSetName: replica1
```

```
# node3.conf
sharding:
  clusterRole: shardsvr
storage:
  dbPath: data/node3/
  wiredTiger:
    engineConfig:
```

```
      cacheSizeGB: .1
net:
  bindIp: 10.0.2.15,localhost
  port: 27013
security:
  keyFile: pki/keyfile
systemLog:
  destination: file
  path: log/node3/mongod.log
  logAppend: true
processManagement:
  fork: true
replication:
  replSetName: replica1
```

```
mkdir -p ./data/{node1,node2,node3}
mkdir -p ./log/{node1,node2,node3}
```

- Iniciar el replica set.

```
mongod -f node1.conf
mongod -f node2.conf
mongod -f node3.conf
```

- Crear replica set.

```
mongo --port 27011
```

```
rs.initiate()
rs.status()
use admin
db.createUser({
  user: "admin",
  pwd: "pass",
  roles: [
    {role: "root", db: "admin"}
  ]
})
db.auth("admin", "pass")
rs.add("10.0.2.15:27012")
rs.add("10.0.2.15:27013")
rs.isMaster()
```

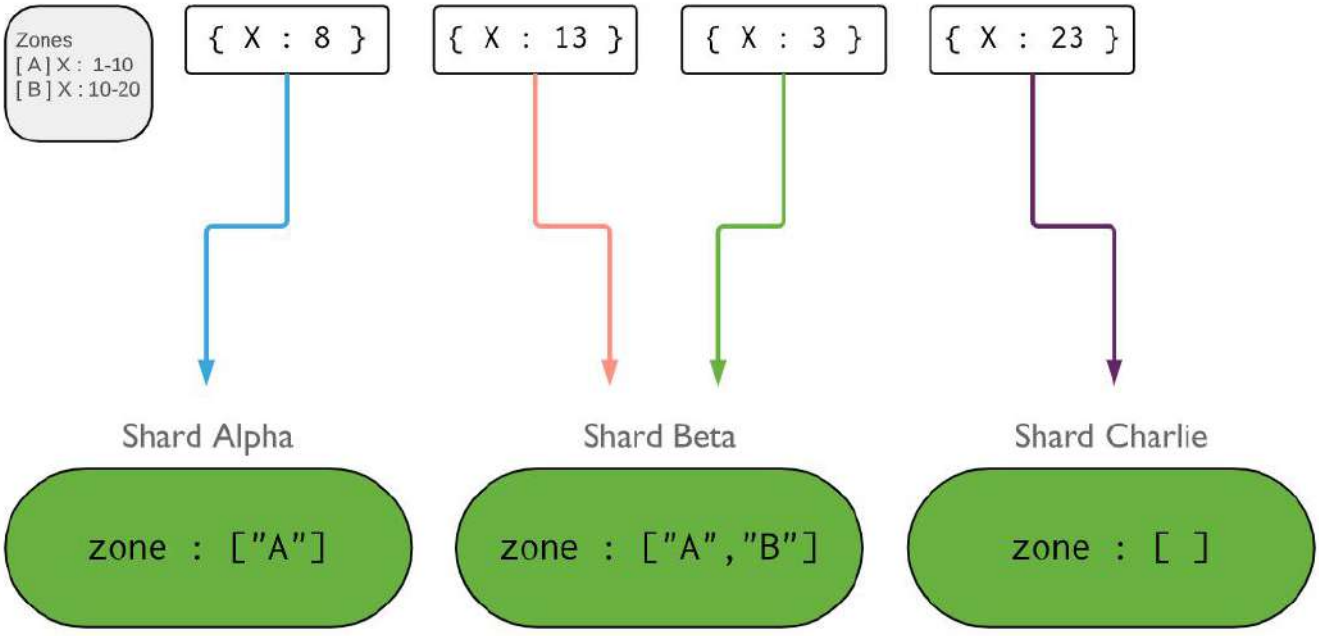
- Conectarse a mongos y añadir el shard.

```
mongo --port 26000 -u "admin" -p "pass" --authenticationDatabase "admin"
```

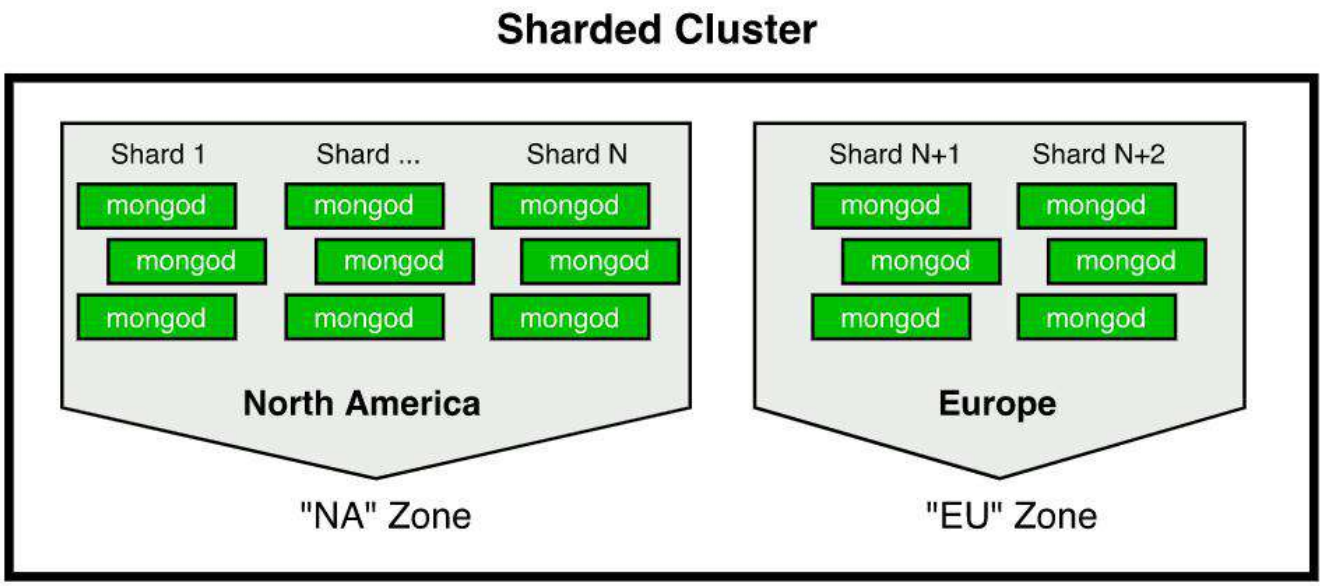
```
sh.addShard("replica1/10.0.2.15:27011")
sh.status()
```

## Zonas en sharded cluster

- En los sharded cluster pueden crearse zonas de datos fragmentados en función de la shard key.
    - Puede asociarse cada zona con uno o más shards del cluster.
    - Un shard puede asociarse con cualquier número de zonas.
- La siguiente imagen ilustra un sharded cluster con tres shards y dos zonas. La zona A representa un rango con un límite inferior de 1 y un límite superior de 10. La zona B representa un rango con un límite inferior de 10 y un límite superior de 20. Los shards Alfa y Beta se encuentran la zona A. Shard Beta también se encuentra en la zona B. Shard Charlie no tiene zonas asociadas. El cluster está en un estado estable y ningún chunks viola ninguna de las zonas (las zonas no pueden compartir rangos y no pueden tener rangos superpuestos).




- En un cluster balanceado, MongoDB migra los chunks cubiertos por una zona solo a los shards asociados con la zona.




- Algunos patrones de despliegue comunes donde se pueden aplicar las zonas son los siguientes:
    - Aislar un subconjunto específico de datos en un conjunto específico de shards.
    - Asegurarse de que los datos más relevantes residen en los shards geográficamente más cercanos a los servidores de aplicaciones.
    - Enrutar los datos a los shards en función del hardware/rendimiento del hardware del shard.
- Cada zona cubre uno o más rangos de valores de shard key. Cada rango que cubre una zona incluye siempre su límite inferior y excluye su límite superior.
- Al elegir una shard key es necesario considerar la posibilidad de utilizar la división de zonas en el futuro, ya que no puede cambiarse la shard key después de fragmentar la colección.
- Más comúnmente, las zonas sirven para mejorar la localidad de los datos para los sharded cluster que abarcan múltiples centros de datos

```
// agrega la zona NYC a dos shards, y las zonas SFO y NRT a un tercer shard:
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "NYC")
sh.addShardTag("shard0002", "SFO")
sh.addShardTag("shard0002", "NRT")

// elimina una zona de un shard particular
sh.removeShardTag("shard0002", "NRT")

// crea rangos de zona
sh.addTagRange("records.users", { zipcode: "10001" }, { zipcode: "10281" }, "NYC")
sh.addTagRange("records.users", { zipcode: "11201" }, { zipcode: "11240" }, "NYC")
sh.addTagRange("records.users", { zipcode: "94102" }, { zipcode: "94135" }, "SFO")

// elimina un rango de zona
sh.removeRangeFromZone("records.user", {zipcode: "10001"}, {zipcode: "10281"})

// devuelve todos los shards en una zona
```

```
use config
db.shards.find({ tags: "NYC" })
```

- Después de asociar una zona con un shard o shards y configurar la zona con un rango de del shard key para una colección fragmentada, el cluster puede tardar un tiempo en migrar los datos afectados para la colección fragmentada. Esto depende de la división de los chunks y la distribución actual de los datos en el cluster. Cuando se completa el balanceo, las lecturas y escrituras de los documentos en una zona determinada se enrutan solo al shard o shards dentro de esa zona.

## La base de datos config

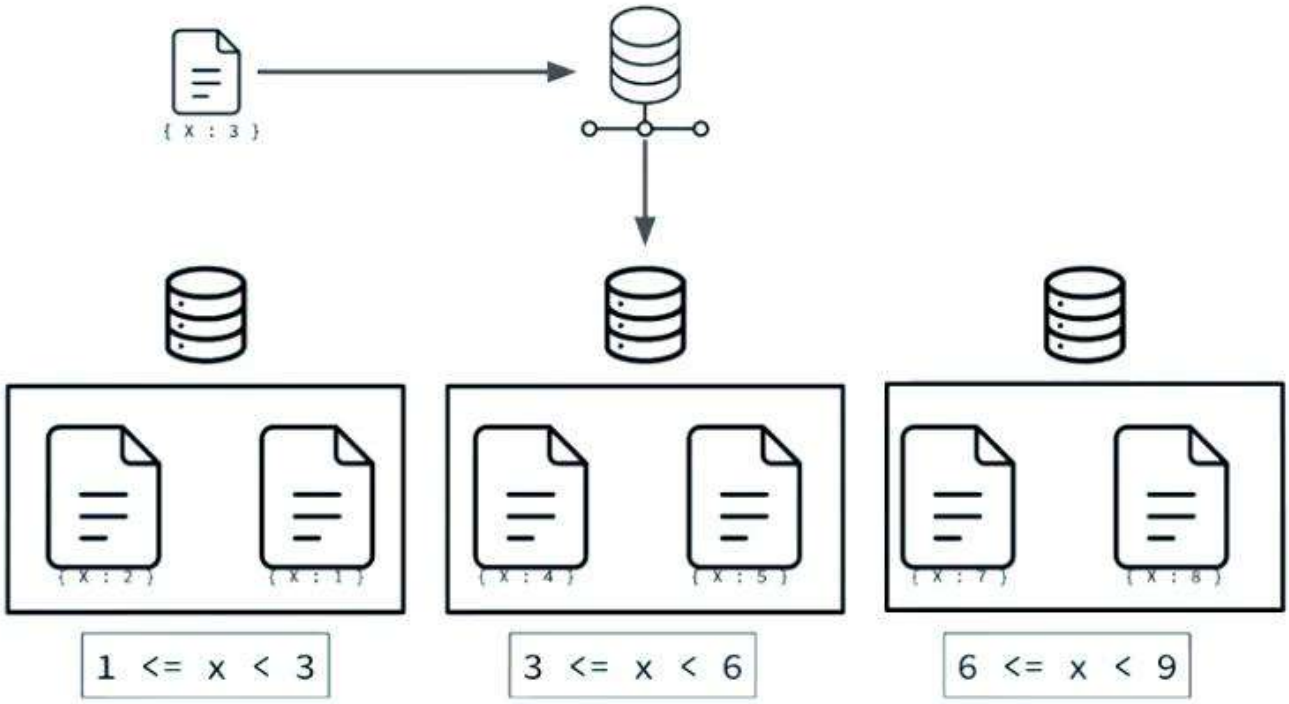
- Contiene datos internos para el correcto funcionamiento de MongoDB y no se debe escribir sobre esta base de datos.
- Puede accederse a ella a desde mongos.

```
// retorna cada base de datos del cluster. Proporciona el primary shard para cada base de datos. El parámetro partitioned indica que el sharding ha sido habilitado para la base de datos.
use config
db.databases.find().pretty()

// colecciones que han sido fragmentadas. Entre la información disponible se encuentra en shard key.
db.collections.find().pretty()

// indica los shards que existe en el cluster
db.shards.find().pretty()

// muestra los chunks del cluster, incluyendo el rango chunk o los valores del shard key. Valores como el mínimo o el maximo para un chunk son estabelecidos. {"$minkey" :1} hace mención al mínimo valor posible o al infinito negativo, mientras que {"$axkey" :1} hace mención al máximo valor posible o al infinito positivo.
db.chunks.find().pretty()

// otro tipo de información de mongos.
db.mongos.find().pretty()
```

## Shard keys en detalle

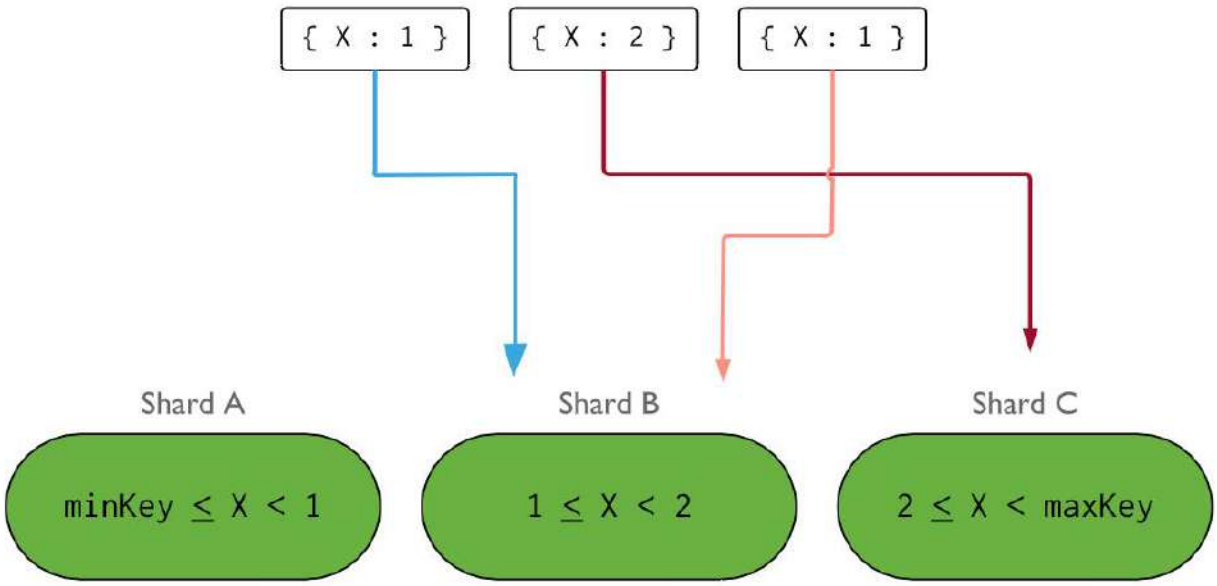
- Todas las colecciones fragmentadas deben tener un índice que admita la shard key.
- El índice puede ser un índice en la shard key o un índice compuesto donde la shard key es un prefijo del índice.
    - Si la colección está vacía, sh.shardCollection crea el índice en la shard key si dicho índice no existe.
    - Si la colección no está vacía, primero debe crearse el índice antes de usar sh.shardCollection.

- El shard key es el campo o campos indexados que MongoDB utiliza para particionar los datos en una colección fragmentada y distribuirlo en los shards de un cluster.
    - Una colección fragmentada solo puede tener una única shard key.
- Cada grupo fragmentado de datos es denominado como chunk. El valor del campo que se toman como shard key ayuda a definir el límite inclusivo inferior y el exclusivo límite superior de cada chunk. Un shard puede contener varios chunks.

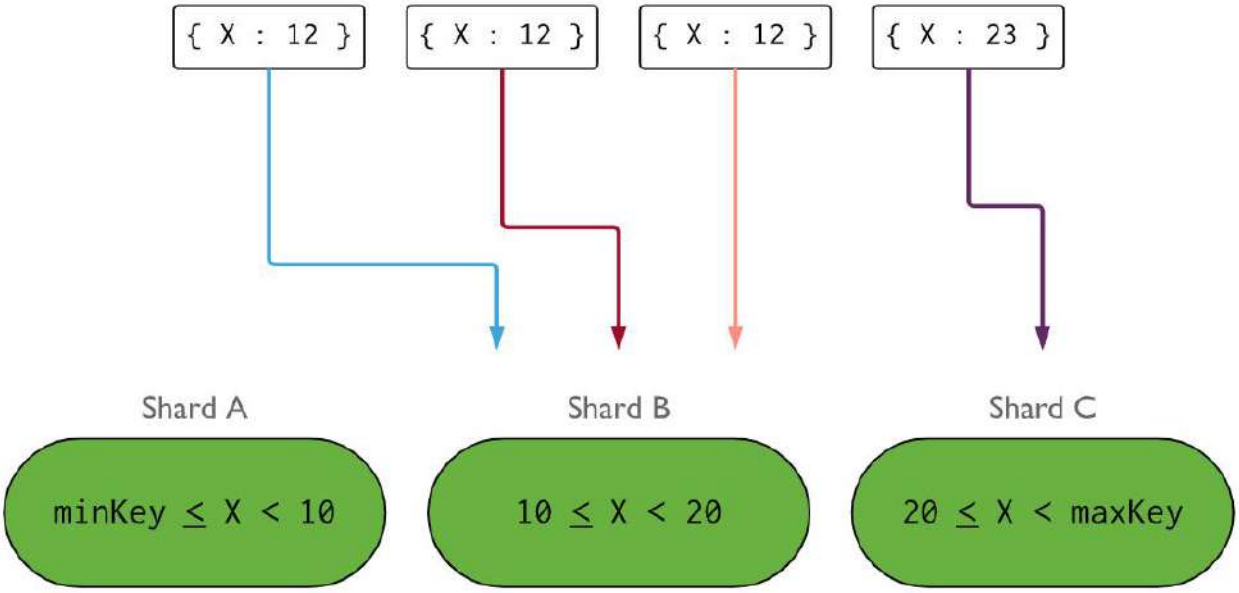



- Por defecto, la capacidad máxima de un chunk es de 64 MB. Cuando se supera ese valor, se crea un nuevo chunk. Si un chunk creado posee un límite inferior y límite superior iguales, entonces este chunk nunca podrá ser fragmentado. Son los denominados jumbo chunks y reducen la efectividad de la escalabilidad horizontal. Esto puede ser mitigado creando shard keys compuestas por varios campos.
    - Incrementar el tamaño del chunk también puede ayudar a eliminar los jumbo chunks.
    - Los jumbo chunks no pueden ser migrados.
    - Los jumbo chunks incrementan su tamaño más allá del tamaño mínimo del chunk.
- Un chunk puede ser partido (comentado anteriormente) o migrado (sucede cuando el sistema entiende que es necesario balancear los datos de un shard a otro).
- Dependiendo de en qué shard está manteniendo un determinado chunk, un nuevo documento es enrutado a el shard específico y solamente a ese. Mientas los datos están siendo migrados, sigue siendo accesibles por la aplicación.
- Es necesario por tanto que el shard key esté presente en todo documento que vayan a insertarse.
- En las operaciones de lectura tambiéns se redirige la operación hacia el shard concrecto considerando el valor del shard key (routed queries). Si en la consulta no está la shard key, entonces mongos tiene que redireccionar la consulta a todos los shards (scatter gatter).
- Los shards keys son inmutables. No es posible cambiar los campos de shard key.

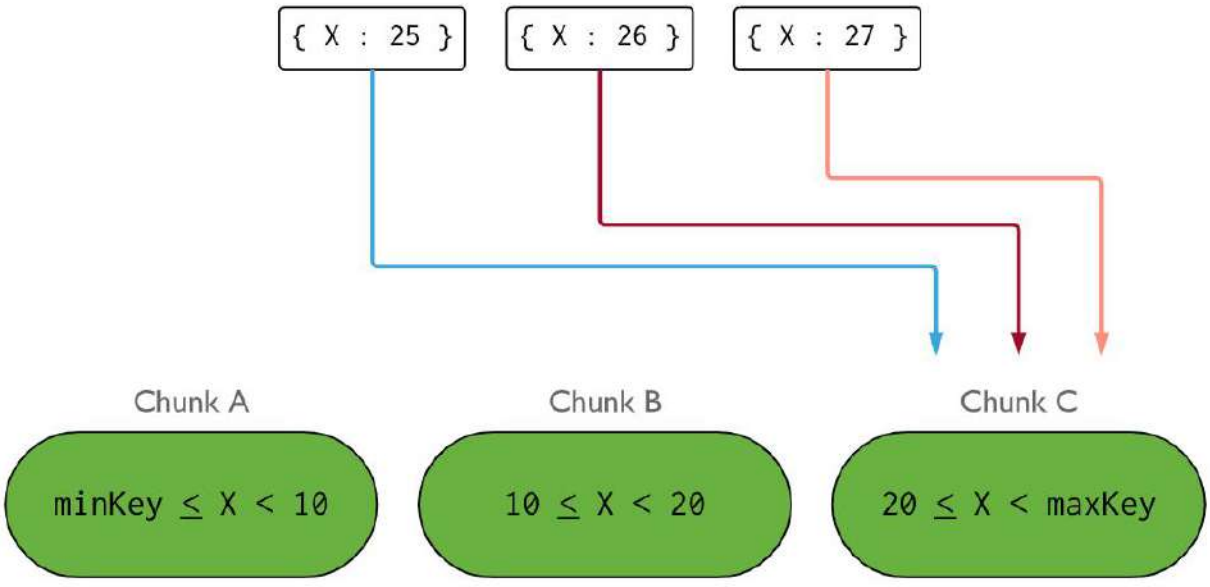
- Cuando se realiza un update, también hay que enviar la shard key. Cualquier actualización que no contenga el shard key o el campo _id, arrojará un error.
- En los sharded cluster, si no se usa el campo _id como shard key, la aplicación debe garantizar la unicidad de los valores en el campo _id para evitar errores. Esto se hace generalmente utilizando un ObjectId estándar generado automáticamente.
- La elección del shard key es importante. El objetivo es garantizar una buena distribución de escrituras. Hay que tener en cuenta: cardinalidad, frecuencia de escritura y cambios monotónicos.
    - Cardinalidad: número de elementos dentro de un conjunto de valores. Es el número de posibles valores únicos de shard keys. Tener una alta cardinalidad proporcionará más chunks y con más chunks, el número de shards también puede crecer.
        - La cardinalidad puede aumentar con el uso de shard keys compuestos. Por tanto, es recomendable una alta cardinalidad (muchos posibles valores únicos.
        - Si una shard key tiene una cardinalidad de 2 (por ejemplo, cuando se toma como shard key un campo que es de tipo booleano), entonces no puede haber más de 2 chunks dentro del grupo fragmentado, cada uno de los cuales almacena un único valor de shard key.




    - Frecuencia de escritura: representa cuán a menudo un valor unico ocurre en los datos. Escribiendo el mismo valor, el documento pertenecerá al mismo chunk y al mismo shard, lo cual es una mala práctica. Es conveniente una baja frecuencia (muy poca repetición de valores únicos).




- Cambios monotónicos: los cambios monotónicos significan que los posibles valores del shard key para un nuevo documento cambian de una forma lineal y predecible.
  - Es preferible que los nuevos documentos contengan valores del shard key que caigan dentro de los límites de los chunks. Un campo de tipo fecha o incremental (como el _id) no es un buen campo para utilizar como shard key porque todos van destinados al último chunk.
  - Cambios de los valores incrementales o decrementales en progresiones numéricas provocan hotspotting.
  - Por tanto, es recomendable cambios no monotónicos (cambios no lineales en los valores). Si el modelo de datos requiere fragmentación en una clave que cambia monótonamente, es conveniente usar la fragmentación hash.




- También es conveniente utilizar un shard key que va a ser utilizado siempre que sea posible en las consultas que realice la aplicación. De esta forma se garantizará el aislamiento de la lectura hacia un único shard (read isolation).

- En una colección fragmentada no puede haber campos únicos, a no ser que pertenezcan al shard key. Esto es debido a que no hay manera de verificar que un campo sea único en los distintos shards.
- Para una colección fragmentada, solo el índice del campo _id y el índice en la shard key o un índice compuesto donde la shard key es un prefijo, puede ser único:
    - No puede fragmentarse una colección que tenga índices únicos en otros campos.
    - No pueden crear índices únicos en otros campos para una colección fragmentada.

# Índices únicos

- La restricción única en los índices garantiza que solo un documento pueda tener un valor para un campo en una colección. Para las colecciones fragmentadas, estos índices únicos no pueden imponer la singularidad porque las operaciones de inserción e indexación son locales para cada shard.
- No se puede especificar una restricción única en un índice hasheado.
- Para una colección fragmentada con rangos, solo los siguientes índices pueden ser únicos:
    - El índice del shard key.
    - Un índice compuesto donde el shard key es un prefijo.
    - El índice _id por defecto: sin embargo, el índice _id solo impone la restricción de unicidad por shard si el campo _id es la shard key o el prefijo de la shard key.
        - Si el campo _id no es la shard key o el prefijo de la shard key, el índice _id solo impone la restricción de unicidad por shard y no entre los shards (aunque ciertamente es muy difícil que exista dos _id iguales en shards distintos).
        - Si el campo _id no es la shard key, ni el prefijo de la shard key, MongoDB espera que las aplicaciones impongan la unicidad de los valores del _id en los shards.
- Las restricciones de índice únicas significan que:
    - Una colección no puede fragmentarse si posee índices únicos.
    - Una colección ya fragmentada no puede crear índices únicos en otros campos distintos de la shard key.
- A través del uso de un índice único en un shard key, MongoDB puede forzar la unicidad de los valores del shard key.
    - MongoDB fuerza la unicidad en la combinación entera de la clave y no en componentes individuales de la shard key. Para forzar la unicidad en los valores de la shard key se suministra el parámetro unique al método sh.shardCollection().
- Un valor de una shard key única no puede existir en un solo shard en un momento dado. Si una shard key tiene una cardinalidad de 4, entonces no puede haber más de 4 chunks, cada uno de los cuales almacena un único valor del shard key.
- Si se necesita asegurar que un campo siempre sea único en una colección fragmentada existen tres posibles alternativas.

## Primera opción

- MongoDB puede imponer la unicidad de la shard key (ya comentado anteriormente). Para shard keys compuestas, MongoDB aplicará la exclusividad en toda la combinación de la clave, y no para un componente específico de la shard key.

```
db.runCommand( { shardCollection : "test.users" , key : { email : 1 } , unique :
true } );
```

- Limitaciones:
  - Solo se puede imponer la unicidad en un solo campo de la colección usando este método.
  - Si se utiliza una shard key compuesta, solo puede imponerse unicidad en la combinación de claves de componentes en la shard key.

### Segunda opción

- Utilizar una colección secundaria para imponer la singularidad.
  - Crear una colección mínima que solo contenga el campo único y una referencia a un documento en la colección principal. Si siempre se inserta en la colección secundaria antes de insertarla en la colección fragmentada, MongoDB producirá un error si se intenta usar una clave duplicada.
- Esta colección debe contener tanto una referencia al documento original (es decir, el ObjectId) como la clave única.

```
// otra forma de cambiar de base de datos (en este caso, a records)
db = db.getSiblingDB('records');

const primary_id = ObjectId();

db.proxy.insert({
   "_id" : primary_id
   "email" : "example@example.net"
})

// si la operación ha sido exitosa, se continúa insertando el documento en la otra
colección
db.information.insert({
   "_id" : primary_id
   "email": "example@example.net"
   // información adicional
})
```

### Tercera opción

- Utilizar identificadores únicos garantizados (es decir, UUID) como ObjectId únicos.

## Fragmentar una colección

```
use m103
show collections
```

- Habilitar fragmentación en una base de datos (hay que conectarse a mongos).

```
sh.enableSharding("m103")
```

- Buscar un documento de la colección a fragmentar para tener como referencia la estructura que permite elegir una correcta shard key.

```
db.products.findOne()
```

- Crear un índice sobre el campo elegido.

```
db.products.createIndex( { "sku" : 1 } )
```

- Fragmentar una colección a partir de un namespace y una shard key.

```
sh.shardCollection("m103.products", {"sku" : 1 } )
sh.status()
```

## Eliminar o actualizar un shard

- Eliminar un shard puede llegar a ser una operación muy lenta porque el balanceador debe equilibrar los chunks entre el resto de shards.
    - Hay que utilizar primero el comando removeShard. En este punto, el shard comenzará a ser "drenado" (draining) y ninguna nueva operación de escritura será redireccionada a este shard.
    - Posteriormente hay que esperar a que el balanceador termine de equilibrar los chunks.
    - Seguidamente, mover las bases de datos no fragmentadas (utilizando el comando movePrimary) que consideran al shard a borrar como primario. Puede utilizarse el comando db.databases.find() para saber cuáles son las bases de datos que consideran al shard a borrar como primario.
    - Finalmente hay que ejecutar el comando removeShard de nuevo.

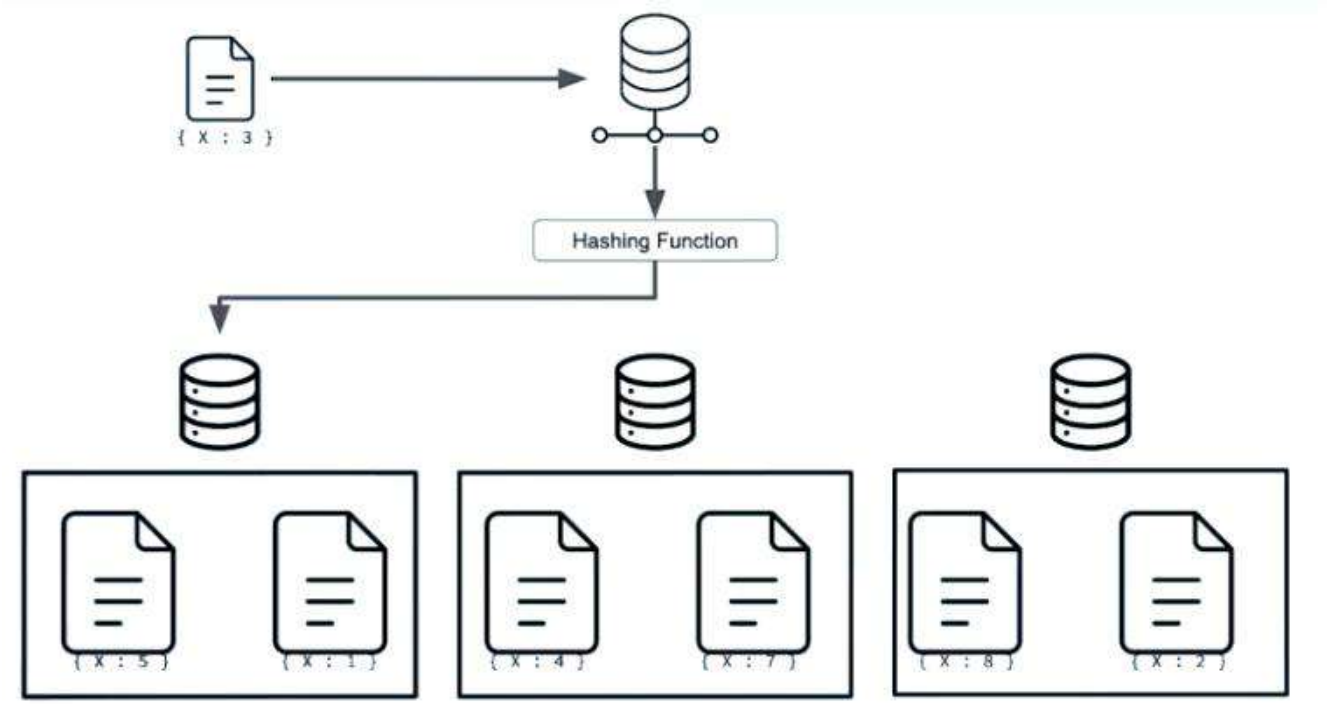
```
sh.startBalancer()

// draining started successfully (avisa también de que hay bases de datos que son
necesario borrar o mover con movePrimary)
db.adminCommand({removeShard: "test-rs1"});

// draining ongoing (y muestra el número de chunks que falta para migrar y también
```

```
el número de dbs que manualmente hay que borrar o migrar con movePrimary)
db.adminCommand({removeShard: "test-rs1"});

// cuando dbs y chunks sean 0, entonces se habrá eliminado correctamente el shard.

// una vez finalizada la operación de draining, si el shard primario es el
borrado, entonces hay que migrar las bases de datos que indicado comando anterior
removeShard a otro shard (movePrimary no retorna hasta que la migración se ha
producido satisfactoriamente)
db.adminCommand( { movePrimary: "fizz", to: "mongodb1" })

// para finalizar la migración, es importante volver a ejecutar removeShard
db.adminCommand({removeShard: "test-rs1"});
```

- En cuanto a la actualización de las instancias mongod a versiones superiores de MongoDB:
  - Es recomendable comenzar a actualizar los secundarios de los replica set en primer lugar.
  - A continuación, actualizar los primarios de los replica set (ejecutando stepDown antes de iniciar la actualización).
  - La actualización de los miembros secundarios puede hacerse de forma paralela, pero la de los miembros primarios es conveniente que no se realice de esta manera.
  - Después de actualizar los miembros primarios y secundarios de los replica set, se actualizan los config servers. Sin embargo, es posible perfectamente actualizar los config server antes que los miembros primarios y secundarios de los replica set.
  - Finalmente se actualiza la instancia o instancias de mongos.

## Hashed shard keys

- Se trata de una shard key donde el índice es hasheado, es decir los datos son distribuidos no por el valor del shard key, sino por el hash producto del valor.



- Las hashed shard keys pueden ser útiles para proveer una mejor distribución de shard keys con cambios monotónicos, produciendo valores más impredecibles que se distribuyan en diferentes chunks y por tanto, en diferentes shards.
- Sin embargo, los hashed shard key pueden ser ineficientes cuando se realizan consultas con rangos de ese campo debido a que los valores se encuentran distribuidos prácticamente de forma aleatoria.
- No es posible soportar operaciones de lecturas aisladas geográficamente utilizando sharding zoned.
- El shard key debe ser un campo único de tipo no array.
- Los índices hasheados no soportan ordenamiento rápido.

## Fragmentar una colección con hashed shard keys

```
use m103
show collections
```

- Habilitar fragmentación en una base de datos.

```
sh.enableSharding("m103")
```

- Buscar un documento de la colección a fragmentar para tener como referencia la estructura que permite elegir una correcta shard key.

```
db.products.findOne()
```

- Crear un índice sobre el campo elegido, indicando en este caso que se trata de un índice de tipo hashed.

```
db.products.createIndex( { "sku" : "hashed" } )
```

- Fragmentar una colección a partir de una hashed shard key.

```
sh.shardCollection("m103.products", {"sku" : "hashed" } )
sh.status()
```

## Balanceo

- Los datos deberían estar distribuidos de forma homogénea por los shards.

- MongoDB balancea los chunks a medida que se insertan documentos, para evitar que un shard posea muchos chunks y trata de buscar la distribución homogénea.
- El balanceador de carga se encuentra en el miembro primario del config server replica set. Si el proceso del balanceador detecta que hay un imbalanceo, comienza una ronda de balanceo. El balanceador puede migrar chunks en paralelo.
- Un determinado shard no puede participar en más de una migración al mismo tiempo, así que para calcular el número máximo de shards que pueden participar en una migración al mismo tiempo puede tomarse floor(n/2), donde n es el número de shards.
    - Por tanto, a veces será necesario varias rondas para poder homoegeneizar los datos. Con tres shards, solamente será posible la migración de un chunk al mismo tiempo en una ronda. Con cuatro shards, será posible dos migraciones de chunks en una ronda.
- Puede lanzarse el balanceador de forma manual mediante el comando startBalancer (solamente se puede ejecutar desde mongos). Este proceso no inmediato y puede tardar unos segundos. Posee los siguientes parámetros:
    - timeout (opcional): límite de tiempo (en milisegundos) para habilitar el equilibrador. Su valor por defecto es 60000 (60 segundos).
    - interval (opcional): la frecuencia (en milisegundos) en la que se comprueba si se ha iniciado una ronda de balanceo.

```
sh.startBalancer(timeout, interval)
```

- Para parar el balanceador se usa el método stopBalancer (solamente se puede ejecutar desde mongos). Si se detiene el balanceador en medio de una ronda, el balanceador únicamente para después después de finalizar la ronda. Posee los siguientes parámetros:
    - timeout (opcional): límite de tiempo (en milisegundos) para desactivar el equilibrador. Su valor por defecto es 60000 (60 segundos).
    - interval (opcional): la frecuencia (en milisegundos) en la que comprueba si la ronda de equilibrio se ha detenido.

```
sh.stopBalancer(timeout, interval)
```

- También se puede habilitar o deshabilitar el balanceador para una determinada colección fragmentada.

```
sh.disableBalancing(<namespace>)
```

- O habilitarse o deshabilitase el balanceador a nivel global.

```
sh.setBalancerState(false)
sh.getBalancerState()
```

```
sh.setBalancerState(true)
```

- El balanceador realmente es un proceso que corre en el background y monitoriza el número de chunks en cada shards. El balanceador solo comienza a mover chunks después de que la distribución de los chunks para una colección fragmentada haya alcanzado ciertos umbrales. Los umbrales se aplican a la diferencia en la cantidad de chunks entre el shard con más chunks para la colección y el shard con el menor número de chunks para esa colección. El balanceador posee los siguientes umbrales:

| Número de chunks | Umbral de migración |
|---|---|
| Menos de 20 | 2 |
| 20-79 | 4 |
| 80 o más | 8 |

- Añadir un shard a un cluster crea un imbalanceo, ya que el nuevo shard no tiene chunks. Mientras que MongoDB comienza a migrar los datos al nuevo shard, puede pasar un tiempo antes de que el cluster se encuentre balanceado. Eliminar un determinado shard de un cluster crea un similar imbalanceo, ya que los chunks residiendo en el shard deben ser redistribuidos por el cluster.
- MongoDB no puede mover un chunk si el número de documentos en el chunk es mayor que 1.3 veces el resultado de dividir el tamaño del chunk configurado (64 megabytes por defecto) por el tamaño promedio del documento.
    - db.collection.stats() incluye el campo avgObjSize, que representa el tamaño promedio de documento en una colección.

## Config Servers y Cluster Metadata

- El número de config server en un sharded cluster es siempre un número impar (3,5,7,...) y mínimo 3. Y formando un replica set.
- Los config servers almacenan los metadatos de un sharded cluster. Los metadatos reflejan el estado y la organización de todos los datos y componentes dentro del sharded cluster. Los metadatos incluyen la lista de chunks en cada shard y los rangos que los definen.
- Las instancias de mongos almacenan en caché estos datos y los utilizan para enrutar las operaciones de lectura y escritura a los shards correctos. mongos actualiza la caché cuando hay cambios de metadatos para el cluster, como divisiones de chunks o agregaciones de un shard. Los shards también leen metadatos de chunks de los config servers
- Los config servers también almacenan información de configuración de autenticación, como el control de acceso basado en roles o la configuración de autenticación interna del cluster.
- Cada sharded cluster debe tener sus propios config servers.
- Las siguientes restricciones se aplican a la configuración de un replica set cuando se usa para servidores de configuración:
    - Debe tener cero árbitros.

    - No debe tener miembros retrasados.
    - Debe generar índices (es decir, ningún miembro debe tener la configuración buildIndexes establecida en falso).

- Escrituras a config servers:

    - La base de datos admin contiene las colecciones relacionadas con la autenticación y autorización, así como otras colecciones system.* para uso interno.
    - La base de datos config contiene las colecciones de los metadatos del cluster fragmentado. MongoDB escribe datos en la base de datos de configuración cuando cambian los metadatos, como después de una migración de chunks o una división de chunks.
    - Los usuarios deben evitar escribir directamente en la base de datos config en el curso de una operación o mantenimiento normal.
    - Al escribir en los config servers, MongoDB utiliza un write concern de "majority".

- Lecturas de config servers:

    - MongoDB lee desde la base de datos admin los datos de autenticación y autorización y otros usos internos.
    - MongoDB lee desde la base de datos config cuando se inicia un mongos o después de un cambio en los metadatos, como después de una migración de chunks. Los shards también leen metadatos de chunks de los servidores de configuración.
    - Al leer en los config servers, MongoDB utiliza un read concern de "majority".

- Si el replica set de los config servers pierde su nodo primario y no puede elegir a otro, los metadatos del cluster se vuelven de solo lectura.

    - Aún pueden leerse y escribir datos de los shards, pero no se realizarán migraciones de chunks o divisiones de chunks hasta que el replica set del config server pueda elegir un nodo primario.

- Si todos los config servers dejan de estar disponibles, el cluster puede dejar de funcionar. Para garantizar que los config server permanezcan disponibles e intactos, las copias de seguridad de los config servers son fundamentales.

    - Los datos en los config servers son pequeños en comparación con los datos almacenados en un cluster, y los config servers tiene una carga de actividad relativamente baja.

- Para los shards y el mongos, si el número de intentos fallidos consecutivos de monitorear los replica set de los configservers (o el resto de shards) excede el valor del parámetro replMonitorMaxFailedChecks, la instancia de mongos o mongod se vuelve inutilizable hasta que reinicie la instancia.

- Para acceder a la base de datos config se puede utilizar el comando use desde la shell.

```
use config
```

- Las colecciones disponibles son las siguientes: changelog, chunks, collections, databases, lockpings, locks, mongos, settings, shards, version

## Write concern en sharding

- Los parámetros w y j también tienen sentido en sharding, pero esta vez no se refieren a miembros de un replica set, sino a shards completos.
    - Las instancias de mongos pasan el valor de write concern a los shards.

## Operaciones bulk

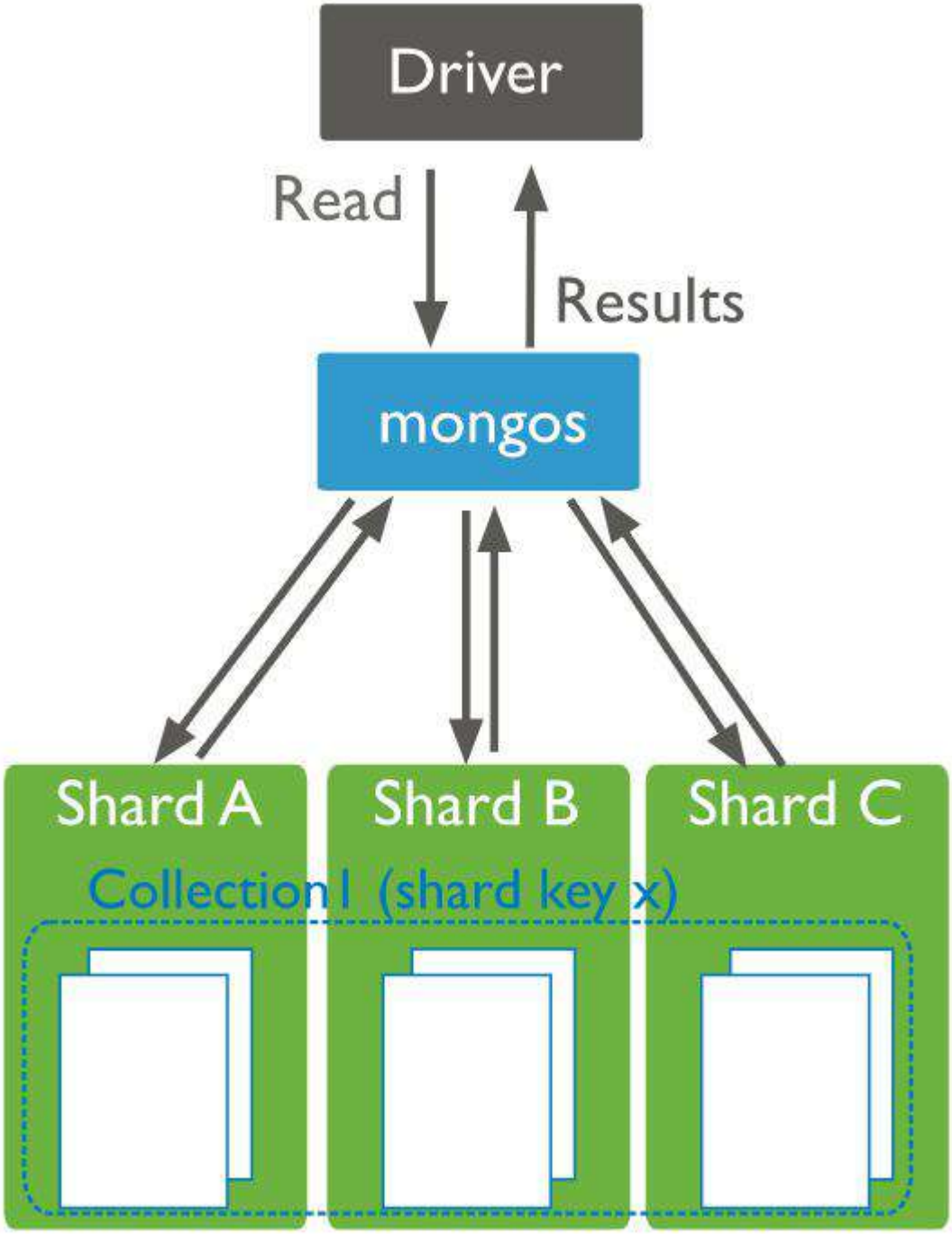
- La latencia puede aumentar considerablemente cuando se realizan operaciones bulk ordenadas porque hay que esperar la respuesta de los distintos shards. En operaciones bulk desordenadas no existen ese problemas porque se ejecutan paralelamente.

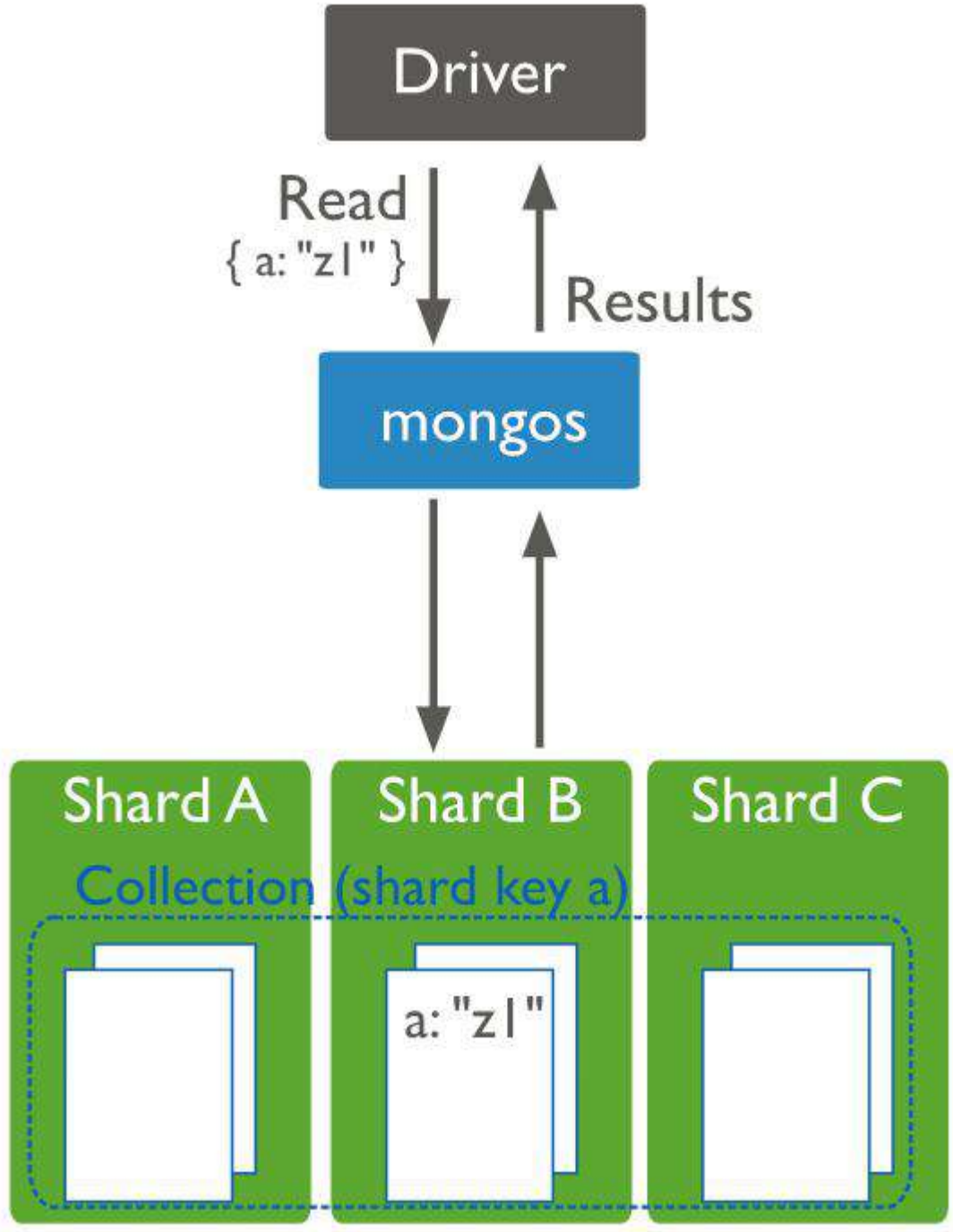
## Routed Queries vs Scatter Gather

- En el contexto de un replica set, existe una tabla de chunks. Mongos mantiene una copia local de esa tabla, que se encuentra en los config servers.

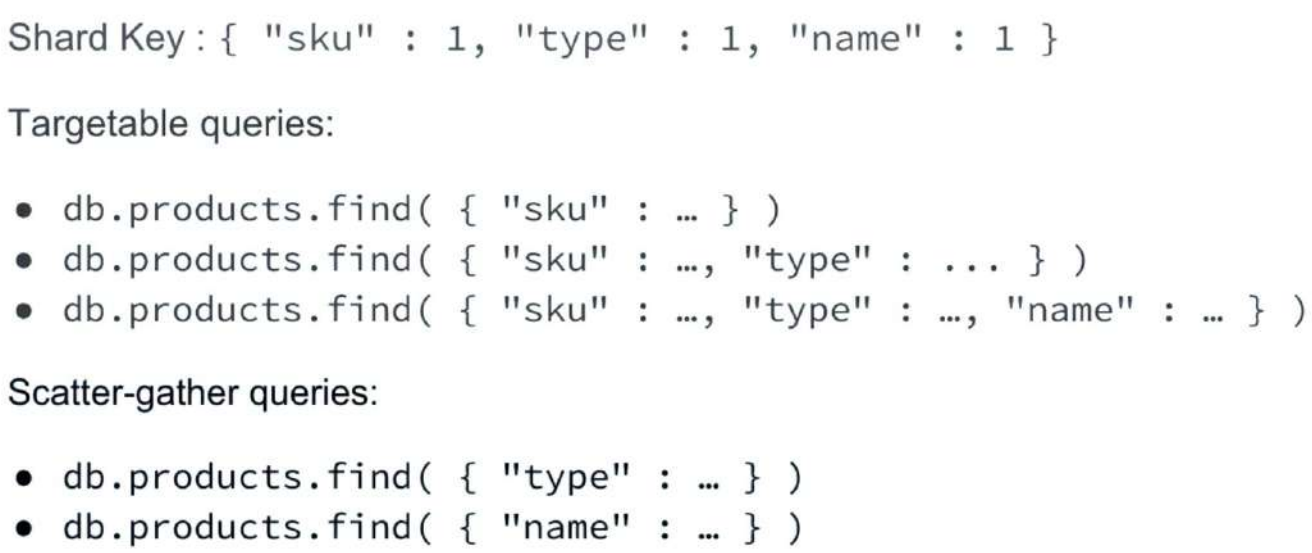
| Shard | Data |
|---|---|
| 1 | *minKey -> 10000000* |
| 2 | *10000000 -> 20000000* |
| 3 | *20000000 -> maxKey* |

- minkey representa el valor más bajo del shard key, mientras que maxKey es el valor máximo del shard key. No es necesario definir los valores para estos dos parámetros.
- Las consultas de rango sobre la shard key, habitualmente requerirá que mongos haga peticiones a varios shards porque podría ser necesario devolver documentos de dos chunks distintos ubicados en diferentes shards, dependiendo del tamaño del rango.


Driver
Read
Results
mongos
Shard A
Shard B
Shard C
Collection1 (shard key x)




- En algunos casos, cuando la shard key o un prefijo de la shard key es parte de la consulta, mongos realiza una operación específica, enrutando las consultas a un subconjunto de shards en el cluster.
- Utilizando un índice compuesto como shard key, todavía sería posible realizar consultas dirigidas a un único shard. Basta con conservar el prefijo en la query.




- Para comprobar si la consulta es dirigida o no, puede utilizarse el método explain.
  - SINGLE_SHARD: obtiene el conjunto de resultados desde un único shard sin necesidad de unir los resultados.
  - SHARD_MERGE: obtiene el conjunto de resultados de varios shard y une los resultados.
  - SHARDING_FILTER: mongos compara la shard key del documento con los metadatos en los servidores de configuración.

```
db.products.find({"sku" : 1000000749 }).explain()
```

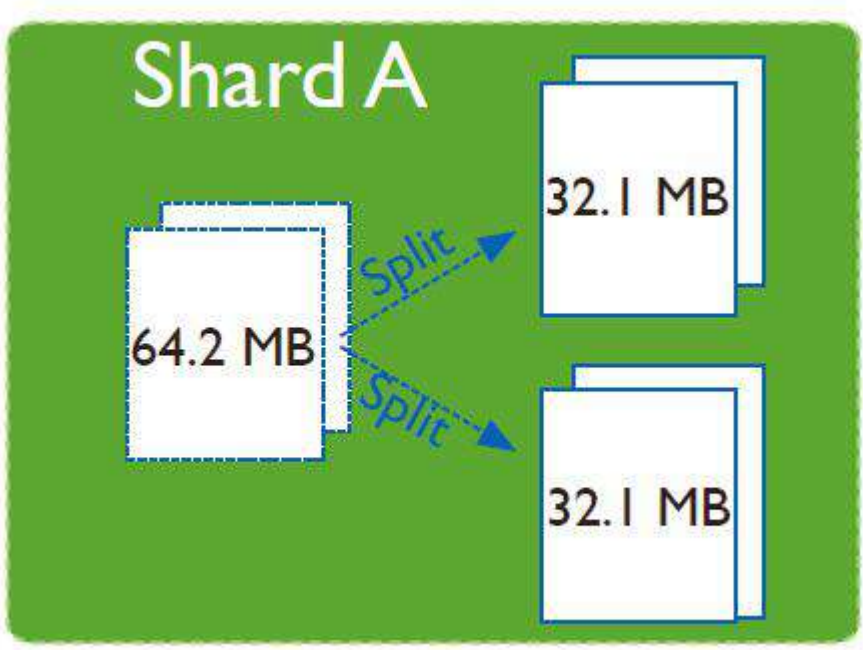
- Los métodos updateMany() y deleteMany() son operaciones de difusión siempre, a menos que el documento de consulta especifique la shard key en su totalidad.
- Cada shard, a su vez, utilizará el índice del shard key u otro índice más eficiente para cumplir la consulta.

## Creando/diviendo chunks

- En la mayoría de las situaciones, un sharded cluster creará/dividirá y distribuirá los chunks automáticamente sin la intervención del usuario. Sin embargo, en un número limitado de casos, MongoDB no puede crear suficientes chunks o distribuir datos lo suficientemente rápido como para soportar la carga requerido.
- Por ejemplo, si se desea insertar un gran volumen de datos en un clúster que no está balanceado, o si la inserción de datos masiva provoca un desequilibrio (como en el caso de un aumento monótono de los datos).
- Diviendo los chunks de una colección fragmentada vaciada puede ayudar con el rendimiento en estos casos.
- Solo se debe dividir chunks en una colección vacía. La división manual de chunks para una colección poblada puede llevar a rangos y tamaños de chunks impredecibles, así como a un comportamiento de equilibrio ineficiente o ineficaz.

- Para dividir manualmente chunks vacíos, debe ejecutarse el comando split.

```
// crear chunks de documentos en la colección myapp.users utilizando el campo de
correo electrónico como shard key
for ( var x=97; x<97+26; x++ ){
  for ( var y=97; y<97+26; y+=6 ) {
    var prefix = String.fromCharCode(x) + String.fromCharCode(y);
    // prefix = aa, ag, am, ... zs y zy
    db.adminCommand( { split: "myapp.users", middle: { email : prefix } } );
  }
}
```




- Normalmente, MongoDB divide un chunk después de una inserción si el chunk excede el tamaño máximo del chunk. Sin embargo es posible dividir el chunk manualmente si:
    - Hay una gran cantidad de datos en el cluster y muy pocos chunks, como es el caso después de implementar un cluster utilizando datos existentes.
    - Se espera añadir una gran cantidad de datos que inicialmente residirían en un único chunk o shard. Por ejemplo, insertar una gran cantidad de datos con un valores de shard key entre 300 y 400, pero todos los valores de los shard keys entre 250 y 500 están en un único chunk.
- MongoDB proporciona el comando mergeChunks para combinar rangos de chunks contiguos en un solo shard para convertirlo en un chunk único.

```
// los bounds son los límites inferior y superior para el nuevo chunk creado
db.adminCommand( { mergeChunks : <namespace>,
  bounds : [
    { <shardKeyField>: <minFieldValue> },
```

```
    { <shardKeyField>: <maxFieldValue> }
  ] } )
```

- El balanceador puede migrar chunks divididos recientemente a un nuevo shard inmediatamente si el movimiento beneficia a futuras inserciones. El balanceador no distingue entre divisiones de chunks manuales y aquellas que realizadas automáticamente por el sistema.
- Puede determinarse los actuales rangos de los chunks en el cluster ejecuando el comando sh.status().
- Para dividir chunks manualmente hay que utilizar el comando split, ya sea con los campos middle o find. El mongo shell provee el método sh.splitFind() y sh.splitAt()
- El método splitFind() divide el chunk que contiene el primer documento devuelto que coincide con esta consulta en dos partes del mismo tamaño. Debe especificase el namespace de la colección fragmentada. La consulta splitFind no necesita utilizar la shard key, aunque casi siempre tiene sentido hacerlo.

```
// divide el chunk que contiene el documento con zipcode=63103een la colección
people de la base de datos record
sh.splitFind( "records.people", { "zipcode": "63109" } )
```

- El método splitAt() para dividir un chunk en dos, utilizando el documento consultado como el documento límite inferior del nuevo chunk.

```
// divide el chunk que contiene el valor de 63109 del campo zicode en la colección
people de la base de datos record
sh.splitAt( "records.people", { "zipcode": "63109" } )
```

- El método splitAt() no necesariamente divide el chunk en dos chunk exactamente iguales. La división ocurre en la ubicación de coincidencia de la consulta, independientemente de dónde esté ese documento en el chunk.

## Leer en un cluster (replicación o sharding)

- Leer desde del miembro primario implica que se está leyendo la última actualización de los datos (strong consistency). Leyendo en un miembro secundario, esto no está garantizado (eventual consistency), aunque ciertamente no existe problema si el miembro primario cae porque las lecturas siguen siendo posible a través de los miembros secundarios en un replica set.
- Cuándo leer de miembros secundarios es buena idea:
    - Offloading work: cargas intensivas como analíticas, donde la demora de la respuesta no es crítica. No es necesario tampoco que recojan la última información actualizada.
    - Local reads: cuando el miembro secundario está geográficamente cerca del cliente y se requiere muy baja latencia.

- Cuándo leer de miembros segundarios no es buena idea:
    - En general.
    - Cuando el miembro primario está sobrecargado de lecturas y no importa que los clientes recojan datos obsoletos, puede leerse directamente sobre los miembros secundarios.
    - Leer desde un shard directamente es una muy mala práctica (tanto primario como secundario, aunque secundario es aún peor). Debe hacerse desde un mongos
- Al utilizar findAndModify en un entorno fragmentado, la consulta debe contener la shard key para todas las operaciones contra el sharded cluster para las colecciones fragmentadas.
- No se puede utilizar un índice geoespacial como shard key al fragmentar una colección. Sin embargo, puede crearse y mantenerse un índice geoespacial en una colección fragmentada si se usa campos distintos al shard key.
- Las siguientes operaciones geoespaciales son soportadas:
    - $geoNear como etapa de agregación.
    - Operadores de consulta $near y $nearSphere (comenzando en MongoDB 4.0).
    - Comando geoNear (en desuso en MongoDB 4.0).
- En versiones tempranas de MongoDB, $near y $nearSphere no son soportadas por colecciones fragmentadas; en lugar de ello, se utilizaba $geoNear y el obsoleto comando geoNear.
- También se puede utilizar $geoWithin y $geoIntersect para consultas en datos geespaciales para un sharded cluster.

## Desarrollo

- Operaciones multi-actualización son siempre operaciones broadcast.
- Los métodos updateMany() y deleteMany() son operaciones broadcast, a menos que el documento de consulta especifique la shard key en su totalidad.
- Todas las operaciones insertOne() son dirigidas a un shard .Cada documento en en el array contenido en insertMany() es dirigido a un single shard, pero no garantiza que todos los documentos en el array sean insertado en un único shard.
- Las operaciones updateOne(), replaceOne() y deleteOne() deben incluir la shard key o el _id en el documento de consulta. MongoDB retorna un error si esos métodos son usados sin el shard key o el _id.
- Una instancia de mongos enruta una consulta a un cluster:
    - Determinando la lista de shards que deben recibir la consulta.
    - Estableciendo un cursor a todos shards destino.
    - Después, mongos une los datos de cada shard y retorna los resultados. Algunas modificadores de consulta, tales como el sorting son ejecutados en un shard como el primario antes de que mongos retorne los resultados.

- Para las operaciones de agregación que se ejecutan en múltiples shards, si las operaciones no requieren ejecutarse en el shard primario, estas operaciones pueden enrutar los resultados de nuevo al mongos donde los resultados son unidos.

- Hay dos casos en los que un no es pipeline no es apto para ejecutarse en mongo:

    - El primer caso ocurre cuando la parte de fusión del pipeline dividido contiene una etapa que debe ejecutarse en un shard primario. Por ejemplo, si $lookup requiere acceso a una colección no fragmentada en la misma base de datos que la colección fragmentada en la que se ejecuta la agregación, la combinación está obligada a ejecutarse en el shard primario.
    - El segundo caso ocurre cuando la parte de la combinación de la tubería dividida contiene una etapa que puede escribir datos temporales en el disco, como $group, y el cliente ha especificado allowDiskUse: true. En este caso, suponiendo que no haya otras etapas en la pipeline de fusión que requieran el shard primario, la fusión se ejecutará en un shard seleccionado al azar en el conjunto de shard a los que apunta la agregación.

- Hay implicaciones con respecto a las operaciones sort, skip y limit en un cluster con shards. En primer lugar, estas operaciones son realizadas localmente en cada shard y finalmente son unidas en el primary shard.

- mongos también tiene específicos comportamientos para sort, limit y skip.

    - sort: si el resultado de la consulta no está ordenado, la instancia de mongos abre un cursor de resultados que "redondea" los resultados de todos los cursores en los shards.

    - limit: si la consulta limita el tamaño del conjunto de resultados utilizando el método de cursor limit, la instancia de mongos pasa ese límite a los shards y luego vuelve a aplicar el límite al resultado antes de devolver el resultado al cliente.

    - skip: si la consulta especifica un número de registros para omitir usando el método de cursor skip, mongos no pueden pasar el salto a los shards, sino que recuperan los resultados no omitidos de los shards y omiten el número apropiado de documentos cuando se ensambla el resultado completo. Cuando se usa junto con un limit, mongos pasará el límite más el valor de skip a los shards para mejorar la eficiencia de estas operaciones.

- Si el pipeline comienza con un $match en una shard key, el pipeline completo se ejecuta solo en el shard correspondiente.

- Para las operaciones de agregación que deben ejecutarse en múltiples shards, si las operaciones no requieren que se ejecuten en el shard primario de la base de datos, estas operaciones direccionarán los resultados a un shard aleatorio para fusionar los resultados y evitar la sobrecarga del shard primario de esa base de datos. La etapas $out, $lookup y $facet requieren ejecutarse en el shard primario de la base de datos.

- Un merge en un despliegue fragmentado hará que todas las etapas subsiguientes del pipeline se realicen en la misma ubicación que la fusión

- El aggregation framework también es manejado por mongos de forma particular dependiendo del stage.

- A veces será necesario realizar un merge en mongos y otras veces no. Depende en gran medida de si es necesario recoger datos de múltiples shards. mongos determinará qué fases se ejecutarán en cada shard y qué fases necesitan ser ejecutadas en un único shard donde los resultados de otros shards serán unidos. La unión de los datos se realiza generalmente en un shard aleatorio, pero hay algunos casos en donde esto no se produce: $out, $facet, $lookup y $graphLookup, donde el primary shard se encargará de ello.

- Para las operaciones que se ejecutan en múltiples shards, si las operaciones no requieren que se ejecuten en el shard primari, estas operaciones pueden enviar los resultados de regreso a los mongos donde se fusionan los resultados.

- La devolución de explain en una agregación incluirá tres objetos json. mergeType muestra dónde ocurre la etapa de la fusión ("primaryShard", "anyShard" o "mongos").