Big Data Aplicado

Big_Data
Aplicado

Curso Especialización
Inteligencia_Artificial y
Big_Data

REPLICA SET

Alta disponibilidad

- Cualquier sistema de base de datos que se precie, debe garantizar al 100 % que sus datos estén disponibles en todo momento.
- Es lo que se conoce como alta disponibilidad y para conseguirlo MongoDB utiliza la replicación.

¿Qué es REPLICA SET?

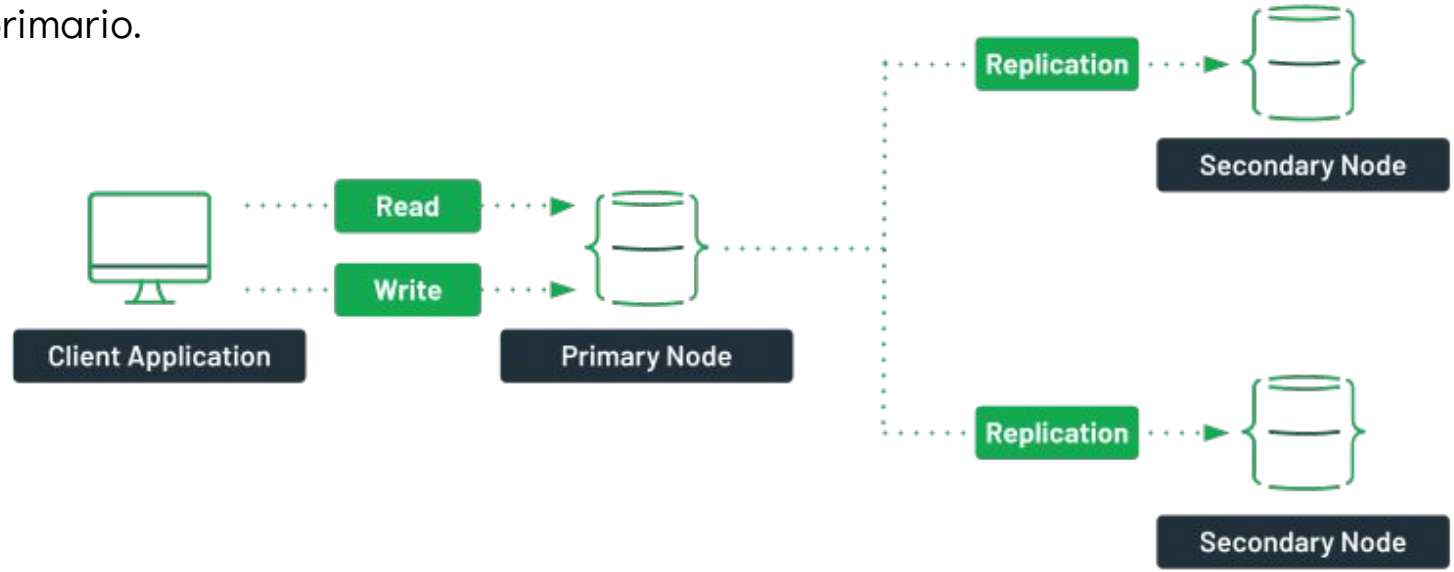
- Es un grupo de instancias mongod configuradas para replicar el contenido de determinadas bases de datos
- Provee de redundancia y disponibilidad a la base de datos
- Es conveniente que las instancias de mongo estén en la misma versión para evitar problemas.

Referencia de Replica Set en MongoDB

Referencia de métodos de Replica Set

¿Qué es REPLICA SET?

Es un grupo de instancias mongod configuradas para replicar el contenido de determinadas bases de datos según las operaciones que va recibiendo un servidor primario.



¿Cómo funciona REPLICA SET?

- Un set de réplicas sólo puede tener 50 miembros en total
- MongoDB funciona con **un solo servidor primario** , y varios secundarios.
- Los nodos secundarios que replican las operaciones que realiza el nodo primario cuando este las anota en el log de operaciones de mongo, conocido como **oplog** .
- Para sincronizar los datos entre servidores de MongoDB, se comunican a través de pings o latidos, informando de su estado y de la versión de sus datos. Así los otros servidores son capaces de saber si hay que elegir un nuevo servidor principal, o si sus datos están actualizados.

OPERACIONES DE ESCRITURA Y LECTURA

- **OPERACIONES DE ESCRITURA** (updates o inserts) → SOLO EN PRIMARIO
- **OPERACIONES DE LECTURA** . Por defecto en el primario, pero puede cambiarse:
  - **Primary** : siempre se lee del nodo primario
  - **primaryPreferred** : si está disponible, las lecturas se realizan en el nodo primario, pero si no lo estuviera, habilita las lecturas de los secundarios.
  - **Secondary** : siempre se lee del nodo secundario.
  - **secondaryPreferred** : si está disponible, las lecturas se realizan en el nodo secundario, pero si no lo estuviera, habilita las lecturas del primario.
  - **Nearest** : el nodo que menor latencia tenga
- Cuando leemos de los nodos secundarios corremos el riesgo de que los datos no estén actualizados después de una escritura en el nodo primario.

¿Cómo funciona REPLICA SET?

- Si un nodo primario cae o se pierde la comunicación, durante 10 segundos, se le considera caído y se tiene que hacer una votación para decidir qué nodo pasa a ser el nuevo nodo primario.
- Mientras no tenemos un nodo primario el cluster se quedará en modo de solo lectura, por lo que no se podrá escribir nada.
- El proceso de caída y reelección dura aproximadamente 1 minuto:
  - De 10 a 30 segundos de margen para determinar la caída del nodo primario e iniciar una nueva elección
  - De 10 a 30 segundos para la elección de un nuevo nodo primario mediante el sistema de votaciones.

TIPOS DE SERVIDORES

- Tenemos distintos tipos de servidores y/o componentes.
- Los nodos tienen distintas características::
  - **VALOR DE SU VOTO** : Es lo que vale su voto para elegir un nuevo nodo primario. Lo normal es que valga 1 (votes=1). Si vale 0 no podrán votar (votes=0).
  - **PRIORIDAD PARA SER ELEGIDO** : Cuando vale 0 (priority=0) quiere decir que no puede ser votado. La prioridad por defecto es 1 (priority=1).Por otro lado, si queremos asegurarnos que un servidor, siempre que esté disponible, sea elegido primario, le pondremos más prioridad que a los demás
  - **PUEDEN ESTAR OCULTOS** : Puede haber servidores **ocultos** que no pueden ser votados (hidden=true).

TIPOS DE SERVIDORES

- **Servidor con PRIORIDAD 0:**
  - Son servidores SECUNDARIOS que no pueden ser elegidos como principales.
  - Aceptan operaciones de lectura
  - Mantienen una copia de los datos
  - Pueden votar en las elecciones de un nuevo servidor principal
  - No pueden lanzar unas elecciones

```
cfg = rs.conf()
cfg.members[2].priority = 0
rs.reconfig(cfg)
```

COMPONENTES

- **Servidor OCULTO:**
  - Son servidores SECUNDARIOS que no pueden ser elegidos como principales (siempre tienen prioridad 0) ni utilizados por los clientes para realizar consultas.
  - Mantienen una copia de los datos
  - Pueden votar en las elecciones de un nuevo servidor principal
  - No pueden lanzar unas elecciones

```
cfg = rs.conf()
cfg.members[2].priority = 0
cfg.members[2].hidden = true
rs.reconfig(cfg)
```

COMPONENTES

- **Servidor RETARDADO (delayed):**
  - Son servidores SECUNDARIOS que replican los datos, pero con un retardo configurado previamente, reflejando un estado anterior del conjunto de datos. Por ejemplo, si lo configuramos con una hora de retraso, este servidor tendrá los datos tal y como estaban una hora antes.
  - Deben ser ocultos y tener prioridad 0 por lo que: votan en las elecciones, no pueden ser elegidos y no pueden lanzar unas elecciones
  - Su “retraso” debe ser más pequeño que la capacidad del opLog

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
cfg.members[0].secondaryDelaySecs = 3600
rs.reconfig(cfg)
```

COMPONENTES

- **Servidor ÁRBITRO:**
  - Son servidores que no tienen copia de los datos.
  - Su única función es votar a la hora de elegir un nuevo nodo primario cuando tenemos un número par de servidores, con la intención de deshacer el empate ante una elección de servidor principal.
  - Los nodos árbitro nunca cambian de estado hacia otro tipo de nodo. Es decir, nunca dejan de serlo.
  - Apenas consumen recursos, por lo que podemos utilizar servidores compartidos con otras tareas.

COMPONENTES

- **OpLog:**
  - Para soportar la replicación, el nodo primario almacena todos los cambios en su oplog
  - El log de operaciones es una colección que almacena todas las operaciones de la BBDD con el propósito de que un nodo secundario pueda replicar dichos cambios en sus datos y asegurar que las bases de datos sean idénticas.
  - El servidor principal recibe las operaciones de escritura que aplica a sus datos y después guarda en el oplog.
  - Los servidores secundarios copian esta colección de forma asíncrona (del servidor principal o de otro servidor secundario) y aplican también los cambios a sus conjuntos de datos.

COMPONENTES

- **OpLog:**
  - El oplog crea un timestamp para cada entrada. Esto permite que un secundario sepa qué entradas necesita transferir para ponerse al día. Si paramos un secundario y lo reiniciamos más adelante, utilizará el oplog para obtener todos los cambios que ha perdido mientras estaba offline.
  - El oplog se almacena en una **colección limitada (capped)** y ordenada de un tamaño determinado. La opción oplogSize define en MB el tamaño del archivo.
  - Tiene un tamaño máximo, por lo que cuando está llena empieza a borrar lo más antiguo para escribir lo nuevo.
  - CUIDADO con las actualizaciones "masivas" o con hacer muchas pequeñas actualizaciones, porque llenarán rápidamente el OpLog y se pueden perder entradas del log que podría no haber replicado algún nodo.

Consistencia en la escritura

- MongoDB tiene un mecanismo para garantizar la consistencia de los datos. en la escritura (writeConcern), ofreciendo un mecanismo que garantiza que un determinado dato se ha replicado en más o menos servidores..
- Dependiendo del nivel de configuración de la consistencia, las inserciones, modificaciones y borrados pueden tardar más o menos.
- Si reducimos el nivel de consistencia, el rendimiento será mejor, a costa de asegurar un poco menos que los datos se han escrito y replicado en los distintos servidores..
- Podemos configurar los siguientes aspectos:
  - w:valor. Número de servidores o la mayoría (valor "majority"), que tienen que confirmar que lo tienen escrito
  - wtimeout:5000. Si se alcanza este tiempo sin confirmación se cancela el cambio
- Ejemplo de cómo podemos modificar una inserción:

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

Elección del nodo primario

La elección de un nodo primario sucede cuando:

- Se está inicializando un nuevo conjunto de réplicas y todavía no hay ningún servidor principal. Es decir, al principio.
- Caen el servidor principal o su conexión. Con la información de los latidos, se decide si es necesario llevar a cabo la elección de un nuevo servidor principal.
- Si pasamos el servidor principal a ser secundario de forma manual (con el comando stepDown).

Elección del nodo primario

- Un set de réplicas sólo puede tener 7 miembros con voto
- Para elegir un nodo como primario tiene que tener un número de votos igual o mayor a la parte entera de la mitad del número total de votos mas 1.

  votos_necesarios = |nº total de votos / 2 | + 1
- El **número total de votos** no se calcula por el número de servidores que votan, sino por el número de votos totales configurados (los servidores caídos también se cuentan).
- Lo normal es que cada servidor tenga 1 voto.
- Ejemplo:
  - Si tenemos 3 servidores, para elegir servidor tendrá que tener 2 votos |3/2|+1, por lo que tiene que haber mínimo 2 servidores que puedan votar. Si se perdieran dos nodos de los 3 no podría volver a elegirse un primario y quedaría en solo lectura.

Elección del nodo primario

- Tenemos que configurar un número impar de votos para evitar los empates.
- En caso de tener un número par de votos tenemos que añadir algún nodo o cambiar los pesos de alguno de los nodos que existen. La opción que menos recursos consume es añadir un nuevo árbitro que no requiere mucho espacio ni procesamiento.
- Hay que añadir nuevos miembros al conjunto antes de quedarnos sin suficientes votos para elegir nuevo primario.
- Si no se puede elegir un nodo principal quedará en modo de solo lectura hasta que se vuelva a lanzar otra votación o se pueda volver a levantar alguno de los nodos.
- MongoDB prefiere dejar el cluster en sólo lectura antes de que haya varios nodos principales.

Elección del nodo primario

- Condiciones para que un nodo pueda ser elegido como principal:
  - Tenga los datos actualizados. Para saber si está actualizado se utiliza la propiedad optime que tiene cada servidor (relacionada con el opLog), y que guarda una marca de tiempo con la última actualización aplicada.
  - Tenga el valor de prioridad mayor que 0.
  - No esté oculto.
  - Pueda conectar con otros nodos hasta acumular un número de votos suficiente para alcanzar la mayoría.
- Parámetros para configurar los tiempos de espera para latidos y elecciones:
  - **heartbeatIntervalMillis (2sg)** → tiempo que pasa para un nuevo latido.
  - **electionTimeoutMillis (10sg)** → número de milisegundos que tienen que pasar sin comunicación del primario antes de lanzar una nueva elección.

Elección del nodo primario

- Proceso:
  - El primario envía heartbeats periódicos (≈ cada 2s) (heartbeatIntervalMillis).
  - Los secundarios reciben esos heartbeats y resetean su contador⏳ de milisegundos desde el desde el último latido.
  - Cuando el contador de un nodo llega al valor de electionTimeoutMillis desde la anterior respuesta del primario:
    - Asume que no hay primario
    - Si cumple las condiciones
      - se postula como candidato
      - pide votos al resto
  - Si el contador de otro nodo llega al valor de electionTimeoutMillis y no hay candidatos que cumplan las condiciones también se postulará como candidato.

Elección del nodo primario

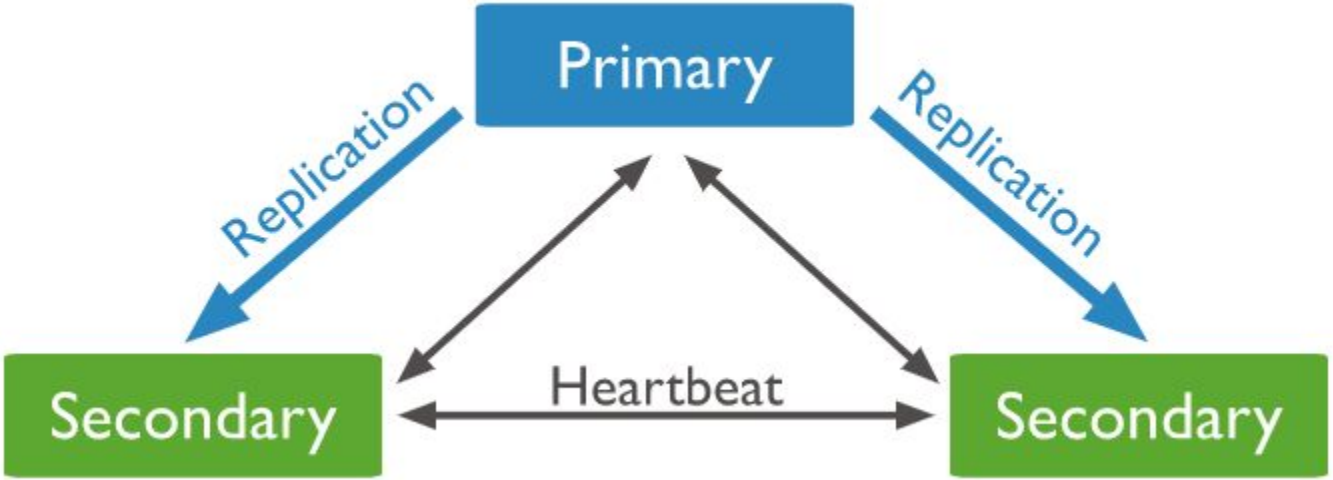
- Después del conteo de votos:
  - Si un nodo tiene más votos que la mayoría, entonces automáticamente se convierte en el primario.
  - Si no hay un nodo con más votos que la mayoría, el algoritmo de votación vuelve a iniciar la votación pasados unos pocos segundos.
  - Si pasados varios intentos no hay ningún nodo con suficientes votos se elige de forma aleatoria a un nodo de entre los que cumplan las condiciones de elección.

Elección del nodo primario

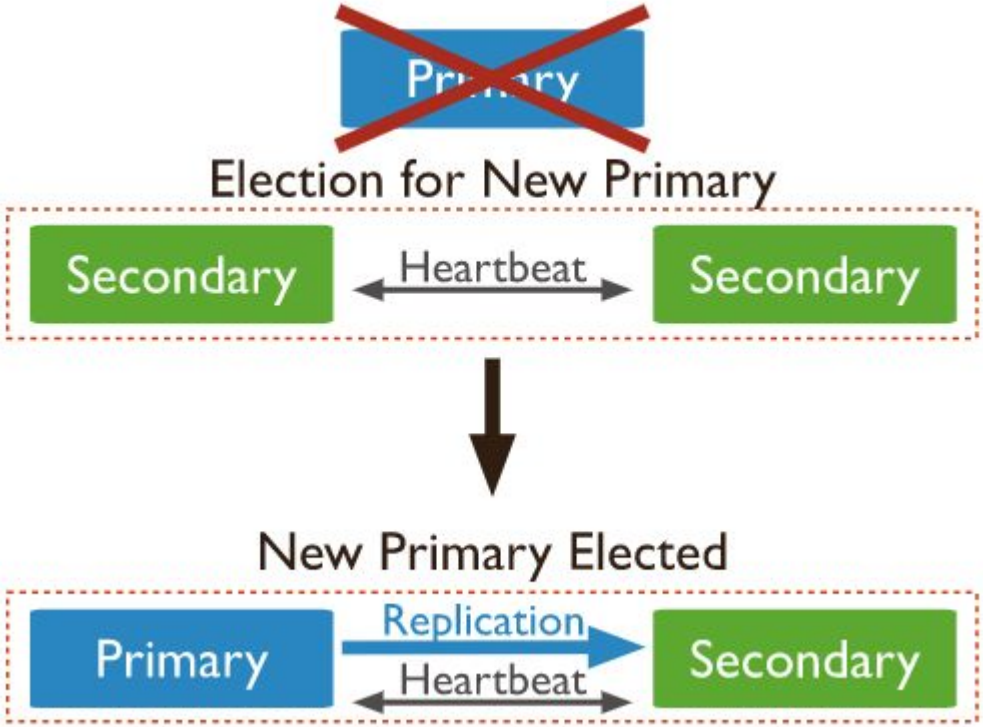
- Tenemos que tener en cuenta la tolerancia a fallos, es decir, cuántos servidores pueden fallar sin que suframos problemas para elegir un nuevo servidor principal
- A continuación podremos ver la relación entre el número de nodos y la mayoría de votos requerida para elegir un nuevo nodo primario. Esto nos da la tolerancia ante el fallo que tiene el conjunto

Number of Members	Majority Required to Elect a New Primary	Fault Tolerance
3	2	1
4	3	1
5	3	2
6	4	2

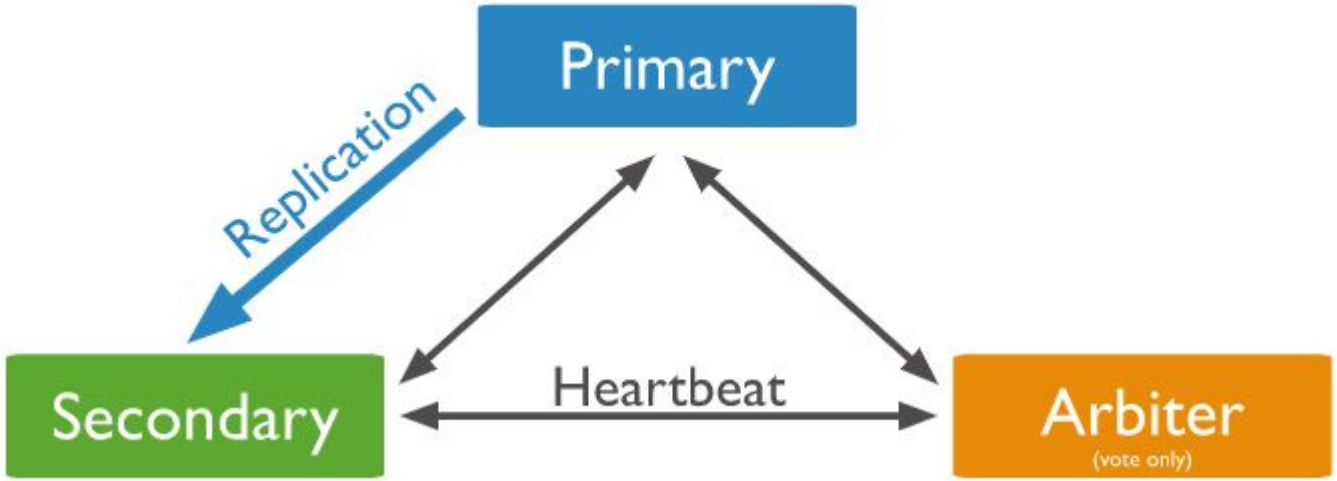
TRES SERVIDORES CON COPIA



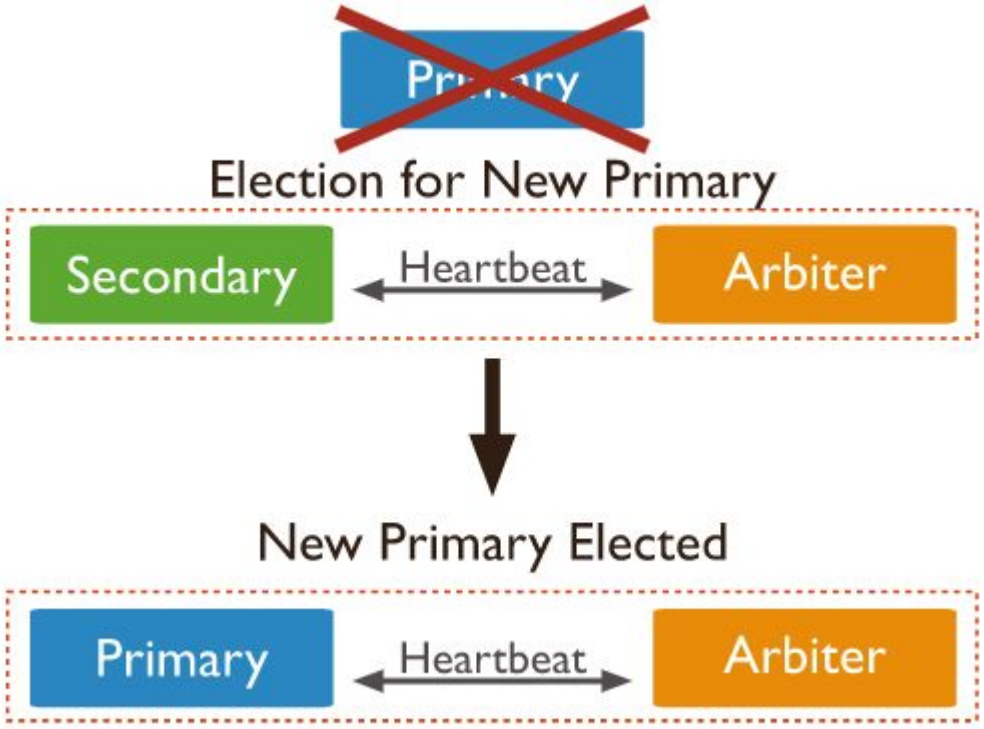
TRES SERVIDORES CON COPIA



DOS SERVIDORES CON COPIA



DOS SERVIDORES CON COPIA



REPARTO GEOGRÁFICO

- La distribución de miembros del conjunto de réplicas entre dos centros de datos **proporciona ventajas sobre un único centro de datos** .
- En una distribución de **dos** centros de datos:

Si uno de los centros de datos deja de funcionar

Los datos todavía están disponibles para lecturas, a diferencia de la distribución de un único centro de datos.

REPARTO GEOGRÁFICO

- Dependiendo de qué centro de datos deja de funcionar

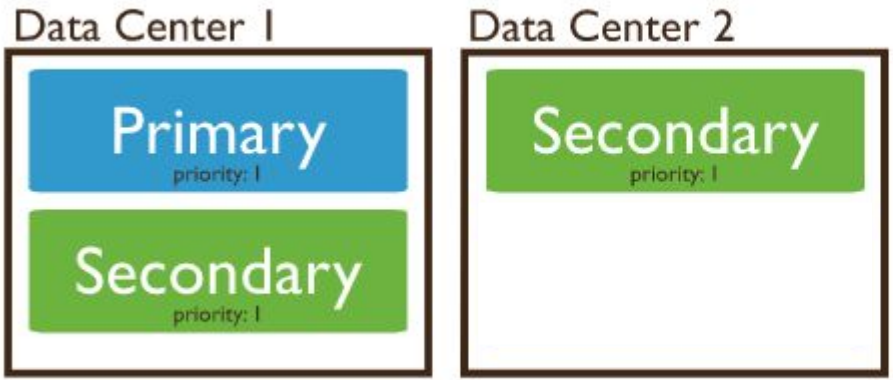
Si deja de funcionar el centro de datos con una **minoría** de miembros	→	El conjunto de réplicas SÍ PUEDE ELEGIR PRIMARIO y sigue trabajando normalmente.
Si deja de funcionar el centro de datos con la **mayoría** de los miembros	→	El conjunto de réplicas no puede elegir primario y pasa a ser de solo lectura.

- Lo ideal es distribuir los miembros en tres centros de datos o más. Si el costo del tercer centro de datos es prohibitivo, una posibilidad es distribuir los nodos que contienen datos entre los dos centros de datos y almacenar algún miembro con voto en la nube.

REPARTO GEOGRÁFICO

- Si tuviéramos tres nodos podríamos distribuirlos en 1, 2 o 3 centros de datos.
- ¿Qué podría pasar aquí?

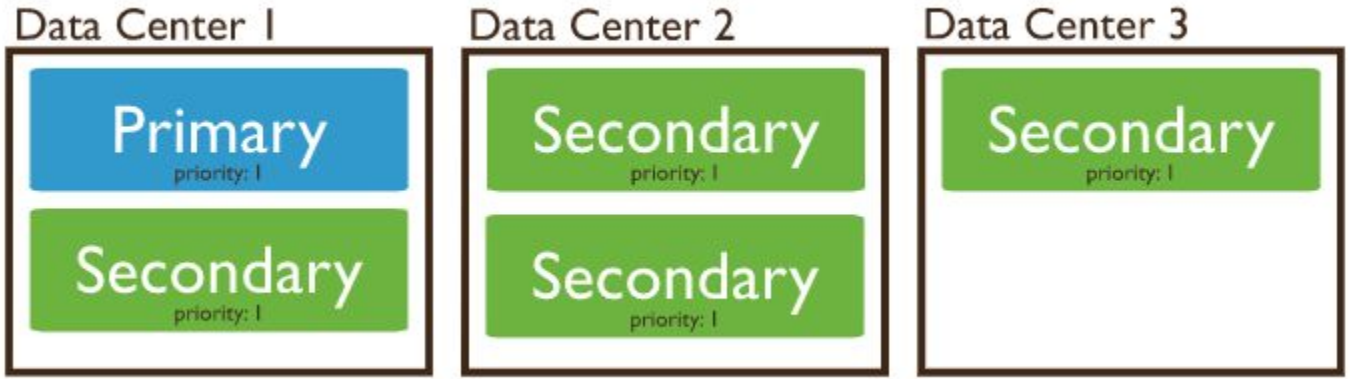
El número de votos necesarios para elegir un nuevo primario será de 2, por lo que



- Si el centro de datos 1 deja de funcionar, no se puede elegir primario, y el conjunto de réplicas pasa a ser de solo lectura.
- Si el Centro de datos 2 deja de funcionar, el Centro de datos 1 puede realizar una elección.

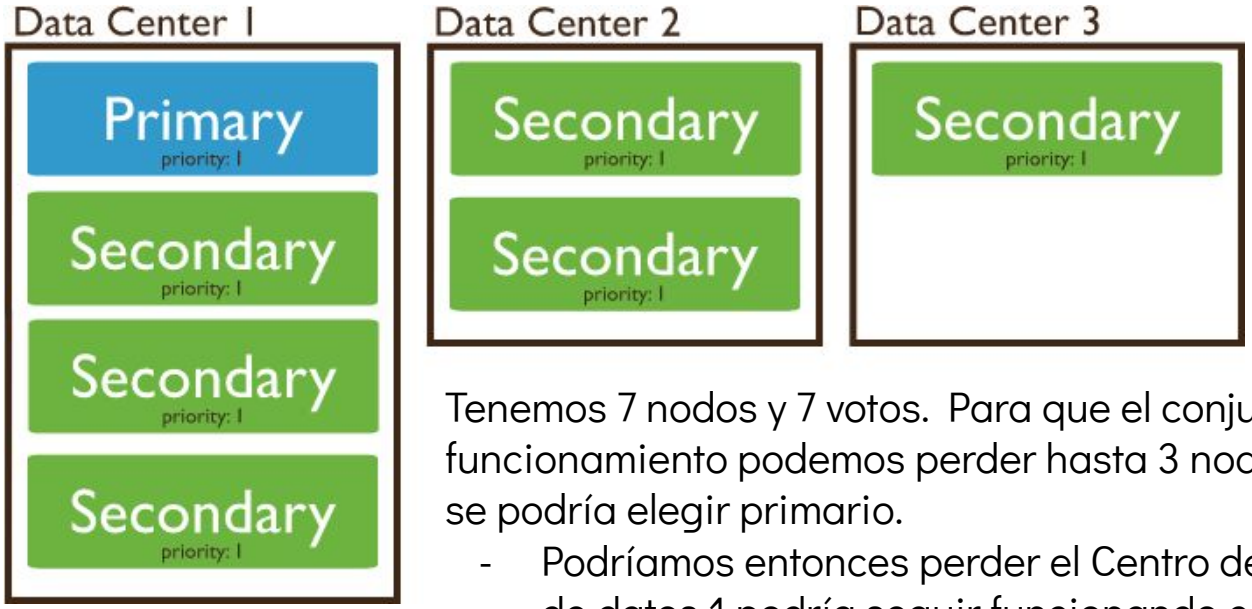
REPARTO GEOGRÁFICO

- Para un conjunto de réplicas con 5 miembros
- ¿Qué podría pasar aquí?



Como el número de votos mínimo es de 3, si algún centro de datos deja de funcionar, los miembros restantes pueden realizar una elección y por lo tanto todo va a seguir funcionando.

REPARTO GEOGRÁFICO



Tenemos 7 nodos y 7 votos. Para que el conjunto siga en funcionamiento podemos perder hasta 3 nodos. Es decir, con 4 votos se podría elegir primario.

- Podríamos entonces perder el Centro de datos 2 y el 3 y el centro de datos 1 podría seguir funcionando correctamente.
- Si perdemos los 4 nodos del centro de datos 1, ya no se podría elegir nodo primario y el conjunto pasaría a ser de solo lectura

REPARTO GEOGRÁFICO

¿Por qué está configurado así?

Estamos estableciendo un orden de prioridad en cuanto a elección de nodo primario. Nos interesa, que alguno de los miembros del Centro de datos 1 sea nodo primario. Si ese centro de datos no está disponible, queremos que sea elegido alguno del Centro de datos 2, y de ninguna manera el miembro del Centro de datos 3 puede datos funcionar como primario.

CREAR REPLICA SET EN DOCKER

Normalmente, cada instancia mongod se coloca en un servidor físico y todos en el puerto estándar (27017).

En nuestro caso, vamos a crear un conjunto de tres réplicas utilizando Docker.

Lo primero que necesitamos es crear un directorio para incluir el docker-compose.yml y cargarlo en Visual Studio Code.

Primero lanzamos los contenedores de Mongo utilizamos el docker-compose.yml, y luego entramos en uno de ellos e iniciamos el cluster.

CREAR REPLICA SET EN DOCKER

Vamos a probar a crear un Set de Réplicas con Docker. Para ello vamos a utilizar la imagen mongo:6.

- **Fichero docker-compose.yml**
  - Por cada nodo le ponemos el nombre del contenedor y un puerto distinto en la parte izquierda:

```yaml
services:
  mongo1:
    image: mongo:6
    container_name: mongo1  # <-- Cambiarlo
    ports:
      - 27117:27017  # <-- Cambiar el número de la izquierda
    command: ["--replSet", "rs0", "--bind_ip_all"]
    volumes:
      - mongo1-data:/data/db   # <-- Cambiarlo
    networks:
      - mongo-cluster
```

CREAR REPLICA SET EN DOCKER

- **Fichero docker-compose.yml**
  - Configuramos los volúmenes:

```
volumes:
  mongo1-data:
  mongo2-data:
  mongo3-data:

networks:
  mongo-cluster:
```

CREAR REPLICA SET EN DOCKER

- **Lanzamos el cluster:**

```
docker-compose up -d
```

ó

```
docker compose up --build
```

- **Entramos en mongo1 para configurar el cluster**

```
docker exec -it mongo1 mongosh
```

- **Inicializamos el cluster**

```
rs.initiate()
```

- **Añadimos los otros dos nodos al cluster**

```
rs.add("mongo2:27017")
rs.add("mongo3:27017")
```

CREAR REPLICA SET EN DOCKER

- **Comprobar el estado. Busca el optime**

  ```
  rs.status()
  ```

- **Paramos el contenedor que sea primario y volvemos a comprobar el estado**

  ```
  rs.stepDown()
  ```

- **Para conectarnos desde compass**

  ```
  mongodb://localhost:27117/?directConnection=true
  ```

- **Parar y eliminar el cluster**

  ```
  docker-compose down -v
  ```

- **Para apagar desde la consola de MongoDB un nodo**

  ```
  db.adminCommand({"shutdown" : 1})
  ```

CREAR REPLICA SET EN DOCKER

- **AÑADAMOS UN ÁRBITRO:**
  - Añadimos el nodo en el docker-compose (no hace falta el bing de puertos):

```
arbitro:
  image: mongo:8
  container_name: arbitro
  command: mongod --replSet rs0 --port 27017
  networks:
    - mongo-cluster
```

  - Añadimos el nodo en el cluster

```
rs.addArb("arbitro:27017")
```

CREAR REPLICA SET EN DOCKER

- En la consola vemos la configuración `rs.conf()`

```
{
  _id: 'rep1',
  version: 5,
  term: 2,
  members: [
    {
      _id: 0,
      host: 'localhost:27666',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 1,
      host: 'localhost:27667',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 2,
      host: 'localhost:27668',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    }
  ],
  protocolVersion: Long('1'),
  writeConcernMajorityJournalDefault: true,
  settings: {
    chainingAllowed: true,
    heartbeatIntervalMillis: 2000,
    heartbeatTimeoutSecs: 10,
    electionTimeoutMillis: 10000,
    catchUpTimeoutMillis: -1,
    catchUpTakeoverDelayMillis: 30000,
    getLastErrorModes: {},
    getLastErrorDefaults: { w: 1, wtimeout: 0 },
    replicaSetId:
ObjectId('66174997ed216477cc7df20a')
  }
}
```

CREAR REPLICA SET EN DOCKER

- **Comprobar el estado**

  ```
  rs.status()
  ```

  Comprobamos por ejemplo qué nodos son secundarios y primarios o el campo que indica el nivel de actualización de cada nodo.

CREAR REPLICA SET EN WINDOWS

Normalmente, cada instancia mongod se coloca en un servidor físico y todos en el puerto estándar (27017).

En nuestro caso, vamos a crear un conjunto de tres réplicas, y en vez de hacerlo en tres máquinas distintas, vamos a lanzar tres ejecuciones de mongod en la misma máquina con tres puertos diferentes (del 27666 al 27668).

Lo primero que necesitamos es crear tres directorios para los datos de cada instancia del servidor (datos1, datos2, datos3). Lo más fácil es crearlos en el directorio donde se encuentran los ejecutables de mongo:

```
C:\Program Files\MongoDB\Server\7.0\bin>mkdir datos1 datos2 datos3
```

También necesitamos tener la consola de mongo. Para ello tenemos que descargar mongosh desde la web de MongoDB, descargarlo en versión ZIP o MSI. Nos vale con la versión ZIP, que descargamos y descomprimimos.

CREAR REPLICA SET EN WINDOWS

Primero creamos las tres instancias desde una consola de administrador para evitar problemas.

Al crear las instancias hay que indicar:

- el mismo nombre del replicaSet que vamos a utilizar
- en cada instancia un puerto y un directorio de datos distinto.

```
cd "C:\Program Files\MongoDB\Server\7.0\bin"

start mongod --port 27666 --dbpath datos1 --replSet rep1

start mongod --port 27667 --dbpath datos2 --replSet rep1

start mongod --port 27668 --dbpath datos3 --replSet rep1
```

CREAR REPLICA SET EN WINDOWS

- Accedemos a la consola de cada instancia desde el lugar donde hemos descomprimido la consola descargada.

```
start mongosh localhost:27666
start mongosh localhost:27667
start mongosh localhost:27668
```

- Iniciamos el replica set dentro de la consola de uno de ellos

```
rs.initiate()
```

- En la misma consola en que hemos iniciado el Replica Set añadimos los dos segundos servidores a la réplica

```
rs.add("localhost:27667")
rs.add("localhost:27668")
```

- Vemos el estado en cada instancia

```
rs.status()
rs.hello()
```

CREAR REPLICA SET EN WINDOWS

- En la consola vemos la configuración
  `rs.conf()`

```
{
  _id: 'rep1',
  version: 5,
  term: 2,
  members: [
    {
      _id: 0,
      host: 'localhost:27666',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 1,
      host: 'localhost:27667',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 2,
      host: 'localhost:27668',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    }
  ],
  protocolVersion: Long('1'),
  writeConcernMajorityJournalDefault: true,
  settings: {
    chainingAllowed: true,
    heartbeatIntervalMillis: 2000,
    heartbeatTimeoutSecs: 10,
    electionTimeoutMillis: 10000,
    catchUpTimeoutMillis: -1,
    catchUpTakeoverDelayMillis: 30000,
    getLastErrorModes: {},
    getLastErrorDefaults: { w: 1, wtimeout: 0 },
    replicaSetId:
ObjectId('66174997ed216477cc7df20a')
  }
}
```

CREAR REPLICA SET EN WINDOWS

- Para poder agregar un nodo árbitro tenemos que crear una instancia de servidor de forma normal y utilizar la función addArb.
  ```
  rs.addArb("localhost:27669")
  ```

- Para convertir un nodo primario en secundario ejecutamos en su consola el comando stepDown. Comenzará entonces la elección de un nuevo nodo primario.
  ```
  rs.stepDown()
  ```

- Para apagar desde la consola de MongoDB un nodo.
  ```
  db.adminCommand({"shutdown" : 1})
  ```

Big Data Aplicado

Big_Data
Aplicado

Curso Especialización
Inteligencia_Artificial y
Big_Data