# EJERCICIOS

REPLICACIÓN EN MONGODB

La replicación en MongoDB es un proceso mediante el cual se crean copias idénticas de una base de datos en varios servidores o nodos (al conjunto de servidores con los mismos datos se le llaman **conjunto de réplica** o **Replica Set**).

En un conjunto de replicación, hay al menos dos nodos: un nodo primario y uno o más nodos secundarios.

Las operaciones de lectura pueden dirigirse a cualquiera de los nodos, mientras que las operaciones de escritura solo se pueden realizar en el nodo primario.

MongoDB utiliza un proceso llamado elección de primario para determinar automáticamente cuál de los nodos será el primario en caso de fallo.

Interfaz de usuario gráfica

Descripción generada automáticamente

El objetivo principal de la replicación es mejorar la disponibilidad y la redundancia de los datos, así como la tolerancia a fallos.

PARTICIONADO EN MONGODB

El particionado, también conocido como sharding, es un enfoque para dividir los datos en fragmentos o particiones (shards) y distribuirlos en múltiples servidores o nodos.

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza media

Los componentes de un sharded cluster son:

* **Shards** (Fragmentos): cada una de las máquinas del clúster, que almacena un subconjunto de los datos de la colección. Cada shard es una instancia de mongod.
* **Servidores de Configuración**: Cada servidor de configuración es una instancia de mongod que almacena metadatos sobre el cluster. Los metadatos mapean los trozos con los shards, para saber qué trozos se encuentran en un determinado shard.
* **Enrutadores**: Cada router es una instancia mongos que enruta las lecturas y escrituras de las aplicaciones a los shards. Las aplicaciones no acceden directamente a los shards, sino al router.

Diagrama

Descripción generada automáticamente

El particionado (sharding) se utiliza para lograr una mayor escalabilidad y rendimiento al distribuir los datos en múltiples servidores.

### EJERCICIO 1

Vamos a crear tres contenedores con mongo (siendo mongo1 el nodo principal y mongo2 y mongo3 los secundarios) dentro de una misma red y que formen un conjunto de réplica (iabdrs).

Vamos a hacerlo usando Docker Compose en una máquina virtual con Ubuntu 22.

El archivo [docker-compose.yml](https://aitor-medrano.github.io/iabd2223/sa/resources/docker-compose-replicaset.yml) que tenemos que crear tiene las especificaciones para crear los tres contenedores.

*nano docker-compose.yml*

**docker-compose.yml**

*services:*

*mongo1:*

*image: mongo*

*container\_name: mongo1*

*command: mongod --port 27017 --replSet iabdrs --bind\_ip\_all*

*volumes:*

*- ./scripts:/scripts*

*- mongodb\_mongo1\_a\_db:/data/db*

*- mongodb\_mongo1\_a\_config:/data/configdb*

*ports:*

*- 27117:27017*

*restart: always*

*networks:*

*- mongo-network*

*links:*

*- mongo2*

*- mongo3*

*mongo2:*

*image: mongo*

*container\_name: mongo2*

*command: mongod --port 27017 --replSet iabdrs --bind\_ip\_all*

*volumes:*

*- ./scripts:/scripts*

*- mongodb\_mongo2\_a\_db:/data/db*

*- mongodb\_mongo2\_a\_config:/data/configdb*

*ports:*

*- 27118:27017*

*restart: always*

*networks:*

*- mongo-network*

*mongo3:*

*image: mongo*

*container\_name: mongo3*

*command: mongod --port 27017 --replSet iabdrs --bind\_ip\_all*

*volumes:*

*- ./scripts:/scripts*

*- mongodb\_mongo3\_a\_db:/data/db*

*- mongodb\_mongo3\_a\_config:/data/configdb*

*ports:*

*- 27119:27017*

*restart: always*

*networks:*

*- mongo-network*

*volumes:*

*mongodb\_mongo1\_a\_db:*

*mongodb\_mongo1\_a\_config:*

*mongodb\_mongo2\_a\_db:*

*mongodb\_mongo2\_a\_config:*

*mongodb\_mongo3\_a\_db:*

*mongodb\_mongo3\_a\_config:*

*networks:*

*mongo-network:*

*driver: bridge*

Veamos una explicación para el fichero anterior:

* **services 🡪** este es el comienzo de la sección donde se definen los servicios de Docker Compose.
* **mongo1 🡪** este es el nombre del primer servicio que se define.
* **image: mongo** **🡪** especifica la imagen de Docker que se utilizará para este servicio. En este caso, se utiliza la imagen oficial de MongoDB llamada "mongo".
* **container\_name: mongo1** **🡪** establece el nombre del contenedor Docker que se creará para el servicio mongo1.
* **command: mongod --port 27017 --replSet iabdrs --bind\_ip all** 🡪 comando que se va a ejecutar al iniciar el contenedor (inicializar mongod). En este caso, se ejecuta **/usr/bin/mongod** (el ejecutable de MongoDB) con opciones que se especifican (--bind\_ip all para que sea accesible desde el exterior y --replSet para indicar que pertenece al conjunto de réplica iabdrs
* **volumes 🡪** aquí se definen los volúmenes que se van a montar en el contenedor.
  + **./scripts:/scripts 🡪** monta en /scripts del contenedor el contenido de la carpeta local ./scripts
  + **mongodb\_mongo2\_a\_db:/data/db 🡪** persiste los datos del directorio /data/db del contendor (en este directorio guarda mongodb los datos) en el volumen mongodb\_mongo2\_a\_db
  + **mongodb\_mongo2\_a\_config:/data/configdb 🡪** persiste los datos del directorio /data/configdb (en este directorio guarda mongodb la configuración) del contendor en el volumen mongodb\_mongo2\_a\_db
* **ports 🡪** aquí se mapean los puertos del host a los puertos del contenedor.
  + **- 27017:27017** **🡪** mapea el puerto 27017 del host al puerto 27017 del contenedor. Esto permite que la instancia de MongoDB dentro del contenedor sea accesible desde el host en el puerto 27017.
* **restart: always 🡪** indica que se reinicien los contendores cuando se reinicie el sistema.
* **networks 🡪** especifica la red a la que se conectará el contenedor.
  + **- mongo-network** **🡪** Indica que el contenedor se unirá a la red personalizada llamada "mongo-network".
* **links** **🡪** establece enlaces (o conexiones) entre los contenedores.
  + **- mongo2** **🡪** crea un enlace a **mongo2**, lo que permite que **mongo1** se comunique con **mongo2**.
  + **- mongo3** **🡪** similar al anterior, se crea un enlace a **mongo3** para la comunicación.
* Las líneas correspondientes a mongo2 y mongo3 son similares.
* **volumes** 🡪 crea los volúmenes para persistir los datos
* **networks 🡪** aquí se define la sección "networks" que se utiliza para configurar redes personalizadas en Docker.
  + **mongo-network 🡪** este es el nombre de la red personalizada que se está creando. En este caso, se llama "mongo-network".
  + **driver: bridge** 🡪 establece el controlador de red que se utilizará para esta red personalizada. En este caso, se utiliza el controlador "bridge", que es el controlador de red predeterminado y se utiliza comúnmente para conectar contenedores en la misma máquina host.

Para inicializar el servicio todavía tenemos que ejecutar un script para su configuración e inicialización, dicho script los vamos a guardar en la carpeta scripts en nuestra ubicación actual (es lo que hemos indicado en volumes):

*mkdir scripts*

*nano scripts/init.js*

**init.js**

*var config = {*

*"\_id": "iabdrs",*

*"version": 1,*

*"members": [*

*{*

*"\_id": 1,*

*"host": "mongo1:27017",*

*"priority": 2*

*},*

*{*

*"\_id": 2,*

*"host": "mongo2:27017",*

*"priority": 1*

*},*

*{*

*"\_id": 3,*

*"host": "mongo3:27017",*

*"priority": 1*

*}*

*]*

*};*

*rs.initiate(config, { force: true });*

*rs.status();*

Este script de bash se utiliza para configurar un conjunto de réplicas en MongoDB:

1. Se define un objeto **config** que contiene la configuración del conjunto de réplicas. Este objeto especifica el ID del conjunto de réplicas, su versión y los miembros del conjunto con sus respectivos IDs, hosts y prioridades.
2. Se utiliza el comando **rs.initiate(config, { force: true })** para iniciar el conjunto de réplicas con la configuración definida en el objeto **config**. El parámetro **{ force: true }** se utiliza para forzar la reinicialización del conjunto de réplicas si ya existe una configuración previa.
3. Se utiliza el comando **rs.status()** para ver como ha quedado nuestro conjutno de réplica

Ya podemos lanzar Docker Compose desde la carpeta donde tenemos docker-compose.yml y la carpeta scripts:

*sudo docker-compose up -d*

Veamos los parámetros del comando:

* **up**: se utiliza para crear y arrancar los contenedores.
* **-d**: se utiliza para ejecutar los contenedores en segundo plano, lo que significa que la aplicación se ejecutará en segundo plano sin bloquear la terminal actual.

Ya sólo nos queda ejecutar el script de inicialización sobre el nodo principal:

*sudo docker exec mongo1 sh -c "mongosh < /scripts/init.js"*

Al ejecutarse el script, se inicializa el conjunto de réplicas y se obtiene su estado (mediante rs.status() ) el cual se muestra por consola y podemos observar cómo ha creado los tres nodos, diferenciando el nodo principal de los secundarios.

Con los siguientes comandos podemos ver los servicios y contenedores de docker-compose activos:

*sudo docker-compose ls*

*sudo docker-compose ps*

Una vez que ya hemos arrancado todo, podemos conectarnos a nuestro conjunto de réplica mediante mongosh desde la terminal del contenedor o desde otro equipo que tenga instalado mongosh.

*sudo docker exec -it mongo1 bash*

*mongosh*

*mongosh mongo2:27017*

*mongosh mongo3:27017*

*mongosh IP:27117*

*mongosh IP:27118*

*mongosh IP:27119*

Al conectarnos mongosh nos indica si estamos conectados a un nodo primario o a un nodo secundario.

*iabdrs [direct: primary] test>*

*iabdrs [direct: secondary] test>*

**TRABAJANDO CON LAS RÉPLICAS**

Mediante rs.status() obtendremos información sobre los nodos.

Vamos a añadir datos desde el nodo primario.

*for (i=0; i<1000; i++) {*

*db.pruebas.insertOne({num: i})*

*}*

Estos 1.000 documentos se han insertado en el nodo principal, y se han replicado a los secundarios. Para comprobar la replicación, abrimos un nuevo terminal y nos conectamos a un nodo secundario:

*iabdrs [direct: secondary] test> db.pruebas.countDocuments()*

Pero que podamos leer no significa que podamos escribir. Si intentamos escribir en un nodo secundario obtendremos un error:

*iabdrs [direct: secondary] test> db.pruebas.insertOne({num : 1001})*

*MongoServerError: not primary*

**TOLERANCIA A FALLOS**

Vamos a comprobar la tolerancia a fallos, para ello vamos a degradar a nodo secundario el nodo primario mediante rs.stepDown():

*iabdrs [direct: primary] test> rs.stepDown()*

Con rs.status() podemos ver quien es ahora el nodo primario.

**DOCKER-COMPOSE**

Si queremos para docker-compose:

*sudo docker-compose down*

Y para volverlo a levantar

*sudo docker-compose up -d*

Para ver los detalles de un contenedor:

*sudo docker inspect mongo1*

### EJERCICIO 2

Crea un conjunto de réplica con 4 nodos de nombre iabdrs4 (en una máquina virtual con Ubuntu 22) en la cual insertaremos los datos de 1000 ciudades (1000cities.json) en una nueva base de datos dbciudades.

Una vez creado, se pide:

* 1. Obtener el estado del conjunto de réplicas
  2. Consultar una ciudad desde un nodo secundario.
  3. Degradar el nodo primario
  4. Averiguar cuál es el nuevo nodo primario.

### EJERCICIO 3

Vamos a crear dos conjuntos de réplicas de dos nodos cada una, y a su vez, particionaremos los datos en dos shards usando docker-compose en una máquina virtual con Ubuntu 22.

Además, vamos a añadir un único router y un servidor de configuración.

Así pues, crearemos:

* un contenedor para el router (router1) el cual ejecuta el servicio mongos y que va a conectar con el servidor de configuración.
* un contenedor para el servidor de configuración (configsvr1) indicándole mediante el parámetro --configsvr su propósito así como la réplica a la que pertenece (todo servidor de configuración debe pertenecer a una réplica, aunque en nuestro caso sólo hemos creado uno).
* dos nodos (mongo-shard1a y mongo-shard1b) para el primer shard (parámetro --shardsvr) que pertenecen al conjunto de réplicas iabdshard1.
* dos nodos más (mongo-shard2a y mongo-shard2b) para el segundo shard (parámetro --shardsvr) que pertenecen al conjunto de réplicas iabdshard2.

Creamos el archivo docker-compose.yml con la definición de los contenedores:

*nano docker-compose.yml*

**docker-compose.yml**

*services:*

*## Router*

*router1:*

*image: mongo*

*container\_name: router1*

*command: mongos --port 27017 --configdb rs-config-server/configsvr1:27017 --bind\_ip\_all*

*ports:*

*- 27117:27017*

*restart: always*

*volumes:*

*- ./scripts:/scripts*

*- mongodb\_cluster\_router1\_db:/data/db*

*- mongodb\_cluster\_router1\_config:/data/configdb*

*networks:*

*- mongo-network-sharded*

*## Config Servers*

*configsvr1:*

*image: mongo*

*container\_name: configsvr1*

*command: mongod --port 27017 --configsvr --replSet rs-config-server --bind\_ip\_all*

*volumes:*

*- ./scripts:/scripts*

*- mongodb\_cluster\_configsvr1\_db:/data/db*

*- mongodb\_cluster\_configsvr1\_config:/data/configdb*

*ports:*

*- 27118:27017*

*restart: always*

*links:*

*- shard1a*

*- shard1b*

*- shard2a*

*- shard2b*

*networks:*

*- mongo-network-sharded*

*## Shards*

*## Shards shard1*

*shard1a:*

*image: mongo*

*container\_name: shard1a*

*command: mongod --port 27017 --shardsvr --replSet rshard1 --bind\_ip\_all*

*volumes:*

*- ./scripts:/scripts*

*- mongodb\_cluster\_shard1a\_a\_db:/data/db*

*- mongodb\_cluster\_shard1a\_a\_config:/data/configdb*

*ports:*

*- 27119:27017*

*restart: always*

*networks:*

*- mongo-network-sharded*

*shard1b:*

*image: mongo*

*container\_name: shard1b*

*command: mongod --port 27017 --shardsvr --replSet rshard1 --bind\_ip\_all*

*volumes:*

*- ./scripts:/scripts*

*- mongodb\_cluster\_shard1b\_a\_db:/data/db*

*- mongodb\_cluster\_shard1b\_a\_config:/data/configdb*

*ports:*

*- 27120:27017*

*restart: always*

*networks:*

*- mongo-network-sharded*

*## Shards shard2*

*shard2a:*

*image: mongo*

*container\_name: shard2a*

*command: mongod --port 27017 --shardsvr --replSet rshard2 --bind\_ip\_all*

*volumes:*

*- ./scripts:/scripts*

*- mongodb\_cluster\_shard2a\_a\_db:/data/db*

*- mongodb\_cluster\_shard2a\_a\_config:/data/configdb*

*ports:*

*- 27121:27017*

*restart: always*

*networks:*

*- mongo-network-sharded*

*shard2b:*

*image: mongo*

*container\_name: shard2b*

*command: mongod --port 27017 --shardsvr --replSet rshard2 --bind\_ip\_all*

*volumes:*

*- ./scripts:/scripts*

*- mongodb\_cluster\_shard2b\_a\_db:/data/db*

*- mongodb\_cluster\_shard2b\_a\_config:/data/configdb*

*ports:*

*- 27122:27017*

*restart: always*

*networks:*

*- mongo-network-sharded*

*volumes:*

*mongodb\_cluster\_router1\_db:*

*mongodb\_cluster\_router1\_config:*

*mongodb\_cluster\_configsvr1\_db:*

*mongodb\_cluster\_configsvr1\_config:*

*mongodb\_cluster\_shard1a\_a\_db:*

*mongodb\_cluster\_shard1a\_a\_config:*

*mongodb\_cluster\_shard1b\_a\_db:*

*mongodb\_cluster\_shard1b\_a\_config:*

*mongodb\_cluster\_shard2a\_a\_db:*

*mongodb\_cluster\_shard2a\_a\_config:*

*mongodb\_cluster\_shard2b\_a\_db:*

*mongodb\_cluster\_shard2b\_a\_config:*

*networks:*

*mongo-network-sharded:*

*driver: bridge*

Antes de lanzar los contenedores mediante docker-compose vamos a crear los scripts de inicialización:

* scripts:
* configserver-init.js
* shard1-init.js
* shard2-init.js
* router-init.js
* init.sh

*mkdir scripts*

*nano scripts/configserver-init.js*

**configserver-init.js**

*rs.initiate(*

*{*

*\_id: "rs-config-server",*

*configsvr: true,*

*version: 1,*

*members: [*

*{ \_id: 0, host : "configsvr1:27017" },*

*]*

*}*

*)*

*nano scripts/shard1-init.js*

**shard1-init.js**

*rs.initiate(*

*{*

*\_id: "rshard1",*

*version: 1,*

*members: [*

*{ \_id: 0, host : "shard1a:27017" },*

*{ \_id: 1, host : "shard1b:27017" },*

*]*

*}*

*)*

*nano scripts/shard2-init.js*

**shard2-init.js**

*rs.initiate(*

*{*

*\_id: "rshard2",*

*version: 1,*

*members: [*

*{ \_id: 0, host : "shard2a:27017" },*

*{ \_id: 1, host : "shard2b:27017" },*

*]*

*}*

*)*

*nano scripts/router-init.js*

**router-init.js**

*sh.addShard("rshard1/shard1a:27017")*

*sh.addShard("rshard1/shard1b:27017")*

*sh.addShard("rshard2/shard2a:27017")*

*sh.addShard("rshard2/shard2b:27017")*

*nano init.sh*

**init.sh**

*#!/bin/bash*

*sudo docker exec configsvr1 sh -c "mongosh < /scripts/configserver-init.js"*

*sudo docker exec shard1a sh -c "mongosh < /scripts/shard1-init.js"*

*sudo docker exec shard2a sh -c "mongosh < /scripts/shard2-init.js"*

*sleep 20*

*sudo docker exec router1 sh -c "mongosh < /scripts/router-init.js"*

Ahora ya podemos lanzar los contendores

*sudo docker-compose up -d*

Ejecutamos init.sh que llama a todos los ficheros de configuración e inicialización:

*sh init.sh*

**CONEXIÓN AL ROUTER**

El router es un nodo de MongoDB que se va a encargar de aceptar las peticiones de los clientes y enrutarlas al shard adecuado.

Para conectarnos tenemos varias opciones:

* Nos conectamos directamente al contenedor y en el terminar al poner mongosh veremos que ejecuta mongos:

*sudo docker exec -it router1 bash*

*mongosh*

*[direct: mongos] test>*

* Nos conectamos desde otro equipo que tenga mongosh al puerto 27117, que es donde habíamos colocado nuestro router:

*mongosh IP:27117*

*[direct: mongos] test>*

Podemos comprobar el estado del shard con sh.status()

**HABILITANDO EL SHARDING**

Una vez hemos creado la estructura necesaria para soportar el particionado vamos a insertar un conjunto de datos para posteriormente particionarlos.

Para ello, vamos a insertar diez mil usuarios en una colección:

*[direct: mongos] test> use iabd*

*switched to db iabd*

*[direct: mongos] iabd> for (var i=0; i<40000; i++) {*

*db.usuarios.insertOne({"login":"usuario" + i, "nombre":"nombre" + i\*2, "fcreacion": new Date()});*

*}*

Una vez creados, comprobamos que se han insertado:

*db.usuarios.countDocuments()*

*db.usuarios.findOne()*

Como podemos observar, interactuar con mongos es igual a hacerlo con mongosh.

Ahora mismo no sabemos en cuál de los dos shards se han almacenado los datos. Además, estos datos no están particionados, es decir residen en sólo uno de los shards.

Para habilitar el sharding a nivel de base de datos y que los datos se repartan entre los shards disponibles, ejecutaremos el comando sh.enableSharding(<nombreDB>) :

*sh.enableSharding("iabd")*

Si volvemos a comprobar el estado del shard (sh.status()), tenemos que se ha creado la nueva base de datos y nos indica cuál es su shard primario.

Para terminar de habilitar el sharding para una determinada colección, tenemos que crear un índice sobre la shard key.

*db.usuarios.createIndex({"login": 1})*

Una vez habilitado el shard ya podemos fragmentar la colección:

*sh.shardCollection("iabd.usuarios", {"login": 1}, false)*

El método shardCollection particiona una colección a partir de una shard key y recibe tres parámetros:

* nombre de la colección, con nomenclatura de nombreBD.nombreColeccion
* nombre del campo para fragmentar la colección, es decir, el shard key. Si quisiéramos indicar que queremos utilizar una clave hash, lo haríamos indicando como valor hashed ({"login": "hashed"}). Si queremos utilizar rangos, lo indicamos con valor 1 ( {"login": 1} ).
* booleano que indica si el valor utilizado como shard key es único. Para ello, el índice que se crea sobre el campo debe ser del tipo unique.

Este comando divide la colección en fragmentos (chunks), la cual es la unidad que utiliza MongoDB para mover los datos. Una vez que se ha ejecutado, MongoDB comenzará a balancear la colección entre los shards del clúster. Este proceso no es instantáneo. Si la colección contiene un gran conjunto de datos puede llevar horas completar el balanceo.

Si ahora volvemos a comprobar el estado del shard veremos que la propiedad nChunks muestra el número de fragmentos que tenemos y rango de valores que alberga cada fragmento y en que shard está. También podemos comprobar la distribución de una colección mediante el método getShardDistribution

*db.usuarios.getShardDistribution()*

Vamos a insertar 10.000 usuarios más a ver qué sucede:

*[direct: mongos] iabd> for (var i=40000; i<60000; i++) {*

*db.usuarios.insertOne({"login":"usuario" + i, "nombre":"nombre" + i\*2, "fcreacion": new Date()});*

*}*

*[direct: mongos] iabd> db.usuarios.countDocuments()*

*20000*

1 sh.splitFind("iabd.usuarios", {"login":1})

Si ahora comprobamos el estado del shard (sh.status() o db.usuarios.getShardDistribution()), los datos se deberían haber repartido entre los shards disponibles.

Si al insertar más datos no se reparten los datos de forma automática, podemos forzarlo el siguiente comando, creando los chunks:

*sh.splitFind("iabd.usuarios", {"login": "usuario10000"})*

*sh.splitFind("iabd.usuarios", {"login": "usuario20000"})*

Ya hemos creado los chunks y si comprobamos en qué shard están veremos que están un único shard.

{

data: '866KiB',

docs: 10000,

chunks: 1,

'estimated data per chunk': '866KiB', 'estimated docs per chunk': 10000

}

---

Totals

{

data: '2.52MiB', docs: 30000,

Shard iabdshard2 at iabdshard2/mongo-shard2a:27017,mongo-shard2b:27017

}

Vamos a mover algún chunk a otro shard:

Movemos el chunk que contiene en el campo login el valor usuario5000 al shard rshard2:

*sh.moveChunk("iabd.usuarios", { "login": "usuario5000" }, "rshard2")*

Si comprobamos en qué shards están los datos veremos que están repartidos en los 2 shards.

}

### EJERCICIO 4

Crea un clúster de MongoDB con tres shards e importar los datos de 1000 ciudades (1000cities.json) con docker-compose en una máquina virtual con Ubuntu 22.

Una vez creado, se pide:

1. Particionar los datos por el nombre de la ciudad.
2. Una vez cargado los datos, obtener el estado del sharding.
3. Si los datos no están particionados, forzar el split de los mismos.
4. Tras ello, vaciar la colección y volver a importar los datos. Una vez importados, obtener de nuevo el estado del sharding.