



Mongo DBA

Versión 3.6

Rubén Gómez García

Version 1.1.0 2018-02-01

Table of Contents

1. Introducción	1
1.1. Big Data	1
1.2. Arquitecturas	3
1.3. BBDD Relacionales	4
1.4. El mundo Big Data	5
1.5. Teorema CAP	6
2. Introducción a MongoDB	8
2.1. Historia	8
2.2. Conceptos clave	9
2.3. Documentos	11
2.4. Instalación	12
2.5. La primera vez	13
2.6. El formato JSON	14
2.7. El formato BSON	17
2.8. Esquema dinámico	19
2.9. mongoimport/mongoexport	21
2.10. Interactuando con MongoDB	22
2.11. Curosres	23
2.12. Inserciones	25
2.13. Query Language Bases	25
2.14. Sorting	28
2.15. Arquitecturas MongoDB	29
3. CRUD	32
3.1. CRUD	32
3.2. Inserción y actualización	32
3.3. Borrar documentos	36
3.4. Bulk Write Operations	37
3.5. Wire Protocol	38
3.6. Comandos	38
4. Replicación I	42
4.1. Introducción	42
4.2. Replicación de datos	43
4.3. Conceptos sobre Replicación	48
4.4. Levantando un Replica Set	49
4.5. Inicializando un Replica Set	51
4.6. Administrando un Replica Set	53
4.7. Lecturas y Escrituras	54
5. Ejercicios - Replicación I	58

5.1. Ejercicio 1	58
5.2. Ejercicio 2	59
5.3. Ejercicio 3	60
5.4. Ejercicio 4	61
5.5. Ejercicio 5	62
6. 1. Sharding	64
6.1. Introducción	64
6.2. Replicación y Sharding	65
6.3. Chunks y operaciones	66
6.4. Arquitectura del Sharded Cluster	67
6.5. Creando un Sharded Cluster	68
6.6. Usando Sharding en Colecciones	72
6.7. Trabajar en un Sharded Cluster	73
6.8. Eligiendo Shard Keys	73
6.9. Ejemplos de Sharded Clusters	74
6.10. Pre-Splitting	75
6.11. Sharding basado en Hash	76
6.12. Eliminando un Shard	77
6.13. Consejos y buenas prácticas	78

Chapter 1. Introducción

1.1. Big Data

1.1.1. ¿Qué es Big Data?

- Volúmen → No hablamos de **GB**, si no de **TB**, **PB**, **EB** o incluso **ZB**
- Velocidad → Información generada a gran velocidad, como la de la bolsa
- Variedad → Nuestra información puede ser estructurada o no serlo
- Veracidad → No toda la información es igual de fiable

1.1.2. ¿Qué dificultades encontramos para ello?

- Complejidad → Parte de la información puede no ser entendida, debido a la no estructuración de la misma
- Almacenamiento → Hablamos de cantidades enormes de información, ¿Cómo gestionar tal cantidad de datos a nivel de almacenamiento?
- Rendimiento → ¿Cómo se puede procesar de manera eficiente la información para mejorar el rendimiento?

1.1.3. ¿De dónde viene la información que se procesa en Big Data?

- Sensores meteorológicos
- Geográfica
- Textos
- Chats de redes sociales
- Sensores de tráfico
- Imágenes de satélite
- Toda esta información son cantidades de datos que no pueden ser gestionadas por aplicaciones normales o datawarehouses y en su mayoría son producidas por sistemas automáticos.
- Suele ser información no estructurada, y crece a gran velocidad.
- En ocasiones, **Big Data** no significa **gran cantidad** si no **datos difíciles**.
- Ante todo esto, hay que tener en cuenta las **4 V's**

Volume

Big data comes in one size: large. Enterprises are awash with data, easily amassing terabytes and even petabytes of information.

TB, Records, Transactions, Tables,, Files

Velocity

Often time-sensitive, big data must be used as it is streaming in to the enterprise in order to maximize its value to the business.

Batch, Near time, Real time, Streams

Variety

Big data extends beyond structured data, including semi-structured and unstructured data of all varieties: text, audio, video, click streams, log files and more.

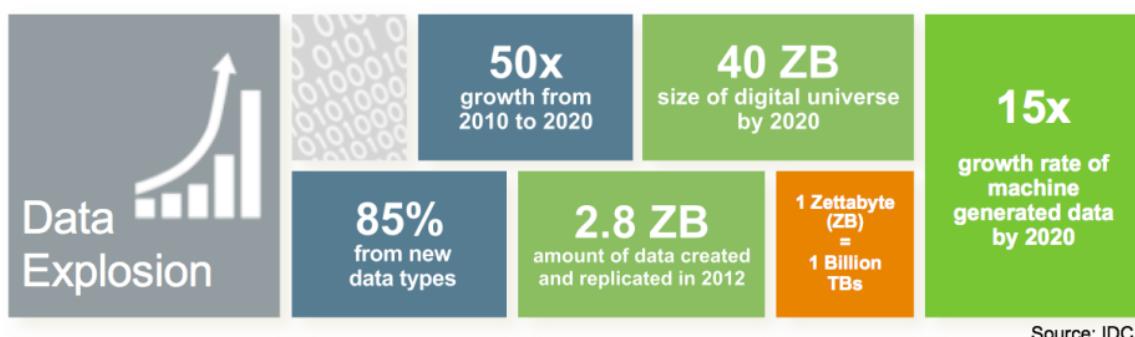
Structured, Unstructured, Semistructured

Value

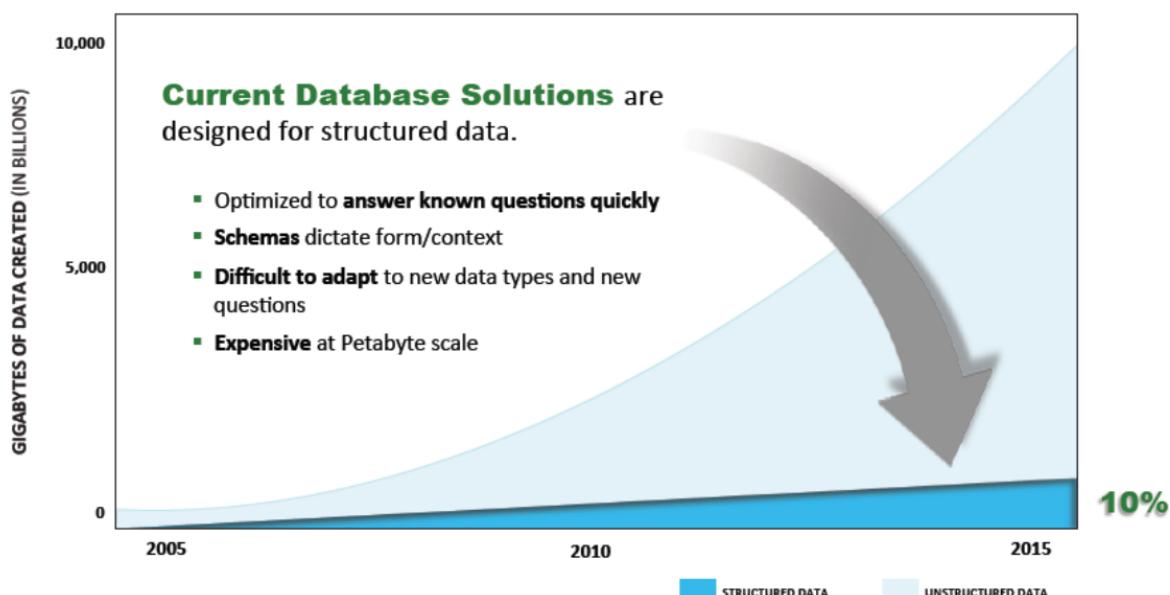
Veracity

Quality and provenance of received data
Good, Bad, Undefined, Inconsistency, Incompleteness, Ambiguity

- En los últimos años, la información ha crecido en gran medida:

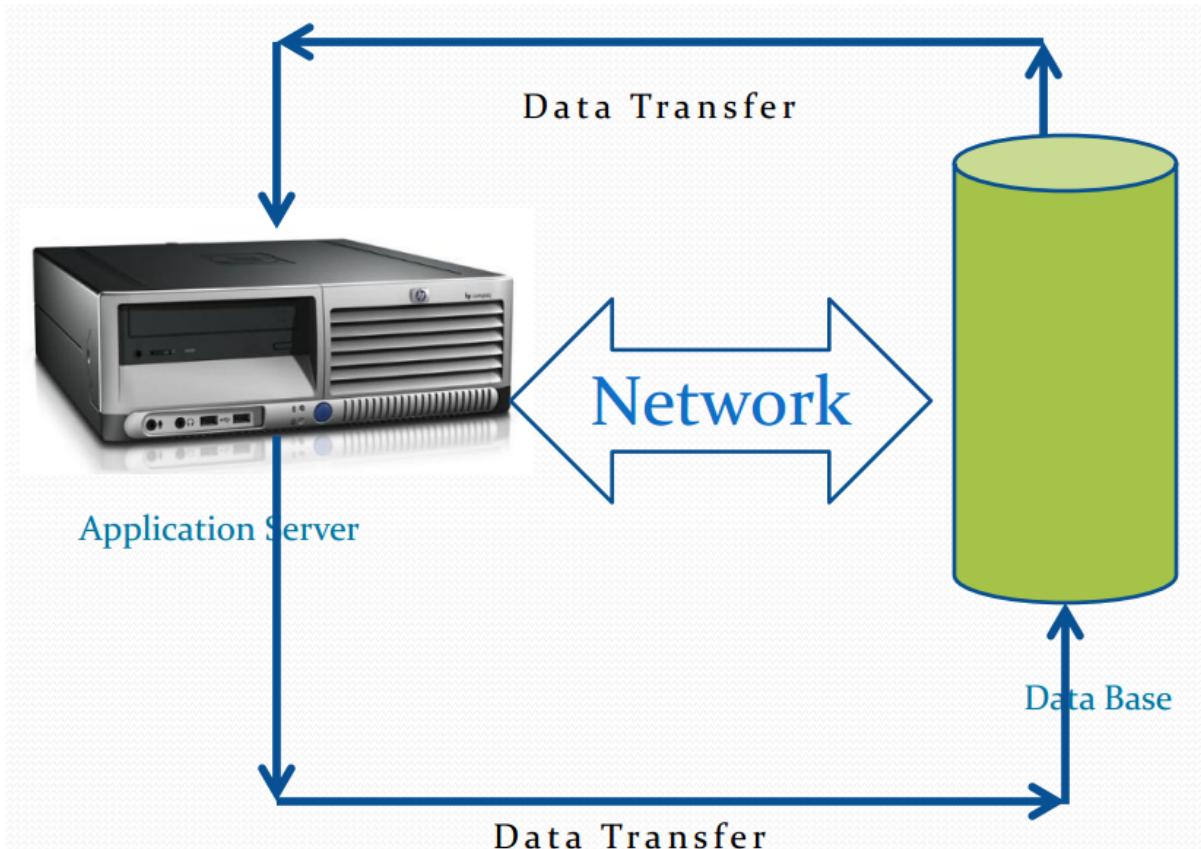


¿Qué aportan las soluciones tradicionales?



1.2. Arquitecturas

Aplicación estándar:



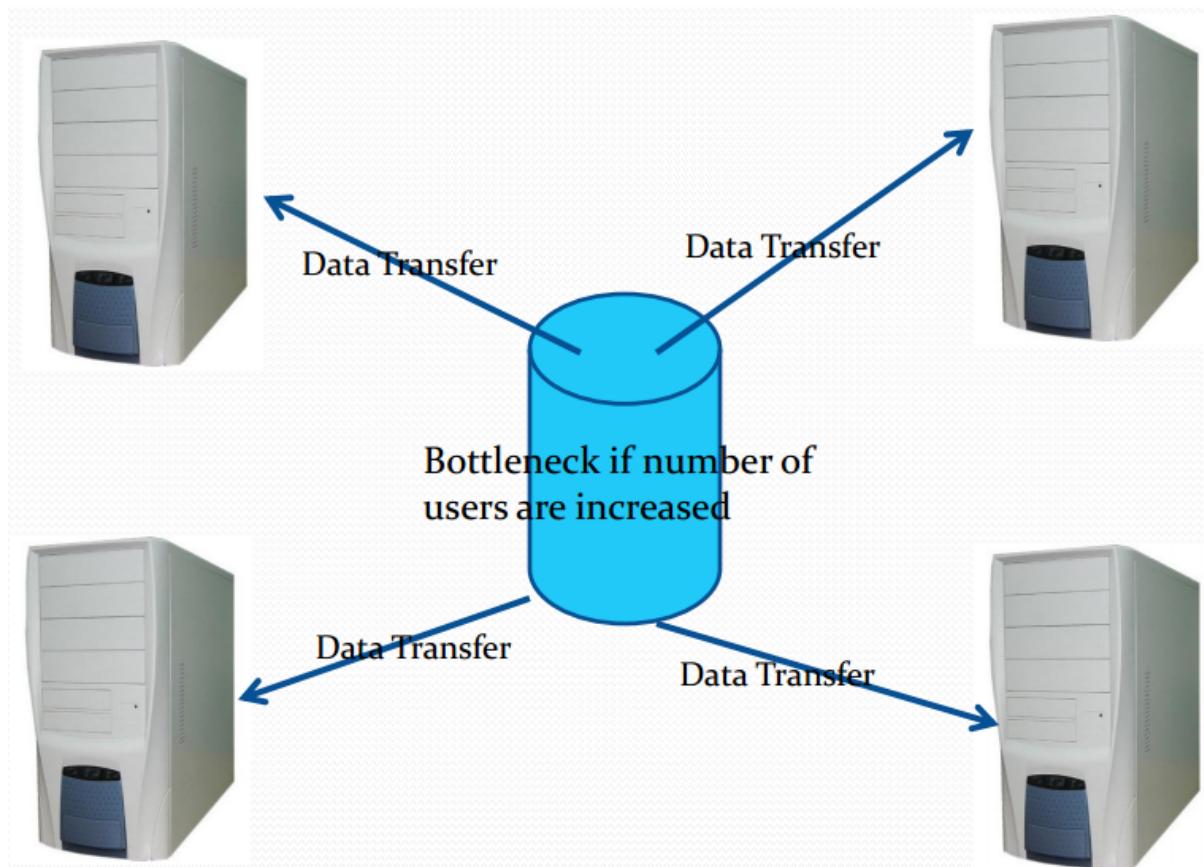
Problemas:

- No tenemos una red dedicada entre la aplicación y la base de datos
- El resto del tráfico de la compañía hace uso de la misma red
- No se controla la producción de la información
- El cambio del formato de los datos es problemático
- El tamaño de la base de datos tiene límites

Ante estas situaciones, en una aplicación normal nos encontramos con que:

- Gran parte del tiempo se consume en transferir la información para su proceso a través de la red
- Para mejorar el rendimiento, se mejora el hardware (más procesador, más ram...)
- La lectura física de datos es un cuello de botella, más datos para leer suponen más tiempo para hacerlo
- La capacidad de almacenamiento tiene un límite físico, al igual que la RAM de la máquina

El siguiente paso a la hora de trabajar fue el de gestionar un sistema distribuido:



Este nuevo enfoque permite:

- Tolerancia parcial a fallos → La caída de algunas máquinas supone una caída de rendimiento, pero no una parada de servicio
- Recuperacion → Una máquina caída puede volver al sistema tras realizar ciertas tareas
- Disponibilidad de la informacion → La caída de una máquina no significa la pérdida de información
- Consistencia → Los resultados han de ser los mismos independientemente de si ha habido caídas
- Escalabilidad → Se pueden añadir más máquinas para escalar el sistema

1.3. BBDD Relacionales

1.3.1. ¿Qué ventajas aportan los enfoques tradicionales de BBDD relacionales?

- Esquemas con tablas normalizadas
- Permite cruce de tablas (Joins)
- Cumple el estándar de transacciones ACID
 - Atomicity: Si una operación tiene varios pasos, se ejecutan todos o ninguno. (Transacciones)
 - Consistency: Se garantiza que la información presentada será siempre la misma
 - Isolation: Una operación no va a afectar a otras

- Durability: Los resultados de las operaciones van a persistir en el tiempo

1.3.2. ¿Y qué contrapartidas ofrecen los enfoques tradicionales de BBDD relacionales?

- Hacer uniones (Joins) de tablas grandes suponen un coste muy elevado, y una carga al sistema inasumible
- Las soluciones para escalado (Shardings) es compleja y frágil
- La disponibilidad del sistema no está asegurada

1.4. El mundo Big Data

1.4.1. ¿Qué enfoques aportan las BBDD NoSQL?

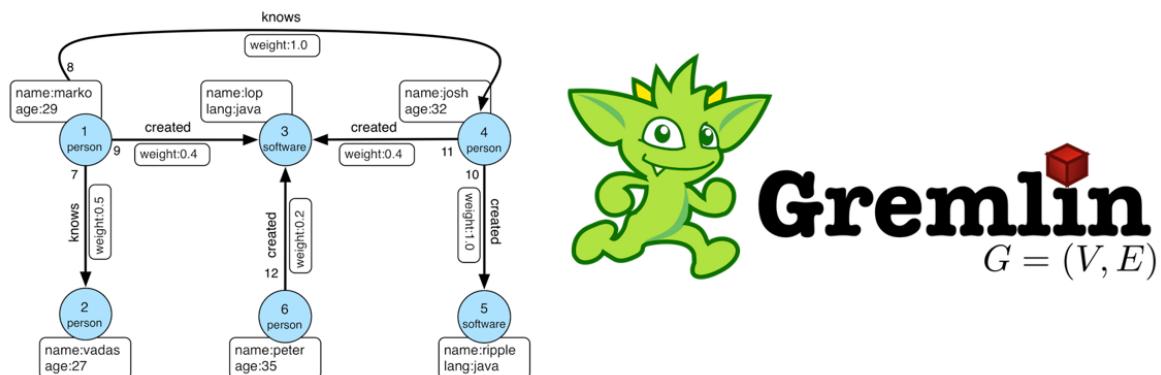
- Distribuye la información a través de múltiples nodos
- Es más laxo con la consistencia
- Es más laxo con los esquemas
- Busca optimizar la información para adaptarla a las necesidades actuales

Hay 4 grandes enfoques a las bases de datos no-relacionales:

- Grafos
- Clave-Valor
- Documento
- Column Family

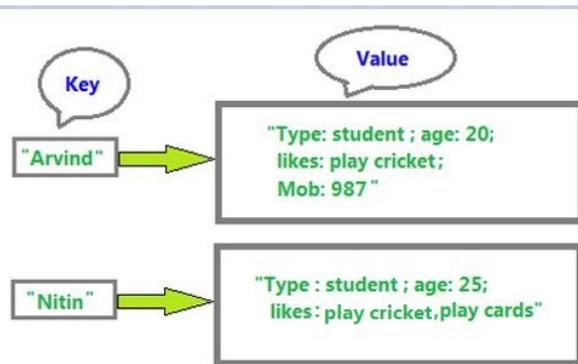
Grafos

- Los elementos están distribuidos como vértices de un grafo, siendo las uniones entre vértices las relaciones de ambos (con propiedades).

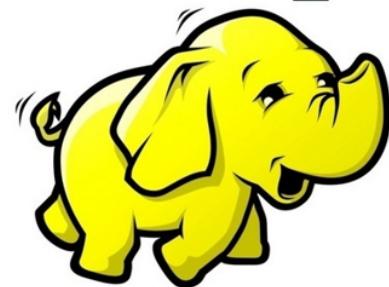


Clave-Valor

- Existen claves que enlazan a un conjunto de valores asociados a la misma de cualquier tipo

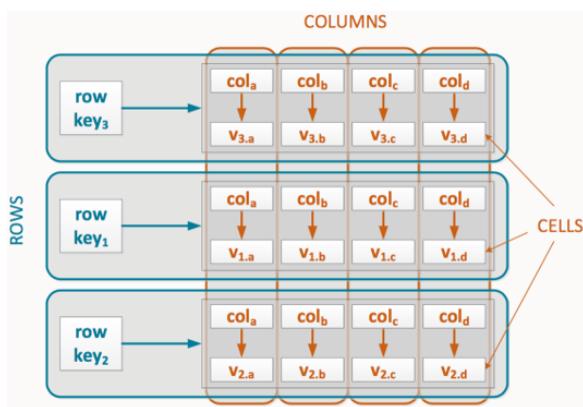


hadoop



Column Family

- Las claves enlazan con un conjunto de columnas



Documento

- Se almacenan conjuntos de documentos (JSON) que pueden ser consultados en su totalidad o en parte.

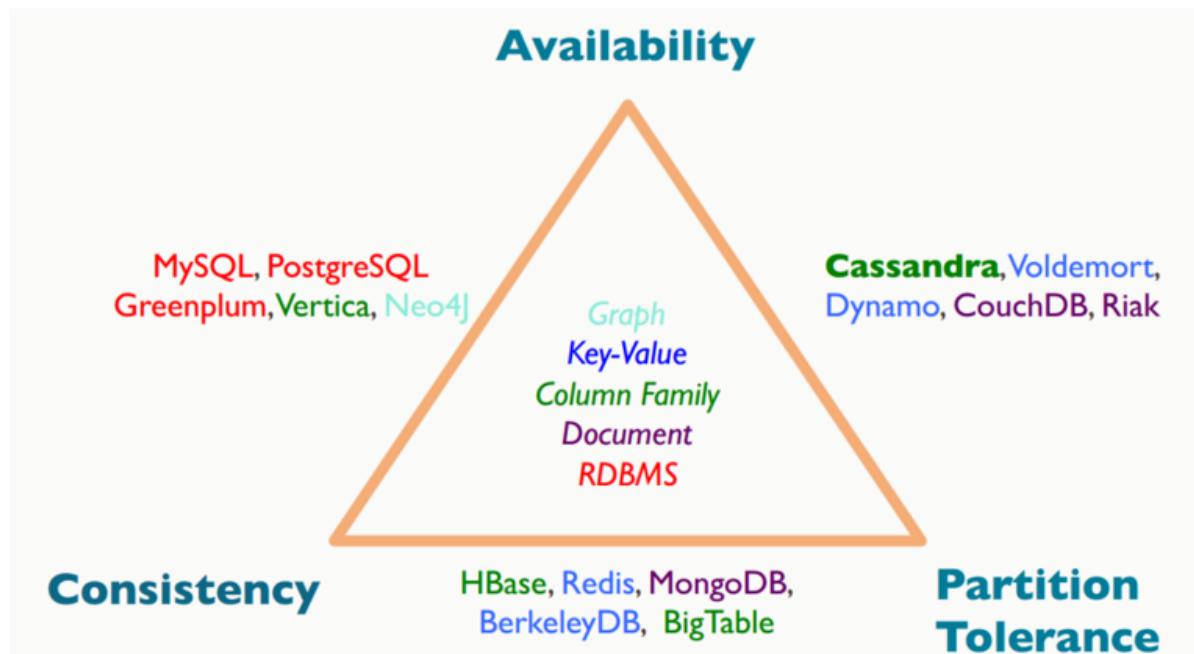
```
Students Document
{
  "_id": "5373aadadac133aad5b6660",
  "name": "kaushal patel",
  "email": "kp@example.com",
  "courses": {
    "course_name": "java",
    "fees": "5000",
    "duration": "3",
    "professor": "g.r."
  }
}
```



mongoDB

1.5. Teorema CAP

Ningún sistema puede tocar al mismo tiempo los 3 vértices del triángulo. Por lo tanto, para acercarse a algún vértice, deben alejarse de otro.



Chapter 2. Introducción a MongoDB

2.1. Historia

El nombre de **MongoDB** viene la la palabra inglesa **humongous**, cuyo significado es **enorme**.

Es un sistema de base de datos NoSQL, orientado a documentos, desarrollado bajo el concepto OpenSource

MongoDB guarda las estructuras de datos en documentos **BSON**, un tipo de documento que surge a raíz del **JSON**

Se desarrolló en 2007 por la compañía **10gen**

Esta tecnología está presente en múltiples entornos, como MTV, Craigslist, Foursquare...

MongoDB como compañía:



Leading Organizations Rely on MongoDB



2.2. Conceptos clave

En los últimos años, el desarrollo del hardware ha empezado a centrarse no sólo en la mejora de los componentes de manera aislada, si no en sistemas formados por varios servidores, y que a su vez poseen varios cores.

Estas arquitecturas que dominan el panorama actual (y este gran concepto que surge de **La nube**) no siempre son aprovechadas por los sistemas tradicionales, con lo que cada vez surgen más soluciones orientadas a explotar dichos entornos.

La finalidad, es poder escalar nuestro sistema sin que incremente el coste del mismo de manera notable.

Estas soluciones es lo que entendemos como **Big Data**.

Estos sistemas, no sólo están orientados a aprovechar estas arquitecturas, si no también a ser capaces de lidiar con tipos de datos no estructurados, o con una estructura dinámica, algo que es impensable para los sistemas **RDBMS**

Escalado

Uno de los primeros conceptos que debemos asimilar a la hora de trabajar con **MongoDB** es el concepto de el escalado.

Existen 2 tipos de escalados:

- Escalado Vertical → Para aumentar la capacidad de nuestro sistema ampliamos la máquina

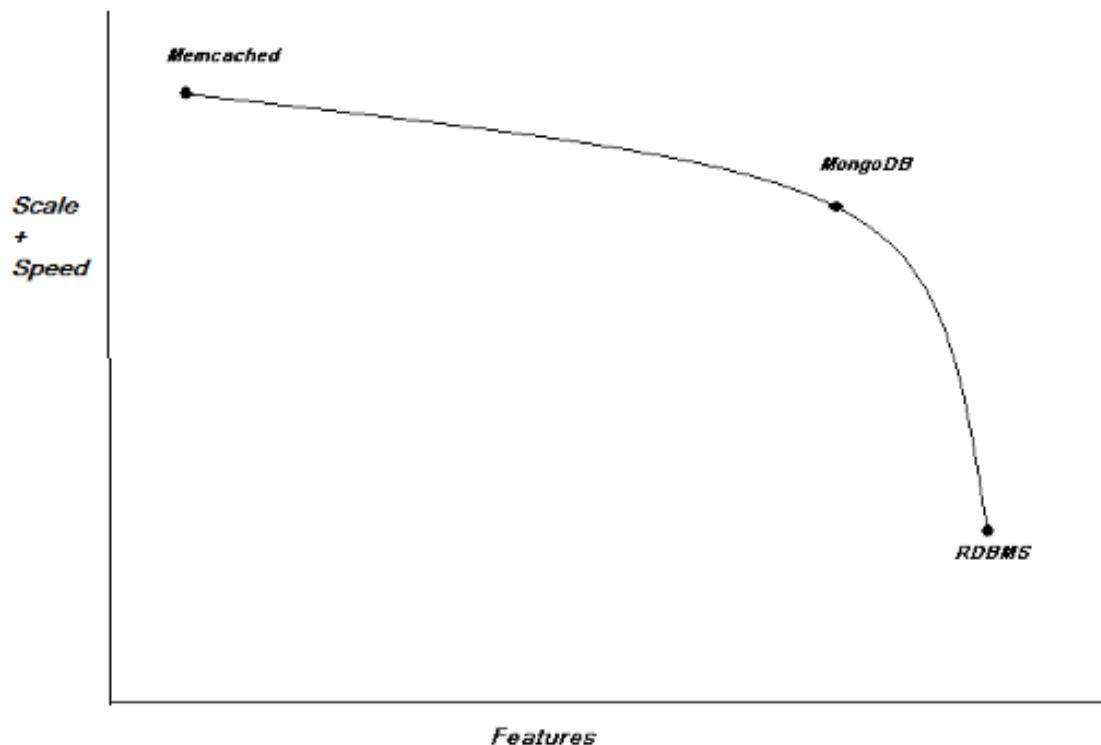
mejorando los componentes (añadiendo memoria, o cores, por ejemplo). Esto, aunque puede suponer una mejora del sistema, no evita una caída del servicio si en la máquina hay algún problema. Además, existe un techo a partir del cuál es muy caro subir.

- Escalado Horizontal → Para aumentar la capacidad de nuestro sistema, añadimos más máquinas. En principio esto es bastante económico, pero exige trabajo adicional, ya que debemos realizar labores de gestión para coordinar los servidores, saber cómo reaccionar a la caída de una máquina, asegurar la disponibilidad de la información en caso de caídas...

La ventaja de este último tipo de escalado, es que se puede realizar con el denominado **Commodity hardware**, es decir, máquinas que no son servidores especializados, si no ordenadores normales (aunque no nos dejemos engañar, deben ser buenas máquinas, del orden de 1000€-2000€ por máquina)

Pero ¿Añadir una máquina siempre mejora el rendimiento de nuestro sistema de la misma manera?

No, el escalado de un sistema depende de la tecnología que estemos usando, y esto es debido a que algunas **features** no obtienen una mejora de un escalado horizontal.



¿Qué decisión ha tomado **MongoDB**? fomentar el escalado eliminando aquellas **features** que lo dificultan, pero manteniendo gran parte de la funcionalidad original de una **RDBMS**.

¿Qué elimina entonces **MongoDB**?

- Los **Joins** → Una consulta que obtenga datos de varias tablas necesita recuperar toda la

información para poder trabajar con ella. Esto hace que no obtengamos un mejor rendimiento de distribuir los datos en varios servidores

- Las **Transacciones** → Si tras varias operaciones decidimos deshacer lo realizado, tendríamos que ponernos de acuerdo con todos los demás nodos afectados para dar marcha atrás, con el coste que ello supone.

Pongamos como ejemplo una tabla SQL muy grande, que está troceada en varios servidores para poder aprovechar un escalado.

Esta tabla tiene unas tablas de dimensiones, que debido a su limitado espacio están duplicadas en todos los servidores. Esta opción parece permitir un buen escalado del sistema, pero en cuanto se hiciera un **Join** de dicha tabla con otra, no aprovecharíamos las ventajas del escalado.

Así mismo, gestionar transacciones complejas tendría el mismo problema, puesto que una operación pesada que trabaje sobre varias tablas puede suponer una caída notable del rendimiento.

No hay solución a esto, ¿Qué hace **MongoDB**? pues no permite realizar ninguna de estas dos operaciones.

En **MongoDB** no existen **Transacciones ni Joins**

2.3. Documentos

Llegados a este punto, tenemos claro que **MongoDB** no posee ni **transacciones ni joins**. ¿Podemos considerarlo entonces un sistema relacional?

Desde el punto de vista estricto, **NO**.

Pero podemos usar un **data model** que nos permita trabajar con estructuras de tipo **clave/valor**, que no es relacional, pero se acerca.

Es como trabajar con un **HashMap**, tenemos una operación **Get(K)** para obtener el valor asociado a una clave, y otra **Set(K,V)** para añadir/modificar el valor asociado a una clave.

¿Qué vamos a usar para tal fin? la notación **JSON**.

Una notación **JSON** es independiente del lenguaje, y es de hecho muy popular (hay muchos lenguajes preparados para trabajar con documentos de tipo **JSON**).

Es decir, la información almacenada en un documento tipo **JSON** es fácilmente extraíble, y por lo tanto una opción muy versátil.

Un ejemplo de objeto **JSON** sería el siguiente

```
{"nombre":"Marta","edad":33,"apellidos":"Domingo Santos"}
```

¿Cómo podemos trabajar entonces aproximándonos a un sistema **RDBMS**?

Usuarios

Nombre	Edad	ID
Pepe	30	1

Préstamos

ID	Libro
1	3
1	5

```
Select * from usuario, préstamos  
where usuario.id=1  
and préstamos.id=usuario.id;
```

Pues creando un documento **JSON** cuya información sea equivalente, por ejemplo:

```
{_id:1, Edad:30, Nombre:"Pepe", Prestamos:[3,5]}
```

Obtener esta información sería tan sencillo como buscar por el campo **_id** de dicho documento

```
db.users.find({_id:1})
```

- En realidad, **MongoDB** hace uso de **BSON**, acrónimo de **Binary JSON**, lo cuál posee implicaciones propias, que estudiaremos más adelante.

2.4. Instalación

2.4.1. Windows

Para descargar **MongoDB** podemos ir a su página oficial y elegir el paquete que queremos usar según nuestro S.O. y paquete que queremos

<https://www.mongodb.com/download-center>

The screenshot shows the MongoDB download center homepage. The top navigation bar includes links for SOLUTIONS, CLOUD, CUSTOMERS, and RESOURCES. Below the navigation, there are five main sections: Community Server (highlighted in green), Enterprise Server, Ops Manager, Compass, and Connector for BI. The 'Community Server' section is expanded, showing the 'Current Stable Release (3.2.10)'. It includes a release date (09/30/2016), release notes, and changelog links. It also lists supported operating systems: Windows, Linux, OSX, and Solaris. A dropdown menu for 'Version' shows 'Windows Server 2008 R2 64-bit and later, with SSL support x64'. A large green button labeled 'DOWNLOAD (msi)' is prominently displayed. At the bottom of the page, there are links for 'Current Release', 'Previous Releases', and 'Development Releases'.

Una vez descargado e instalado, si se ha optado por una instalación manual, habrá que añadir a la variable de entorno **PATH** la ruta donde se encuentran los binarios de **MongoDB**, que por defecto es en

C:\Program Files\MongoDB\Server\3.XX\bin

(siendo XX la subversión que hayamos instalado)

Una vez hecho esto, podremos invocar a los comandos de **MongoDB** desde cualquier ruta. Los dos comandos que más vamos a usar son:

- **mongod** → Levanta el servidor **MongoDB**
- **mongo** → Shell que se conecta al servidor especificado

2.4.2. Linux RPM

Para descargar MongoDB e instalarlo en una distribución Linux basada en RedHat como CentOS (Mismo kernel que la versión oficial de pago), debemos agregar el repositorio oficial de MongoDB:

```
echo "[mongodb-org-3.6]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/amazon/2013.03/mongodb-org/3.6/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-3.6.asc" | sudo tee /etc/yum.repos.d/mongodb.repo
```

NOTE

Este fichero se ha agregado ya al repositorio de yum de la máquina virtual entregada

Instalamos la última versión estable de MongoDB

```
sudo yum -y install mongodb-org mongodb-org-shell
```

2.5. La primera vez

Una vez instalado **MongoDB**, vamos a realizar una primera toma de contacto. Para ello vamos a realizar dos acciones:

- Crear el directorio donde **MongoDB** almacenará la información
- Levantar el servidor con los parámetros necesarios para que trabaje usando como directorio de almacenamiento el especificado anteriormente
 - Creamos el directorio para guardar los datos.

Suponemos que vamos a trabajar en un directorio concreto (por ejemplo **C:\MongoPruebas**). Sobre este directorio crearemos la ruta donde volcará los datos **MongoDB**

Versión de Windows

```
cd C:\MongoPruebas  
md data  
md data\db
```

Versión de Linux

```
mkdir -p data/appdb
```

- Levantamos el servidor **MongoDB**, indicándole desde dónde leer los datos (Damos por supuesto que nos encontramos en **C:\MongoPruebas**)

Versión de Windows

```
mongod --dbpath data\db
```

Versión de Linux

```
mongod --dbpath data/db
```

Una vez hecho esto, lanzamos el shell de **MongoDB** para acceder por primera vez

```
mongo
```

Y una vez dentro, insertamos un documento **JSON** y pedimos que lo muestre posteriormente

```
db.names.insert({'name':'Nombre Guay'})  
db.names.find()
```

¿Qué aparece al hacer el find? ¿Por qué?

2.6. El formato **JSON**

MongoDB es un tipo de sistema **NoSQL** orientado a documentos.

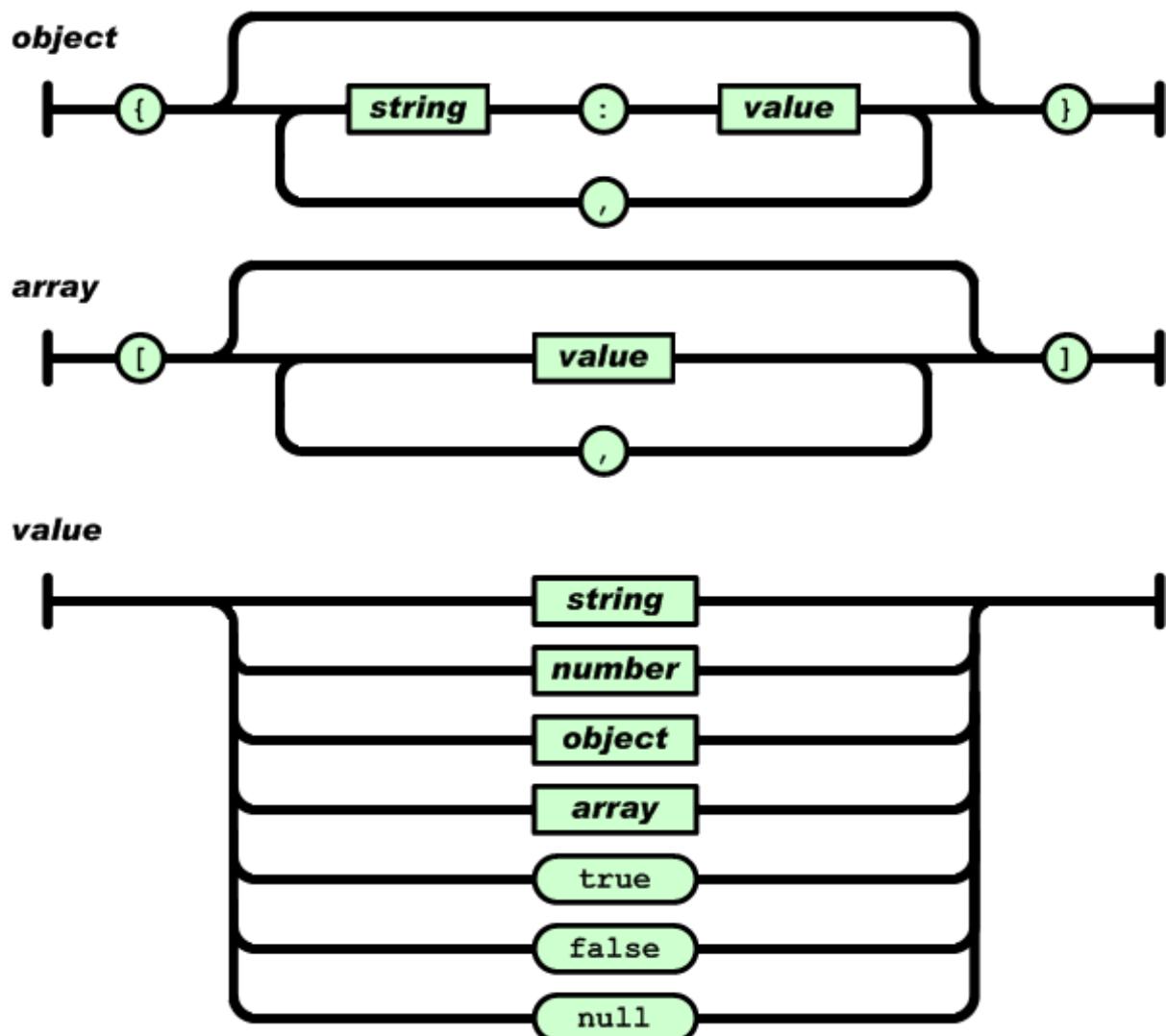
La razón por la que **MongoDB** ha escogido el documento tipo **JSON** y no otro (como por ejemplo **XML**) atiende a dos motivos:

- Los objetos **JSON** son más ligeros que sus equivalentes en **XML**
- Son más **legibles** por parte de los humanos

La sintaxis de un documento **JSON** puede ser consultada directamente desde la web <http://www.json.org/>

Un documento JSON es un conjunto de tuplas **clave/valor**, donde el **valor** puede ser de uno de los siguientes tipos:

- String → Cadena de texto
- Number → Número, tanto decimal como entero
- Boolean → Valores **true** y **false**
- Null → Tipo especial **null** para especificar un campo sin valor
- Array → Un array de elementos
- Object → El valor puede ser también otro objeto JSON, con lo que existen documentos embebidos



Un ejemplo de un objeto JSON

```
{
    "name": "Pepe", //String
    "age": 27, //Number
    "married": true, //Boolean
    "School": null, //Null
    "likes": ["Tennis", "Football"], //Array
    "Address": { //Document
        "city": "Madrid",
        "country": "Spain"
    }
}
```

Unas consideraciones a tener en cuenta:

- Los campos deben empezar con letra
- Nombres como # y 2323 no son válidos como tal, pero si son válidos si se escapan, es por ello que se recomienda siempre el uso de las comillas
- Un **JSON** es un array asociativo, donde la clave es el nombre del campo, y valor el contenido del mismo

Ejercicio:

Convierte el siguiente documento **XML** en un **JSON** equivalente

```
<person>
    <name>John</name>
    <age>25</age>
    <address>
        <city>New York</city>
        <postalCode>10021</postalCode>
    </address>
    <phones>
        <phone type="home">212-555-1234</phone>
        <phone type="mobile">646-555-1234</phone>
    </phones>
</person>
```

Solución:

```
db.person.insert(  
{  
    "name" : "John",  
    "age" : 25,  
    "address" : { "city" : "New York", "postalCode" : "10021" },  
    "phones" : [ {"phone":"212-555-1234", "type" : "home"},  
                {"phone":"646-555-1234", "type" : "mobile"} ]  
})
```

2.7. El formato BSON

BSON es un estándar cuya especificación se encuentra en <http://bsonspec.org/>

Su finalidad es pasar el documento **JSON** a formato binario para mejorar la eficiencia, y con ello obtener:

- Una lectura más rápida
- Mejora de eficiencia al hacer uso de data types concretos

Un ejemplo claro se puede entender si nos fijamos en el siguiente documento:

```
{_id:"XX",  
 name:"James",  
 prefs:{.....},  
 active:true}
```

Imaginemos que el documento anidado en **prefs** tiene un tamaño considerable, y no nos interesa, por lo que queremos saltarlo. En un **JSON** no quedan más opciones que leerlo para saber dónde acaba, sin embargo, en un **BSON** sabemos cuánto ocupa ese campo, y por lo tanto podemos saltar esa cantidad de Bytes y así ser más eficientes en dicha lectura.

Aunque internamente **MongoDB** hace uso de **BSON**, es el propio driver el que se encarga de gestionarlo, por lo que esto siempre será transparente para nosotros.

Imaginemos que tenemos un documento simple como el mostrado a continuación:

```
{a:3,b:"XYZ"}
```

¿Cómo se guardaría en disco dicha información?

En primer lugar, se establece el formato en **BSON**, especificando qué tipo de dato contiene cada campo, como se muestra a continuación

Tamaño documento	Tipo	Nombre campo	Valor campo	Tipo	Nombre campo	Valor campo	Valor	Fin
	int32	a\0	3	str	b\0	4	XYZ\0	\0

Ahora, rellenamos cuánto ocupa cada campo teniendo en cuenta que:

- El campo Length of document es un entero de 32 bits, por lo que ocupa 4 Bytes
- Cada tipo ocupa 1 Bytes
- Cada carácter de un texto ocupa 1 Bytes (Y tienen que acabar con el carácter especial '\0')
- El número empleado para guardar el tamaño de una cadena de texto es un int32 (4 Bytes)
- El documento cierra con el carácter '\0', de 1 Bytes

Quedaría tal como se muestra

Tamaño documento	Tipo	Nombre campo	Valor campo	Tipo	Nombre campo	Valor campo	Valor	Fin
	int32	a\0	3	str	b\0	4	XYZ\0	\0
4 Bytes	1 B	2 B	4 B	1 B	2 B	4 B	4 B	1 B

Con estos datos, ya podemos calcular el tamaño total en disco de nuestro **BSON**

Tamaño documento	Tipo	Nombre campo	Valor campo	Tipo	Nombre campo	Valor campo	Valor	Fin
23	int32	a\0	3	str	b\0	4	XYZ\0	\0
4 Bytes	1 B	2 B	4 B	1 B	2 B	4 B	4 B	1 B

Cabe resaltar que el formato **BSON** es más eficiente no sólo porque nos permita accesos a disco mejorados, si no porque para ciertas operaciones, no hay que solicitar más espacio.

Por ejemplo, una operación de incremento de un número en un objeto **JSON** puede requerir un dígito más, lo cuál exige desplazar todo el documento para crear un espacio para dicho dígito.

Con **BSON**, se puede hacer el incremento **in situ**, ya que sólo hay que reemplazar el valor binario de dicho número.

Evidentemente con un texto, esto sigue pasando, aunque **MongoDB** usa ciertas estrategias para minimizar el impacto de esto (como solicitar siempre un espacio extra para dichos campos, para permitir su crecimiento posterior))

¿Cómo funcionan las aplicaciones que trabajan con **MongoDB**?

MongoDB Server guarda los documentos en **BSON**, cuando una aplicación realiza una query, el server lo devuelve así, en formato **BSON** (aunque al enviarlo este es decorado con una cabecera, pero por el resto se envía el documento tal cual se almacena en disco)

La aplicación cliente guarda esto en la ram, y es el **Mongo Driver** el encargado de realizar la transformación de este binario en una representación para el tipo de lenguaje que usemos, por ejemplo un objeto Java (por lo general se transforma en un **JSON**).

MongoDB posee dos utilidades que usaremos a lo largo de este curso que permiten pasar de **JSON** a **BSON** y viceversa, son:

- mongoimport → Permite enviar en formato **BSON** objetos que están en formato **JSON**, para poder almacenarlos directamente en **MongoDB**.
- mongoexport → Permite traer objetos **BSON** y traducirlos a su equivalente **JSON**, para poder usarlos posteriormente con otras herramientas.

Hay que tener muy claro dos cosas

- Lo que usa internamente **MongoDB** y lo que almacena son **BSON**
- Lo que se transmite vía red **sigue siendo objeto BSON**, con lo que la transmisión de dichos datos es eficiente. El driver es el encargado de coger el **BSON** de memoria y traducirlo a **JSON** (o lo que sea necesario)

2.8. Esquema dinámico

MongoDB se define como una Base de Datos NoSQL **Schemaless**, es decir, sin estructura definida.

En realidad, no es cierto que no posea estructura, y de hecho precisa estructura para realizar tareas críticas para el escalado como es el **Sharding**, pero esta estructura es muchísimo más flexible que la que posee un sistema **RDBMS**, permitiendo su alteración constante.

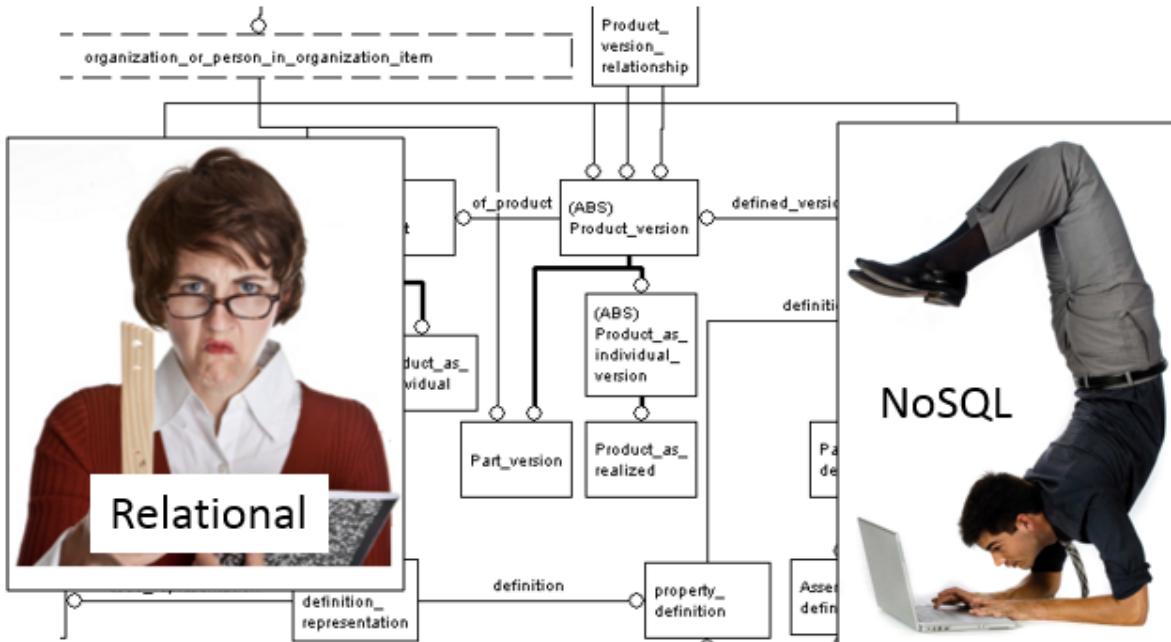
Es por ello que es mucho más adecuado hablar de **MongoDB** como una base de datos que posee un esquema dinámico.

¿Qué significa entonces decir que posee un esquema dinámico?

Pues muy sencillo, los esquemas se resuelven (y se pueden crear y modificar) en tiempo de ejecución. En una Base de Datos relacional, los esquemas deben estar definidos antes de trabajar (si quieras guardar algo en una tabla, dicha tabla y su base de datos deben existir).

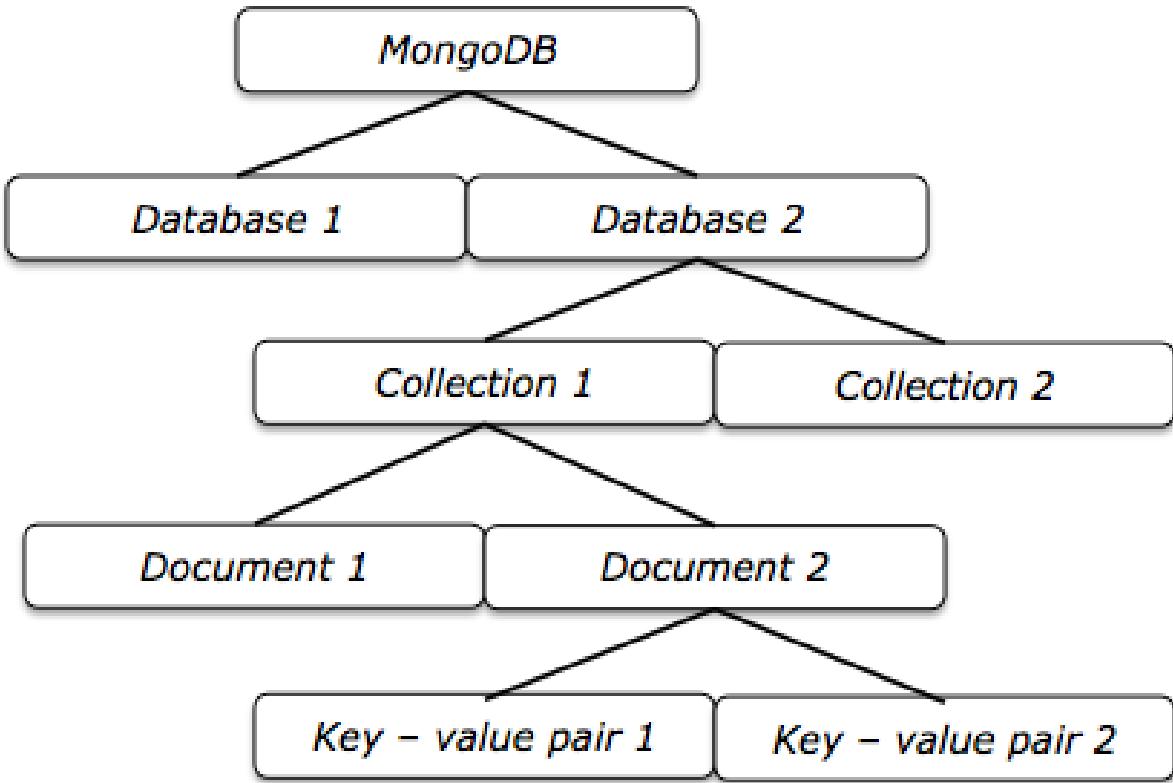
En **MongoDB**, aunque es posible definirlos antes de trabajar con ellos, no es obligatorio, y gracias a esto obtenemos una serie de ventajas:

- Mucha mayor flexibilidad.
- Permite trabajar de manera mucho más ágil, y realizar iteraciones de forma mucho más sencilla.
- Permite polimorfia.



En resumidas cuentas:

- MongoDB posee un servidor que posee una serie de **Bases de datos**, que se pueden crear de manera dinámica.
- Estas **Bases de datos** son un conjunto de **colecciones** (que serían el equivalente a tablas en RDBMS).
- Una **colección** es un conjunto de documentos.
- Los **documentos** no tienen por qué tener la misma estructura.
- Se puede crear una **subcolección**, que no es otra cosa que forma de organizar las **colecciones**, pero es a todos los efectos una **colección**



2.9. mongoimport/mongoexport

Como ya sabemos, **MongoDB** hace uso de documentos **JSON** a la hora de trabajar, pero internamente los gestiona como **BSON**.

Esto implica que los datos que guardamos dentro de **MongoDB** no sean legibles directamente, y que por lo tanto necesitaremos utilidades adicionales para extraer **JSON** de la base de datos y viceversa.

Es muy importante recordar que, aunque estas utilidades reciben o devuelven objetos **JSON**, internamente también trabajan con **BSON**, por lo que en todo momento, la información que viaja por la red es de tipo **BSON**.

Para esta finalidad disponemos de dos herramientas:

- **mongoexport** → Permite extraer datos de **MongoDB** y volcarlos a un fichero como tipo **JSON**
- **mongoimport** → Permite procesar un fichero con datos tipo **JSON** y volcarlos directamente en **MongoDB**

mongoexport se lanza por línea de comandos, tal como se muestra en el siguiente ejemplo

```
mongoexport --db test --collection colección --out salida.json
```

Esta invocación realizaría la descarga de objetos **JSON** existentes en la colección **colección** de la base de datos **test**, sobre el fichero destino **salida.json**

Las opciones de invocación más importantes son:

- **--db** → Base de datos sobre la que queremos trabajar

- --collection → Colección sobre la que queremos trabajar
- --out → Fichero sobre el que queremos volcar los datos
- --host → Host:puerto (siendo puerto opcional) al que queremos atacar
- --port → Puerto al que queremos atacar
- --type → Tipo de fichero a generar (json|csv|tsv)
- --fields → Listado (separado por comas) de campos a exportar
- --query → Especifica la query para la extracción de datos (como se especificaría dentro de un find)

mongoimport se lanza por línea de comandos, los siguientes ejemplos serían válidos

```
mongoimport --db test --collection colección --file entrada.json
mongoimport --db test --collection colección < entrada.json
```

Esta invocación realizaría la carga de objetos **JSON** en la colección **colección** de la base de datos **test**, desde el fichero origen **entrada.json**

Las opciones de invocación más importantes son:

- --db → Base de datos sobre la que queremos trabajar
- --collection → Colección sobre la que queremos trabajar
- --file → Fichero sobre el que queremos cargar los datos
- --host → Host:puerto (siendo puerto opcional) al que queremos atacar
- --port → Puerto al que queremos atacar
- --type → Tipo de fichero a generar (json|csv|tsv)
- --drop → Borra la colección destino antes de la carga
- --headerline → Si trabajamos con ficheros csv o tsv usa la primera línea del documento como nombre de los campos (si no, considera la primera línea un documento)
- --upsertFields → Si el documento existe y queremos actualizarlo, pero lo identificamos por otro campo que no sea su **_id**

2.10. Interactuando con MongoDB

Vamos a conocer las bases necesarias para movernos por **MongoDB** como pez en el agua.

En primer lugar, hay que acordarse de arrancar la base de datos, con el siguiente comando (siendo **data\db** en este caso la ruta donde almacenará los datos):

```
mongod --dbpath data\db
```

Una vez hecho esto, nos podemos conectar mediante el comando **mongo**

```
mongo
```

Esto, por defecto, nos va a conectar a la máquina situada en **localhost**, al puerto **27017**, y a la base de datos **test**.

Evidentemente, podemos especificar una base de datos distinta, o un host/puerto distinto.

```
mongo otraBBDD  
mongo host:puerto/otraBBDD
```

Una vez dentro, podemos cambiar de base de datos mediante el comando **use** seguido de la base de datos destino

```
use test
```

Podemos consultar las bases de datos existentes con el comando **show databases**, pero recordad que **MongoDB** es **schemaless**, por lo que podemos ir a una base de datos **que todavía no exista** y en cuanto trabajemos en ella, se creará automáticamente.

Un punto importante es saber que **mongo** carga por defecto un script llamado **.mongorc.js**, si queremos añadir funciones o variables para su uso, este es un buen lugar.

Algunas de las opciones más importantes a la hora de arrancar **mongo** son:

- **--shell** → Ejecuta el script que le pasemos como parámetros
- **--nodb** → Arranca la consola sin conectarla a ningún **mongod**
- **--norc** → Si no queremos que se cargue automáticamente el fichero **.mongorc.js**
- **--quiet** → Reduce la salida de mensajes
- **--eval** → Ejecuta la instrucción **Javascript** pasada como parámetro
- **--verbose** → Aumenta la salida de mensajes

2.11. Cursosores

El shell de **MongoDB** es un intérprete de **Javascript**.

Esto nos permite realizar scriptings sobre la BBDD, y posee además una serie de variables predefinidas (como db).

Si hacemos una query como **db.products.find()** nos devuelve resultados (en caso de existir dicha colección en nuestra **BBDD**), pero si nos fijamos en profundidad, veremos que tan sólo nos devuelve 20 elementos.

Podemos solicitar los próximos 20 elementos mediante el comando **it..**

Esto es porque cuando nosotros lanzamos una **query** contra **MongoDB**, el sistema no devuelve de

golpe todo los valores, devuelve un **cursor** para poder recorrerlos.

Cuando voy a hacer una consulta de dicho tipo, puedo almacenar el **cursor** en una variable, y trabajar con él mediante **Javascript**. Para ello usaremos los métodos:

- `hasNext()` → Indica si posee más datos el cursor
- `next()` → Nos devuelve el siguiente elemento.

Pongamos por ejemplo que tenemos una colección de elementos (que vamos a crear directamente en nuestra **BBDD** con el siguiente comando desde la shell **mongo**)

```
for (var i=0;i<1000;i++){
    db.cursos.insert({x:i});
}
```

Ahora vamos a recorrer todos los documentos y mostrar sólo los cuyo valor para x sea par

```
var c=db.cursos.find()
while (c.hasNext()){
    var temp=c.next();
    if (temp.x%2==0){
        printjson(temp);
    }
}
```

Ya hemos visto que cuando lanzamos una **query** contra **MongoDB** no se devuelve de golpe todos los datos, obtenemos un cursor.

Algunas funciones útiles para poder trabajar de una manera un poco distinta son:

- `.skip(x)` → Salga los x primeros valores devueltos.
- `.limit(x)` → Limita la respuesta a x elementos.
- `.toArray()` → Devuelve todos los elementos como un **JSONArray** (No devuelve un cursor).

Partiendo de la colección vista anteriormente, ¿Qué devolverá la query mostrada a continuación?

```
db.cursos.find().skip(3).limit(2)
```

Respuesta:

Los documentos con los valores de x 3 y 4.

Hay un pequeño truco aquí, el orden en el que se devuelven los elementos determina el resultado. En este caso el orden es el considerado **natural** o de inserción, no confundir con el orden de los valores de los propios elementos.

2.12. Inserciones

Para insertar un **documento** en **MongoDB** usamos la función **insert()**.

A la hora de insertar hay que especificar qué **colección** va a recibir el documento (recordar, la variable **db** contiene la **base de datos** actual)

El uso de **insert()** es muy sencillo, tan sólo hay que pasarle el documento que queremos que inserte (y si la **colección** no existe, se crea automáticamente). De hecho, si nos vamos a una **BBDD** que no exista, esta también se crea.

En el siguiente ejemplo insertamos un docuemnto en una **BBDD** que no existe, y en una **colección** que tampoco existe

```
use nuevaBBDD
db.nuevacoleccion.insert({x:1,b:"algo"})
```

Cada documento insertado en **MongoDB** ha de poseer un campo **_id**.

Este campo sirve de identificador exclusivo del documento, y puede ser proporcionado por el usuario explícitamente como en el siguiente ejemplo:

```
db.foo.insert({_id:1,b:"algo"})
```

Si no lo especificamos, el sistema crea el campo con un valor de tipo **ObjectID**, que es un identificador único.

```
db.foo.insert({x:0})
```

El documento tendría la forma de:

```
{ "_id" : ObjectId("581c6542bba504f1d31b499e"), "x" : 0 }
```

2.13. Query Language Bases

Ya hemos visto como trabaja **MongoDB** con cursores, pero trabajando directamente con el grueso de una colección.

Por lo general, no vamos a querer trabajar de esta manera, si no que realizaremos búsquedas de documentos concretos, o en base a unos valores. También es posible que no nos interese el documento por completo, si no tan sólo unos campos concretos.

En esta sección vamos a intentar conocer cómo trabajar correctamente con **MongoDB**, ya que las **queries** son **Data driven**, es decir, orientadas a obtener resultados en función de los datos.

Antes de continuar conviene tener en cuenta una cosa, cuando lanzamos una **query**, esta se va a ejecutar siempre del lado del servidor (evaluación perezosa), es decir, si ponemos algo como

```
db.cursores.find().skip(3).limit(2)
```

Es el servidor quien se salta 3 registros y coge sólo dos, **nunca** es el cliente **mongo**.

Aclarado este punto, vamos a ver cómo funciona la función **find()**

Esta función puede recibir dos parámetros, que deben ser documentos **JSON**.

- El primer parámetro es la consulta, la **query** que queremos lanzar en el servidor
- El segundo parámetro es la **proyección**, cómo queremos recibir los datos extraídos

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 } )  
.limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

Vamos empezar conociendo cómo funciona la **proyección** en **MongoDB**. Para ello vamos a trabajar con un documento vacío a la hora de lanzar queries con **find()**, ya que todavía no hemos visto cómo cambiar dichas **queries**

Una **Proyección** define qué campos vamos a querer extraer de los documentos devueltos en una **query**.

Para ello, tan sólo hay que enviar un documento con los campos que queremos mostrar u ocultar. Se siguen las siguientes reglas:

- Los campos enviados en dicho documento sólo pueden contener valor **1** o **0**
- Si contienen un **1** quiere decir que queremos mostrarlo, si contienen un **0** que no queremos
- Un documento que sólo contiene exclusiones (campos con valores **0**), muestra todos los campos que no se han especificado
- Un documento que contiene al menos un campo de inclusión, sólo muestra los campos incluídos
- El campo **_id** se muestra **siempre**, salvo que se excluya explícitamente.

La mejor manera de entender esto es mediante ejemplos. Partimos de una colección **misi** que posee un documento como el mostrado a continuación:

```
{_id:1,campo1:1,campo2:2,campo3:3}
```

Vamos a lanzar los siguientes ejemplos, ¿Qué obtendremos?

```
db.misi.find({},{})
db.misi.find({},{"campo1":0})
db.misi.find({},{"campo2":1})
db.misi.find({},{"_id":1})
db.misi.find({},{"_id":0})
db.misi.find({},{"_id":0,"campo1":1})
```

Resultado:

- db.misi.find({},{}) → { "_id" : 1 "campo1" : 1, "campo2" : 2, "campo3" : 3 }
- db.misi.find({},{"campo1":0}) → { "_id" : 1, "campo2" : 2, "campo3" : 3 }
- db.misi.find({}, {"campo2":1}) → { "_id" : 1, "campo2" : 2 }
- db.misi.find({}, {"_id":1}) → { "_id" : 1 }
- db.misi.find({}, {"_id":0}) → { "campo1" : 1, "campo2" : 2, "campo3" : 3 }
- db.misi.find({}, {"_id":0,"campo1":1}) → { "campo1" : 1 }

Entendido ya cómo funciona la proyección, vamos a pasar al primer parámetro de la función **find()**, la **query**.

La **query** es un documento **JSON** que sirve para filtrar la información que queremos buscar en la colección.

Este documento, contiene una serie de tuplas clave/valor, y nos va a devolver como resultado todos los documentos que cumplan con dichos criterios

Por poner un ejemplo, queremos buscar en la colección **misi** aquel documento que tenga en **campo1** el valor **1**, podríamos hacerlo de la siguiente manera

```
db.misi.find({"campo1":1})
```

Recordad, que esto es combinable con la proyección, imaginemos que sólo nos interesa el valor **campo2** y no queremos ver el **_id** del documento:

```
db.misi.find({"campo1":1}, {"_id":0, "campo2":1})
```

La función **find()** posee múltiples comandos para realizar búsquedas más complejas, las veremos en el tema dedicado al **CRUD**, pero mencionaremos algunas de ellas por tener una visión general:

- \$gte → Mayor o igual
- \$in → Contiene un valor concreto en el array
- \$type → Verifica tipo de dato
- \$gt → Mayor qué
- \$lt → Menor qué

- \$lte → Menor o igual
- \$or → Or lógico
- \$not → Negación
- \$exists → El campo debe existir
- \$nin → Not In

Algunos ejemplos de uso de estos comandos (que estudiaremos más adelante) serían los siguientes

Busca documentos cuyo campo **price** sea mayor o igual a 200

```
db.coleccion.find({price:{$gte:200}},{name:1,price:1})
```

Busca documentos que contengan el campo **campo**

```
db.col.find({campo:{$exists:true}})
```

Busca documentos que contengan el campo **a** con valor 3, o el campo **b** con valor 5

```
db.misi.find({$or:[{a:3},{b:5}]})
```

2.14. Sorting

A la hora de recibir la información solicitada a **MongoDB**, podemos solicitar recibir los resultados en un orden concreto.

Para tal finalidad podemos hacer uso de la función **sort()**, cuya sintaxis es muy simple, recibe un documento con los campos sobre los que quiere ordenar los resultados, y los valores de dichos campos pueden ser: **1** → En caso de orden ascendente **-1** → En caso de orden descendente

Un ejemplo de ordenación sería el siguiente:

```
db.foo.find().sort({a:1,b:-1})
```

A la hora de realizar una ordenación hay que tener en cuenta varias cosas:

- La ordenación se realiza en el servidor (evaluación perezosa), es por ello que **mongo** va a recibir los datos **YA** ordenados
- Si no existe un campo concreto, se considera que tiene valor **null**
- Si el campo contiene diferentes tipos de datos para distintos documentos, se impone el orden de tipos (**null** va por delante de **number**)

Para evitar por ejemplo que ordene los **null** por delante de otros datos, podemos filtrar previamente los resultados eliminando dichos documentos:

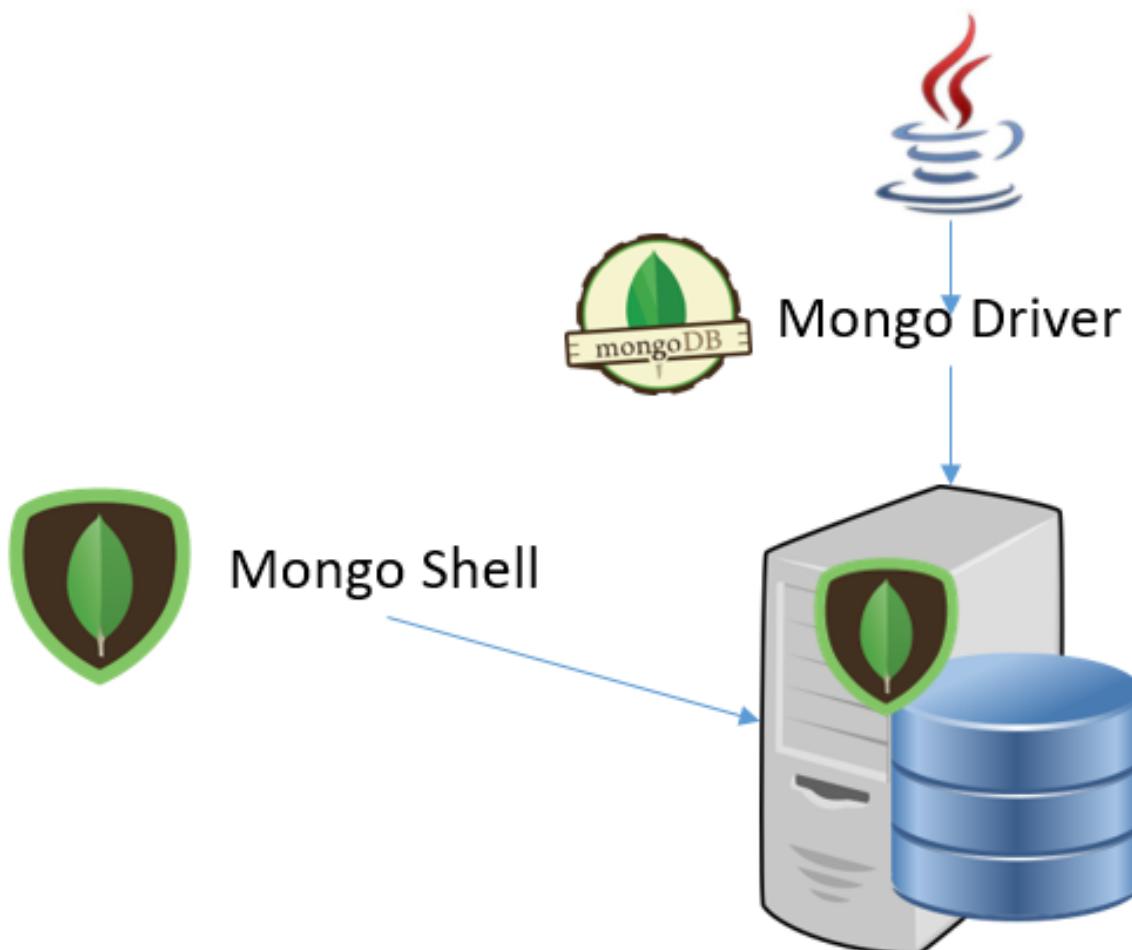
```
db.foo.find({a:{$exists:true},b:{$exists:true}}).sort({a:1,b:-1})
```

2.15. Arquitecturas MongoDB

Como punto final al presente tema, vamos a conocer las distintas arquitecturas que podemos usar en **MongoDB**:

- Standalone → Servidor en solitario, rápido de montar, útil para prototipado
- Replica Set → Asegura la persistencia de datos y la disponibilidad
- Sharded Cluster → Permite el escalado, complejo de mantener

Servidor en modo **Standalone**

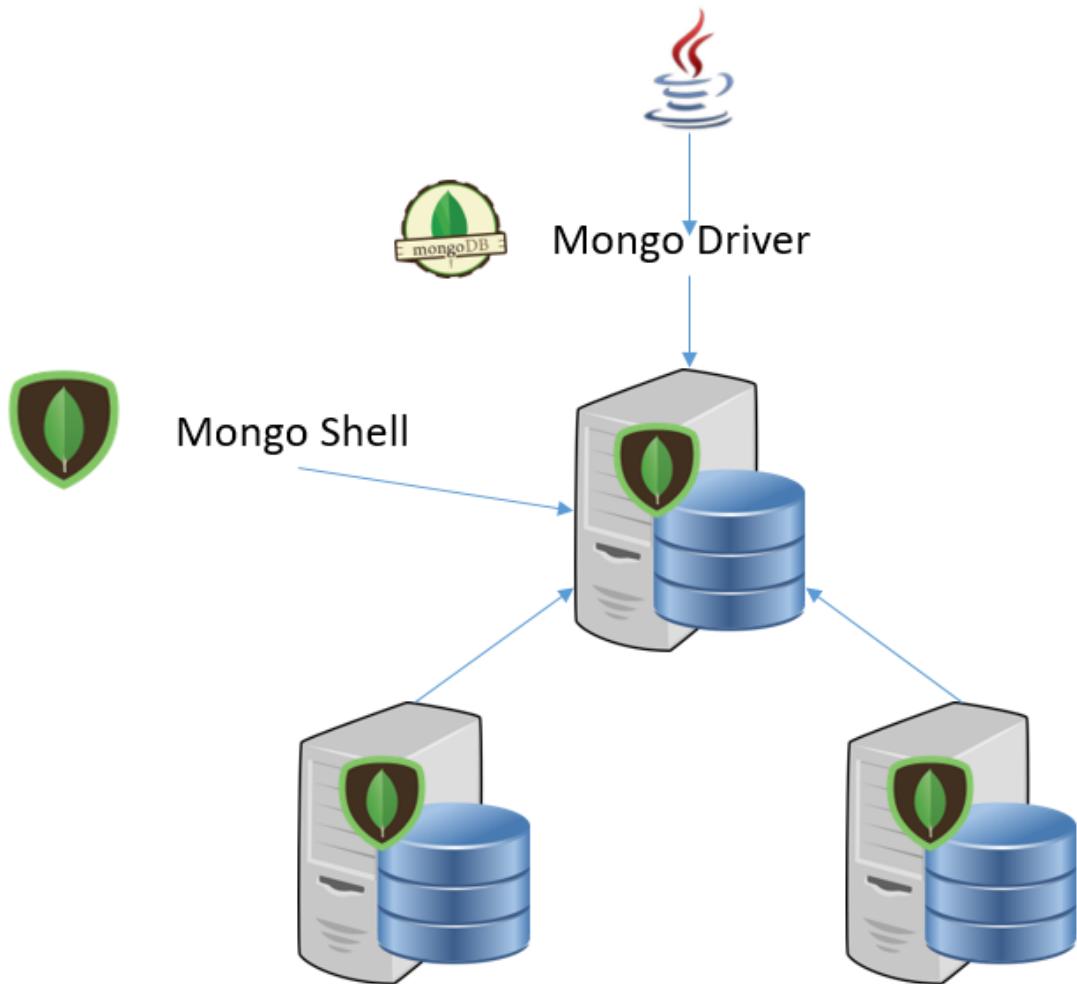


Servidor en modo **Standalone**

- Ventajas
 - Simple
 - Barato
- Desventajas

- No hay redundancia de datos
- No existe escalado

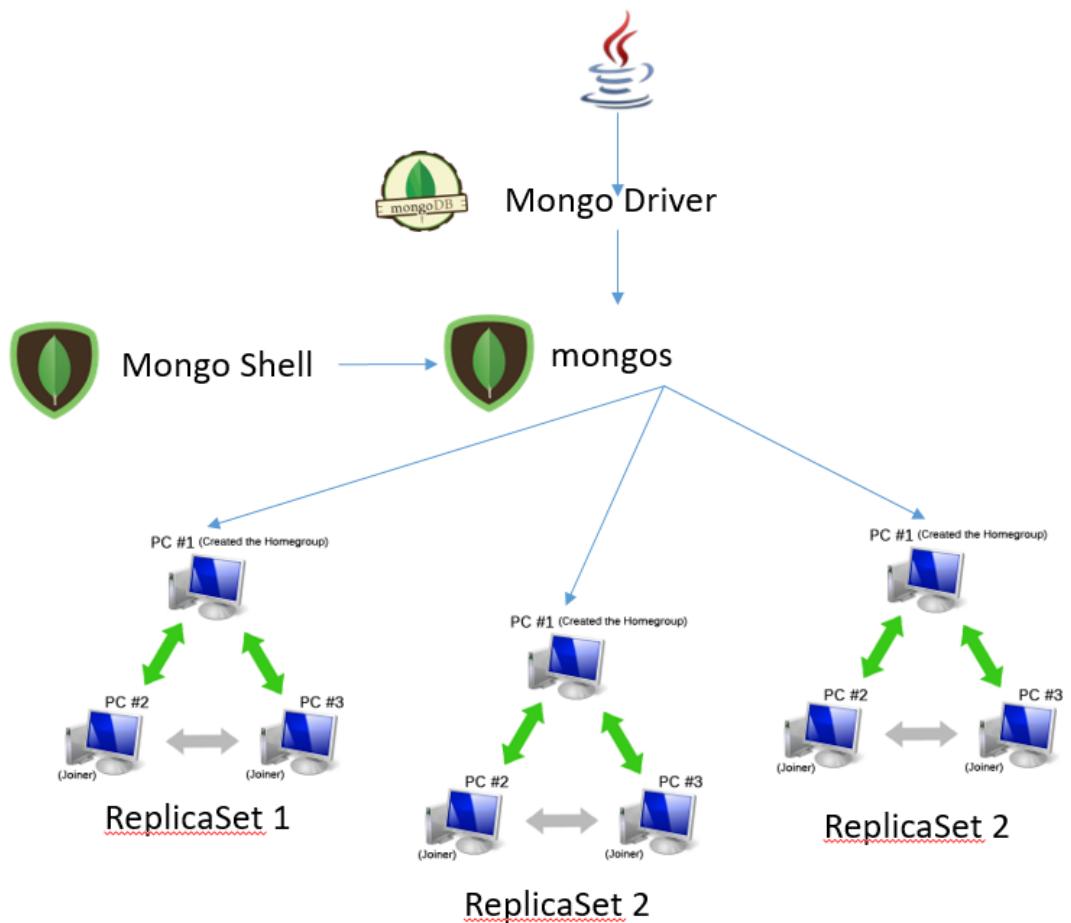
Servidor en modo **Replica Set** (En este ejemplo, disponemos de un **Primario** y dos **Secundarios**)



Servidor en modo **Replica Set**

- Ventajas
 - Disponibilidad
 - Durabilidad
- Desventajas
 - Más complejo
 - No escala

Servidor en modo **Sharded Cluster**



Servidor en modo **Sharded Cluster** (Cada Shard es un **ReplicaSet**)

- Ventajas
 - Disponibilidad
 - Durabilidad
 - Escalabilidad
- Desventajas
 - Coste
 - Complejidad

Chapter 3. CRUD

3.1. CRUD

El término **CRUD** es el acrónimo para Create, Read, Update and Delete.

Este término, hace referencia a las cuatro funcionalidades básicas de la persistencia de una base de datos. Fue popularizado por James Martin en su libro "Managing the Data-baes Environment" (1983).

- Create → La posibilidad de añadir nuevos datos al sistema
- Read → La lectura de los datos existentes
- Update → Poder alterar datos ya almacenados
- Delete → Borrar datos existentes

El **find()** que vimos en la introducción corresponde a la operación **Read**, el resto vamos a ir conociéndolas

3.2. Inserción y actualización

Ya vimos en el tema anterior cómo insertar documentos:

```
db.nuevacoleccion.insert({x:1,b:"algo"})
```

Cabe resaltar que en **MongoDB**, cualquier documento insertado ha de tener un identificador único llamado **_id**. Este identificador puede ser proporcionado a la hora de la inserción, como en el siguiente ejemplo

```
db.temperature.insert({ _id:100,x : 3 , y : 4 })
```

Pero también podemos no proporcionarlo, con lo que **MongoDB** creará automáticamente dicho campo con un valor **ObjectId** (generado pseudoaleatoriamente)

Pero además de poder insertar documentos, podemos también actualizarlos, cumpliendo así otra funcionalidad especificada en **CRUD**.

Una de las formas de actualizar es mediante la función **update()**.

```
db.users.update(  
  { age: { $gt: 18 } },  
  { $set: { status: "A" } },  
  { multi: true }  
)
```

← collection
← update criteria
← update action
← update option

Un ejemplo de actualización sería el siguiente:

```
db.sample.update({_id:100},{_id:100,x:"Hello",y:5})
```

NOTE

En **MongoDB 3.2** han aparecido tres nuevos operadores, **updateOne()**, **updateMany()** y **replaceOne()**, son simplificaciones del método principal **update()**

Como podemos observar en la imagen, la función **update()** recibe 3 documentos:

- Criterio de actualización → Es el equivalente al **where** de una consulta **SQL**, su sintaxis es análoga a la que podemos emplear en la función **find()**
- Acción de actualización → Puede contener un documento con el que reemplazar los encontrados, o unas acciones para realizar sólo actualizaciones parciales
- Opciones de actualización (Opcional) → Distintas opciones para realizar la actualización

Hay que tener cuidado, esta sintaxis es válida para **Mongo 3.0** y superiores, en versiones anteriores las opciones de actualización estaban desglosadas y no venían en un documento propio

Era algo como esto:

- db.collection.update(<where>,<documentNewVersion_OR_PartialUpdate>[,<upsert>][,<multi>])

A la hora de trabajar con la función **update()** hay que plantearse 3 cosas:

- ¿Queremos actualizar sólo un documento o todos los que cumplan dicho criterio? → Opciones de actualización, **multi**
- ¿Queremos realizar una actualización parcial o total? → Operadores de actualización
- ¿Queremos crear un nuevo documento si no encontramos ninguno que cumpla nuestro criterio? → Opciones de actualización, **upsert**

Actualizaciones totales o parciales

Cuando actualizamos un documento puede interesarnos reemplazarlo por otro nuevo o bien modificar sólo parte del documento original, sin alterar el resto de campos.

- Si queremos reemplazar el documento por otro nuevo, tan sólo tenemos que poner nuestro nuevo documento como segundo parámetro del **update()** (**importante**: No se puede modificar el campo **_id** de un documento, el nuevo documento ha de contener el mismo **_id** o bien no especificar dicho campo)
- Si tan sólo queremos modificar el documento existente, tenemos que hacer uso de los **operadores de actualización**

Operadores de actualización de campos

- **\$inc** → Aumenta el valor de un campo la cantidad indicada { **\$inc**: { "field1": -2, "field2":1, ... } }
- **\$mul** → Multiplica el valor del campo por el número indicado { **\$mul**: { "field": 3 } }

- `$rename` → Renombra un campo { `$rename:` { "nombreViejo1": "nombreNuevo1", "nombreViejo2": "nombreNuevo2", ... } }
- `$setOnInsert` → Cuando usamos **upsert**, si insertamos documento nuevo, realizamos la acción descrita en este comando
- `$set` → Actualiza o crea dicho campo { `$set:` { "field1": "valor", ... } }
- `$unset` → Elimina el campo indicado { `$unset:` { "field1": "", ... } }
- `$min` → Actualiza el campo sólo si el valor especificado es menor que el actual { `$min:` { "field1": 50, ... } }
- `$max` → Actualiza el campo sólo si el valor especificado es mayor que el actual { `$min:` { "field1": 200, ... } }
- `$currentDate` → Actualiza el valor de un campo a la fecha actual como tipo **Date** o **timestamp** (desde Mongo 3.0 estos tipos se consideran distintos a la hora de compararse)

En el operador **currentDate** hay que especificar o bien **true** (pone la fecha como **Date**) o un documento que especifica qué tipo de fecha vamos a usar (bien "timestamp" o "date")

```
db.products.update(
  { _id: 1 },
  { $currentDate: { ultimaMod: { $type: "timestamp" } } }
)
```

```
db.products.update(
  { _id: 1 },
  { $currentDate: { ultimaMod: true } }
)
```

Ejemplo de uso de **SetOnInsert**

```
db.products.update(
  { _id: 1 },
  {
    $set: { item: "apple" },
    $setOnInsert: { defaultQty: 100 }
  },
  { upsert: true }
)
```

Operadores de actualización de Arrays

- `$` → Actúa sobre el primer elemento del array que cumple la condición
- `$addToSet` → Añade un elemento al array si no está ya en él
- `$pop` → Elimina el primer (con -1) o último elemento (con 1) del array
- `$pullAll` → Elimina todos los valores especificados del array

- **\$pull** → Elimina todos los valores del array que cumplen una condición
- **\$pushAll** → **Deprecated** Añade varios elementos a un array (usamos **\$push** con **\$each**)
- **\$push** → Añade un elemento a un array (Puede usarse con operadores propios como **\$each**, **\$slice**, **\$position...**)

Ejemplo de **\$**

```
{ "_id" : 1, "grades" : [ 80, 85, 90 ] }
db.students.update(
  { _id: 1, grades: 80 },
  { $set: { "grades.$" : 82 } }
)
```

Ejemplo de **\$addToSet**

```
{ "_id" : 1, "grades" : [ 80, 85, 90 ] }
db.students.update(
  { _id: 1},
  { $addToSet: { grades: "57" } }
)
```

Ejemplo de **\$push**

```
{ "_id" : 1, "grades" : [ 75 ] }
db.students.update(
  { _id: 1},
  { $push: { grades: { $each: [ 80, 85, 90 ] } } }
)
```

Ejemplo de **\$pop**

```
{ "_id" : 1, "grades" : [ 75, 80, 85 ] }
db.students.update(
  { _id: 1},
  { $pop: { grades: -1} } )
)
```

Las opciones de actualización es un documento que puede contener 3 campos distintos:

- **upsert** → Boolean, especificamos si queremos realizar **upserts** o no.
- **multi** → Boolean, especificamos si queremos realizar sólo el primer documento encontrado o todos los que cumplan el criterio
- **writeConcern** → especificamos la política de **write concern** si queremos no usar la por defecto. Lo veremos más adelante **{ w: value, j: boolean, wtimeout: número }**

Hay que indicar que el comportamiento de **upsert** tiene ciertas lagunas que cabe mencionar. Por ejemplo, si haciendo un **upsert** se crea un documento nuevo, sólo se crea con los valores que se pueda deducir (si la query de búsqueda es algo como **{x:1,price:{\$gte:200}}**, el documento creado sólo contendrá el **{x:1}**).

Del mismo modo si hacemos un **\$inc** sobre un campo no existente toma como valor por defecto para dicho campo el **0**

Esto queda más claro en el siguiente ejemplo:

```
db.sample.update({x:9,y:{$gt:20}},{$inc:{y:3}}, {multi:true,upsert:true})
db.sample.find().pretty()
{ "_id" : ObjectId("582059ee31f525fca22a64ab"), "x" : 9, "y" : 3 }
```

La función **save()** permite insertar o actualizar un documento en una colección. Si el documento que pasamos a la función **save()** no contiene campo **_id**, se crea un nuevo documento para dicha colección. Si el documento que pasamos contiene **_id**, reemplaza el documento existente en la colección con el que nosotros proporcionamos.

```
db.gatitos.save({_id:1,ternura:98}) // cambia el documento por el proporcionado
db.gatitos.save({nombre:"misi",ternura:78}) //inserta un nuevo documento
```

A veces, es más cómodo trabajar con variables intermedias, como se muestra en el siguiente ejemplo

```
var t=db.foo
myObj=t.findOne()
myObj.y=400
t.save(myObj)
```

Por cierto, si en algún momento tenéis interés en ver cuál es el código de una función, se puede hacer poniendo su identificador (sin los paréntesis)

```
db.gatitos.save
```

3.3. Borrar documentos

Borrar un documento es análogo a buscar un documento, pero llamamos a la función **remove()** en lugar de a la función **find()**.

A dicho método, le pasamos como parámetro un documento con el criterio de eliminación. Esto procederá a eliminar todos los documentos que cumplan dicho criterio.

```
db.gatitos.remove({ternura:{$lt:60}}) //borra todos los gatitos con ternura inferior a  
60  
db.periquitos.remove({}) //borra todo
```

Un punto importante, es que podemos usar expresiones regulares, tanto en el **find()** como en el **remove()**

```
db.gatitos.insert({name:"Pepe"})  
db.gatitos.insert({name:"T1000"})  
db.gatitos.insert({name:"T200Mejorado"})  
    //eliminamos todos los gatitos T1000,  
    //o similares, porque es evidente que  
    //son parientes de terminator, y mejor curarse en salud  
db.gatitos.remove({name:/^T[0-9]{1,3}/})
```

3.4. Bulk Write Operations

A veces para optimizar nuestros procesos, no queremos lanzar de manera individual las operaciones, si no que queremos agruparlas.

En **MongoDB** nos proporcionan las **Bulk Write Operations**, que no son otra cosa que agrupar operaciones para lanzarlas en bloque.

Existen dos tipos distintos de **Bulk Write Operations** que podemos usar:

- Ordenados → Si queremos que el orden de ejecución de nuestras instrucciones sea el que hemos especificado
- Desordenados → Si no importa que se ejecuten en un orden distinto al especificado

Los **Ordered Bulk Write Operations** se ejecutan en servidor de manera individual, conservando el orden proporcionado en el momento de su creación.

Su uso es sencillo, primero creamos el objeto, le añadimos instrucciones, y por último solicitamos su ejecución.

Para crear este objeto usamos el método **initializeUnorderedBulkOp()** sobre la colección que queremos trabajar

```
var bulk = db.collection.initializeUnorderedBulkOp();  
bulk.insert({_id:1,x:1})  
bulk.insert({_id:2,x:2})  
bulk.insert({_id:3,x:3})  
bulk.execute()
```

Los **Unordered Bulk Write Operations** agrupan, al igual que sus hermanos, un conjunto de operaciones que queremos lanzar en servidor, pero no tienen por qué lanzarse en el orden en el

que fueron insertadas. Esto es más eficiente, ya que permite paralelizar las operaciones.

Su uso es análogo al anterior, aunque se inicializa con el método **initializeOrderedBulkOp**

```
var bulk = db.collection.initializeOrderedBulkOp();
bulk.find({_id:1,x:1}).remove()
bulk.find({_id:2}).remove()
bulk.find({_id:3}).update({$inc:{x:1}})
bulk.execute()
```

En ambos casos, podemos realizar diversas operaciones, vamos a comentar las más comunes (siendo **bulk** nuestro *Bulk Object):

bulk.insert() → Añade una operación
bulk.find().removeOne() → Elimina el primer documento que cumpla la condición
bulk.find().remove() → Elimina los documentos que cumplan la condición
bulk.find().replaceOne() → Reemplaza un documento
bulk.find().updateOne() → Añade una actualización unitaria
bulk.find().update() → Añade una actualización múltiple
bulk.find().upsert().update() → permite especificar el **upsert** de la operación posterior
bulk.execute() → Solicita la ejecución del conjunto de operaciones añadidas
bulk.getOperations() → Devuelve un array con las operaciones añadidas al **Bulk Object**

Para consultar el total de operaciones disponibles, podéis consultar la documentación oficial en docs.mongodb.org/manual/reference/method/Bulk/

3.5. Wire Protocol

El protocolo de comunicación de **MongoDB** es un mecanismo de comunicación simple basado en petición/respuesta a través de un puerto, realizado a través de un puerto TCP/IP.

En estos paquetes se envían las operaciones de consulta y actualización de datos, que pueden ser de los siguientes tipos:

- Query → Una consulta a la base de datos
- Insert → Insertar un datos
- Update → Actualizar
- Remove → Eliminar
- GetMore → Cuando solicitamos datos con una query, si hay muchos valores devueltos, se entregan en bloques

El punto que hay que tener en cuenta en esta sección es que existe una operación **GetMore** para cuando los datos devueltos exceden una cierta cantidad, de tal manera que vamos trayendo dichos datos en bloques.

3.6. Comandos

Existen no obstante otra serie de operaciones que ya exceden el ámbito **CRUD** y están más

relacionadas con la administración. Estas operaciones las vamos a denominar **comandos**, y podemos hacer uso de ellas mediante la colección especial **\$cmd** o la función **db.runCommand()**. De hecho, las más utilizadas tienen incluso un **helper** o método directo para hacer uso de las mismas.

El siguiente ejemplo muestra 3 formas distintas de obtener información sobre nuestro nodo

```
db.$cmd.findOne({isMaster:1})  
db.runCommand({isMaster:1})  
db.isMaster()
```

Además, existen ciertos **comandos** que sólo pueden ser ejecutados desde la base de datos **admin**, no obstante, podemos hacer uso de la función **_adminCommand()** para no tener que movernos. Los siguientes ejemplos son pues, equivalentes

```
use admin  
db.runCommand( {buildInfo: 1} )
```

```
db._adminCommand({buildInfo:1})
```

Aunque hay un gran número de comandos, los más comunes son un grupo bastante reducido. Vamos a centrarnos en ellos.

Para estudiar los comandos, vamos a agruparlos en:

- Comandos de usuario
- Comandos de administración

(Para un listado completo de comandos se puede consultar el manual oficial de **MongoDB** en <https://docs.mongodb.com/manual/reference/command/>)

Los comandos de usuario más comunes son:

- **aggregate** → Permite realizar agregaciones (equivalente a los group by de **SQL**). Importantes cambios en **Mongo 3.2**
- **mapReduce** → Permite realizar operaciones map/reduce en **MongoDB**
- **count** → Cuenta los elementos
- **findAndModify** → Es una operación considerada atómica.

Una visión general de los comandos de administración más comunes (ampliaremos posteriormente) son:

- **getLastError** → Estado de éxito de la última operación lanzada (**Innecesario desde Mongo 2.6**, ya devuelve la propia operación el resultado asociado a su **write concern**)
- **isMaster** → Devuelve información del rol de un miembro en un **Replica Set**

- drop → Eliminar una colección (no dejándola vacía como hace el **remove({})**)
- createCollection → Crear una colección
- compact → Defragmenta una colección y reconstruye sus índices
- serverStatus → Devuelve una colección de métricas del servidor
- replSet[Freeze | StepDown | Reconfig | GetConfig] → Para trabajar con **Replica Sets**
- addShard | removeShard | listShardsenableSharding | split | shardCollection → Creación/borrado/listado de **Sharded Clusters**
- enableSharding | split | shardCollection → Habilitar y administrar **sharding** en colecciones y **esquemas**
- createIndex | listIndexes | dropIndexes → Para crear/borrar/listar índices sobre una colección
- createUser | dropUser | createRole | ... → Operaciones de gestión de roles y usuarios
- fsync → Vuelca los datos en memoria a disco
- explain → Para conocer cómo resuelve el servidor una operación (si usa índices, cuántos documentos lee...)

Para ejecutar un comando, en ocasiones tenemos que enviar parámetros, como se muestra en el código siguiente

```
db.runCommand({<command_name>:<value>, param1:value1...})
```

Muchos comandos no necesitan parámetros. Por ejemplo, **serverStatus** no necesita parámetros

```
db.runCommand({serverStatus:1})
```

Otro ejemplo que puede llevar parámetros es crear un índice para una colección

```
db.runCommand(createIndexes:"colección", indexes:[{key:{x:1}, unique:true}])
```

En este caso, suele ser mucho más cómodo usar el helper, puesto que está mucho más simplificado

```
db.colección.createIndex({x:1},{unique:true})
```

Otros comandos de suma utilidad a la hora de administrar nuestro servidor **MongoDB** son los comandos **currentOp()** y **killOp()**.

Esto nos permite ver qué operaciones están ejecutándose actualmente, y finalizarlas si lo consideramos necesario.

- db.currentOp() → Obtenemos un listado de las operaciones que se están ejecutando actualmente
- db.killOp(opid) → Matamos la operación con el **opid** proporcionado

Por lo general, un atributo que debemos vigilar de cerca al lanzar el **currentOp** es **secs_running**

Un ejemplo para poder mostrar cómo se pueden usar estos comandos:

En un shell mongo lanzar el siguiente código:

```
while (1){ db.temp.insert({timeStamp:new Date()});}
```

Ahora, desde otra terminal mongo (conectada al mismo esquema), vamos a obtener el **opid** de dicha operación y acabar con ella mediante **killOp()**

```
db.currentOp() //cogemos el opid  
db.killOp(opid) //cogido del punto anterior
```

A la hora de trabajar con **colecciones**, también tenemos una serie de **comandos** útiles para administrar nuestra base de datos.

En primer lugar, podemos consultar qué colecciones existen actualmente en nuestro esquema:

```
show collections
```

Otro comando útil será consultar el total de documentos que posee una colección

```
db.temp.count()
```

Así mismo, cada colección posee sus propias estadísticas, y pueden ser bastante interesantes (nos puede dar datos como tamaño, los índices que tiene...)

```
db.temp.stats()
```

Se pueden borrar colecciones mediante el comando **drop()**, que elimina los documentos y **la colección**

```
db.temp.drop()
```

Por último, y aunque a priori en **MongoDB** no es necesario crear las colecciones antes de usarlas, en ocasiones si nos interesa hacerlo (por ejemplo porque queremos crearla como **capped collection**, o preasignar un tamaño en disco para la misma)

```
db.createCollection("logcutre", { capped : true, size : 5242880, max : 5000 } )
```

Chapter 4. Replicación I

4.1. Introducción

Hasta ahora todo lo que hemos trabajado en **MongoDB** se centra en la funcionalidad ofrecida, pero no en su arquitectura.

A la hora de hacer uso de sistemas en producción, el esfuerzo no se centra tan sólo en el desarrollo del producto, si no también en intentar que nuestro sistema sea "a prueba de balas", o lo que es lo mismo, que un fallo en el sistema no comprometa ni su servicio, ni su integridad.

En resumidas cuentas, pase lo que pase, **el espectáculo debe continuar**

Show must go on



Muchos sistemas clásicos requieren capas adicionales para gestionar, por ejemplo, la disponibilidad de copias de seguridad de los datos con los que trabajamos (por ejemplo, usando discos montados sobre un **Raid**). Esto quiere decir que el sistema en sí es agnóstico a dicho backup o duplicado, y este surge como un parche al mismo ante una necesidad (la de no perder datos). Y por supuesto, para el caso que hemos mencionado, las copias están en una misma máquina física (la que posee el **Raid**)

En **MongoDB** hablar de disponibilidad de la información no es hablar de copias en una misma máquina, ya que la arquitectura propia de **MongoDB** está enfocada a evitar escenarios en los que la pérdida de una única máquina pueda comprometer nuestro sistema. **MongoDB** realiza las copias de los datos entre distintas máquinas, minimizando así la posibilidad de pérdida de datos.

Estas estructuras de máquinas enfocadas a disponer de distintas **réplicas** de una misma

información es el denominada **Replica Set**.

Un **Replica Set** es un conjunto de máquinas que van a replicar la información existente entre ellas. Una de las mismas se va a levantar como la máquina **Maestro**, que va a ser la única que pueda recibir solicitudes de escritura, mientras que el resto serán las máquinas **Esclavo**, que van a replicar los cambios existentes en la máquina **Maestro**, pero sobre las cuales se pueden realizar lecturas (no por defecto, pero es configurable). Es decir, tenemos la posibilidad de escalar nuestro sistema para poder atender un mayor número de lecturas si usamos los **Esclavos** para tal fin.

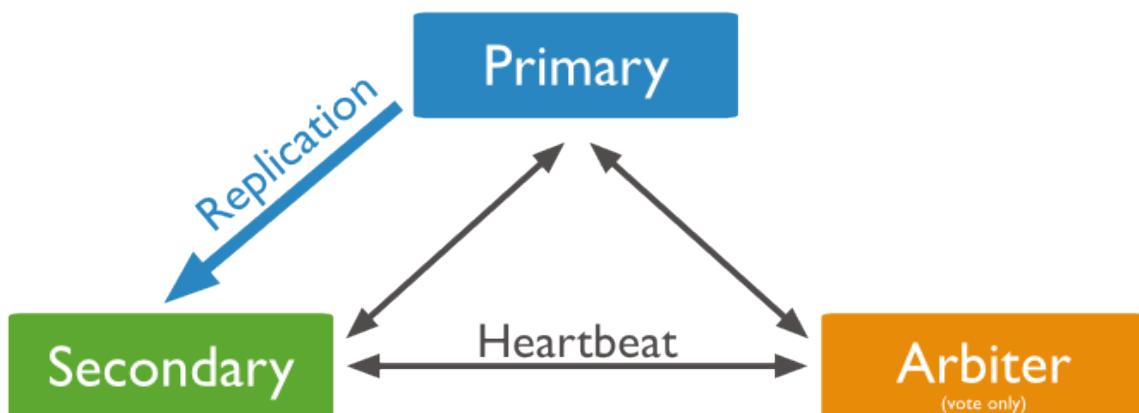
¿Qué ventajas ofrece el uso de un **Replica Set**?

- Alta disponibilidad → En caso de caerse una máquina, el resto pueden seguir dando servicio. Incluso si se cae el **Maestro**, uno de los **Esclavos** ocupará su puesto y el servicio podrá proseguir con normalidad.
- Seguridad de la información → Al existir réplicas de los datos, evitamos perder nuestra información si sucede algo que evite recuperar una máquina.

Hay que tener en cuenta que la finalidad de las réplicas es ofrecer seguridad para evitar la pérdida de datos, pero no son un **Backup** a nuestro sistema. Un **Backup**, bien entendido, es una copia de un estado dado de nuestro sistema (con la finalidad de poder revertir a dicho estado si fuera necesario). Es decir, tener una réplica de nuestros datos no nos libra de realizar **Backups** de los mismos. Por ejemplo ¿Qué pasaría si pasamos un proceso que realiza unos cambios erróneos sobre nuestros datos? que necesitaríamos poder volver a un estado anterior, cosa que no ofrecen las réplicas.

Existe no obstante un tipo de **Esclavos** con una configuración especial (que retrasa la ejecución de cambios un tiempo definido) que puede ser usado para solventar un problema como el expuesto, no obstante sigue siendo muy recomendable disponer de copias de seguridad, puesto que nunca se sabe para qué podemos necesitarlas.

Ejemplo de **Replica Set** con un **Arbiter** (lo veremos más adelante)



4.2. Replicación de datos

A la hora de trabajar con **Replica Sets**, hay que tener muy claro qué sucede con la información.

En primer lugar, debemos tener presente que cualquier cambio sobre los datos se realiza siempre

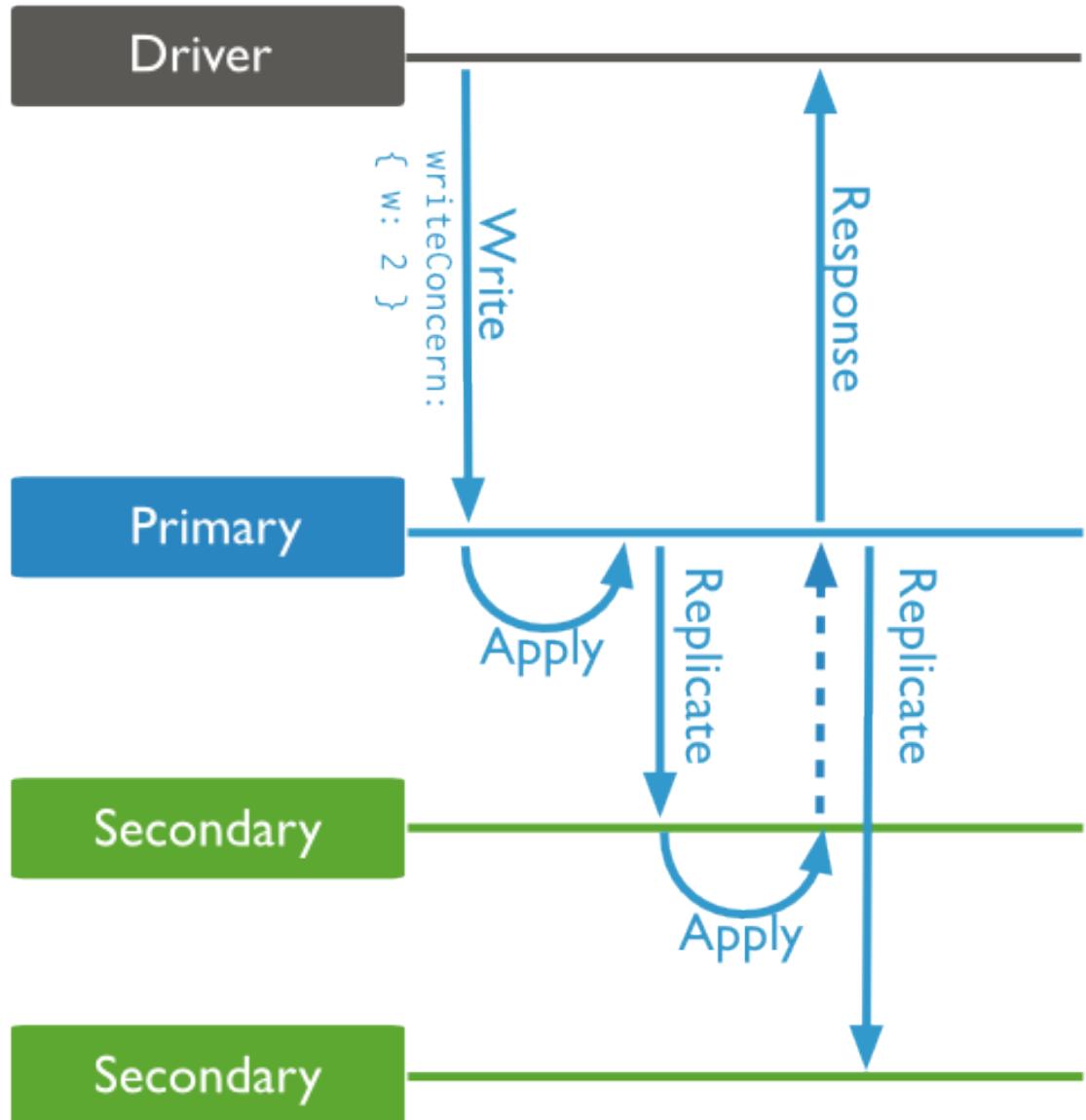
sobre el **Primary** (es decir, el **Maestro**). En **MongoDB** no existe la posibilidad de realizar escrituras sobre nodos que no sean **Primary**.

Los cambios realizados en el nodo **Primary** son replicados a los nodos **Secondary** de manera **Asíncrona**.

En un flujo normal, el cliente escribe directamente sobre el nodo **Primary**, y desde dicho nodo se realizan las réplicas a los **Secondaries**. En cuanto alcancemos el **Write Concern** correspondiente (esto lo estudiaremos más adelante), devolveremos una señal **ack** al cliente para indicar que la operación ha tenido éxito. (**NOTA:** Aunque en principio se replica del nodo **Primary**, es posible reaplicar datos desde un nodo **Secondary**, ya lo veremos más adelante)

En la siguiente imagen vemos cómo se realizaría una actualización de datos disparada por el cliente sobre el nodo **Primary**, para el cuál hemos pedido un **Write Concern** de **2** (en resumidas cuentas, esto es el total de máquinas que deben haber recibido la orden de escritura antes de dar por buena la operación).

Al escribir la primera de las dos réplicas, ya alcanzamos el nivel de **Write Concern** solicitado, con lo que enviamos la señal **ack** al cliente, pero continuamos realizando la réplica al nodo restante.



Más adelante entenderemos cómo funciona el **Write Concern**, que es empleado por las operaciones de escritura para dar por buena una operación. De momento tengamos presente que el **Write Concern** que existe por defecto tiene el valor **w** a **1**, es decir, con llegar la instrucción a la máquina **Primary**, el cliente recibe el **ack**.

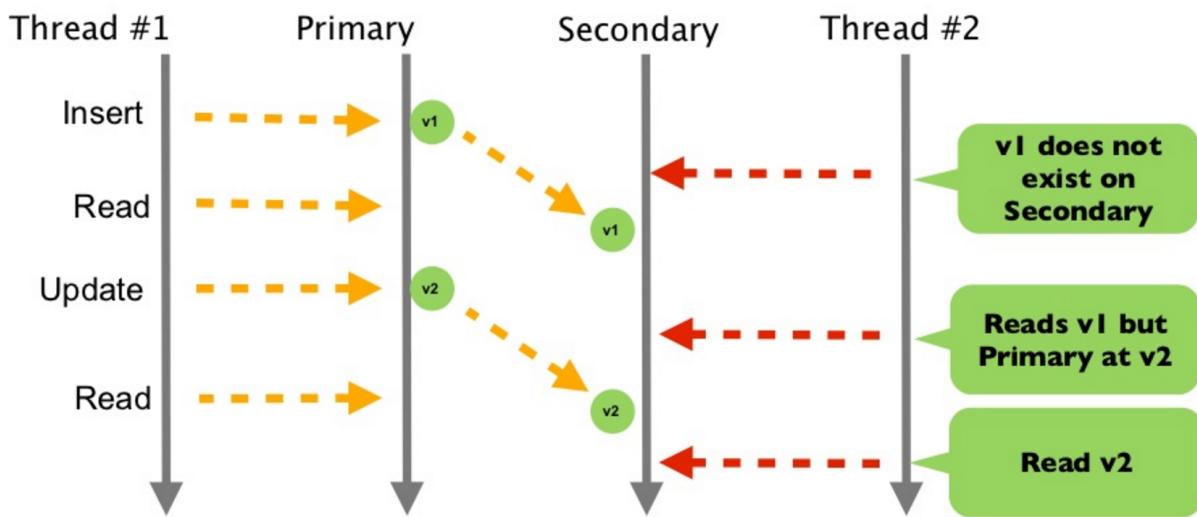
Al ser la réplica de datos una operación asíncrona, se puede dar el caso de haber disparado una actualización de datos (de la cual hemos recibido el **ack**) en el **Primary**, y realizar una lectura de datos de un nodo **Secondary** sobre el cuál todavía no se ha replicado el cambio. Es decir, podemos estar leyendo un dato ligeramente desfasado.

Al ser las réplicas asíncronas, este escenario puede suceder, aunque su marco temporal es muy reducido. Esta situación de **inconsistencia** se solventa generalmente en cuestión de milisegundos, pero tenemos que tener presente que esta situación existe.

En **MongoDB** no podemos decir que existe siempre una situación de **Consistencia** absoluta, pero las inconsistencias se solventan en un período de tiempo muy reducido. Es por ello que en **MongoDB** hablamos de **Eventual Consistency**. Este término nos indica que por lo general va a existir consistencia, y que en el caso de que no existe en un momento dado, el sistema solo se volverá

consistente.

NOTA: Evidentemente si no permitimos lecturas de los nodos **Secondary**, la consistencia será fuerte, pero impediremos la posibilidad de escalar a nuestro sistema.



Otro punto crítico con respecto a la replicación de datos consiste no sólo en saber cómo se replican los datos, si no en saber qué datos se replican.

Voy a explicarme, los datos almacenados en **MongoDB** son datos binarios, dependientes del **Storage Engine** usado por cada nodo. ¿Quiere esto decir que todos los miembros de un **Replica Set** han de compartir el mismo **Storage Engine**? No, puesto que no se hace una replicación a nivel de datos binarios.

La replicación de datos en un **Replica Set** es a nivel de instrucciones, es decir, es **Statement Based**.

Una operación de inserción de un dato implica un cambio en un fichero almacenado en disco (por ejemplo escribir una serie de bytes en una posición de un fichero), si el dato insertado afecta además a índices, habría que tocar más ficheros.

Evidentemente podríamos dedicarnos a tener los mismos ficheros físicos y enviar cambios sobre estos (lo cual exigiría que todos los nodos usaran el mismo **Storage Engine**), sin embargo, **MongoDB** opta por otra opción, enviar a los demás nodos una instrucción **CRUD** de **MongoDB**, con los cambios que haya que realizar. Esto permite usar distintos **Storage Engines** e incluso distintas versiones, conviviendo juntas.

Una ventaja adicional de dicha decisión, es la posibilidad de realizar una actualización de la versión de **MongoDB** de nuestro sistema, modificando los nodos uno a uno, ya que la comunicación entre ellos se realiza de manera agnóstica, usando sólo **Statements**

Ya hemos oido hablar de dos elementos propios de **MongoDB**, el **journal** y el **oplog**. Iremos conociéndolos en profundidad, pero para avanzar tenemos que tener una idea básica de ellos:

- Journal → Es un fichero que trabaja a bajo nivel, (se puede desactivar), indicando qué cambios vas a realizar sobre los ficheros (algo del estilo "pon estos bytes en tal posición de tal fichero"). Antes de realizar el cambio sobre dichos ficheros, se indican en el **Journal**. Se usa para re establecer la consistencia de la base de datos si hubo una parada inesperada (al levantarse, ve

que había pendiente unos cambios y realiza un **crash recovery**, y devuelve la consistencia a la base de datos)

- Oplog → Por su parte, el **Oplog** almacena las transacciones de alto nivel realizadas sobre la base de datos (los inserts, updates y removes realizados). Este fichero es el usado para replicar los cambios desde el nodo **Primary** a los **Secondaries** (el **Oplog** se almacena en el **Primary**, y los **Secondaries** lo sincronizan trayéndose los cambios realizados sobre dicho fichero)

En resumen, ¿Qué usamos para sincronizar los datos en los nodos de un **Replica Set**? El **Oplog**.

Y ahora viene un concepto crítico a la hora de trabajar con **MongoDB**. ¿Qué almacena dicho **Oplog**? Podemos estar tentados a pensar que almacena las instrucciones tal como las realizamos, pero esto es un error, ya que las operaciones lanzadas **pueden ser transformadas al insertarse en el Oplog**.

Este es uno de los conceptos más importantes en **MongoDB**. Las operaciones almacenadas en el **Oplog** han de ser **idempotentes**, es decir, si ejecutamos dos veces la misma operación, no va a haber diferencia con respecto a ejecutarla una vez.

En la práctica esto se traduce en que **MongoDB** transforma muchas de las operaciones que lanzamos.

Pongamos por ejemplo que tenemos un documento en una colección

```
db.prueba.find()  
{_id:1,x:1}
```

y lanzamos una operación para incrementar el valor de **x** en **1**

```
db.prueba.update({_id:1},{x:{$inc:1}})
```

El **Oplog** almacenará una instrucción distinta, que sea **Idempotente**, por ejemplo:

```
db.prueba.update({_id:1},{_id1,x:2})
```

No importa que lancemos 4 veces dicha operación, el resultado será el mismo

Pero el **Oplog** no sólo puede transformar una operación en otra distinta, **puede transformarla en múltiples operaciones**, y este es otro concepto crítico

Imaginemos que tenemos una colección que posee una serie de documentos con un campo **age**, y queremos eliminar a todas las personas con **age** igual a **30**. Lanzaremos la siguiente instrucción, que eliminará por ejemplo 4 documentos

```
db.personas.remove({age:30})
```

El **Oplog** almacenará una operación **idempotente** por cada documento eliminado, por ejemplo algo

como (se supone que los `_id` pertenecen a los documentos con el campo `age` igual a 30):

```
db.personas.remove({_id:1:7})
db.personas.remove({_id:1:53})
db.personas.remove({_id:1:21})
db.personas.remove({_id:1:93})
```

En resumen, **una operación sobre el nodo primario puede convertirse en múltiples operaciones sobre los nodos secundarios**, el Primary ejecutará un único **Statement**, pero los **Secondaries** ejecutarán tantos **Statements** como haya en el **Oplog**

4.3. Conceptos sobre Replicación

Antes de ponernos a crear un **Replica Set**, hay un par de conceptos más que necesitamos aclarar.

En primer lugar vamos a hablar de el factor de replicación (**Replication Factor**). Cuando hablamos del factor de replicación estamos hablando del total de copias que va a haber de un dato en nuestro sistema. Esto quiere decir que un **Replication Factor** de 3 indica que existen un total de 3 copias de un dato (un error bastante frecuente es creer que existe el dato original y 3 copias, esto no es así).

¿Cuál es entonces el **Replication Factor** de un **Replica Set**? muy fácil, es igual al número de miembros existentes, puesto que cada miembro alberga una copia del dato (quedan fuera de este concepto los **Arbiter**, que los veremos luego y los nodos **Delayed**).

Más adelante conoceremos el concepto de **Sharded Cluster**, ahora mismo no vamos a entrar en ello, pero si es necesario que tengamos la idea de que un **Sharded Cluster** es un conjunto de **Replica sets**, y estos (que actuán como replication clusters), son a su vez un conjunto de servidores **mongod**

Una de las ventajas de usar un **Replica Set** es que este, gestiona automáticamente las caídas o **failovers**. Si una máquina se cae, el **Replica Set** seguirá trabajando. Un caso especial es si se cae el nodo **Primary**, ya que entre los nodos restantes se debe promocionar un nuevo nodo **Primary**, y esto no es instantaneo, puede llegar a **tardar unos 10 segundos**. Además, para poder elegir un nuevo nodo **Primary** deben poder **votar** al menos, más de la mitad (ahora lo veremos con más detalle).

Otra ventaja importante es **automatic node recovery**, es decir, que se permite que una máquina caída vuelva a ingresar en el **Replica Set** y obtenga un estado de consistencia.

Recordemos que, aunque se pueden permitir las lecturas de los nodos **Secondaries**, las escrituras siempre se deben lanzar sobre el **Primary**

En un **Replica Set**, todos los nodos se monitorizan entre ellos. Si cae el nodo **Primary** existe un **consenso** para determinar que dicho nodo está caído (de esta manera evitamos situaciones en las que por ejemplo, por un tema de red, el **Replica Set** se parte en dos conjuntos y ciertos nodos no vean al **Primary**, pero los otros si lo vean)

En caso de que caiga el nodo **Primary**, se van a realizar unas **elecciones**, que determinarán qué nodo se alza como nuevo nodo **Primary**. Para poder promocionar un nodo **Secondary** a **Primary**

debe haber un **consenso**, es decir, que al menos **más de la mitad** voten por dicho nodo.

En un **Replica Set** con 3 nodos, si cae el **Primary**, siguen existiendo 2 nodos levantados, con lo que cumplimos la regla para poder elegir un nuevo **Primary**. Uno de estos nodos se promocionará a **Primary**, el driver del cliente gestionará automáticamente este cambio y se podrá seguir trabajando sin problemas (El driver es **Replica Set Aware**, es decir, habla con todos los miembros)

Si se diera el caso de que tenemos un **Replica Set** con sólo 2 miembros, si cae el **Primary** no habría posibilidad de elegir un nuevo **Primary**, por no alcanzar el **consenso**. No podríamos seguir trabajando.

¿Qué pasa si resulta que cae un **Primary**, pero se incorpora tras un período de tiempo de nuevo a nuestro **Replica Set**?

Esta situación está contemplada por **MongoDB**, y la va a gestionar automáticamente. No obstante hay algunos conceptos que debemos tener en cuenta.

Cuando el **Primary** se reconecta al **Replica Set**, es posible que posea operaciones de escritura que no se hubiesen portado a los secundarios, y que existan nuevas operaciones que nuestro nodo no posea.

Imaginemos que cuando nuestro nodo se cayó, la última operación replicada fue **w5**, pero iba a realizar las operaciones **w6** y **w7**, que no ha replicado. Para volver a estar sincronizado hace los siguientes pasos:

- Hace rollback de las operaciones **w6** y **w7**, las archiva (en **archived**) y es responsabilidad del administrador decidir qué hacer con ellas
- Se sincroniza con el nuevo **Primary**, recibiendo todas las operaciones realizadas tras **w5**, que es el último punto en común entre ellos.

Si eres un nodo **Secondary**, es mucho más fácil, busca el último punto en común y hace un **tail** de todo lo restante en **Oplog**.

Más adelante veremos cómo podemos evitar estas situaciones (usando un **write concern** que exija más de la mitad hayan recibido la instrucción), pero es importante que tengamos muy claro que sólo existen **Rollbacks** bajo la situación descrita.

4.4. Levantando un Replica Set

Vamos a crear un **Replica Set** en una única máquina. Sobra decir que en producción jamás habría que crear así un **Replica Set**, y cada nodo debería estar en una máquina distinta a los demás.

Para poder realizar esta operación vamos a tener que:

- Dar puertos distintos a cada miembro (no pueden usar los mismos puertos al estar en la misma máquina)
- Dar rutas distintas para almacenar los datos (con `--dbpath`)
- Usar el mismo **Replica Set Name**, puesto que si no poseen el mismo nombre, aunque nos inviten a un **Replica Set**, no nos unimos

Recordemos, que un **Sharded Cluster** es un conjunto de **Replica Sets**. Si nuestra finalidad a futuro es formar parte de dicho clúster, es elegante definir los nombres de cada **Replica Set** con algo descriptivo como **shardalgo_1**, **shardalgo_2**...

Vamos a ir haciendo los pasos en nuestro sistema (estos ejemplos vienen con comandos del shell de **windows**, si usáis **UNIX** tendréis que usar sus análogos)

Creo los directorios para cada nodo del **Replica Set** (suponiendo que trabajo en **c:\mongo**)

Windows

```
cd c:\mongo  
mkdir replicaset  
cd replicaset  
mkdir 1 2 3  
cd c:\mongo
```

Linux

```
mkdir -p replicaset/[1,2,3]  
mkdir logs
```

Una vez creados los directorios donde los distintos miembros del **Replica Set** van a almacenar sus datos, voy a levantar cada uno de los procesos que van a formar parte del mismo

Levanto un primer server (esto sigue siendo **windows**).

Windows

```
start /b mongod --port 27001 --replSet abc --dbpath replicaset/1 --logpath log/log.1  
--logappend --oplogSize 50 --smallfiles
```

Si trabajamos en **UNIX**, podemos hacer uso de la opción **--fork** para levantar el proceso en otro hilo, tal como mostramos a continuación

```
[vagrant@localhost ~]$ mongod --port 27001 --replSet abc --dbpath mongo/replicaset/1  
--logpath logs/log.1 --logappend --oplogSize 50 --smallfiles --fork  
about to fork child process, waiting until server is ready for connections.  
forked process: 15450  
child process started successfully, parent exiting
```

Hemos especificado una serie de datos a la hora de levantar el servidor:

- **--port** → El puerto donde va a estar escuchando
- **--replSet** → El nombre del **Replica Set** al que va a poder unirse
- **--dbpath** → Path donde almacenará los datos
- **--oplogSize** → tamaño del **Oplog** (tiene un tamaño por defecto, pero permite especificarlo)

- `--smallfiles` → Tamaño de los ficheros de almacenamiento comienzan siendo más pequeños

Para levantar el resto de procesos, tendré que lanzar la misma instrucción, cambiando los puertos donde escuchamos, la ruta donde almacena los datos y la de los logs

Windows

```
start /b mongod --port 27002 --replSet abc --dbpath replicaset/2 --logpath log/log.2
--logappend --oplogSize 50 --smallfiles
start /b mongod --port 27003 --replSet abc --dbpath replicaset/3 --logpath log/log.3
--logappend --oplogSize 50 --smallfiles
```

Linux

```
[vagrant@localhost ~]$ mongod --port 27002 --replSet abc --dbpath mongo/replicaset/2 --logpath log.2 --logappend
--oplogSize 50 --smallfiles --fork
about to fork child process, waiting until server is ready for connections.
forked process: 15511
child process started successfully, parent exiting
[vagrant@localhost ~]$ mongod --port 27003 --replSet abc --dbpath mongo/replicaset/3 --logpath log.3 --logappend
--oplogSize 50 --smallfiles --fork
about to fork child process, waiting until server is ready for connections.
forked process: 15547
child process started successfully, parent exiting
```

¿Basta con esto para tener nuestra réplica funcionando?

No. Si nos vamos a los logs veremos un mensaje del estilo a:

```
Did not find local replica set configuration document at startup; NoMatchingDocument:
Did not find replica set configuration document in local.system.replset----
```

Es decir, hemos levantado 3 procesos **mongod**, pero no existe una configuración de nuestro **Replica Set**. Hasta que no lo configuremos no trabajaremos como **Replica Set**.

4.5. Inicializando un Replica Set

Ahora que tenemos levantados los 3 procesos **mongod**, necesitamos especificar cuál es la configuración de nuestro **Replica Set**, para poder inicializarla. Esto se puede conseguir con tan sólo dos pasos sencillos:

- Especificar la configuración de nuestro **Replica Set**
- Usar (si queremos) los datos existentes en uno de los miembros para inicializar nuestro **Replica Set**. Este miembro lo debemos establecer como **Primary** y sus datos se copiarán a los **Secondaries** (que deberían estar vacíos). Si todos los miembros están vacíos, es indiferente qué miembro elijamos para levantar el **Replica Set**.

En nuestro caso es indiferente qué miembro se convierta en **Primary**, vamos a conectarnos por ejemplo al miembro que está en el puerto **27001** para comenzar a trabajar

```
mongo --port 27001
```

Lo primero que vamos a tener que hacer es crear el documento de configuración. En su versión más simple debemos especificar el nombre del **Replica Set** en el campo **_id** y dar un listado de miembros del mismo en el campo **members**. Sirva como ejemplo el mostrado debajo (Estamos usando el nombre de la máquina, no se recomienda usar **localhost** puesto que puede generar conflictos. Asegurarnos de incluir nuestro nombre en el fichero **hosts** si no disponemos de servidor **dns**)

```
cfg={  
  _id:"abc",  
  members:[  
    {_id:0,host:"machine_name:27001"},  
    {_id:1,host:"machine_name:27002"},  
    {_id:2,host:"machine_name:27003"},  
  ]}
```

Una vez hayamos definido la configuración, vamos a inicializar el **Replica Set**. Siempre que queramos realizar alguna operación sobre el **Replica Set** y no tengamos claro qué podemos hacer, podemos solicitar la ayuda de la shell **mongo** como se muestra en el ejemplo:

```
rs.help()
```

Mirando esa ayuda, vemos que existe un **rs.initiate(cfg)**, que parece ser lo que necesitamos para levantar nuestro **Replica Set**, vamos a lanzarlo:

```
rs.initiate( cfg )
```

Una vez hecho esto, nuestra shell debería cambiar de mostrar:

```
abc:OTHER>
```

a mostrar algo como (podría ser también **SECONDARY**, puesto que no hemos especificado ninguna prioridad en la configuración para indicar quién queremos sea **Primary**, lo estudiaremos más adelante)

```
abc:PRIMARY>
```

Ya tenemos funcionando nuestro **Replica Set**.

4.6. Administrando un Replica Set

Ahora que ya tenemos nuestro **Replica Set** montado, vamos a conocer algunas utilidades que nos permitirán tanto monitorizar como administrarlo.

En primer lugar es importante conocer cuál es el estado de nuestro **Replica Set**. Para ello disponemos del comando **rs.status()**

```
rs.status()
```

Este comando nos dará información del **Replica Set** y cada uno de sus miembros.

- **health** → Salud del nodo. **1** significa que está bien
- **state** → Nos informa si el nodo es **1-Primary** o **2-Secondary** (en realidad hay 10 estados, <https://docs.mongodb.com/manual/reference/relica-states/>, entre los que se incluye **7-Arbiter** o **8-Down**). El campo **statStr** es la representación **humana** de este campo
- **optime** → Fecha de la última operación (o lo que es lo mismo, última actualización del **Olog**). La más reciente es siempre la del **Primary**, los **Secondaries** pueden no tener la última versión (es útil para elegir nuevo **Primary**, cogemos aquel con su **Olog** más reciente).
- **optimeDate** → Igual al anterior pero más legible para humanos
- **pingMs** → Tiempo de respuesta al contactar con dicho miembro

Otros comandos útiles para administrar nuestro **Replica Set** son:

- **rs.conf()** → Si en algún momento queremos obtener la configuración de nuestro servidor, podemos hacerlo con **rs.conf()**, devuelve un documento con la configuración actual (útil para realizar cambios y aplicarlos, así partimos de un documento sobre el que realizaremos modificaciones). **OJO**, hay que usar **rs.reconfig(cfg)** para aplicar los cambios, jamás hagáis algo como **system.replset.save()** de dicho documento
- **rs.reconfig(cfg)** → Cambia la configuración existente del **Replica Set** por la proporcionada en el documento **cfg**
- **rs.add(host:port)** → Añade un nuevo miembro al **Replica Set** (dicho host debe tener un **mongod** levantado con la opción **--replSet** con el nombre adecuado, y estar levantado en el puerto indicado)
- **rs.remove(host:port)** → Elimina un miembro del **Replica Set**
- **rs.syncFrom(host:port)** → Realizamos un **sync**, trayendo los datos del host indicado
- **rs.freeze(secs)** → Durante **secs** segundos, no se puede convertir en **Primary**
- **rs.stepDown(secs)** → Si el nodo es **Primary** se vuelve **Secondary**, y además, durante **secs** segundos, no se puede convertir en **Primary**.
- **db.isMaster()** → Ya conocímos este comando, pero ahora cobra más importancia, ya que nos dice el rol que ocupa este nodo

4.7. Lecturas y Escrituras

Hemos visto que las escrituras deben pasar siempre por el nodo **Primary**, pero hemos dicho que si podemos trabajar con **Eventual Consistency**, no deberíamos tener problemas en leer de un nodo **Secondary**.

Conectémonos puesta un nodo **Secondary** e intentemos hacer una lectura. ¿Qué sucede?

```
mongo --port 27002
abc:SECONDARY> db.algo.find()
error: { "$err" : "not master and slaveOk=false", "code" : 13435 }
```

Por defecto, las lecturas de los nodos **Secondaries** no están permitidas, pero podemos especificar que permitimos lecturas de dichos nodos mediante `*rs.slaveOk()`

```
abc:SECONDARY> rs.slaveOk()
abc:SECONDARY> db.algo.find()
```

Recordemos que esto sólo lo debejmos hacer si nos podemos permitir trabajar con **Eventual Consistency**. Si quisiéramos dejar de permitir las lecturas, valdría con hacer **rs.slaveOk(0)**

Vamos a simular una caída en nuestro sistema, y ver cómo se adapta nuestro **Replica Set** a ello.

En primer lugar, obtengamos el **PID** de nuestro proceso

Windows:

```
nestat -a -n -o
```

UNIX

```
ps -edaf
```

Una vez obtenido, matemos el proceso que está usando el puerto del nodo **Primary** (podría ser por ejemplo el que está levantado en el puerto **27001**). Usar el administrador de tareas de windows para matarlo, o el **kill -9** en Unix.

Al haber matado al nodo **Primary**, debería haber en el **Replica Set** unas elecciones, y uno de los actuales nodos **Secondary** deberían ocupar el puesto del nodo caído

Vamos a ver si ha sucedido lo que esperamos. Nos conectamos a un nodo de los que eran **Secondaries**

```
mongo --port 27002
```

Ahora, lanzamos un **rs.status()**, lo cuál debería mostrar la configuración actual de nuestro **Replica Set**, indicando que hay un nodo caído, y quién ocupa actualmente el puesto **Primary**

```
rs.status()
```

Si nuestro nodo es el nuevo **Primary**, nos quedamos en él, si no, nos conectamos al nodo que haga de **Primary**. Ahora insertamos un documento:

```
db.foo.insert({_id:"post failover"})
```

Esto no debería haber dado ningún problema, se ha alzado un nuevo **Primary** y podemos seguir trabajando a pesar de tener un nodo caído.

Vamos a levantar ahora el nodo que se había caído:

Windows:

```
start /b mongod --port 27001 --replSet abc --dbpath replicaset/1 --logpath log/log.1  
--logappend --oplogSize 50 --smallfiles
```

Unix:

```
mongod --port 27001 --replSet abc --dbpath replicaset/1 --logpath log/log.1  
--logappend --oplogSize 50 --smallfiles --fork
```

Es interesante consultar el log de dicho nodo y ver qué pasos da para unirse nuevamente al **Replica Set**. Fijarse también en que anteriormente era **Primary**, pero al ingresar nuevamente en el **Replica Set**, ingresa de **Secondary**.

Vamos a confirmar que el nodo está unido correctamente y que la sincronización ha funcionado bien, para ello nos conectarmos a él y lanzaremos una query para confirmar que la inserción que habíamos hecho previamente, está presente (acordarse de permitir lecturas en **Secondaries**):

```
mongo --port 27001  
abc:SECONDARY> rs.slaveOk()  
abc:SECONDARY> db.foo.find()
```

Para estos ejemplos hemos permitido realizar lecturas desde nodos **Secondary**, cosa que conseguíamos mediante el **rs.slaveOk()**. Por favor, recordad que esto sólo debe hacerse si podemos trabajar con **Eventual Consistency**.

No obstante, a la hora de usar una aplicación, delegamos en el **driver** de qué nodo va a realizar las lecturas.

Hay que recordar que las escrituras **siempre** se realizan en el nodo **Primary**, y desde él se

propagan a los nodos **Secondaries**. Esto suele tardar unos milisegundos, pero podría ser más tiempo (debido a múltiples factores).

Si nos podemos permitir leer de un **Secondary**, descargamos de parte del trabajo al **Primary**, lo cual puede ser bastante necesario en muchas ocasiones.

Algunas razones por las que nos puede interesar leer de nodos **Secondaries** son:

- Geografía → Puede que nuestros nodos **Secondary** sean cercanos (y obtengamos la respuesta mucho más rápido), y el **Primary** sea lejano.
- Separar cargas de trabajo → Imaginemos que estamos lanzando unas operaciones analíticas pesadas, mejor saturar un nodo **Secondary**
- No saturar un nodo → Puede funcionar si la presión es ejercida por muchas lecturas, pero si el cuello de botella es la escritura, para mejorar esto debemos pasar al **Sharding**
- Disponibilidad → Un **Replica Set** puede llegar a tardar unos 10 segundos en descubrir que su **Primary** está caído. Si lo sabemos de antemano, podemos decidir ir a otro nodo directamente y ahorrar tiempo

Como ya hemos dicho, las lecturas en el **driver** son gestionadas por él, y como es **Replica Aware** el sabe quien es el **Primary** al que enviar las escrituras y quienes son los **Secondaries**. (Esto también tiene sentido cuando veamos **Sharded Cluster**, ya que **mongos** hace uso de estas mismas políticas)

De hecho las políticas de lectura para los **drivers** son bastante ricas, podemos especificar distintas políticas que pueden dotar de flexibilidad a nuestra aplicación:

- Primary → Opción por defecto, las lecturas se realizan todas del **Primary**
- PrimaryPreferred → Intentamos leer siempre del **Primary**, pero si detectamos algún problema (imaginamos que está caído por ejemplo), vamos a un **Secondary**.
- Secondary → Leemos exclusivamente de los nodos **Secondaries**, así evitamos saturar el **Primary**
- SecondaryPreferred → Intentamos leer de los nodos **Secondaries**, pero si no podemos, atacamos al **Primary**
- Nearest → Como tenemos un estado de cada nodo (obtenido mediante **heartbeats**), sabemos la latencia en la comunicación con cada uno. Usamos dicha información para atacar al nodo con menor latencia (puede que sea porque está más cerca, o está menos saturado...)

Como consejo:

- Cuando no tengamos claro qué hacer, usemos la opción por defecto, es decir **Primary**. Con ella tenemos **Strong Consistency**.
- Si usamos servidores que estén geográficamente distante, usar **Nearest** es una opción que mejorará drásticamente el rendimiento de nuestra aplicación.
- Para cargas de trabajo pesadas o analíticas, usar **Secondary** (no interesa saturar un nodo **Primary**)

Y recordar, **una escritura en el maestro puede transformarse en múltiples escrituras en los**

esclavos

Chapter 5. Ejercicios - Replicación I

5.1. Ejercicio 1

Para este ejercicio usaremos el fichero **replication.js**. Vamos a crear un **Replica Set** con tres miembros partiendo de un miembro con datos. En este ejercicio todavía no creamos el **Replica Set**

Desde la línea de comandos, ir a un directorio sobre el que trabajaréis y crear 3 directorios (de nombre 1, 2 y 3) para guardar los datos de cada miembro del **Replica Set**

Arracamos un servidor **mongod** con la siguiente instrucción (desde el directorio actual, y que apunte al directorio creado anteriormente)

```
mongod --dbpath replicaSetEjercicios\1 --port 27001 --smallfiles --oplogSize 50
```

En otro terminal, lanzamos un shell **mongo** que cargue el script **replication.js**

```
mongo --port 27001 --shell replication.js
```

Y ejecutamos **homework.init()**, que se encarga de añadir datos a nuestra base de datos:

```
homework.init()
```

Por último, ejecutar **homework.a()** para obtener un resultado y confirmar que todo es correcto. ¿Qué resultado obtienes?

```
homework.a()
```

Solución:

Desde el directorio donde vayamos a trabajar hacemos:

```
mkdir replicaSetEjercicios  
mkdir replicaSetEjercicios\1 replicaSetEjercicios\2 replicaSetEjercicios\3  
mongod --dbpath replicaSetEjercicios\1 --port 27001 --smallfiles --oplogSize 50  
mongo --port 27001 --shell replication.js
```

Una vez conectados a **mongod** lanzamos:

```
homework.init()  
homework.a()
```

RESULTADO: 5001

5.2. Ejercicio 2

En este ejercicio vamos a crear un **Replica Set** partiendo de una instancia existente de **mongod**. Para ello, vamos a parar nuestro proceso **mongod** y levantarla nuevamente con el parámetro **--replSet XXX**, siendo XXX el nombre que nos apetezca dar a nuestro **Replica Set**.

Una vez hecho esto, nos conectamos con **mongo** y lanzamos:

```
rs.initiate()
```

NOTA: **mongod** cogerá el nombre de tu **host** automáticamente. Asegúrate de que tu **hostname** está en tu fichero **hosts**

En el **Ejercicio 1** habíamos cargado datos. Confirmemos que están correctamente cargados haciendo una consulta sobre ellos:

```
use replication
db.foo.find()
```

Ahora, para confirmar que este ejercicio está finalizado, ejecutemos **homework.b()**. ¿Qué resultado obtienes?

```
homework.b()
```

Solución:

Matamos el **mongod** que hay levantado, con **CTRL-C** debería bastar si se ha levantado como se ha especificado

Lanzamos nuevamente nuestro proceso con la opción **--replSet Test** (o el nombre que hayamos elegido)

```
mongod --dbpath replicaSetEjercicios\1 --port 27001 --smallfiles --oplogSize 50
--replSet Test
```

Ahora lanzamos **rs.initiate()** y **homework.b()**

```
rs.initiate()
homework.b()
```

RESULTADO: 5002

5.3. Ejercicio 3

Ahora, vamos a agregar dos miembros a nuestro **Replica Set**, (porque un **Replica Set** de un miembro, es poco útil, la verdad).

Para ello, vamos a levantar dos procesos **mongod** nuevos, usando los puertos **27002** y **27003**, y los directorios creados en el **Ejercicio 1**, **replicaSetEjercicios\2** y **replicaSetEjercicios\3**. **Muy importante**, recordar que tenemos que levantarlos con el mismo nombre de **ReplicaSet** asignado al levantar nuestro **mongod** en el **Ejercicio 2**

Una vez hayamos levantado las dos instancias **mongod**, hay que añadirlas al **Replica Set** (que actualmente tiene tan sólo 1 miembro).

Podemos añadir miembros a nuestro **Replica Set** mediante el comando **rs.add(host:puerto)**, y consultar el estado de nuestro **Replica Set** con **rs.status()**.

Una vez creado nuestro **Replica Set** con los dos nuevos miembros, podemos conectarnos a uno de los **Secondary** con una shell **mongo** para hacer queries (acordarse de activar el **rs.slaveOk()**).

Una vez confirmado que todo está corriendo bien, y que los **Secondaries** se han sincronizado, ejecutar desde el **Primary** la instrucción **homework.c()** (perteneciente al script **replication.js**). ¿Cuál es el resultado?

Solución:

Arrancar 2 nuevos mongod (desde el directorio de trabajo)

```
start /b mongod --dbpath replicaSetEjercicios\2 --port 27002 --smallfiles --oplogSize 50 --replSet Test
start /b mongod --dbpath replicaSetEjercicios\3 --port 27003 --smallfiles --oplogSize 50 --replSet Test
```

Añadir los nuevos miembros al **Result Set** desde la consola **mongo** del **Primary** (recordar poner el nombre de vuestro host, evitad poner **localhost**)

```
rs.add("localhost:27002")
rs.add("localhost:27003")
```

Ahora lanzamos **homework.c()** desde el **Primary**. Si por algún casual hubiera salido, podemos conectarnos al **Primary** con nuestro **replication.js** de la siguiente manera

```
mongo --port 27001 --shell replication.js
```

Lanzamos **homework.c()**

```
homework.c()
```

RESULTADO: 5

5.4. Ejercicio 4

Vamos ahora a eliminar el primer miembro (el levantado en el puerto **27001**).

Para ello, vamos a pararlo en primer lugar (dado que el resto de miembros son suficientes para un **consenso**, se elegirá un nuevo **Primary**).

Podemos sencillamente matarlo como ya hicimos en la parte de teoría, o podemos hacer dos pasos, **rs.stepDown()** para dejar de ser **Primary** y luego matarlo (esto es mucho más seguro), bien como hemos hecho antes o desde la consola con **db.shutdownServer()**

Una vez se haya reconfigurado el **Replica Set**, conectarnos con la shell de **mongo** a uno de los nodos restantes (usando el script **replication.js**), y revisar la configuración actual del **Replica Set** usando **rs.status()**

```
rs.status()
```

Si no estamos en el nodo **Primary** (ahora sabemos cuál es gracias al **rs.status()**, salir y conectarlos al nodo **Primary**.

Una vez sepamos que estamos en el **Primary**, eliminar el nodo caído usando bien **rs.reconfig()** o **rs.remove()**.

Una vez finalizado, ejecutar **homework.d()**. ¿Qué valor te da?

```
homework.d()
```

OJO: Asegúrate de que los miembros de tu **Replica Set** tienen como **_id** 0,1 y 2. Si no, el resultado podría no ser el esperado

Solución:

Usamos **rs.stepDown()** desde el **Primary** (al que nos hemos conectado con **mongo**) para ceder la presidencia:

```
rs.stepDown()
```

Ahora matamos el proceso (por ejemplo en windows ejecutamos **netstat -a -n -o**, obtenemos su **PID** y lo matamos)

Nos vamos al nuevo nodo **Primary** (sabemos cuál es porque usamos el **rs.status()**). En este ejemplo el **Primary** está en el puerto **27003**

```
mongo --port 27003 --shell replication.js
```

Ahora, quitamos el nodo caído:

```
mongo --port 27003 --shell replication.js
```

Opción A: Ahora, cogemos la configuración actual con **rs.config()**, aplicamos cambios y reconfiguramos con **rs.reconfig()**

```
var cfg=rs.config()
//modificamos cfg para quitar el nodo que falla, en este caso imaginemos que es el primero
var miembros=cfg.members
cfg.members=miembros.splice(1,2)//cogemos 2 elementos desde la posición 1
rs.reconfig(cfg)
```

Opción B: Usamos **rs.remove()** para retirar el nodo caído (cuidado, usar el nombre de la máquina que corresponda en lugar de **localhost**)

```
rs.remove(localhost:27001)
```

Por último, lanzo **homework.d()**

```
homework.d()
```

RESULTADO: 6

5.5. Ejercicio 5

El estado actual de nuestra **Replica Set** contiene un número par de miembros. Esto no es deseable, no obstante, vamos a continuar así para finalizar estos ejercicios. Ahora toca trabajar con el **Oplog**.

Nota: El orden de estos pasos es muy importante.

Primero, vamos a ir al nodo **Secondary** de nuestro **Replica Set**. Ahora nos vamos a la base de datos local (con **use local**) y vemos el **Oplog** (No es necesario usar el **rs.slaveOk()** porque la base de datos **local** no se replica, es propia de cada nodo):

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

Vamos a consultar ahora datos sobre el **Oplog**

```
db.oplog.rs.stats()
```

¿Qué resultado devolverá la siguiente consulta?

```
db.oplog.rs.find({}).sort({$natural:1}).limit(1).next().o.msg[0]
```

NOTA: Si hemos jugado con nuestro **Replica Set** tanto que lo hemos llenado (Recordad que es una colección circular, cuando llega a su máximo va sobreescritiendo), y por lo tanto es posible que no tengamos los datos correctos. Si es así, hay que repetir desde 0 todos los ejercicios.

Solución:

Nos conectamos al **Secondary**

```
mongo --port 27002 --shell replication.js
```

Me voy a la base de datos **local** y lanzo las operaciones que me indicaron

```
use local
db.oplog.rs.find()
db.oplog.rs.stats()
```

Por último, lanzo la operación que debe devolverme el resultado finalizado

```
db.oplog.rs.find({}).sort({$natural:1}).limit(1).next().o.msg[0]
```

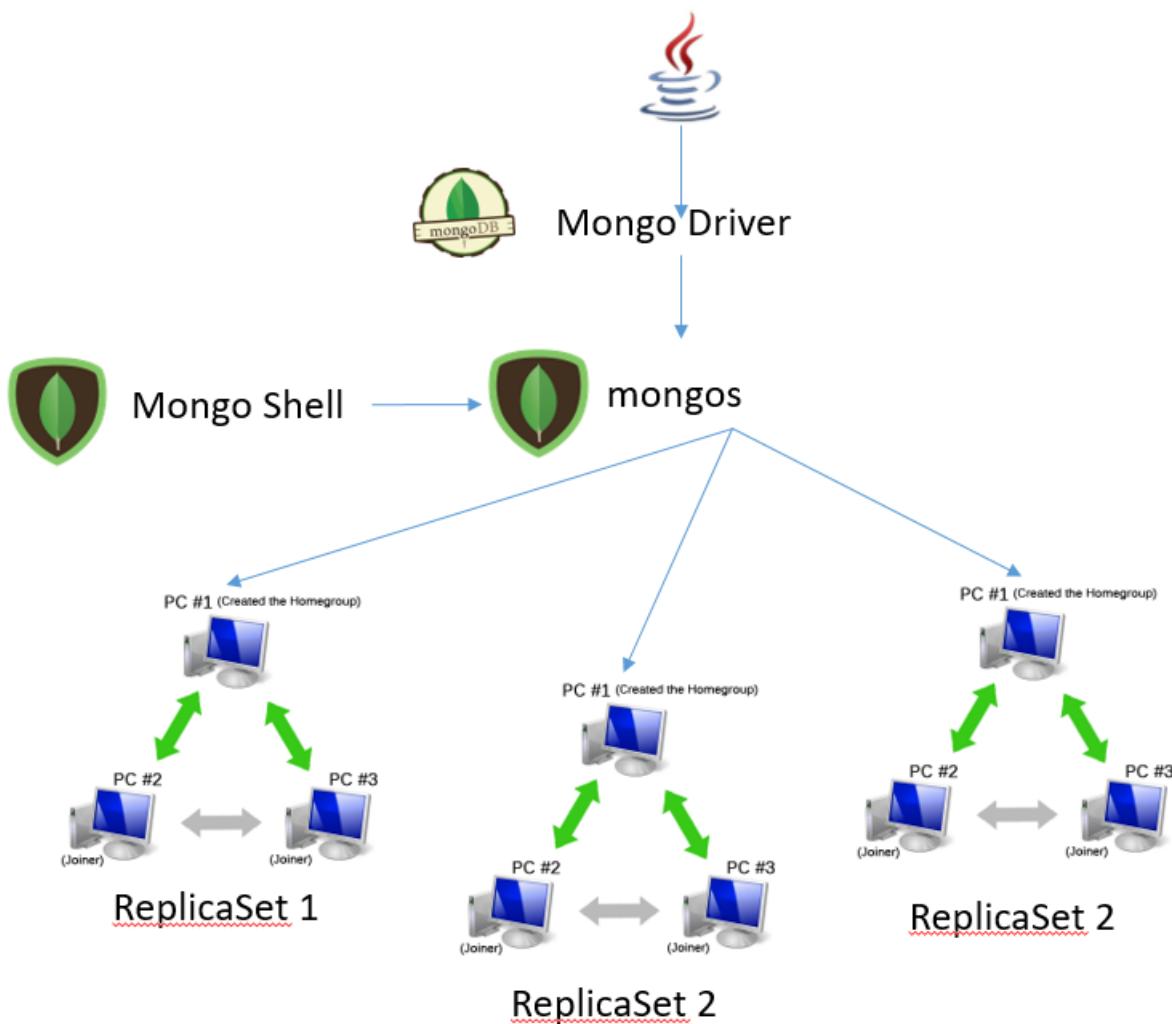
RESULTADO: R

Chapter 6. 1. Sharding

6.1. Introducción

Un **Shard** es una partición. En un **Sharded Cluster**, cada documento va a estar en una única partición del clúster, es posible que se mueva debido al **Balancer**, pero estará en un único **Shard**.

Para asegurar la disponibilidad, cada **Shard** es por debajo un **Replica Set**.



En **MongoDB**, cuando trabajamos con un **Sharded Cluster**, distribuimos los documentos a través de los distintos **Shards** existentes.

¿Cómo realizamos dicha distribución? A través de su **ShardKey**. Antes de distribuir una colección, elegimos una **ShardKey**, y asignamos rangos a los distintos **Shards**. Cuando buscamos un documento, podemos ir directamente al **Shard** que tiene asignado el rango al que pertenece su **ShardKey**.

Esto, además, implica que los documentos que estén cercanos (por el orden que dicta su **ShardKey**), suelen estar también en el mismo **Shard**.

Los rangos se especifican siempre con el valor mínimo (incluído en el rango) y el valor máximo (excluido del rango), es decir **[low-high]**

Un ejemplo, un clúster con tres shards, en el que particionamos por una clave **name**:

low	high	shard
jane	joe	S2
joe	kyle	S0
kyle	Mark	S1

Entender cómo funciona esto es crítico, puesto que nos permite hacer queries con desigualdades de manera eficiente. Por ejemplo, la siguiente query:

```
db.users.find({name:/^jo/})
```

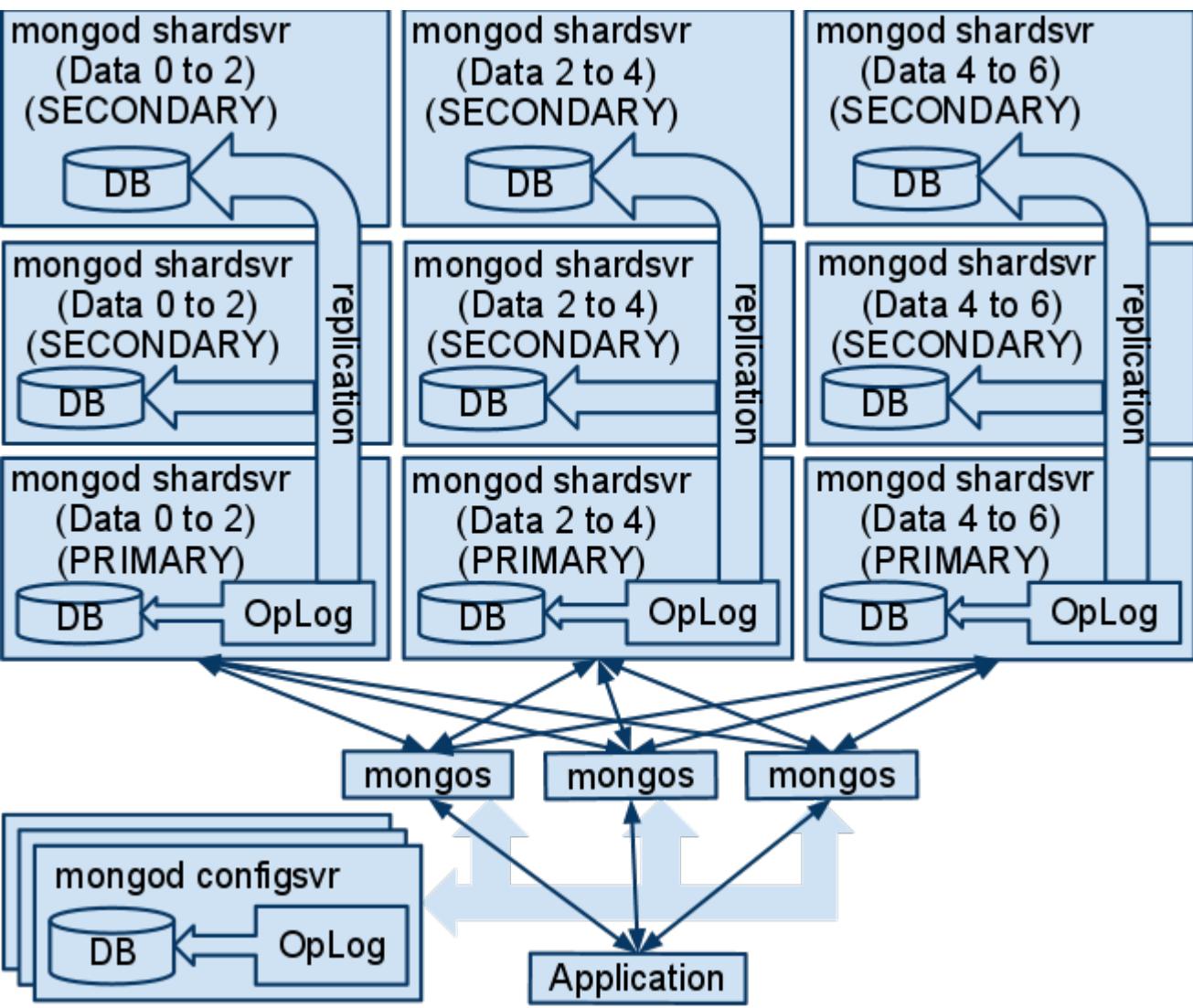
Como está definido en qué rangos trabaja cada **Shard**, dicha query buscará datos sólo en los **Shards S2 y S0**.

NOTE Si lanzamos una query que no posea **ShardKey**, será procesada en todos los **Shards**.

6.2. Replicación y Sharding

Es importante tener claro cómo funciona de manera conjunta el **Replica Set** dentro de un **Sharded Cluster**.

Cada documento asignado a un **Shard**, está asignado al **Replica Set** que implementa dicho **Shard**, y por lo tanto en dicho **Shard** van a existir tantas copias del documento en cuestión como factor de replicación tenga el **Replica Set**



6.3. Chunks y operaciones

Un **Chunk** es un conjunto de datos que abarca un rango de una **ShardKey**, como podía ser en el ejemplo anterior [Jane-Joe).

Un **Chunk** está por definición en un **Shard**, y tiene un tamaño por defecto de **64MB**, aunque es configurable (se puede configurar este tamaño desde **1MB** hasta **1024MB**). El tamaño es importante, puesto que con tamaños más pequeños existen más migraciones.

El sistema realiza dos operaciones automáticamente con respecto al **Sharding**:

- Split → Partir un **Chunk** en dos.
- Migrate → Movre un **Chunk** de un **Shard** a otro.

Imaginemos que un **Chunk** está creciendo más de la cuenta. ¿Cómo lo solucionamos?

Pues muy fácil, hacemos un **split**, es decir creamos dos rangos donde antes sólo había uno.

Por ejemplo, tenemos el rango **[Joe-Kyle)**, y este crece demasiado. El sistema calcula cuál es la mediana de dicha **ShardKey** y crea dos rangos a partir de dicha mediana:

- **[Joe-Kyle) → [Joe-Kate) , [Kate-Kyle)**

Esto, a bajo nivel, no es nada más que un cambio de **metadata**, puesto que no se ha movido ningún dato del **Chunk**. Todo está en el mismo servidor.

Split es por lo tanto una operación barata, puesto que realiza cambios en el **metadata**.

El siguiente paso sería migrar uno de los rangos o **Chunks** a otro servidor, esto ya es más costoso.

Ya hemos visto la operación **Split**, que nos sirve para partir un **Chunk** en dos. Pero esto sólo tiene sentido si posteriormente distribuimos uno de los **Chunks** a otro **Shard** (para mantener un equilibrio, no queremos tener un **Shard** con muchos **Chunks**).

Esta operación, llamada **Migración**, copia todos los datos de un **Chunk** a otro **Shard**, y cuando ha realizado dicha copia, borra los datos del **Shard** donde estaban originalmente, y además, modifica la **metadata** para indicar en qué **Shard** debemos buscar dicho **Chunk** ahora.

La operación **Migration** es más costosa que **Split**, pero es necesaria para mantener un equilibrio en el **Sharded Cluster**.

Durante la operación de **Migración**, se puede seguir trabajando con el **Chunk**, y los cambios realizados sobre dicho **Chunk** se propagarán también al **Shard** donde se está migrando.

¿Cuándo se hacen las migraciones? Cuando lo decide un componente llamado **Balancer**, (que podemos desactivar desde **mongos** con `sh.stopBalancer()`). El **Balancer** se basa **únicamente** en el número de **Chunks** que hay en cada **Shard**, no importa el consumo de cpu o tamaño.

Para cambiar el tamaño del **Chunk** hay que hacer los siguientes pasos:

- 1 → Conectarse a un proceso **mongos** usando la shell **mongo**
- 2 → Ir a la base de datos **config**

```
use config
```

- 3 → Realizar la siguiente operación (indicando el tamaño que queremos usar)

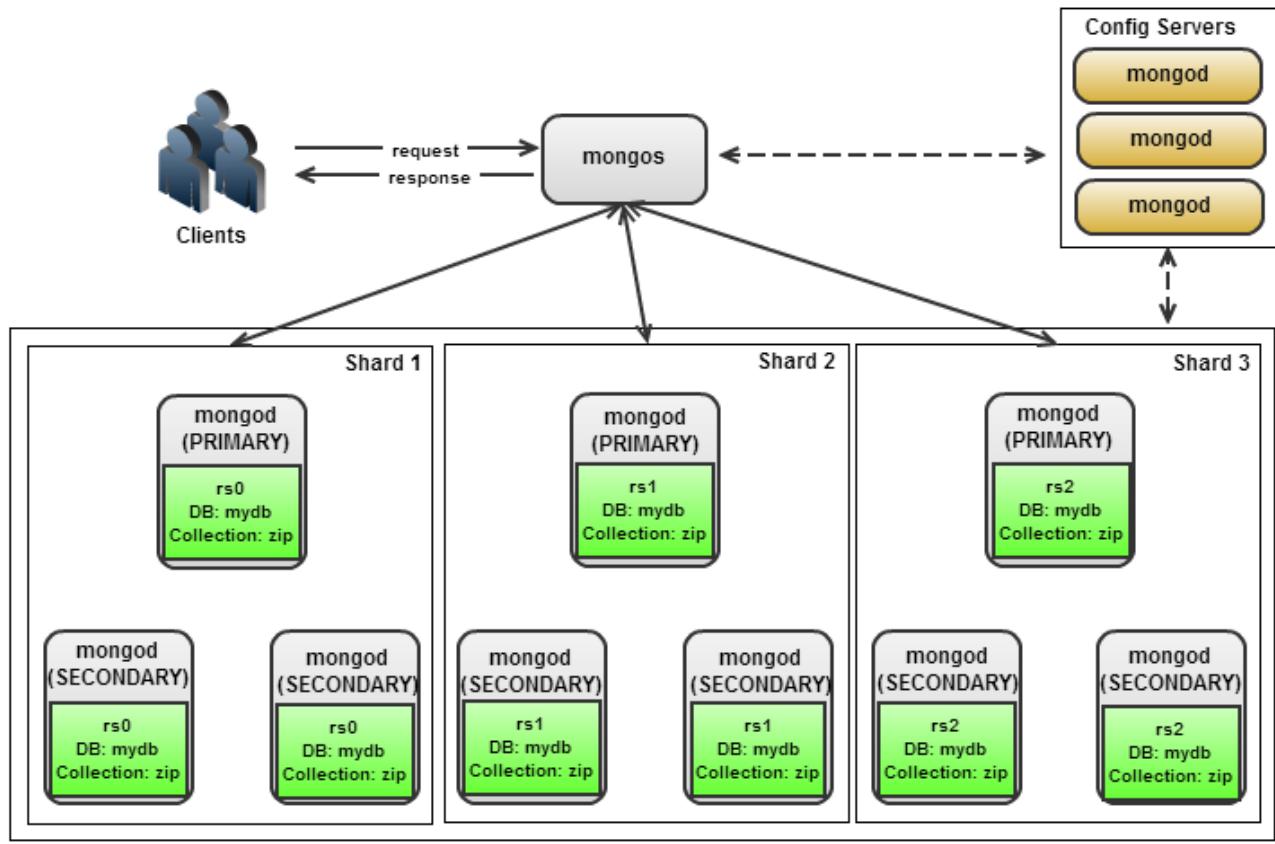
```
db.settings.save({ _id:"chunksize", value:<TamañoEnMB>})
```

6.4. Arquitectura del Sharded Cluster

Cada **Shard** está compuesto por varios servidores **mongod**, que van a pertenecer a **Replica Sets** que a su vez serán **Shards** del clúster.

Existen también otros procesos **mongod** que mantienen la **metadata**, a estos procesos los llamamos **Config Servers**.

Y por último, existen uno o varios procesos **mongos**, que son los frontales de nuestro clúster. La finalidad de los **mongos** es hacer transparente la arquitectura a la aplicación externa, sobre ellos mandamos nuestras peticiones y ellos las redirigirán al/los **Shards** correspondientes.



mongos posee el conocimiento de cómo está montado nuestro clúster, dónde están los rangos de cada **Chunk** de cada **colección**, y también se comunica con los **Config Servers**, que es de donde obtiene inicialmente estos datos.

- **mongos** no tiene persistencia, la información se obtiene de los **config servers** y es cacheada por *mongos.
- Los procesos **mongod** de los **Shards** guardan los datos de nuestra base de datos
- Los procesos **mongod** de los **Config Servers** guardan la **metadata** de la configuración.

Por lo general, en un clúster de producción hay 3 **config servers** (idénticos en datos), pero en desarrollo podemos usar 1. En producción usamos 3 por si hay discrepancias que pueda ganar la **mayoría**.

Si cae un **Config Server** y no hay mayoría para elegir un nuevo **Primary** (en versiones modernas trabajan como **Replica Set**, en antiguas NO), no podemos realizar cambios sobre la configuración, pero mientras haya uno levantado, podemos leer la **metadata**.

Si no tenemos la **metadata**, no sabemos qué **Shard** tiene qué datos.

6.5. Creando un Sharded Cluster

Antes de crear un clúster es siempre importante planificar qué buscamos. En **MongoDB** debemos hacernos siempre estas preguntas:

- ¿Cuántos **Shards** queremos inicialmente en nuestro clúster?
- ¿Cuáles el **Replication Factor** que buscamos en nuestro clúster?

En el ejemplo siguiente, montaremos un **Sharded Clúster** con 2 **Shards** y un **Replication Factor** de 2. Tened en cuenta que esto lo vamos a montar sobre una única máquina física, pero en **Producción** deberíamos tener tantas máquinas como procesos **mongod portadores de datos** vayamos a necesitar (Los **Config Servers**, los **mongos** y los **Arbiters** pueden levantarse en máquinas donde ya estén estos procesos)

Vamos a necesitar para este ejemplo:

- Servidores **mongod portadores de datos** → **ReplicationFactor x ShardsNumber**, en nuestro caso **2 x 2**, es decir, **4**
- Varios procesos **mongos** (por ejemplo 1, son procesos muy ligeros, no hay problema, pero como sólo tengo una máquina...)
- **Config Servers**, vamos a levantar 2, también son muy ligeros, así que no nos preocupemos.

Sobra decir que esta configuración tiene un fin pedagógico, y no debería ser aplicada en **Producción**, ya que no asegura la **disponibilidad**.

Para todo esto, elegiremos una ruta de trabajo (por ejemplo **mongoDB\shardTest**).

- 1 → Creamos los directorios para nuestros **Replica Sets** de cada **Shard** y nuestros **Config Servers**.

Windows y Unix

```
mkdir s0_0 s0_1 s1_0 s1_1 config_0 config_1
```

- 2 → Levantamos los **Config Servers** (Ojo, a partir de **Mongo 3.2** se pueden crear como **Replica Set**, vamos a hacerlo así)

Windows

```
start /b mongod --configsvr --dbpath config_0 --port 26050 --logpath log.cfg0  
--logappend --replSet configRS  
start /b mongod --configsvr --dbpath config_1 --port 26051 --logpath log.cfg1  
--logappend --replSet configRS
```

Unix

```
mongod --configsvr --dbpath config_0 --port 26050 --logpath log.cfg0 --logappend  
--replSet configRS --fork  
mongod --configsvr --dbpath config_1 --port 26051 --logpath log.cfg1 --logappend  
--replSet configRS --fork
```

- 3 → Levantamos los **Shards**, levantando cada miembro del **Replica Set** de cada **Shard**

Windows

```
start /b mongod --shardsvr --replSet S0 --dbpath s0_0 --logpath log.s0_0 --port 27000  
--logappend --smallfiles --oplogSize 50  
start /b mongod --shardsvr --replSet S0 --dbpath s0_1 --logpath log.s0_1 --port 27001  
--logappend --smallfiles --oplogSize 50  
start /b mongod --shardsvr --replSet S1 --dbpath s1_0 --logpath log.s1_0 --port 27100  
--logappend --smallfiles --oplogSize 50  
start /b mongod --shardsvr --replSet S1 --dbpath s1_1 --logpath log.s1_1 --port 27101  
--logappend --smallfiles --oplogSize 50
```

Unix

```
mongod --shardsvr --replSet S0 --dbpath s0_0 --logpath log.s0_0 --port 27000  
--logappend --smallfiles --oplogSize 50 --fork  
mongod --shardsvr --replSet S0 --dbpath s0_1 --logpath log.s0_1 --port 27001  
--logappend --smallfiles --oplogSize 50 --fork  
mongod --shardsvr --replSet S1 --dbpath s1_0 --logpath log.s1_0 --port 27100  
--logappend --smallfiles --oplogSize 50 --fork  
mongod --shardsvr --replSet S1 --dbpath s1_1 --logpath log.s1_1 --port 27101  
--logappend --smallfiles --oplogSize 50 --fork
```

- 4 → Arrancamos los procesos **mongos**, dando las direcciones de los **Config Servers**. Evitar usar **localhost**. Es muy recomendable levantar los **mongos** en el puerto por defecto de **MongoDB**, el **27017**, ya que no queremos que nadie se conecte por error a un **mongod** (salvo administradores)

Windows

```
start /b mongos --configdb configRS/localhost:26050,localhost:26051 --logpath  
log.mongos0 --logappend --port 27017
```

Unix

```
mongos --configdb configRS/localhost:26050,localhost:26051 --logpath log.mongos0  
--logappend --port 27017 --fork
```

- 5 → Iniciamos el **Replica Set** de nuestro **Config Server** (nos conectamos a uno de los 2 y añado el otro. Acordarse de sustituir el **host**)
 - 5.1 → Conectarse a un **mongod** del **Config Server**

```
mongo --port 26050
```

- 5.2 → Iniciar el **Replica Set** y añadir el otro nodo

```
rs.initiate()
rs.add("localhost:26051")
```

- 6 → Inicializar uno de los **Shards**
 - 6.1 → desde el shell **mongo** conectado a **mongos**, nos conectamos a uno de los **Shards**

```
db=connect("localhost:27000/test")
```

- 6.2 → Iniciamos el **Replica Set** y añadimos el otro miembro del **Shard** (Ojo, revisar que el **hostname** que nos ha dado por defecto es correcto, si no, cambiarlo).

```
rs.initiate()
//esta parte es sólo para cambiar el nombre del host por defecto
cfg=rs.config()
cfg.members[0].host="localhost:27000"
rs.reconfig(cfg)
//Fin de cambiar el nombre del host por defecto
rs.add("localhost:27001")
```

- 6.3 → Volvemos a conectarnos al **mongos** (también vale salir y volver a conectarse), y añadimos el **Shard**

```
db=connect("localhost:27017/test")
sh.addShard("S0/localhost:27000");
```

- 7 → Repetimos para el otro **Shards**
 - 7.1 → desde el shell **mongo** conectado a **mongos**, nos conectamos a uno de los **Shards**

```
db=connect("localhost:27100/test")
```

- 7.2 → Iniciamos el **Replica Set** y añadimos el otro miembro del **Shard**.

```
rs.initiate()
//esta parte es sólo para cambiar el nombre del host por defecto
cfg=rs.config()
cfg.members[0].host="localhost:27100"
rs.reconfig(cfg)
//Fin de cambiar el nombre del host por defecto
rs.add("localhost:27101")
```

- 7.3 → Volvemos a conectarnos al **mongos** (también vale salir y volver a conectarse), y añadimos el **Shard**

```
db=connect("localhost:27017/test")
sh.addShard("S1/localhost:27100");
```

Una vez hecho esto, ya tenemos nuestro **Sharded Cluster** levantado y funcionando, aunque NO estamos usándolo correctamente, puesto que para ello hay que decir qué bases de datos admiten **Sharding**, qué colecciones vamos a trocear, etc...

Podemos confirmar que no estamos haciendo nada porque las siguientes queries no devuelven datos:

```
use config
db.chunks.find()
db.shards.find()
```

Adicionalmente, recordemos que desde **mongos** podemos consultar la configuración de nuestro **Sharded Cluster** siempre a través de dicha base de datos, **config**. Algunas de las colecciones más interesantes son **chunks**, **databases**, **locks**, **mongos**, **shards**, **settings**...

6.6. Usando Sharding en Colecciones

Ahora que ya tenemos **Shards**, podemos activar el sharding en las colecciones (por defecto NO tienen sharding).

A la hora de aplicar **Sharding** sobre una **colección**, la ***ShardKey** establece cómo vamos a dividir dicha colección, y por lo tanto es crítico elegir una buena clave para tal fin.

Veamos un pequeño ejemplo en el que crearemos una colección y activaremos el **Sharding** sobre ellas

- 1 → Creamos una colección en una base de datos y simulamos que posee datos (recordad, todo a través del **mongos**)

```
use sharddb
for (i=0;i<2000;i++){ db.foo.insert({_id:i,x:"ES "+i}); }
```

- 2 → Activamos el **Sharding** para la base de datos

```
sh.enableSharding("sharddb")
```

- 3 → Para poder hacer **Sharding** sobre la colección, tenemos que crear previamente un índice sobre lo que queremos que sea nuestra **ShardKey**. En el ejemplo usaremos el único campo que ya posee un índice creado automáticamente, el **_id**.

```
sh.shardCollection("sharddb.foo", {_id:1}, {unique:true})
```

- 4 → Confirmamos que se ha realizado el **Sharding** correctamente, y que se han dividido los documentos en varios **Chunks** mediante `sh.status()` (también podemos llamar al método `stats()` de la colección para conocer los datos de la misma)

```
sh.status()
```

6.7. Trabajar en un Sharded Cluster

Trabajar con un **Sharded Cluster** es muy similar a trabajar con un servidor en solitario de **MongoDB**, gracias a la labor que realiza el **mongos**.

Sin embargo, hay varias cosas que debemos tener en cuenta a la hora de trabajar con un **Sharded Cluster**:

- Es **obligatorio** incluir la **ShardKey** al insertar un nuevo documento (que esté **Sharded**). Sin ella no podemos insertarlo
- Se pueden crear índices en las colecciones, y estos se crean en todos los **Shards** (tanto e los **Primary** como en los **Secondaries**)
- El mejor índice de todos los existentes es el que debemos usar como **ShardKey**, puesto que por lo general nos permite reducir el número de máquinas a las que atacamos con una query.
- Podemos obtener información interesante de lo que sucede al lanzar una query contra el **Shard** con `explain("executionStats")`, bastante útil para saber si tenemos una query de tipo **scatter gather**, es decir, que se lance en todos los **Shards**
- Podemos lanzar **Scatter Queries** que usen índices, pero siempre será peor que si usan la **ShardKey**, porque sobrecargan todos los **Shards**
- Usar parte de la **ShardKey** al hacer una query también suele salir rentable, porque como trabaja por rangos, atacará sólo a los **Shards** que corresponda.

6.8. Eligiendo Shard Keys

Un tema crítico a la hora de trabajar en un **Sharded Cluster** es saber elegir correctamente la **ShardKey** de una colección. A la hora de elegirla debemos tener en cuenta varios puntos:

- La **ShardKey** debería ser un campo sobre el que se realicen querys, es decir, debería ser la forma de atacar a dicha colección.
- La **cardinalidad** debe ser buena. Si disponemos de pocos valores posibles para nuestra **ShardKey**, no vamos a poder distribuir correctamente nuestros documentos. Como ejemplo exagerado, imaginemos que tenemos 5 posibles valores para dicha clave, y que uno de esos valores crezcan mucho. No podremos hacer un **Split** porque no podemos crear un nuevo rango.
- La **Monotonidad** de la clave también tiene que ser tenida en cuenta (La **Monotonidad** significa que una función sólo crece o decrece, en **MongoDB** hace referencia a que nuestro rango aumenta constantemente, bien por el límite inferior o por el superior). Un ejemplo claro es un campo de tipo **Timestamp**, siempre va a ir creciendo, e insertando valores sobre su último rango. Esto se traduce en que un único **Shard** va a ser quien reciba todas las operaciones

de escritura (el que posea el **Chunk** con un rango **[valor,+infinito)**). Ojo, los **ObjectId()** entran en esta categoría.

Conociendo esto, ya sabemos cuales son los puntos a tener en cuenta antes de elegir una **ShardKey** para una colección. Vamos a intentar ponerlo en práctica:

Tenemos una colección **pedidos** que tiene documentos como el siguiente

```
{  
  _id:78,  
  company:"Company Molona",  
  items:["item1","otro_item"],  
  date:ISODate("2010-05-01T00:00:00.000Z"),  
  total:45  
}
```

Una primera opción que podríamos considerar sería usar el **Company**, porque todas nuestras queries van a usar este campo. Sin embargo su cardinalidad puede no ser buena (a lo mejor casi todos los pedidos son de pocas compañías).

Como no vemos otro candidato mejor, tomamos la opción de crear una **ShardKey** compuesta, vamos a crear la clave usando **company** y **date**, puesto que el primer campo aparece con frecuencia y ya nos permite atacar a un número de **Shards**, y el segundo es probable que también se nos facilite.

6.9. Ejemplos de Sharded Clusters

Ya hemos visto que un **Sharded Cluster** posee:

- Servidores que contienen datos, que forman los **Replica Sets** de cada **Shard** (`mongod --shardsvr`)
- Servidores que contienen los metadatos (`mongod --configsvr`)
- Procesos **mongos**, para ser los frontales con los que se accede al clúster.

Pero a la hora de enfrentarnos en **Producción** a montar un clúster, debemos dedicar cómo distribuir físicamente todos estos procesos.

Vamos a ver qué opciones tenemos si dispusiéramos, por ejemplo, de 32 servidores.

Primera opción, elegimos un **Replication Factor** de 3, con lo que cada **Shard** está montado con un **Replica Set** de tres máquinas. Hay que recordar que cada proceso **mongod** que pertenece a los **Shards** debe ir en una máquina física propia, con lo que podríamos llegar a montar 10 **Shards** con **Replica Sets** de 3 máquinas. Nos sobrarían todavía dos máquinas.

Montaremos 3 **Config Servers**, y los **Config Servers** si pueden convivir con otros procesos **mongod**, por lo que podemos distribuirlos en las máquinas que forman los **Shards**. No obstante es recomendable:

- No ponerlos en el **Shard 0**, ya que ahí irán todas las colecciones que no estén **Sharded**

- No vamos a poner dos **Config Servers** en un mismo **Shard**

Ya sólo queda levantar procesos **mongos**. Estos procesos son ligeros, podemos distribuirlos en las máquinas que no usamos, incluso si queremos podemos levantar un **mongos** en cada servidor.

Otra opción, vamos a intentar trabajar con un **Replication Factor** de 2 (Por simplicidad vamos a pensar que disponemos de 16 máquinas).

En este caso crearemos 8 **Shards**, y levantaremos un **mongod** en cada servidor que pertenecerá a uno de estos **Shards**.

Como no es muy correcto tener un **Replica Set** de 2 miembros, añadiremos un tercer miembro de tipo **Arbiter** que puede convivir con otros procesos **mongod**, lo único, asegurarnos de que no estamos en la misma máquina que está algún proceso **mongod** de nuestro **Shard**.

Los **Config Servers** los distribuimos en el clúster, como consideremos (por ejemplo, en máquinas que no tienen **Arbiter**)

Y por último, levantamos un proceso **mongos** en cada máquina.

Nota: Si llegamos a tener un clúster muy grande, de cientos de servidores, si puede llegar a interesarnos tener los **Config Servers** por separado, puesto que ya empiezan a gestionar una cantidad de información considerable. (Esto también podría suceder con los **mongos** en clústers muy grandes).

6.10. Pre-Splitting

En ocasiones, nos vemos en la necesidad de hacer cargas iniciales masivas, **Bulk loads**. En este caso, si estamos realizando inserciones sobre una **Sharded Collection**, vamos a perder mucho tiempo haciendo **splitting** y **migrations** de **Chunks**.

Ante estas situaciones, en las que corremos el riesgo de sobrecargar un **Shard**, nos puede interesar preparar nuestra colección para distribuir los datos futuros en **Chunks** pre-definidos.

Esta situación **NO** es el comportamiento normal, y debemos aplicarla sólo cuando tengamos pensado realizar una carga masiva.

Para poder hacer esto, recurriremos al método `sh.splitAt("database.collection", {"shardkey":splitValue})`, que crea un nuevo **Chunk** usando el valor indicado de la **ShardKey** para partir el **Chunk** inicial.

Vamos a simular esto, concretamente vamos a partir una colección en 100 **Chunks** distintos, sobre un **ShardKey** que puede tomar valores en el rango **[0,1000000]**, para que luego la inserción sea más eficiente:

- 1 → Activamos el **Sharding** para la colección (que debe poseer un índice en el campo **x**, y cuya **base de datos** debe tener el **Sharding Enabled**)

```
sh.shardCollecton("sharddb.pre_demo", {x:1})
```

Ahora, solicitamos que se creen los **Chunks**

```
for (var i=0;i<100;i++){
    sh.splitAt("sharddb.pre_demo",{x:1000000*1/100});
}
```

Podemos consultar los **Chunks** existentes desde la base de datos **Config**, a través de la colección **chunks**:

```
use config
db.chunks.find({ns:/demo/},{_id:0,lastmodEpoch:0,lastmod:0})
```

NOTA: Recordad que **split** sólo crea los rangos de los **Chunks**, pero es el **Balancer** quien se encarga de distribuir los **Chunks** a través de los distintos **Shards**, por lo que hay que esperar un poco desde que los creamos hasta que se redistribuyen (recordad que el **Balancer** sólo se basa en el número de **Chunks** que hay en cada **Shard**, le da igual lo llenos que estén, o lo usados que sean).

6.11. Sharding basado en Hash

Desde **Mongo 2.4** se puede realizar un nuevo tipo de **Sharding**, basado en códigos **Hash**.

Este tipo de **Sharding** se puede usar exclusivamente con índices únicos, si vamos a usar una clave no única habrá que combinarla con otros campos para asegurar su unicidad.

Usar un **Sharding** basado en **Hash** tiene las siguientes características:

- Tiene una alta cardinalidad
- Si tenemos una alta **monotonía** evitamos colapsar un único nodo
- Puede ser problemático con números en coma flotante (ya que los trunca a enteros de 64-bits, esto haría que colisionaran valores como 2.3 y 2.7)
- No podemos hacer búsquedas en rangos sobre sólo un subconjunto de **Shards** basados en su orden original (pues se pierde, al distribuirse en los **Shards** mediante su código **Hash**)
- Los documentos "cercanos" según su orden ya no tienen altas posibilidades de estar en el mismo **Shard**.

Para usar este tipo de **Sharding** tan sólo tenemos que indicar al hacer **Sharding** de la colección que queremos que sea tipo **Hash**

```
sh.shardCollection("database.collection",{_id:"hashed"})
```

Al igual que pasa con cargas iniciales pesadas, es recomendable hacer un **Pre-Splitting** en varios **Chunks**. No obstante, al ser el rango existente siempre el definido por un **Hash**, y distribuir uniformemente los datos, es mucho más fácil definir los **Chunks** iniciales al hacer el **Sharding** de la colección, y para ello tenemos que recurrir al **comando** (que no al **helper**) para poder especificar

este parámetro:

```
{  
    shardCollection: "<database>.<collection>",  
    key: <shardkey>,  
    unique: <boolean>,  
    numInitialChunks: <integer>  
}
```

Quedaría tal como sigue (para hacer 6 **Chunks** iniciales):

```
use admin  
db.runCommand({shardCollection: "database.collection",  
    key:{_id:"hashed"},  
    numInitialChunks: 6  
})
```

Nota: Tambié se pueden hacer **hashed indexes**, siempre y cuando no se hagan sobre **Arrays**. Su creación es simple:

```
db.collection.createIndex({campo:"hashed"})
```

6.12. Eliminando un Shard

En algunas ocasiones podemos vernos ante la necesidad de eliminar un **Shard** de nuestro clúster, pero esta decisión no debe ser tomada a la ligera, ya que va a ser un proceso **lento**.

Los pasos a dar para eliminar un **Shard** de nuestro clúster son:

- Ejecutar el comando **removeShard**, y muy importante, asegurarnos de que el **Balancer** está activo, si no nunca llegará al segundo punto.
- Esperar a que los **Chunks** se migren por completo.
- Mover cualquier base de datos que esté en dicho **Shard** como su primario (usar **movePrimary**)
- Ejecutar el comando **removeShard** de nuevo.

Para poder hacer esto, es útil conocer la siguiente información:

En la base de datos **config**, podemos usar **db.database.find()** para conocer cuales son los **primary shards** para cada base de datos. Recordar que las colecciones que no estén **Sharded** se van al **primary shard**

```
use config  
db.database.find()
```

Al ejecutar el comando **removeShard** nos dice si tenemos la necesidad de mover bases de datos.

```
db.adminCommand({removeShard:"shard01"})
```

Una opción para ver si se han terminado de mover los **Chunks** tras lanzar el comando **removeShard** es volver a lanzarlo, si todavía no ha acabado nos dirá **drainning ongoing**.

6.13. Consejos y buenas prácticas

A la hora de trabajar con un **Sharded Cluster**, es recomendable seguir los siguientes consejos:

- Usar **Sharding** sólo en colecciones grandes, ya que en una colección pequeña no mejoramos el rendimiento, y crearlo supone un trabajo extra.
- Tener cuidad a la hora de elegir el **ShardKey**, porque **NO** se puede cambiar el **ShardKey** una vez creada la colección (habría que migrar a otra colección nueva).
- Si tenemos pensado hacer una carga masiva, es recomendable realizar un **Pre-Splitting**.
- Se consciente de los valores que tengan un comportamiento **monotónico**, porque siempre irán a un **Chunk** concreto.
- Añadir **Shards** al clúster es muy sencillo, pero no es instantaneo (puesto que el **Balancer** se tiene que encargar de mover y redistribuir los **Chunks**).
- Siempre debemos conectarnos al **mongos**, salvo para tareas concretas de administración. **mongos** debe estar levantado en el puerto por defecto de **MongoDB**
- Usar para los nombres de los **Config Servers** nombres lógicos, puesto que si no puede ser complicado de cambiar o gestionar.