UNIDAD 5: BASES DE DATOS NO SQL

Módulo profesional: Análisis de grandes datos (Big Data)

Índice

RESUMEN INTRODUCTORIO

En esta unidad vamos a presentar las características de las bases de datos NoSQL, desde su origen a su aplicación.

Del mismo modo, desde una perspectiva más práctica, conoceremos los fundamentos de MongoDB, desde su instalación hasta como almacenar información en ella, recuperarla y modificarla posteriormente y optimizar el sistema gestor para que pueda soportar volúmenes de datos que realmente puedan ser considerados big data.

Conocer las diferencias y el funcionamiento de una base de datos NoSQL es fundamental en el desarrollo de aplicaciones y en la administración de sistemas, dado que cada vez son y serán más los sistemas de información que se van a construir sobre dichos repositorios de sistemas.

El aprendizaje en bases de datos NoSQL puede presentar una dificultad añadida sobre las bases de datos relacionales. Esto es debido a que, en el segundo caso, la estructura y diseño de estas era idéntico y, por tanto, las particularidades de cada implementación diferente de un fabricante concreto existían, pero no dificultaban mucho pasar, por ejemplo, de un sistema gestor de Microsoft a uno de Oracle.

En el primer caso nos encontraremos que cada fabricante en concreto desarrolla una infraestructura de repositorio diferente a la hora de albergar y gestionar la información almacenada. Por este motivo, aunque la información guardada normalmente puede ser exportada sin problemas (si se opta por cambiar de repositorio) la administración de este puede variar de forma significativa.

Es común ver que, en los sistemas basados en NoSQL, su arquitectura es totalmente distribuida.

En esta unidad nos centraremos en conocer qué es MongoDB, sus principales características, ventajas y desventajas. Analizaremos en qué MongoDB es diferente a otras bases de datos y para comenzar, debemos tener claro que MongoDB rompe el paradigma de las bases de datos relacionales, proponiendo un nuevo paradigma orientado a documentos, muy flexible y fácil de entender para los desarrolladores, almacenando los datos en objetos con una estructura de tipo JSON.

INTRODUCCIÓN

Las bases datos están presentes en prácticamente cualquier sistema de información. Los datos, a nivel mundial, empezaron a crecer de una manera explosiva gracias a la llegada de tecnologías como internet o la telefonía móvil. Debido a la gran cantidad de datos que se manejan es casi imposible manipularlos a través de una base de datos convencional (SQL).

En cambio, con la llegada de las bases de datos no relacionales (NoSQL) se modifica la manera en cómo los sistemas gestores la manejan, ya que cuentan con mucha flexibilidad.

El término de Not only SQL (NoSQL), no está del todo claro cuando nació. Es probable que el primero en utilizarlo fuese el ingeniero Carlos Strozzi, en el año 1998. Aunque quien consiguió popularizarlo fue en 2009 Eric Evans, que volvió a retomar la idea, pero dándole un toque personal, haciendo referencia que este tipo de bases de datos se diferencian de las demás por la cantidad de datos manejados.

La mayoría de las bases de datos relacionales suelen presentar fallos no estructurales, sino en cuanto a la eficiencia al usar determinadas aplicaciones que manejan datos de manera intensiva, como por ejemplo ocurre con los sistemas de indexado de internet o las grandes colecciones de documentos.

A pesar de que las bases de datos relacionales han mantenido una posición de dominancia en el mercado de almacenamiento de información en los últimos cuarenta años y que nada hace prever que vayan a desaparecer en los próximos, es verdad que por primera vez una alternativa se asienta en el mercado.

Nos estamos refiriendo a las bases de datos NoSQL que aspiran a superar las limitaciones y deficiencias que desde hace tiempo se sabían de las bases de datos relacionales pero que realmente no habían sido problemáticas hasta los últimos tiempos, en los cuales el surgimiento y posterior generalización del big data ha dejado claro las limitaciones del modelo relacional.

Las dos bases de datos más populares en lo que es el paradigma NoSql son MongoDb y Cassandra. De modo general, ambas presentan las ventajas comunes de superar el modelo relacional, aunque entre ellas también presentan diferencias que hacen que se deba elegir cuidadosamente cuál de las dos (o una tercera) son las que aportarán mayor flexibilidad a la infraestructura de almacenamiento que vayamos a usar en nuestros proyectos.

CASO INTRODUCTORIO

Trabajas en una editorial especializada en la publicación de contenidos jurídicos como legislación, jurisprudencia, análisis legales, manuales, libros, etc.

La editorial está interesada en desarrollar una biblioteca jurídica universal en internet donde estén publicadas todas las leyes a lo largo de la historia de todos los países, así como la jurisprudencia, que son todas las sentencias dictadas por los juzgados.

Sin duda, el volumen de documentos es amplio. Además, ese tipo de documentos requiere de ser buscado a texto completo, es decir, cualquier persona interesada en consultar esa biblioteca querrá introducir una palabra, una frase, un nombre propio, etc., y que al momento el sistema le devuelva los resultados más pertinentes.

Es un proyecto muy ambicioso y debes encargarte de él y de todas las cuestiones técnicas que lo rodean.

Al final de esta unidad sabrás y comprenderás cuestiones como:

- Distinguir las características de una base de datos relacional (SQL) de una base de datos NoSQL
- Determinar en qué situaciones un sistema de información estará mejor soportado mediante una base de datos relacional o una base alternativa.
- Instalar la base de datos NoSQL más popular que es MongoDB y comenzar a practicar con ella para acostumbrarte a este nuevo paradigma.

1. QUÉ SON LAS BASES DE DATOS NOSQL

Hasta ahora no habías trabajado con bases de datos NoSQL, y en cuanto te asignaron la responsabilidad sobre este nuevo proyecto, fuiste consciente de que tenías que formarte en este sector.

La primera impresión te decía que ni la mejor de las infraestructuras conseguiría que una base de datos relacional fuese capaz de gestionar con flexibilidad y rapidez la gran cantidad de documentos que se pretenden explorar.

Por este motivo, se tomó la decisión de que era preferible que tanto tú como tu equipo técnico os formaseis en bases de datos NoSQL.

Una vez decidida la apuesta por un repositorio NoSQL, debéis elegir uno en concreto. Por un lado, puedes optar por MongoDB, por otro lado, Cassandra, ambas tecnologías muy populares y con grandes similitudes como su vinculación al open source, pero también con grandes diferencias.

Cassandra, a diferencia de MongoDB, almacena la información bajo el paradigma clave-valor. Finalmente, la decisión se decantó hacia MongoDB.

Las **bases de datos relacionales** son todas aquellas en las cuales los datos se acceden a través de relaciones previamente establecidas. Están compuestas por tablas y cada tabla a su vez posee campos y registros. Estas bases de datos fueron propuestas por Edgar Frank Codd, de los laboratorios de IBM en 1970 y podríamos decir que prácticamente todas estas bases de datos utilizan SQL para el mantenimiento y consulta de las mismas.

Ahora que ya sabemos cuáles son las bases de datos relacionales, podemos entender cuáles no lo son. Las bases de datos no relacionales también son conocidas como NoSQL, aunque en algunas ocasiones también se les denomina "no solo SQL", ya que algunos sistemas gestores si que lo soportan.

Se caracterizan por no poseer el tipo habitual de estructura de tablas y relaciones, así como una gran escalabilidad y rendimiento cuando se trata ciertos tipos de datos.

2. VENTAJAS DE LOS SISTEMAS NOSQL

A priori, la descripción del proyecto deja bastante claro que las ventajas y beneficios que aportará el uso de una base de datos relacional van a ser muy superiores a los posibles inconvenientes.

Por un lado, si se realiza un análisis de cómo están trabajando otras editoriales de diferentes sectores, todos sus sistemas de distribución de contenidos online están basados en repositorios que no son SQL.

Probablemente el principal beneficio que aportará la infraestructura de MongoDB o de otra base de datos orientada a documentos es que permitirá la indexación total del contenido. Es decir, cualquier porción de información albergada en la legislación, jurisprudencia, informes, etc., será localizable en unos tiempos razonablemente rápidos, lo que está fuera del alcance incluso de los sistemas gestores de bases de datos relacionales más poderosos del mercado.

Las **bases de datos SQL** o **bases de datos relacionales** están diseñadas para realizar transacciones fiables, pero como se fundamentan sobre un esquema rígido, esto provoca una serie de restricciones que limita su utilización en muchas aplicaciones. Las bases de datos NoSQL surgieron en respuesta a esas limitaciones de las bases de datos relacionales.

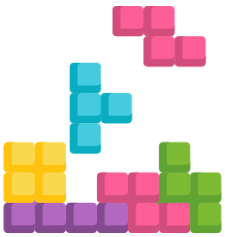
Los sistemas NoSQL nos permitirán manejar grandes cantidades de datos a una gran velocidad, los que ofrece muchas posibilidades para los desarrolladores.

Las bases de datos NoSQL soportan estructuras distribuidas lo que las convierte en fácilmente escalables. Además, como no cuentan con un esquema rígido como el de las bases de datos relacionales se pueden adaptar con más facilidad a los diferentes proyectos.

Los cambios que necesitemos realizar en el esquema de un sistema NoSQL se pueden hacer sin necesidad de detener las bases de datos.

Permiten realizar una escalabilidad en horizontal de forma que si necesitamos aumentar la capacidad del sistema tan solo tendremos que aumentar el número de equipos que no necesitan grandes recursos.

Ventajas de los sistemas NoSQL
Fuente: Elaboración propia



EJEMPLO PRÁCTICO

Trabajas en el departamento de infraestructura y bases de datos de una compañía de distribución eléctrica regional con aproximadamente 100.000 clientes.

Hasta el momento actual, el sistema de facturación a los clientes consistía en que una vez al mes un inspector de la compañía eléctrica visitaba a los clientes y tomaba manualmente la medida de los KW/hora consumidos por cada uno de ellos.

Posteriormente, esa medición se trasladaba al departamento de facturación que calculaba el consumo diferencial desde la anterior medición y aplicaba al cliente la tarifa y descuentos que correspondiese.

Para almacenar el histórico de las mediciones en el departamento de informática se estaba utilizando una base de datos relacional.

Sin embargo, se ha producido un cambio regulatorio y a partir de dentro de 6 meses tendréis que cambiar los contadores de la luz tradicionales por contadores inteligentes, capaces de registrar y tomar la medición del consumo cada minuto para poder cumplir con la ley y cobrar el cliente el KW/hora al coste de producción (por ejemplo, por las noches es muy más barato que en horario diurno).

De esta forma, si antes la base de datos soportaba:
100.000 clientes x 1 medición x 12 meses= 1.200.000 mediciones cada año.

Ahora el total de registros nuevos de la base de datos será:
100.000 clientes x 1 medición x 24 horas al día x 30 días al mes x 12 meses al año=864.000.000 de registro.

Esto quiere decir que el volumen de información (supongamos que cada registro futuro ocupa los mismos bytes que los registros históricos) se multiplica al menos por 720 veces, lo que significa que es un cambio de más de 2 órdenes de magnitud, por lo que no se trata de un cambio incremental.

Dado el tremendo cambio en el orden de magnitud de la información manejada debido a un cambio regulatorio es de presuponer que muchos aspectos de la infraestructura informática deben cambiar.

Aunque es probable que la base de datos haya ofrecido un rendimiento suficiente hasta la fecha, es previsible que al multiplicar casi por 1.000 la información gestionada se presente graves problemas por lo que debemos ser previsores y, como mínimo, realizar pruebas de carga.

En caso de que el resultado de las pruebas de carga previsionales constaten la dificultad para seguir adelante, será necesario evaluar la idoneidad de migrar a una base de datos NoSQL que ofrezca mejor respuesta al resto del internet de las cosas, que es la realidad a la que una compañía eléctrica se enfrenta con el uso de los contadores digitales inteligentes.

Aunque es probable que desde un punto de vista purista este escenario no puede ser calificado como big data porque, aunque se trata de un gran volumen de datos, la variedad de estos no es tal, sin duda un repositorio como MongoDB dará buena respuesta a las necesidades de la compañía comercializadora de energía.

3. PRINCIPALES DIFERENCIAS CON LAS BASES DE DATOS SQL

A lo largo de la historia del uso de sistemas informáticos en la editorial, se han utilizado las bases de datos relacionales. Por ejemplo, la base donde se almacenan los datos sobres los distribuidores y los clientes de los libros que se publican se ha construido mediante una base de datos relacional. En ella se habrá dispuesto de una tabla para registrar cada libro editado, otra para almacenar los datos de cada cliente, en otra se albergarán las compras de libros que ha realizado cada uno de los clientes, etc. Esta normalización es común en las bases de datos relacionales, donde la estructura de cada una de las entidades es homogénea y, salvo excepciones, inamovible a lo largo del tiempo.

Por el contrario, para este nuevo proyecto, será mucho más útil tratar la información como colecciones de documentos que pueden ser heterogéneos entre ellos, sin necesidad de que respeten una estructura común. Por ejemplo, cuando se almacena una ley, probablemente ese documento incluya datos como el rango de esta (autonómica, estatal, europea, internacional…), mientras que los documentos que correspondan a sentencias contendrán información sobre el juzgado donde se emitieron, las partes implicadas, sus respectivos abogados y procuradores, etc.

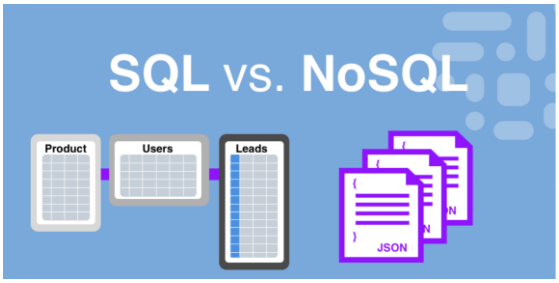
La **diferencia fundamental entre SQL y NoSQL** radica en que cada uno tiene una filosofía diferente de cómo se deben almacenar y recuperar los datos.

En una base de datos relacional o SQL la información se almacena en campos de datos estructurados en columnas definidas, pero esto no ocurre en los sistemas NoSQL.

Una base de datos relacional como MySQL utiliza un esquema, una definición formal de cómo se dispondrán los datos insertados en la base de datos. Por ejemplo, una columna determinada de una tabla puede estar restringida sólo a números enteros, otra a fechas, otra a texto, etc. Como consecuencia, los datos registrados en cada columna estarán muy normalizados. En la estructura de un sistema SQL se registran también las relaciones entre las tablas que forman toda la base de datos.

Esto no ocurre en las bases de datos NoSQL. En NoSQL las estructuras de los datos que almacenemos pueden ser variables y, como consecuencia, en aquellos casos en los que necesitemos que los datos sean completamente consistentes, lo ideal será utilizar sistemas SQL.

Las bases de datos NoSQL están orientadas a colecciones y documentos y acostumbran a usar formatos como JSON. En este caso no existen relaciones fijadas de forma previa entre los datos almacenados. En una base de datos NoSQL tenemos colecciones de documentos en los que se almacena toda la información.



Diferencia entre sistemas SQL y NoSQL
Fuente: https://acodez.in/sql-and-nosql-an-overview/

CUANDO USAR UNA BASE DE DATOS SQL	CUANDO USAR UNA BASE DE DATOS NOSQL
Ámbito educativo para mantener una estructura lógica de la información.	Big Data.
Inteligencia y análisis de negocio para identificar patrones de los datos.	Redes sociales.
Software empresarial para mantener una estructura consistente.	Cloud (Xaas).

Algunas **recomendaciones de uso de sistemas SQL y sistemas NoSQL,** son que, en el caso del MongoDB, como ejemplo de base de datos NoSQL, presenta distintas ventajas frente a bases de datos relacionales. Por ejemplo, las validaciones dentro de los esquemas hacen que MongoDB sea una herramienta flexible y muy completa, que facilita la integridad de los datos y tener control sobre lo que ocurre en el servidor.

Aporta una gran **potencia** al trabajar con índices tanto únicos como parciales y compuestos, que son de gran importancia en las collections para mejorar el rendimiento de las consultas. MongoDB utiliza todo el poder de los índices para ofrecer tiempos de respuesta óptimos, algo imprescindible si tenemos en cuenta que las bases de datos en MongoDB (y en otras bases de datos NoSQL) suelen tener gran volumetría.

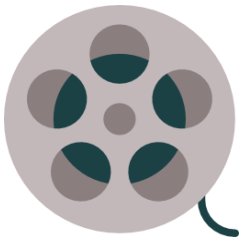
Crear **índices sencillos**, compuestos, sobre arreglos y objetos anidados es muy sencillo en bases de datos NoSQL y son maneras comunes para hacer que las consultas NoSQL funcionen de modo más rápido y eficiente. También en el caso de MongoDB una de las ventajas disponibles es poder afinar las bases de datos realizando análisis sobre las consultas con el método explain. MongoDB y otras bases no relacionales ofrecen muchas posibilidades para organizar los datos y salvaguardarlos, como son el uso de réplicas y el particionado.

El uso de **réplicas** ayuda a mantener los servicios activos y los datos bien protegidos. El uso de particionado permite sacar el mayor provecho del hardware, pudiendo manejar grandes cantidades de datos.

También podemos citar como beneficios de las bases de datos **not only SQL** que suelen incorporar una serie de herramientas que nos permiten analizar cómo funcionan a través de las cuales es posible localizar fallos en nuestro diseño y corregirlo, algo que redundará en un mejor rendimiento a futuro.

De este modo, si mantenemos un buen modelado de datos, evitaremos los **cuellos de botella**. Aunque las bases de datos relacionales, sobre todo aquellas de tipo más empresarial, también cuenta con herramientas que ofrecen gran cantidad de información sobre lo que están haciendo, cuánto dura una operación, qué tipo de operación es, etc., en las bases NoSQL es más frecuente teniendo en cuenta que normalmente están orientadas a soportar una carga de trabajo superior a las bases de datos SQL.

Muchas de estas **ventajas** de las bases NoSQL se podrían resumir diciendo que incluso si no disponemos de una gran infraestructura, con pocos recursos, se puede hacer mucho y lograr una alta disponibilidad de datos de manera eficiente.



VIDEO DE INTERÉS

En el siguiente vídeo se muestran las diferencias entre MongoDB, utilizado en sistemas NoSQL y MySQL que se usa en bases de datos relacionales

https://youtu.be/b_zr8t2g2Ic

4. TIPOS DE BASES DE DATOS NOSQL

Decidir el tipo de base de datos NoSQL a utilizar en el proyecto es una decisión relevante y cada uno de los subtipos de base de datos presenta aspectos positivos y negativos. En este caso, se podría dudar entre una base de datos orientada a documentos y una base de datos orientada a grafos.

Es probable que, para tomar esta decisión, no sea suficiente hacerlo desde una perspectiva informática, sino que también será necesario conocer las características de la información a almacenar. Por ejemplo, puede que haya expertos que opinen que para desplegar una biblioteca universal sea más interesante utilizar una base de datos orientada a grafos, porque permitiría gestionar las relaciones y la semántica de la información almacenada.

Si en vez de información de tipo jurídico se tratase de una biblioteca de información histórica, sí que podría ser más interesante el uso de grafos. Sin embargo, un experto en temas legales y jurídicos probablemente opine que el valor de los grafos no es relevante en el contexto que nos estamos manejando. Por este motivo, las ventajas de un sistema como MongoDb tenga más peso que experimentar con otras tecnologías como los grafos que pueden agregar mayor incertidumbre.

En la complejidad y variedad de los gestores de bases de datos, existen distintas formas, modelos o tipos que varían dependiendo del sistema, del entorno o las distintas funcionalidades que puedan ofrecer.

Entre los más comunes se encuentran los siguientes.

- **Bases de datos orientadas a documentos.** Las bases de datos documentales se distinguen por su estructura, que se centra en un solo objeto de tipo documentos. Cada motor de búsqueda que sigue este paradigma se diferencia en los detalles, pero comparten características como es el almacenamiento de la información en un formato estándar.

  Estos formatos acostumbrar a ser JSON y BSON, que son, sin duda, los más usados en la actualidad. A continuación, vemos un ejemplo de cómo podría ser un formato Json en una base de datos de este tipo.

```
                                    {
 _id : 1 ,
Nombre : “Lidl” ,
url : “ http : / /www.lidl.es “ ,
tipo : “ Documental “
                                    }
```

  Con ese ejemplo podemos apreciar la sencillez de una estructura de base de datos documental, detallando los campo-valor, donde la clave es el campo y el valor es el dato del campo.

- **Bases de Datos Orientadas Clave/Valor.** Tienen una gran similitud a las documentales, ya que guardan la misma información de clave/valor, pero su gran diferencia está en el almacenamiento en clave. Lo explicaremos de la siguiente manera, tomando el ejemplo arriba planteado

```
                                    mongodb =>
{ _id : 1 ,
                  nombre : "MongoDB" ,
                                                u r l : " h t
t p : / /www. mongodb . o r g " ,
                   t i p o : " Documental "
                                    }
```

  Eso significa, que la clave es «MongoDB» y el contenido es el mismo que el del documental.

- **Base de datos orientadas a grafos.** Este tipo de base de datos tiene una manera muy peculiar de organizar la información, ya que se basa en la teoría de grafos, donde los nodos individuales equivaldrían a la estructura de columna y, por tanto, se trataría de bases de datos normalizas.

  Esto quiere decir que las relaciones entre la información solamente son binarias. En este tipo de base de datos cualquier cambio o anomalía en un nodo, solo afectará de manera local.

- **Bases de datos multivalor.** Sin duda, es una de las arquitecturas más complejas. Primero, por su sistema muy particular de gestión de la información, creando características multidimensionales y por supuesto NoSQL. Muchos profesionales las suelen comparar con el sistema relacional clásico, pero no es así, ya que el multivalor es mucho menos rígido.

- **Bases de datos orientadas a objetos.** Como el nombre muy bien lo explica, sus valores están dados por objetos. Al igual que las bases de datos relacionales orientadas a objetos, este tipo da gran facilidad y a aporta interesantes características cuando se desarrolla en lenguajes de programación como Java.

MODELO ORIENTADO A OBJETOS	MODELO RELACIONAL
Clase.	Relación.
Objeto.	Registro.
Variable.	Atributo.
Método.	Procedimiento almacenado.

Diferencias entre el modelo orientado a objetos y el modelo relacional
Fuente: Elaboración propia

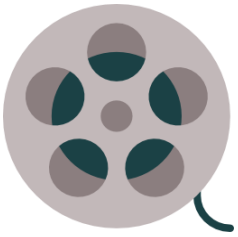
- **Bases de datos tabulares.** Como lo explica su nombre, es la formación de una base a modo de tabla, incorporando elementos entre columnas y filas. Dichas columnas y filas generan intersecciones, a las cuales se les asigna una numeración para establecer un orden a la hora de gestión de datos. Están diseñadas para el almacenamiento de grandes cantidades de datos, que pueden ser o distribuidos.

- **Bases de datos arrays.** Este tipo de base trabaja estructuras donde se sitúan los datos en una cuadricula. Se usan mayormente este tipo de base de datos para la representación de simulaciones, estadísticas y trabajo con sensores. Pueden manejar grandes volúmenes de datos de una manera flexible y escalable.

ARTÍCULO DE INTERÉS

En el siguiente artículo se explican las características y utilidad de las bases de datos orientadas a grafos.

https://www.sngular.com/es/que-son-bases-datos-a-grafos/



VIDEO DE INTERÉS

En el siguiente vídeo podrás ampliar tus conocimientos sobre los tipos de bases de datos NoSQL.

https://youtu.be/bZPZIwdIu8U

5. EJEMPLOS DE BASES DE DATOS NOSQL

Eres consciente de que MongoDB ha sido utilizado en multitud de implantaciones en todo el mundo para la puesta en marcha de sistemas de gestión documental, lo que hace que la decisión de utilizar esta plataforma para crear la biblioteca universal de la editorial se vea reafirmada.

También, sabes que hay que analizar si el dinamismo de la información que va a introducirse en el repositorio no va a ser muy grande. Es decir, por muchas leyes y sentencias que se generen cada día en el mundo nunca va a aproximarse la velocidad de la creación de contenidos a lo que ocurre en las redes sociales.

Por tanto, MongoDB sigue siendo una buena alternativa. Sin embargo, si el proyecto debiese manejar información en tiempo real, como la que se genera en una red social, un repositorio como Cassandra podría resultar más eficiente. En cualquier caso, piensas que es recomendable realizar prototipos y pruebas de carga reales para comparar el rendimiento y no dejarse guiar por decisiones con un fundamento exclusivamente teórico.

Las bases de datos pueden ser muy variadas y clasificarse por diferentes criterios, como los datos documentales, los orientados a grafos, multivalor… A continuación, veremos los distintos tipos de bases de datos y su clasificación.

Entre las **bases de datos documentales** podemos destacar las siguientes:

- MongoDB
- CouchDB
- RavenDB
- BaseX
- Terrastore

En las **bases de datos clave-valor** están las siguientes:

- Apache Cassandra
- Riak
- Redis
- Dynamo
- BigTable

En cuanto a las **bases de datos orientadas a grafos** podemos reseñar las siguientes:

- Neo4j
- Dex
- Sones GraphDB
- Hyper GraphDB
- InfoGrid

En bases de **datos multivalor** están las siguientes:

- Rocket D3 DBMS
- Open QM
- Reality
- Jbase
- OpenInsight

Entre las **bases de datos NoSql orientadas a objetos** están las siguientes, algunas de las cuales, a pesar de seguir implantadas en diversos proyectos, han dejado de ser evolucionadas:

- ObjectDB
- ZooDB
- Realm.io
- Db4o
- GemStone S

Como **bases de datos tabulares** podemos presentar estas:

- Hbase, de Apache
- BigTable, de Google
- LevelDB
- Hypertable

Finalmente, como ejemplo de base de datos de arrays mencionaremos SciDB.

5.1 ¿Cuál es la base de datos NoSql más popular?

MongoDB es desarrollado por MongoDB, Inc. desde el año 2007, bajo la licencia AGPL y es de uso libre y gratuito.

Con una arquitectura diseñada para cumplir las demandas de las nuevas aplicaciones, tales como encontrar una mejor manera de manejar los datos, mejor gestión de los sistemas de almacenamiento distribuido y la libertad de ejecutar en cualquier lugar que se desee debido a su portabilidad.

MongoDB viene a romper la estructura de las clásicas bases de datos relacionales, que durante tantos años habían dominado la manera en cómo se almacena la información de nuestras aplicaciones. Ahora, veamos más en profundidad de que trata MongoDB, pero primero debemos saber que son las bases de datos no relacionales

En los próximos apartados veremos más a fondo todas estas características y cómo nos ayudarán a desplegar distintos tipos de sistemas de información.

MongoDB parte de las siguientes premisas, así que podemos destacar como las principales virtudes de esta base de datos no SQL:

- Ofrece la mejor manera de trabajar con los datos:
  - **Fácil**: Los datos se gestionan de forma natural e intuitiva.
  - **Rápida**: Tiene un alto rendimiento por defecto, por lo cual no hay necesidad de hacer configuraciones complejas.
  - **Flexible**: Se puede modificar la estructura rápidamente.
  - **Versátil**: Soporta muchísimos tipos de datos y de consultas.

- Se pueden ubicar los datos donde se desee:
  - **Disponibilidad**: Permite desarrollar aplicaciones resilentes con altas capacidades de replicación y mantenimiento.
  - **Escalabilidad**: Crece horizontalmente en base a sus características nativas
  - **Aislamiento de cargas**: Genera trabajos de análisis y cargas de trabajo en el mismo cluster.
  - **Localización**: almacena datos a nivel local, en dispositivos específicos.

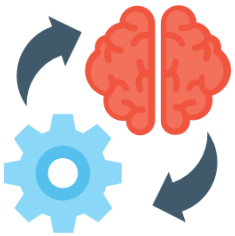
- Libertad para ejecutarse en cualquier ámbito:

- **Portabilidad :** Es una base de datos que corre igual en cualquier lugar.
- **Preparada para la nube**: Disfruta de los beneficios del almacenamiento en distintas nubes.
- **Soporte global**: El servicio de soporte técnico está disponible en más de cincuenta regiones y en seis de los proveedores mayoritarios de servicios de nube.

ENLACE DE INTERÉS

Te recomendamos acceder a la página web oficial del proyecto de MongoDB, a través de la cual puedes realizar la descarga de forma gratuita.

https://www.mongodb.com/



RECUERDA

MongoDB es una base de datos orientada a documentos, esto quiere decir que la información se almacena en documentos similares a JSON con una jerarquía dentro de los mismos. Algunas de las principales características de MongoDB son:

- Gran flexibilidad y escalabilidad.
- Índices secundarios.
- Consultas por rangos.
- Ordenación de datos.
- Agregación.
- Indexación geoespacial.

Por tanto, podemos decir que MongoDB es una herramienta poderosa y fácil de aprender a usar. Veamos a continuación varios conceptos básicos que serán necesarios para entender cómo funciona:

Un **documento (document)** es la unidad básica de datos de MongoDB y se podría decir que es el equivalente a un registro en las bases de datos relacionales (pero ofreciendo muchísima más flexibilidad).

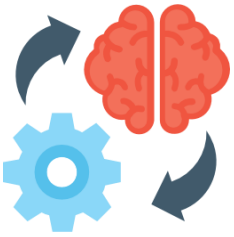
De la misma manera tenemos una **colección (collection).** Esta pudiera ser vista como una tabla en su equivalente relacional, pero muchísimo más dinámica.

MongoDB es distribuida junto con una herramienta muy sencilla, pero a su vez bastante potente, llamada **mongo Shell**. Esta nos permite administrar y manipular todas nuestras bases de datos, ya sean sus collections, documents o hacer algún tipo de query usando el MongoDB query Language. Además, posee un intérprete de JavaScript, que nos permitirá programar una gran variedad de scripts.

PARA SABER MÁS

En la página oficial de MongoDB tenemos acceso al manual de Mongo Shell, en el que encontraremos ejemplos de cómo trabajar con esta herramienta.

https://docs.mongodb.com/manual/mongo/



RECUERDA

Una sola instancia de MongoDB puede contener múltiples e independientes bases de datos y a su vez, cada una de estas, tendrá sus propias colecciones.

Cada documento tendrá una llave especial "_id" que será única para su colección.

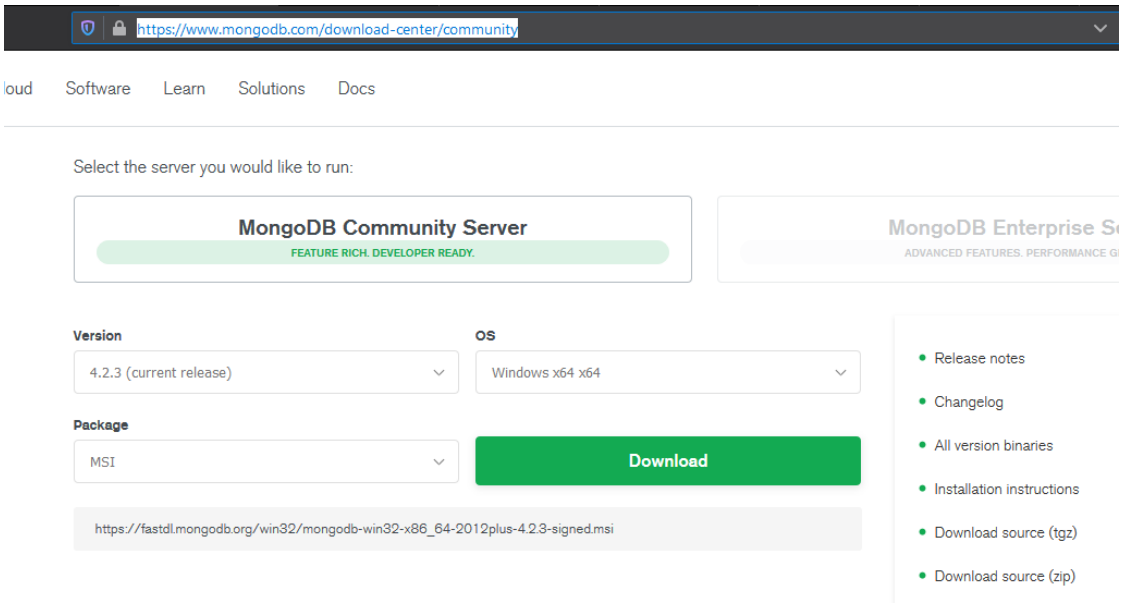
5.2 Instalación de MongoDB

Lo primero que debemos hacer es dirigirnos a la web oficial de MongoDB y, concretamente, nos situaremos en el "Download Center".

Descargamos el instalador para nuestro sistema operativo. En esta guía que desarrollamos a continuación, damos por hecho que usaremos Windows 10 de 64 bits. Es importante saber que MongoDB para esta plataforma solo soporta la arquitectura de 64bits. Descargaremos el asistente de instalación para la Community Edition.

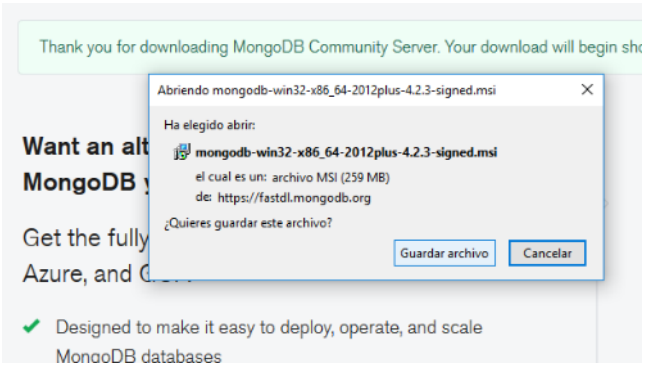
Elegimos MongoDB Comunity Server y descargamos la versión actual (current release) 4.2.3 o la más reciente que se encuentre disponible en el momento de la descarga. Elegimos el paquete MSI en este caso referido a Windows. Si se trata de algún otro sistema operativo podemos seleccionarlo entre las opciones disponibles en la lista desplegable "OS". Comprobaremos

que MongoDB tiene soporte para la gran mayoría de sistemas populares tales como Linux, Mac OS y Windows.



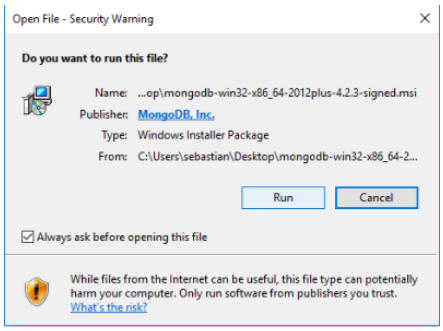
Download Center de MongoDB
Fuente: https://www.mongodb.com/

1. El asistente para Windows pesa unos 259 MB.



Descarga de MongoDB
Fuente: Elaboración propia

2. Después de que lo hayamos descargado, ejecutamos el asistente dando click en el botón "Run".



Instalación de MongoDB
Fuente: Elaboración propia

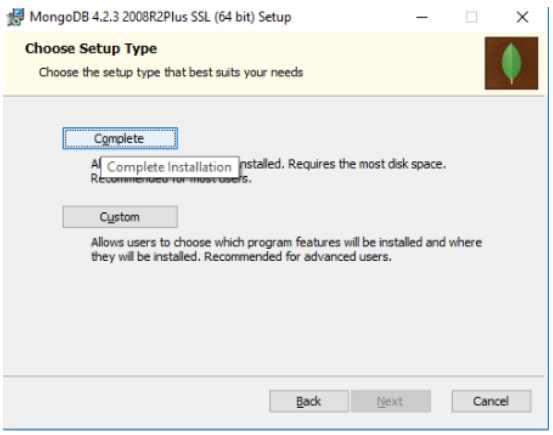
3. Esto nos abrirá el instalador que se verá de la siguiente manera:

Instalador de MongoDB
Fuente: Elaboración propia

4. Este asistente nos guiará durante toda la instalación de manera cómoda y práctica. El siguiente paso es aceptar la licencia publica de MongoDB:

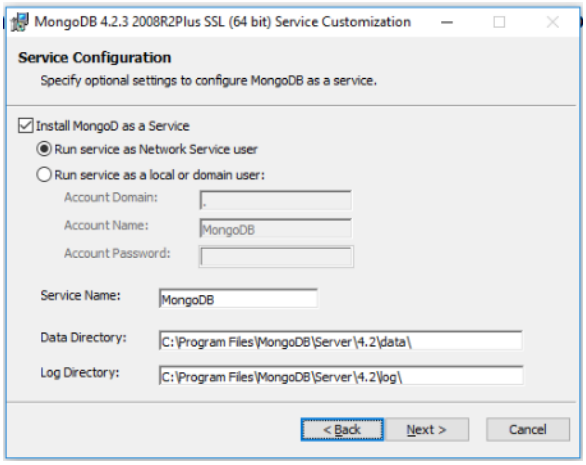
Términos de licencia de MongoDB
Fuente: Elaboración propia

5. Tras aceptar los términos de la licencia, le diremos al asistente que ejecute una instalación completa dando clic en el botón "complete":



Fuente: Elaboración propia

6. Después debemos seleccionar instalar como un servicio. Aquí tendremos la opción de instalarlo como un servicio de la red o como un servicio local. En nuestro caso dejamos la opción por defecto, que es como un servicio de red y le damos a siguiente (next):



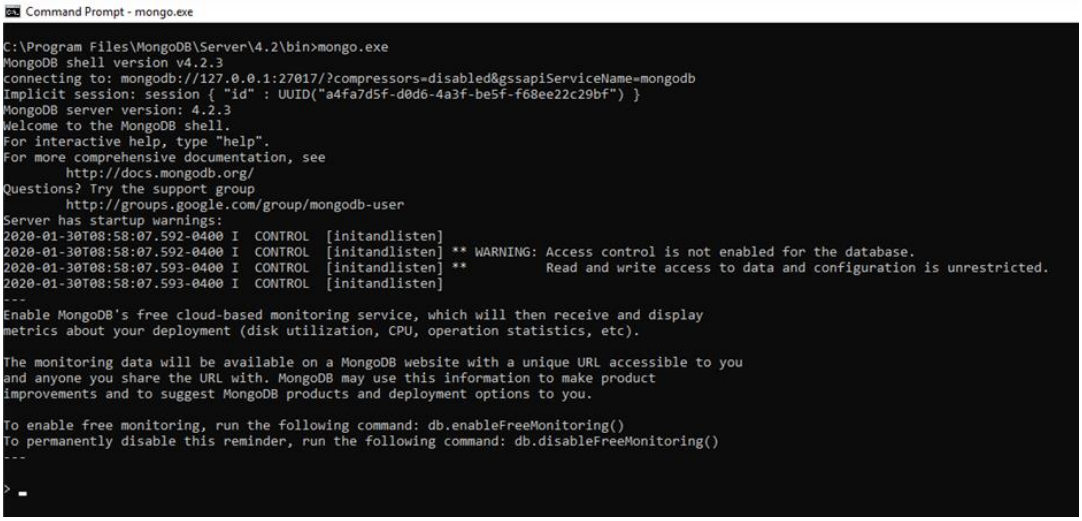
Opciones de instalación de MongoDB
Fuente: Elaboración propia

7. Después de esto debemos decidir si usaremos la interface gráfica de Mongo, que se llama MongoDB Compass. Le indicamos que si, a continuación, siguiente y le damos instalar.

8. Al finalizar el asistente, tendremos instado nuestro MongoDB. Ahora abriremos una consola de comandos, para comprobar si todo está correctamente instalado.

   A continuación, escribiremos lo siguiente:
   "C:\Program Files\MongoDB\Server\4.2\bin\mongo.exe"

   Y le damos a "enter", lo cual nos abrirá el Mongo Shell.



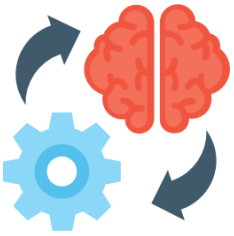
Fuente: Elaboración propia

9. Por tanto, ya hemos completado el primer y fundamental paso para iniciarnos con MongoDB, que es haberlo instalado correctamente. En el próximo apartado veremos más sobre Mongo Shell, sus comandos básicos y como usar esta herramienta.

ENLACE DE INTERÉS

En el siguiente enlace puedes descargar MongoDB de su página oficial:

https://www.mongodb.com/download-center/community



RECUERDA

La ruta de Mongo dependerá de lo que hayamos elegido en el asistente de instalación. Esta que indicamos aquí, es la ruta por defecto.

5.3 MongoDB Shell

Habéis comenzado el proyecto de la biblioteca con un proceso de formación y entrenamiento del equipo. Tras haber instalado un servidor de prueba de MongoDb, estáis comenzando a familiarizaros con el uso básico de esta base de datos.

Os resulta chocante no disponer de algo tan trivial como un comando para crear una base de datos. Así que comenzáis creando una colección de prueba y también insertáis algunos documentos: sentencias, leyes y algún otro documento de los que están disponibles en el fondo digital de la editorial
.
En paralelo a estas pruebas para tomar el pulso de MongoMD te ha parecido interesante comenzar a estimar el coste de adquisición del hardware. Dado que no hay datos de referencia de un proyecto similar, por ser la primera vez que trabajáis en entornos NoSql has encargado a uno de los miembros del equipo que escriba un pequeño script. La idea que tienes es que ese script automatice el proceso de inserción de documentos que sería inviable hacerlo manualmente de uno en uno.

Tu objetivo es llenar de documentos y colecciones el pequeño servidor de pruebas que habéis desplegado hasta que se sature. De esa forma, podrás disponer de una sencilla regla de tres para poder anticipar la volumetría que necesitaréis de servidor cuando el proyecto alcance la etapa de producción.

La misma estimación que debes hacer de tus necesidades de recursos de computación también la harás de espacio de almacenamiento, sobre todo ahora que ya conoces los aspectos básicos de los tipos de datos.

MongoDB viene con una **consola JavaScript** que nos permitirá la interacción con nuestras bases de datos, o instancias de MongoDB, desde la terminal de comandos o consola de nuestro sistema operativo.

Esta consola (Mongo Shell) nos será de mucha utilidad en el momento de tener que realizar labores administrativas sobre nuestras bases de datos, además de para inspeccionar y explorar MongoDB.

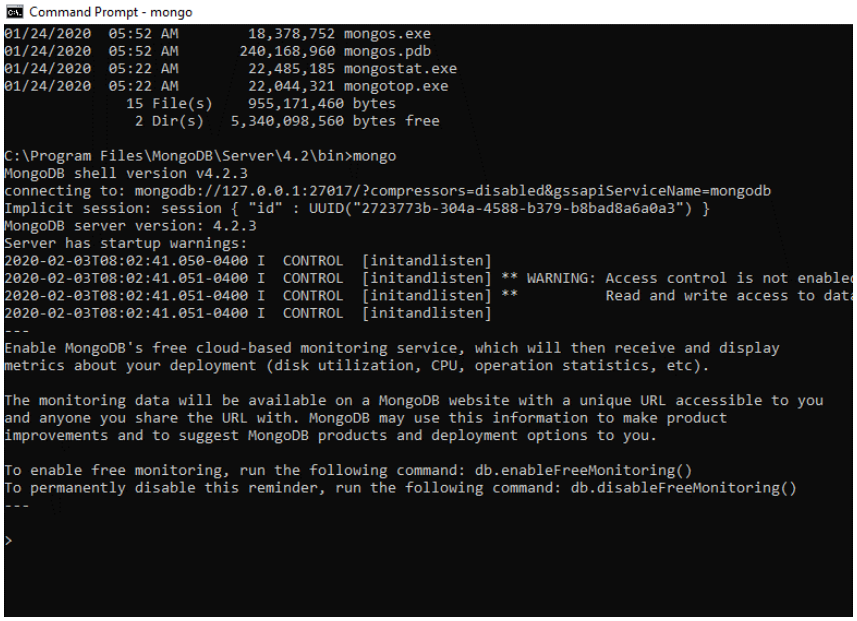
Mongo Shell es una herramienta crucial para el uso de MongoDB y a lo largo de este apartado nos extenderemos en su uso.

5.3.1 Introducción a Mongo Shell

Como vimos en el apartado anterior, **Mongo Shell** se instala por defecto cuando hacemos la instalación de MongoDB. La manera de ejecutar esta Shell es desde la consola de nuestro sistema operativo, que en el caso que estamos desarrollando es Windows:

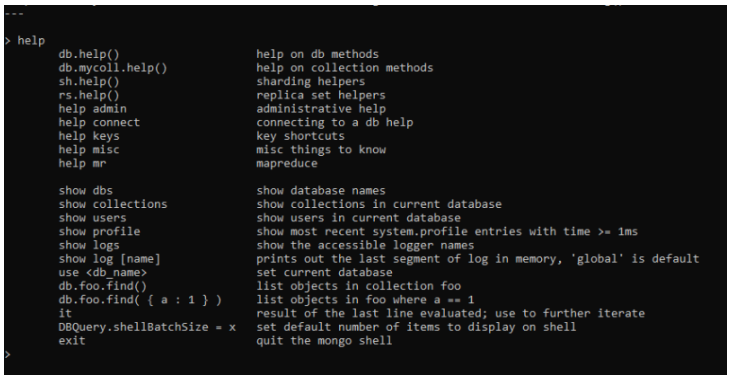
1. Ejecutamos el comando **cmd** y abrimos el símbolo de sistema o command prompt.
2. A continuación, debemos navegar hacia la carpeta de instalación de MongoDB, usando para ello el siguiente comando:

   ```
   cd C:\Program Files\MongoDB\Server\4.2\bin>
   ```
3. Entonces ejecutamos el comando "`mongo.exe`".
4. Se nos mostrará un mensaje de bienvenida que nos aporta información y algunas advertencias.



Mensaje instalación Mongo Shell
Fuente: Elaboración propia

5. Para obtener una lista de comandos básicos podemos ejecutar el comando `help`



Lista de comandos básicos

Fuente: Elaboración propia

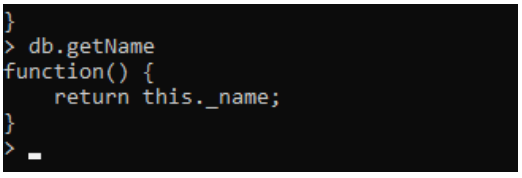
6. Para salir de mongo Shell, solamente debemos escribir el comando exit y presionar la tecla enter.

5.3.2 Mongo Shell help

Como hemos visto, es sumamente sencillo acceder a nuestra consola de MongoDB. Ahora veremos cómo usar la ayuda que ella misma nos proporciona.

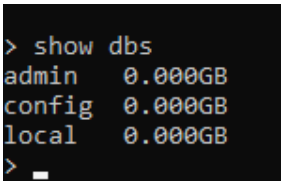
1. **Ayuda para las bases de datos.** Para obtener la lista de comandos básicos aplicables sobre las bases de datos, debemos hacer uso de la función "`db.help()`".

2. Aquí nos muestra funciones útiles para saber el nombre de la base de datos. Por ejemplo, `db.getName()` la cual nos devuelve el nombre de la base de datos sobre la que estamos trabajando. Si queremos ver la implementación del método, podemos ejecutar la función sin los paréntesis y obtendremos dicha implementación del método a través de la consola:



Fuente: Elaboración propia

3. Otro comando útil es **show dbs,** que nos muestra un listado de las bases de datos disponibles.



Fuente: Elaboración propia

4. **Ayuda para las colecciones (collections).** Para ver un listado de las collections existentes en la base de datos actualmente seleccionada, podemos usar el comando `show collections`.

5. Actualmente no tenemos ninguna collection creada en nuestra base de datos. Por defecto, MongoDB se conecta a la base de datos Test, que es una base de datos que se crea durante la instalación de nuestro servidor.

   En Mongo, para crear una base de datos, lo hacemos utilizando el comando **use** con una collection que todavía no existe. Por ejemplo:

   ```
   use Tienda
   ```

   ```
   > use Tienda
   switched to db Tienda
   ```

   Fuente: Elaboración propia

6. Ahora, dentro de esta base de datos, agregaremos una función haciendo uso del comando:

   ```
   db.Productos.insert({ nombre: "Naranjas", precio: 2})
   ```

   ```
   > db.Productos.insert({ nombre: "Naranjas", precio: 2})
   WriteResult({ "nInserted" : 1 })
   ```

   Fuente: Elaboración propia

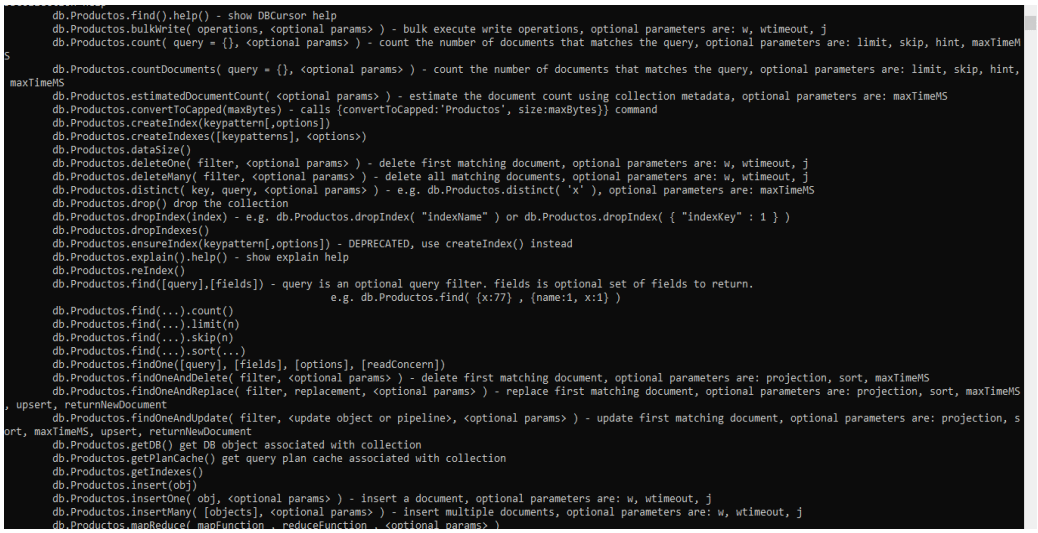
7. Posteriormente explicaremos con más detalle estos comandos. Por el momento los usaremos para ilustrar simplemente como podemos usar la ayuda.

   Después de la acción inicial, ahora sí que podemos usar el comando **`show collections`** y obtendremos el siguiente resultado:

   ```
   > show collections
   Productos
   >
   ```

   Fuente: Elaboración propia

8. Para saber qué otras cosas podemos hacer con las collections, desde nuestra consola usamos el comando **`db.Productos.help().`** Hacemos hincapié en que la collection no es necesario que exista, para que nos muestre la ayuda. Acto seguido vemos la siguiente salida a través de la consola:

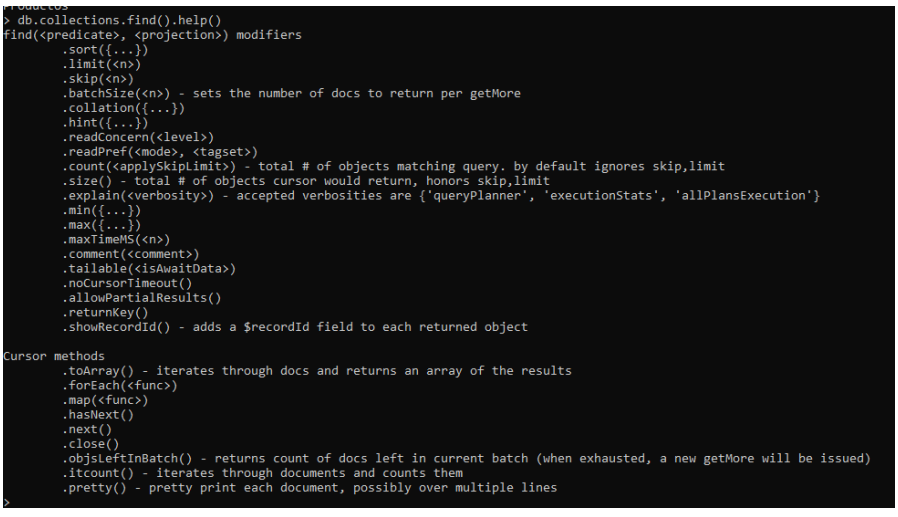


Fuente: Elaboración propia

Podemos ver que están los métodos como el que ya se usó anteriormente para insertar, **Insert**, así como otros muchos más.

9. **Ayuda para el manejo del cursor.** Cuando utilizamos métodos como el **`find()`**, podemos hacer uso del cursor para modificar su comportamiento, haciendo uso también de métodos en JavaScript que nos provee la Mongo Shell.

   Para ver estos métodos, lo podemos conseguir introduciendo el comando: `db.collection.find().help()`, con lo que obtenemos la siguiente salida:



Fuente: Elaboración propia

10. Como vemos, es un listado de los comandos que tenemos disponibles para el cursor. Entre los más útiles destacamos `hasNext()`, `next()` y `forEach()`.

¿SABÍAS QUE…?

Con la tecla de tabulador podemos autocompletar comandos en la consola.

5.4 Escribir scripts en MongoDB

En MongoDB podemos **escribir scripts**, ya sea a través de la consola o Mongo Shell, o desde alguna otra fuente externa. Esto nos servirá para realizar tareas administrativas o para organizar otro tipo de tareas.

Aquí veremos un ejemplo simple en JavaScript, que nos puede servir para ser utilizado desde la Mongo Shell, o desde un servidor web.

Supongamos que tenemos unos datos que queremos ingresar en nuestra base de datos, por ejemplo, la información de unos productos. Para conseguir esto sería muy engorroso hacer los inserts uno a uno. Por lo tanto, para esto podemos crear un script que nos ayudará con esta tarea.

1. Lo primero es crear un archivo vacío con nuestro editor de código preferido. En este caso, a modo de ejemplo, usaremos **Visual Studio Code,** el cual se puede descargar de manera gratuita su web.

2. Creamos el archivo y le colocamos la extensión .js

3. Ahora, escribimos el contenido. En nuestro caso insertaremos una serie de productos:

```
db.Productos.drop();
for (i = 1; i <= 10; i++) {
    db.Productos.insertOne({
        nombre: "nombre" + i,
        precio: 2 * i,
    });
}
```

4. Como se observa, podemos hacer uso de la potencia de JavaScript y este código pudiéramos pegarlo directamente en la consola de Mongo (Mongo Shell) o cargar el archivo usando el comando **load ("ruta al archivo").**

5. Al ejecutar este script veremos la siguiente salida:

```
> db.Productos.drop()
true
> for(i = 1; i <= 10; i++) {
...    db.Productos.insertOne(
...      {
...        nombre: 'nombre'+i,
...        precio: 2*i
...      }
...    )
... };
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5e40291c23ca975c1765a31f")
```

Fuente: Elaboración propia

6. A continuación, para revisar que todo se ejecutó correctamente, simplemente utilizamos el comando **`db.Productos.find().pretty()`** y obtendremos un listado de la collection Productos:

```
> db.Productos.find().pretty()
{
        "_id" : ObjectId("5e40291c23ca975c1765a316"),
        "nombre" : "nombre1",
        "precio" : 2
}
{
        "_id" : ObjectId("5e40291c23ca975c1765a317"),
        "nombre" : "nombre2",
        "precio" : 4
}
{
        "_id" : ObjectId("5e40291c23ca975c1765a318"),
        "nombre" : "nombre3",
        "precio" : 6
}
```

Fuente: Elaboración propia

ENLACE DE INTERÉS

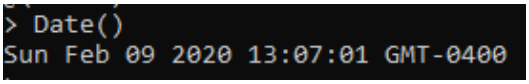
Puedes descargar gratuitamente Visual Studio Code desde la siguiente dirección web:

https://code.visualstudio.com/

5.5 Tipos de datos

MongoDB BSON nos ofrece un mayor tipo de datos que JSON. Concretamente, tenemos soporte nativo para los siguientes tipos:

1. **Date.** Mongo Shell nos ofrece varios métodos para devolver las fechas, ya sea como un objeto de tipo Date o un String. `Date()` nos devuelve la fecha actual como un String.



Fuente: Elaboración propia

`new Date()` devuelve un objeto del tipo Date().

Fuente: Elaboración propia

Internamente, los objetos de tipo Date() son un numero entero de 64 bits representando la cantidad de milisegundos desde el 1 de enero del año 1970. El hecho de que sea de 64 bits nos permite trabajar de manera segura con años que van desde el 0 hasta el 9999.

2. **ObjectId.** Este tipo de datos se utiliza como identificador único del objeto. Por lo tanto, se podría decir que es como una clave única. Dentro de los beneficios aportados por este tipo de datos, se destacan que son de rápida generación y sirven para ordenar.

   Este tipo de dato posee una longitud de 12 bits, los cuales están compuestos por:
   - Una representación de 4 bytes de la fecha de creación, medida en segundos desde la época Unix.
   - 5 bytes aleatorios.
   - 3 bytes en un contador único, que inicia desde el valor aleatorio anterior.

   En MongoDB, cada objeto insertado en una collection debe poseer un ObjectID. Esto nos permite saber la fecha en que el objeto fue creado y también nos permitirá organizar nuestros objetos.

3. **NumberLong.** En MongoDB se tratan todos los números como punto flotantes y para tipos enteros muy grandes existe NumberLong(), que posee 64 bits.

El constructor acepta el entero como un string y si se utiliza alguna función como inc para aumentar su valor, este pasa a ser un punto flotante.

4. **NumberInt.** Se usa para enteros específicamente de 32 bits.

5. **NumberDecimal.** Por defecto, MongoDB trata todos los números como punto flotante de 64 bits, pero NumberDecimal permite crear números punto flotante de 128 bits, capaces de generar una exactitud decimal bastante elevada, ideal para datos de tipo financiero, estadísticas, impuestos y otros de índole científico.

   El constructor de NumberDecimal acepta una cadena string y valores numéricos. A pesar de que se aceptan valores decimales, no es recomendado, porque pueden existir perdidas de precisión. Se almacena en la base de datos en el siguiente formato:
   `NumberDecimal("1000.55").`

   Aquí podemos ver un ejemplo de cómo se almacenan los distintos tipos de datos en MongoDB:

```
{ "_id" : 1, "num" : NumberDecimal( "9.99" ) }
{ "_id" : 2, "num" : NumberLong(10) }
{ "_id" : 3, "num" : NumberDecimal( "10.0" ) }
```

   Para verificar el tipo de datos en MongoDB, este nos provee de dos operadores, instanceof y typeof. Cuando necesitemos saber el tipo de un dato, simplemente haremos la llamada a uno de estos métodos:
   - `instanceof` compara el tipo con el nombre del tipo.
   - `typeof` devuelve el tipo.

¿SABÍAS QUE...?

BSON es un tipo de datos para la serialización, utilizado por MongoDB para almacenar los documents.

Binary + JSON, le dan origen a su nombre

5.6 Operaciones CRUD en MongoDB

El sistema de información que estáis diseñando e implementando estará muy orientado a las operaciones de lectura, que serán las que habrá que optimizar al máximo. Un proyecto tan ambicioso y global como es este, requerirá de un volumen de usuarios muy elevado, por lo que habrá mucha concurrencia.

Las operaciones de actualización o de borrado serán mínimas porque las sentencias nunca se eliminan por el valor histórico que tienen. A lo sumo, se actualizarán indicando que hay otra sentencia posterior que las matiza o las modifica o las deja sin efecto.

Con la legislación ocurre algo parecido. A diferencia de otros documentos que tienden a ser más dinámicos y estar constantemente cambiando, las leyes reciben cambios de forma poco frecuente. En ocasiones pasan meses o años sin que se mueva ni una coma.

Las operaciones más usadas en una base de datos siempre serán insertar, eliminar, modificar y leer algún dato. Estas operaciones son más conocidas por sus siglas en ingles CRUD (Create, Read, Update and Delete). En este modelo se basan casi todas las operaciones de los actuales sistemas y aplicaciones.

En este apartado veremos cómo realizar cada una de estas operaciones sobre nuestra base de datos. Para ello utilizaremos un ejemplo práctico para ir viendo cómo se pueden realizar dichas operaciones.

También aclararemos conceptos nuevos y repasaremos el uso de Mongo Shell y los conocimientos aprendidos en los apartados anteriores.

5.6.1 Insertar

Crear o insertar un nuevo document en la collection de nuestra base de datos es muy simple en la Mongo Shell con el **método insertOne**. Supongamos que queremos almacenar determinada información referida a unas películas. Lo primero que debemos hacer es crear una variable, la cual llamaremos película y esta será una representación de lo que queremos almacenar. Le sumaremos los atributos que deseemos, tales como nombre, fecha y director.

1. Antes de nada, debemos crear una base de datos, porque si no, se almacenará todo en la base de datos que esté actualmente en uso.

Así que escribamos en la consola el siguiente comando: **use tutorial** + tecla enter.

```
> use tutorial
switched to db tutorial
> _
```

Fuente: Elaboración propia

2. Ya hemos creado una nueva DB (Data Base).

3. Abrimos nuestra consola y escribimos:

pelicula = { titulo: "Joker", director:"Todd Phillips", año:"2019"}

Pulsamos enter a continuación y con ello hemos asignado el objeto a la variable.

```
> pelicula = { titulo: "Joker", director:"Todd Phillips", año:"2019"}
{ "titulo" : "Joker", "director" : "Todd Phillips", "año" : "2019" }
```

Fuente: Elaboración propia

4. Podemos ver el contenido de la variable colocando: **pelicula** y pulsamos enter, esto nos mostrará el contenido de la variable:

```
> pelicula
{ "titulo" : "Joker", "director" : "Todd Phillips", "año" : "2019" }
```

Fuente: Elaboración propia

5. A continuación, podemos usar el método **insertOne()** de la siguiente manera:

db.peliculas.insertOne(pelicula) y pulsamos enter para que lo ejecute.

```
> db.peliculas.insertOne(pelicula)
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5e4302d766b042c9e00f6bc8")
}
>
```

Fuente: Elaboración propia

Esto ha sido suficiente para insertar un objeto en nuestra collection películas.

6. Ahora observamos que nos devuelve un parámetro que es **true.** Para mostrar que la operación se hizo correctamente, también nos

muestra un **insertedId ,** que si queremos verificar como se guardó nuestro objeto, podemos escribir lo siguiente en la consola:

db.peliculas.find().pretty() + enter

Esto nos devuelve lo siguiente:

```
> db.peliculas.find().pretty()
{
        "_id" : ObjectId("5e4302d766b042c9e00f6bc8"),
        "titulo" : "Joker",
        "director" : "Todd Phillips",
        "año" : "2019"
}
```

Fuente: Elaboración propia

7. Como podemos observar, se ha agregado una propiedad "_id", la cual posee el mismo valor que nos devolvió la consola en el momento de usar el método insertOne().

¿SABÍAS QUE…?

Cuando pasamos un nombre desconocido a Mongo, este crea esa collection o base de datos automáticamente.

5.6.2 Query

Ya hemos creado un documento. Ahora veamos como leerlo (read), para lo cual podremos usar **find** o **findOne** escribiendo en la consola:

db.peliculas.find() +tecla enter.

```
> db.peliculas.find()
{ "_id" : ObjectId("5e4302d766b042c9e00f6bc8"), "titulo" : "Joker", "director" : "Todd Phillips", "año" : "2019" }
```

Fuente: Elaboración propia

Ahora, veamos cómo hacer consultas más complejas. Para ello, creemos antes más datos para ilustrar mejor nuestro ejemplo. Para conseguir esto fácilmente, haremos uso de otro método para insertar datos que es **insertMany()**:

```
db.peliculas.insertMany([
    { titulo: "Batman Inicia", director: "Christopher Nolan", año:
"2005" },
    { titulo: "El Caballero Oscuro", director: "Christopher Nolan",
año: "2008" },
    { titulo: "El Origen", director: "Christopher Nolan", año: "2010"
},
    { titulo: "The Hangover", director: "Todd Phillips", año: "2009"
},
    { titulo: "The Hangover I", director: "Todd Phillips", año:
"2011" },
    { titulo: "The Hangover II", director: "Todd Phillips", año:
"2013" },
    { titulo: "Isla de Perros", director: "Wes Anderson", año: "2018"
},
    { titulo: "Fantastic Mr.Fox", director: "Wes Anderson", año:
"2009" },
    { titulo: "Moonrise Kingdom", director: "Wes Anderson", año:
"2012" },
    { titulo: "Titanic", director: "James Cameron", año: "1997" },
    { titulo: "Avatar", director: "James Cameron", año: "2009" },
    { titulo: "The Terminator", director: "James Cameron", año:
"1984" },
]);
```

1. Como podemos observar, **insertMany()** acepta como parámetro un array de elementos a insertar separados por una coma ",". Por lo tanto, podemos copiar y pegar esto en nuestra consola para ahorrar tiempo.

```
> db.peliculas.insertMany([{ titulo: "Batman Inicia", director:"Christopher Nolan", año:"2005"},
...  { titulo: "El Caballero Oscuro", director:"Christopher Nolan", año:"2008"},
...  { titulo: "El Origen", director:"Christopher Nolan", año:"2010"},
...  { titulo: "The Hangover", director:"Todd Phillips", año:"2009"},
...  { titulo: "The Hangover I", director:"Todd Phillips", año:"2011"},
...  { titulo: "The Hangover II", director:"Todd Phillips", año:"2013"},
...  { titulo: "Isla de Perros", director:"Wes Anderson", año:"2018"},
...  { titulo: "Fantastic Mr.Fox", director:"Wes Anderson", año:"2009"},
...  { titulo: "Moonrise Kingdom", director:"Wes Anderson", año:"2012"},
...  { titulo: "Titanic", director:"James Cameron", año:"1997"},
...  { titulo: "Avatar", director:"James Cameron", año:"2009"},
...  { titulo: "The Terminator", director:"James Cameron", año:"1984"}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5e43fca74faa63fd708b1220"),
                ObjectId("5e43fca74faa63fd708b1221"),
                ObjectId("5e43fca74faa63fd708b1222"),
                ObjectId("5e43fca74faa63fd708b1223"),
                ObjectId("5e43fca74faa63fd708b1224"),
                ObjectId("5e43fca74faa63fd708b1225"),
                ObjectId("5e43fca74faa63fd708b1226"),
                ObjectId("5e43fca74faa63fd708b1227"),
                ObjectId("5e43fca74faa63fd708b1228"),
                ObjectId("5e43fca74faa63fd708b1229"),
                ObjectId("5e43fca74faa63fd708b122a"),
                ObjectId("5e43fca74faa63fd708b122b")
        ]
}
```

Fuente: Elaboración propia

Después de ejecutarlo, veremos que nos devuelve un array con los ObjectId creados.

2. Entendamos mejor cómo funciona el **find():** La función **find()** recibe como parámetro un **document**, al usarla sin parámetro, es equivalente a que le pasáramos un parámetro vacío, es decir "{}", lo cual provoca que coincida con cualquier objeto dentro de la **collection.**
3. Ahora, supongamos que queremos saber en nuestros datos de ejemplo sobre películas ¿cuáles de las películas que están en esa collection fueron dirigidas por "Christopher Nolan"?

   Para conseguir esto deberíamos pasar un parámetro a la función find(), como vemos a continuación:

   db.peliculas.find({director:"Chistopher Nolan"})

   Esto nos traería como resultado lo observado a continuación:

```
> db.peliculas.find({director:"Christopher Nolan"})
{ "_id" : ObjectId("5e43fca74faa63fd708b1220"), "titulo" : "Batman Inicia", "director" : "Christopher Nolan", "año" : "2005" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1221"), "titulo" : "El Caballero Oscuro", "director" : "Christopher Nolan", "año" : "2008" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1222"), "titulo" : "El Origen", "director" : "Christopher Nolan", "año" : "2010" }
>
```

Fuente: Elaboración propia

4. También podemos colocar más de un parámetro en el objeto query:
   db.peliculas.find({año:"2009"})
   db.peliculas.find({director:"Wes Anderson",año:"2009"})

```
> db.peliculas.find({año:"2009"})
{ "_id" : ObjectId("5e43fca74faa63fd708b1223"), "titulo" : "The Hangover", "director" : "Todd Phillips", "año" : "2009" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1227"), "titulo" : "Fantastic Mr.Fox", "director" : "Wes Anderson", "año" : "2009" }
{ "_id" : ObjectId("5e43fca74faa63fd708b122a"), "titulo" : "Avatar", "director" : "James Cameron", "año" : "2009" }
> db.peliculas.find({director:"Wes Anderson",año:"2009"})
{ "_id" : ObjectId("5e43fca74faa63fd708b1227"), "titulo" : "Fantastic Mr.Fox", "director" : "Wes Anderson", "año" : "2009" }
```

Fuente: Elaboración propia

5. También existen momentos en donde no necesitaremos recuperar todos los datos. Por ejemplo, si quisiéramos solamente el campo de los directores de las películas, es posible hacer la siguiente query:
   db.peliculas.find({},{director:1})

```
> db.peliculas.find({},{director:1})
{ "_id" : ObjectId("5e4302d766b042c9e00f6bc8"), "director" : "Todd Phillips" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1220"), "director" : "Christopher Nolan" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1221"), "director" : "Christopher Nolan" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1222"), "director" : "Christopher Nolan" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1223"), "director" : "Todd Phillips" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1224"), "director" : "Todd Phillips" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1225"), "director" : "Todd Phillips" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1226"), "director" : "Wes Anderson" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1227"), "director" : "Wes Anderson" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1228"), "director" : "Wes Anderson" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1229"), "director" : "James Cameron" }
{ "_id" : ObjectId("5e43fca74faa63fd708b122a"), "director" : "James Cameron" }
{ "_id" : ObjectId("5e43fca74faa63fd708b122b"), "director" : "James Cameron" }
>
```

Fuente: Elaboración propia

6. Ahora veamos que otro tipo de consultas podemos hacer:

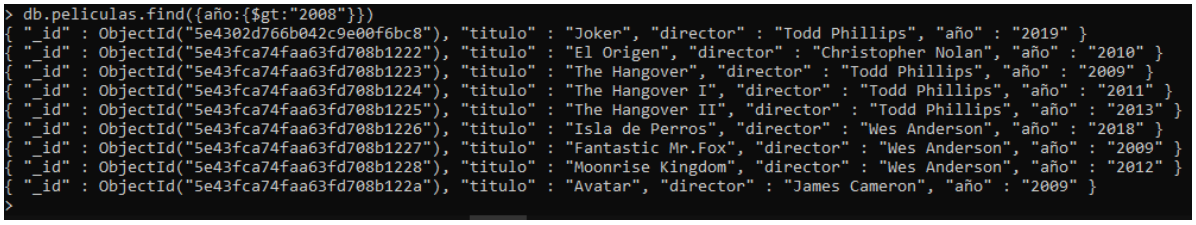
- Podemos hacer uso de condicionales:
  - **$lt:** menor que.

- **$lte:** menor o igual que.
- **$gt:** mayor que.
- **$gte:** mayor o igual que.

7. Veamos cómo usarlos.

   db.peliculas.find({año:{$gt:"2008"}})

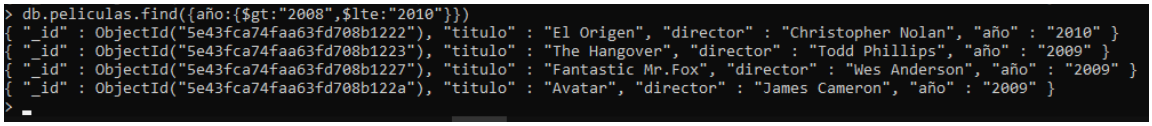
   Debemos especificar un objeto query para el parámetro que deseamos comparar. En este caso usamos **$gt** para buscar todas las películas cuyo año sea mayor que 2008.



Fuente: Elaboración propia

8. Estas queries pueden ser bastante útiles cuando trabajamos con fechas, almacenadas con el tipo de dato Date.

   También podremos hacer combinaciones, por ejemplo:
   db.peliculas.find({año:{$gt:"2008",$lte:"2010"}})

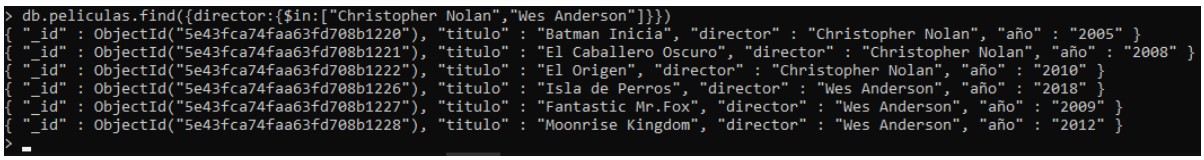


Fuente: Elaboración propia

9. Aquí vemos el resultado de las películas cuyo año es mayor que 2008 y, al mismo tiempo, el año es menor o igual que 2010. Ahora, veamos cómo se usan los operadores **$in** y **$or**.

   $in lo podemos utilizar para comparar un elemento con muchos valores. Por ejemplo:

   db.peliculas.find({director:{$in:["Christopher Nolan","Wes Anderson"]}})



Fuente: Elaboración propia

10. Como podemos observar, el elemento director lo hemos reducido a dos posibilidades "Christopher Nolan" y "Wes Anderson".

¿SABÍAS QUE…?

$in es bastante flexible por lo que podemos comparar incluso con tipos distintos a numéricos y string.

Si usamos en esa misma consulta **$nin** (Este es el resultado opuesto) **db.peliculas.find({director:{$nin:["Christopher Nolan","Wes Anderson"]}})**

```
> db.peliculas.find({director:{$nin:["Christopher Nolan","Wes Anderson"]}})
{ "_id" : ObjectId("5e4302d766b042c9e00f6bc8"), "titulo" : "Joker", "director" : "Todd Phillips", "año" : "2019" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1223"), "titulo" : "The Hangover", "director" : "Todd Phillips", "año" : "2009" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1224"), "titulo" : "The Hangover I", "director" : "Todd Phillips", "año" : "2011" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1225"), "titulo" : "The Hangover II", "director" : "Todd Phillips", "año" : "2013" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1229"), "titulo" : "Titanic", "director" : "James Cameron", "año" : "1997" }
{ "_id" : ObjectId("5e43fca74faa63fd708b122a"), "titulo" : "Avatar", "director" : "James Cameron", "año" : "2009" }
{ "_id" : ObjectId("5e43fca74faa63fd708b122b"), "titulo" : "The Terminator", "director" : "James Cameron", "año" : "1984" }
>
```

Fuente: Elaboración propia

El operador **$or** tiene como principal capacidad poder contener otros operadores condicionales, es decir, dentro de un `$or` podemos comparar distintos valores con distintos campos. Una de sus virtudes es que es muy eficiente. Veamos esto en nuestro ejemplo:

db.peliculas.find({"$or" : [{"director" : { "$in": ["Christopher Nolan","Wes Anderson"] }}, { "año":"2009" }]})

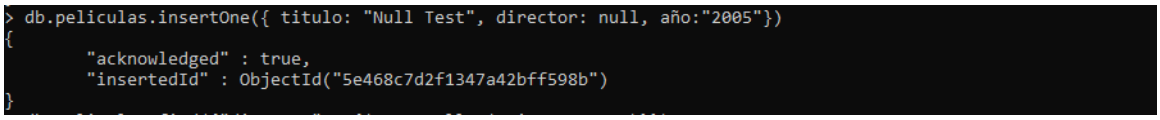
```
> db.peliculas.find({ $or: [ {director : { $in: ["Christopher Nolan","Wes Anderson"] }}, {año:"2009"} ]})
{ "_id" : ObjectId("5e43fca74faa63fd708b1220"), "titulo" : "Batman Inicia", "director" : "Christopher Nolan", "año" : "2005" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1221"), "titulo" : "El Caballero Oscuro", "director" : "Christopher Nolan", "año" : "2008" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1222"), "titulo" : "El Origen", "director" : "Christopher Nolan", "año" : "2010" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1223"), "titulo" : "The Hangover", "director" : "Todd Phillips", "año" : "2009" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1226"), "titulo" : "Isla de Perros", "director" : "Wes Anderson", "año" : "2018" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1227"), "titulo" : "Fantastic Mr.Fox", "director" : "Wes Anderson", "año" : "2009" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1228"), "titulo" : "Moonrise Kingdom", "director" : "Wes Anderson", "año" : "2012" }
{ "_id" : ObjectId("5e43fca74faa63fd708b122a"), "titulo" : "Avatar", "director" : "James Cameron", "año" : "2009" }
>
```

Fuente: Elaboración propia

En principio, puede parecer una operación algo complicada y no ser trivial el comprender el resultado que nos devolvió pero, si nos fijamos, solo están en el resultado aquellos documentos cuya fecha es **2009** o aquellos cuyos directores son **Christopher Nolan** o **Wes Anderson**.

En conclusión, el operador `$or` siempre buscará que alguna de las condiciones se cumpla. Se recomienda el uso de `$in` siempre que sea posible por cuestiones de eficiencia.

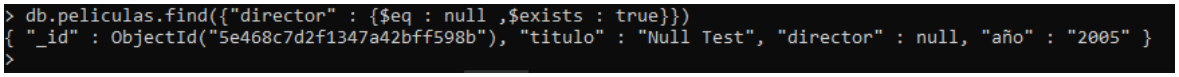
- **`Null`**. Una particularidad de null en Mongo es que no podemos hacer una búsqueda digamos, basada en una comparación con null, simplemente el resultado no sería lo esperado. En este caso debemos hacer uso del operador **`$exists`** para comparar la existencia de un campo que sea null. Por ejemplo:



Fuente: Elaboración propia

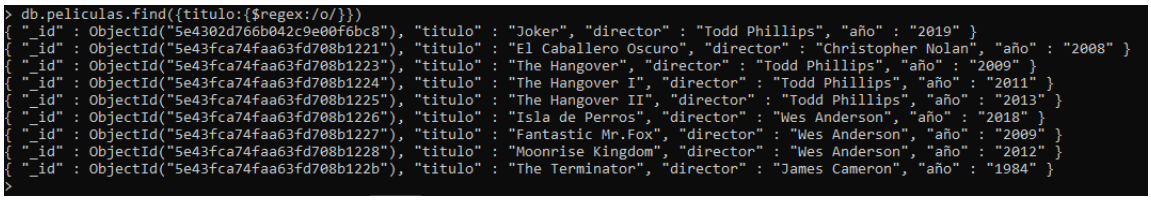
`db.peliculas.find({"director" : {$eq : null ,$exists : true)}})`

Esta búsqueda nos devolverá todos aquellos documentos donde el director tiene un valor que es null.



Fuente: Elaboración propia

- **`$regex`**. MongoDB nos permite hacer comparaciones con expresiones regulares, lo que hace de ello una poderosa herramienta para comparar y buscar cadenas de texto muy específicas.
Por ejemplo, imaginemos que quisiéramos buscar todas las películas cuyo título tenga al menos una letra "o":
`db.peliculas.find({titulo:{$regex:/o/}})`



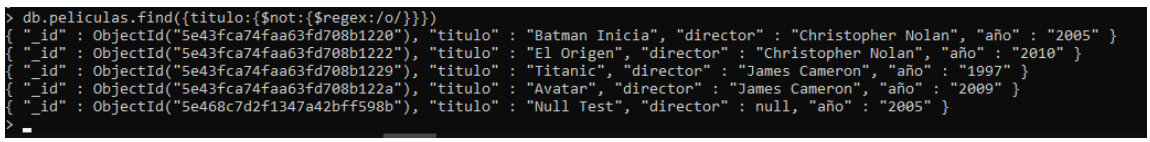
Fuente: Elaboración propia

Con este ejemplo, queda claramente confirmado que las expresiones regulares son una herramienta muy útil y potente.

- **`$not`**. Es un operador que sirve para traer aquellos documents que no coincidan con la consulta. Por ejemplo:

`db.peliculas.find({titulo:{$not:{$regex:/o/}}})`

Esto nos daría todos aquellos documents que no tienen letra "o" en el título:



Fuente: Elaboración propia

Si te fijas, verás que ha surgido lo que podría parecernos un error. Tal vez no te habías dado cuenta de que el resultado tiene una letra "O" en mayúsculas. Esto se debe a que las expresiones regulares son sensibles a las mayúsculas. Para solucionar eso, podemos hacer uso del `flag` **`i`,** que hace que nuestra búsqueda sea independiente de las mayúsculas y de las minúsculas.

```
db.peliculas.find({titulo:{$not:{$regex:/o/i}}})
```

```
> db.peliculas.find({titulo:{$not:{$regex:/o/i}}})
{ "_id" : ObjectId("5e43fca74faa63fd708b1220"), "titulo" : "Batman Inicia", "director" : "Christopher Nolan", "año" : "2005" }
{ "_id" : ObjectId("5e43fca74faa63fd708b1229"), "titulo" : "Titanic", "director" : "James Cameron", "año" : "1997" }
{ "_id" : ObjectId("5e43fca74faa63fd708b122a"), "titulo" : "Avatar", "director" : "James Cameron", "año" : "2009" }
{ "_id" : ObjectId("5e468c7d2f1347a42bff598b"), "titulo" : "Null Test", "director" : null, "año" : "2005" }
>
```

Fuente: Elaboración propia

- **Queries sobre los arrays.** En MongoDB, los arrays vendrán a ser un tipo de dato que continuamente estaremos manejando o almacenando. Estos tienen un comportamiento muy distinto, ya que al trabajar con ellos surgen un mayor número de requerimientos en el momento de hacer las consultas.

  Supongamos que ahora nuestros datos sobre las películas tienen un nuevo atributo llamado **protagonistas,** el cual será un array de nombres. Por ejemplo:

  ```
  {"titulo" : "The Terminator", "director" : "James Cameron",
  "año" : "1984",protagonistas:[" Arnold Schwarzenegger"," Linda
  Hamilton"," Michael Biehn"] }
  {"titulo" : "The Terminator II", "director" : "James Cameron",
  "año" : "1991",protagonistas:[" Arnold Schwarzenegger"," Linda
  Hamilton"," Edward Furlong"," Robert Patrick"] }
  ```

- **En MongoDB,** una consulta como la siguiente:

  ```
  db.peliculas.find({protagonistas:"                    Arnold
  Schwarzenegger"})
  ```

  Nos traería como resultado ambas películas, lo cual es bueno, pues MongoDB trata la información como si fuera un tipo de dato "no array". Pero ahora, ¿qué pasaría si quisiéramos buscar exactamente un objeto en el cual todos los elementos del array coincidan?

  Pues pongámonos manos a la obra, para lo cual introduzcamos ahora más datos en nuestra collection.

  Primero eliminaremos los datos que ya tenemos con el comando:
  `db.collection.drop()`

- A continuación, insertaremos los **nuevos datos** con la información sobre los protagonistas:

```
db.peliculas.insertMany([
    { titulo: "Batman Inicia", director: "Christopher Nolan",
año: "2005", protagonistas: ["Christian Bale", "Michael Caine",
"Liam Neeson", "Gary Oldman"] },
    { titulo: "El Caballero Oscuro", director: "Christopher
Nolan", año: "2008", protagonistas: ["Christian Bale", "Michael
Caine", "Heath Ledger", "Gary Oldman"] },
    { titulo: "El Origen", director: "Christopher Nolan", año:
"2010", protagonistas: ["Leonardo DiCaprio", "Ellen Page",
"Marion Cotillard", "Joseph Gordon-Levitt", "Tom Hardy",
"Cillian Murphy", "Ken Watanabe", "Michael Caine"] },
    { titulo: "The Hangover", director: "Todd Phillips", año:
"2009", protagonistas: ["Bradley Cooper", "Ed Helms", "Zach
Galifianakis"] },
    { titulo: "The Hangover I", director: "Todd Phillips", año:
"2011", protagonistas: ["Bradley Cooper", "Ed Helms", "Zach
Galifianakis", "Ken Jeong"] },
    { titulo: "The Hangover II", director: "Todd Phillips",
año: "2013", protagonistas: ["Bradley Cooper", "Ed Helms",
"Zach Galifianakis", "Justin Bartha", "Ken Jeong"] },
    { titulo: "Isla de Perros", director: "Wes Anderson", año:
"2018", protagonistas: ["Bryan Cranston", "Koyu Rankin",
"Edward Norton", "Bob Balaban", "Bill Murray"] },
    { titulo: "Fantastic Mr.Fox", director: "Wes Anderson",
año: "2009", protagonistas: ["George Clooney", "Meryl Streep",
"Jason Schwartzman", "Bill Murray"] },
    { titulo: "Moonrise Kingdom", director: "Wes Anderson",
año: "2012", protagonistas: ["Jared Gilman", "Kara Hayward",
"Bruce Willis", , "Bill Murray"] },
    { titulo: "Titanic", director: "James Cameron", año:
"1997", protagonistas: ["Leonardo DiCaprio", "Kate Winslet"] },
    { titulo: "Avatar", director: "James Cameron", año: "2009",
protagonistas: ["Sam Worthington", "Zoe Saldaña", "Sigourney
Weaver", "Michelle Rodríguez"] },
    { titulo: "The Terminator", director: "James Cameron", año:
"1984", protagonistas: ["Arnold Schwarzenegger", "Linda
Hamilton", "Michael Biehn"] },
    { titulo: "The Terminator II", director: "James Cameron",
año: "1991", protagonistas: ["Arnold Schwarzenegger", "Linda
Hamilton", "Edward Furlong", "Robert Patrick"] },
]);
```

PARA SABER MÁS

Te invitamos a buscar más información sobre cómo usar las expresiones regulares. Aquí tienes unos sitos donde puedes aprender bastante:

https://www.adictosaltrabajo.com/2015/01/29/regexsam

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_Expressions

Podemos copiar y pegar en la consola. Ello nos generará el siguiente resultado:

```
> db.peliculas.drop()
true
>  db.peliculas.insertMany(
...  [
...  { titulo: "Batman Inicia", director:"Christopher Nolan", año:"2005",protagonistas:["Christian Bale","Michael Caine","Liam Neeson","Gary Oldman"]},
...  { titulo: "El Caballero Oscuro", director:"Christopher Nolan", año:"2008",protagonistas:["Christian Bale","Michael Caine","Heath Ledger","Gary Oldman"]},
...  { titulo: "El Origen", director:"Christopher Nolan", año:"2010",protagonistas:["Leonardo DiCaprio","Ellen Page","Marion Cotillard","Joseph Gordon-Levitt","Tom Hardy","Cillian Murphy","Ken Watanabe","Michael Caine"]},
...  { titulo: "The Hangover", director:"Todd Phillips", año:"2009",protagonistas:["Bradley Cooper","Ed Helms","Zach Galifianakis"]},
...  { titulo: "The Hangover I", director:"Todd Phillips", año:"2011",protagonistas:["Bradley Cooper","Ed Helms","Zach Galifianakis","Ken Jeong"]},
...  { titulo: "The Hangover II", director:"Todd Phillips", año:"2013",protagonistas:["Bradley Cooper","Ed Helms","Zach Galifianakis","Justin Bartha","Ken Jeong"]},
...  { titulo: "Isla de Perros", director:"Wes Anderson", año:"2018",protagonistas:["Bryan Cranston","Koyu Rankin","Edward Norton","Bob Balaban","Bill Murray"]},
...  { titulo: "Fantastic Mr.Fox", director:"Wes Anderson", año:"2009",protagonistas:["George Clooney","Meryl Streep","Jason Schwartzman","Bill Murray"]},
...  { titulo: "Moonrise Kingdom", director:"Wes Anderson", año:"2012",protagonistas:["Jared Gilman","Kara Hayward","Bruce Willis",,"Bill Murray"]},
...  { titulo: "Titanic", director:"James Cameron", año:"1997",protagonistas:["Leonardo DiCaprio","Kate Winslet"]},
...  { titulo: "Avatar", director:"James Cameron", año:"2009",protagonistas:["Sam Worthington","Zoe Saldaña","Sigourney Weaver","Michelle Rodríguez"]},
...  { titulo: "The Terminator", director:"James Cameron", año:"1984", protagonistas:["Arnold Schwarzenegger","Linda Hamilton","Michael Biehn"]},
...  { titulo: "The Terminator II", director:"James Cameron", año:"1991", protagonistas:["Arnold Schwarzenegger","Linda Hamilton","Edward Furlong","Robert Patrick"]}
...  ])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5e46abcc2f1347a42bff5999"),
                ObjectId("5e46abcc2f1347a42bff599a"),
                ObjectId("5e46abcc2f1347a42bff599b"),
                ObjectId("5e46abcc2f1347a42bff599c"),
                ObjectId("5e46abcc2f1347a42bff599d"),
                ObjectId("5e46abcc2f1347a42bff599e"),
                ObjectId("5e46abcc2f1347a42bff599f"),
                ObjectId("5e46abcc2f1347a42bff59a0"),
                ObjectId("5e46abcc2f1347a42bff59a1"),
                ObjectId("5e46abcc2f1347a42bff59a2"),
                ObjectId("5e46abcc2f1347a42bff59a3"),
                ObjectId("5e46abcc2f1347a42bff59a4"),
                ObjectId("5e46abcc2f1347a42bff59a5")
        ]
}
>
```

Fuente: Elaboración propia

Bien, ahora tenemos nuestros datos con los protagonistas de las películas. Hagamos una consulta sencilla, para ver qué películas ha protagonizado "Leonardo DiCaprio".

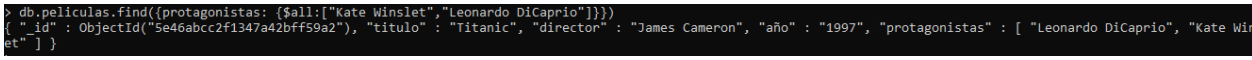
```
db.peliculas.find({protagonistas:"Leonardo DiCaprio"})
```

Esta consulta nos trae como resultado un par de elementos:

```
> db.peliculas.find({protagonistas:"Leonardo DiCaprio"})
{ "_id" : ObjectId("5e46abcc2f1347a42bff599b"), "titulo" : "El Origen", "director" : "Christopher Nolan", "año" : "2010", "protagonistas" : [ "Leonardo DiCaprio", "Ellen Page", "Marion Cotillard", "Joseph Gordon-Levitt", "Tom Hardy", "Cillian Murphy", "Ken Watanabe", "Michael Caine" ] }
{ "_id" : ObjectId("5e46abcc2f1347a42bff59a2"), "titulo" : "Titanic", "director" : "James Cameron", "año" : "1997", "protagonistas" : [ "Leonardo DiCaprio", "Kate Winslet" ] }
>
```

Fuente: Elaboración propia

1. Para hacer una búsqueda más completa y que todos los elementos del array coincidan, podemos usar el operador **`$all`** de la siguiente manera: **`db.peliculas.find({protagonistas: {$all:["Kate Winslet","Leonardo DiCaprio"]}})`**

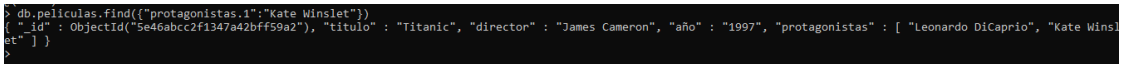


Fuente: Elaboración propia

2. Como se puede observar, el orden de los elementos no afecta el resultado y MongoDB hace la comparación por sí solo. También podemos comparar solo con un elemento del array:

   db.peliculas.find({"protagonistas.1":"Kate Winslet"})

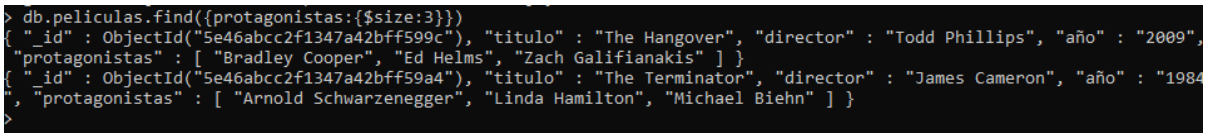
   En este caso, buscariamos que el segundo elemento del array sea "Kate Winslet". Recordemos que los arrays en MongoDB empiezan desde el 0. Tambien cuando hacemos este tipo de consultas debemos agregar "" comillas a la propiedad que se va a usar.



Fuente: Elaboración propia

3. Otro condicional bastante útil para los arrays es **$size,** que nos permitirá comparar el tamaño del array.

   db.peliculas.find({protagonistas:{$size:3}})



Fuente: Elaboración propia

4. En este caso, aplicando esta consulta estamos recuperando aquellos documents donde el array de protagonistas tiene tres elementos. El tipo de consulta `$where` nos permitirá ejecutar código JavaScript como parte de nuestra consulta, es decir, nos permite escribir una función en JavaScript la cual, según sea el caso, si devuelve true, el documento estará en el resultado y en, el caso contrario, debe ser false.

   Veamos un ejemplo para entender más claramente cómo funciona:

```
db.peliculas.find({
    $where: function () {
        return hex_md5(this.titulo) ==
"bedb365b942bd25fa2eaafb2e7b96c4a";
    },
});
```

Fuente: Elaboración propia

5. Aquí, en esta consulta, usamos la condición para buscar el titulo cuyo hash md5 sea igual al que estamos buscando. El uso de este operador está limitado a ciertas funciones y se recomienda usarlo solo cuando es realmente necesario, pues resta mucho rendimiento.

   Hasta ahora hemos revisado los operadores más comunes, aunque es verdad que existen otros, por lo que te proponemos que sigas investigando para sacarle el máximo provecho a MongoDB.

ENLACE DE INTERÉS

Si quieres ampliar tus conocimientos sobre el uso avanzado del operador $where, te recomendamos que accedas al siguiente manual.

https://docs.mongodb.com/manual/reference/operator/query/where/

5.6.3 Update

Ahora, revisaremos otro de los elementos del CRUD, que es el update o actualización. Se usa para modificar los valores de registros que ya están en nuestra collection.

Para conseguir esto tendremos a nuestra disposición los siguientes métodos: updateOne, updateMany y replaceOne.

Cada uno de estos métodos toma como primer parámetro un document para filtrar el registro, que será modificado dentro de la colección. El segundo parámetro llevará los cambios que deben ser aplicados.

MongoDB maneja los updates con la política de que nunca se ejecuten simultáneamente. Esto quiere decir que, si dos updates se hacen simultáneamente, la primera actualización que llegue al servidor se ejecutará y luego la siguiente. Este patrón de diseño se llama patrón de versiones de documentos.

- **updateOne.** Este método se usa para actualizar datos en un documento. Recibe como primer parámetro un documento para la búsqueda y, como segundo parámetro, un documento con los operadores para hacer los cambios, tales como **`$set`**,**`$unset`**, **`$inc`**. Veamos un ejemplo:

```
db.peliculas.updateOne({titulo:"The Hangover I"},{$set
:{titulo:"The Hangover II"}})
```

Aquí estamos actualizando el título de la película usando el operador **$set**.

Como podemos ver, existe un error en nuestros datos con la película **The Hangover**. Por si no te habías dado cuenta, el error está en que son tres películas y, por lo tanto, los títulos deberían ser: The Hangover, The Hangover II y The Hangover III.

- **updateMany.** Al igual que updateOne, se usa para actualizar, con la salvedad de que con este método podremos actualizar muchos documentos a la vez. Veamos un ejemplo:

```
db.peliculas.updateMany({director:"James Cameron"},{$set :{director : "James F Cameron"}})
```

```
> db.peliculas.updateMany({director:"James Cameron"},{$set :{director : "James F Cameron"}})
{ "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 4 }
```

Aquí hemos actualizado el nombre del director de "James Cameron" a "James F Cameron" en una sola línea.

- **replaceOne.** Este método reemplaza el documento deseado por uno completamente nuevo. Veámoslo con un ejemplo:

```
db.peliculas.replaceOne({titulo:"The Hangover"},{titulo:"The Joker",director:"Todd Phillips",año:"2019",protagonistas:["Joaquin Phoenix","Robert De Niro"]})
```

Con este comando hemos remplazado el documento con título "The Hangover".

- Ahora, veamos cómo hacer unas **actualizaciones en arrays**. Tenemos a nuestra disposición una gran variedad de operadores. Echemos un vistazo, por ejemplo, al operador **$push.** Este operador agrega un elemento al final de un array:

```
db.peliculas.updateOne({ titulo: "The Joker" }, { $push: { protagonistas: "Zazie Beetz" } });
```

```
> db.peliculas.updateOne({titulo:"The Joker"},{"$push": {protagonistas : "Zazie Beetz" }  } )
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.peliculas.find({titulo:"The Joker"}).pretty()
{
        "_id" : ObjectId("5e46abcc2f1347a42bff599c"),
        "titulo" : "The Joker",
        "director" : "Todd Phillips",
        "año" : "2019",
        "protagonistas" : [
                "Joaquin Phoenix",
                "Robert De Niro",
                "Zazie Beetz"
        ]
}
>
```

Fuente: Elaboración propia

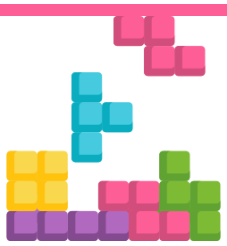
Como se puede observar, hemos agregado un elemento al array de protagonistas de una manera sencilla. Cabe destacar que, de no existir ese array, se crearía con el elemento nuevo.

- Análogamente, también tenemos el **operador `$pull`.** Este operador se utiliza para eliminar un elemento de nuestro array.

```
db.peliculas.updateOne({ titulo: "The Joker" }, { $pull: {
protagonistas: "Zazie Beetz" } });
```

```
> db.peliculas.updateOne({titulo:"The Joker"},{"$pull": {protagonistas : "Zazie Beetz" }  } )
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.peliculas.find({titulo:"The Joker"}).pretty()
{
        "_id" : ObjectId("5e46abcc2f1347a42bff599c"),
        "titulo" : "The Joker",
        "director" : "Todd Phillips",
        "año" : "2019",
        "protagonistas" : [
                "Joaquin Phoenix",
                "Robert De Niro"
        ]
}
>
```

Como se observa, hemos eliminado un elemento del array.



EJEMPLO PRÁCTICO

Se requiere actualizar los datos de la pelicula The Hangover, para que quede así: The Hangover, The Hangover II, The Hangover III.

Esto se realizará usando el método updateOne y lo aprendido anteriormente.

Cambiamos la película II, actualizándola con este comando:

`db.peliculas.updateOne({titulo:"The Hangover I"},{$set :{titulo:"The Hangover II"}})`

Ahora, actualizaremos la tercera con la siguiente operación

`db.peliculas.updateOne({titulo:"The Hangover II",año:"2013"},{$set :{titulo:"The Hangover III"}})`

Así se verán los documentos ya actualizados:

```
> db.peliculas.find({director:"Todd Phillips"})
{ "_id" : ObjectId("5e46abcc2f1347a42bff599c"), "titulo" : "The Hangover", "director" : "Todd Phillips", "año" : "2009", "protagonistas" : [ "Bradley Cooper", "Ed Helms", "Zach Galifianakis" ] }
{ "_id" : ObjectId("5e46abcc2f1347a42bff599d"), "titulo" : "The Hangover II", "director" : "Todd Phillips", "año" : "2011", "protagonistas" : [ "Bradley Cooper", "Ed Helms", "Zach Galifianakis", "Ken Jeong" ] }
{ "_id" : ObjectId("5e46abcc2f1347a42bff599e"), "titulo" : "The Hangover III", "director" : "Todd Phillips", "año" : "2013", "protagonistas" : [ "Bradley Cooper", "Ed Helms", "Zach Galifianakis", "Justin Bartha", "Ken Jeong" ] }
>
```

5.6.4 Delete

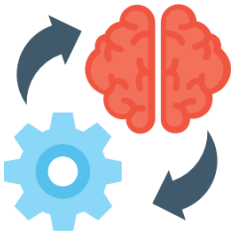
Para la eliminación de elementos, MongoDB nos proporciona los métodos deleteOne y deleteMany. Al igual que en los anteriores métodos, el primer parámetro contiene el documento de búsqueda o de filtro, que nos permite buscar el elemento que deseamos eliminar.

```
db.peliculas.deleteOne({titulo:"The Joker"})
```

```
> db.peliculas.deleteOne({titulo:"The Joker"})
{ "acknowledged" : true, "deletedCount" : 1 }
```

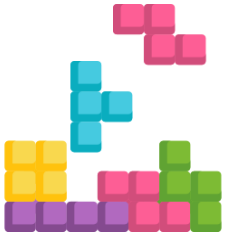
También tenemos el método drop(), el cual nos permitirá desechar una collection de manera fácil y rápida.

```
db.peliculas.drop()
```



RECUERDA

En MongoDB, después de que la información es eliminada, no existe manera de recuperarla, ni de restaurar elementos eliminados.
Solo se podría conseguir eso si antes se dispusiese de un backup de los datos.



EJEMPLO PRÁCTICO

¿Cómo se podrían eliminar todos los elementos de la collection usando el deleteMany()?

El comando sería el siguiente:

db.peliculas.deleteMany({})

Este filtro generará como resultado todos los elementos de la collection y los eliminará.

5.6.5 Capped collections

Normalmente, MongoDB crea collections normales, sin tamaño prefijado. Es decir, ellas van adaptándose a la información según se va insertando. Existe la posibilidad también de crear collections con un tamaño especifico. A este tipo de collections se le denominan Capped Collections y funcionan como un buffer circular, en el cual se van agregando documentos hasta que se llena. A continuación, la información nueva irá reemplazando a la información más antigua.

En este tipo de collections, ciertas operaciones no están permitidas, tales como la eliminación. Esta solo ocurre cuando se hace automáticamente debido al comportamiento anteriormente descrito. Las actualizaciones que hacen que el objeto crezca en tamaño, tampoco están permitidas. Esto garantiza que el orden de los datos ingresados en la collection se mantiene.

Este tipo de collection puede ser útil para gestionar y llevar el control de informaciones de seguridad, tales como datos referidos al login de usuarios.

- **¿Cómo podemos crear este tipo de colecciones?**

  ```
  db.createCollection("log",{capped:true, size:100000,max:10})
  ```

  Aquí hacemos uso del helper createCollection. Como podemos observar, el primer parámetro es el nombre de nuestra collection y el segundo es un document con las especificaciones, como tipo, tamaño y máximo de documentos. Una vez creadas estas collections, no se pueden modificar, sino que deben ser eliminadas y creadas de nuevo.

- También tenemos la posibilidad de convertir una collection normal en una de **tipo capped**, para lo cual podemos hacer uso del siguiente comando:

  ```
  db.runCommand({ convertToCapped: "peliculas", size: 100000 });
  ```

- **Otro método** útil para este tipo de collections es:

  ```
  db.collection.isCapped()
  ```

  Con este método podemos determinar si una collection es del tipo capped.
  Otro aspecto interesante de este tipo de collection es que nos permite el uso del comando tail, que es similar al comando del mismo nombre de Unix, mostrándonos los últimos registros insertados en la collection.

5.7 Map Reduce

En gran medida, vuestro proyecto tiene unos requerimientos similares a los de un buscador. Recuerda lo importante que es que las consultas que realicen vuestros usuarios obtengan respuestas como cuando visitamos un buscador de internet y queremos que los resultados sean coherentes y adecuados a nuestra consulta; para un profesional que consulta una base de datos jurídica esto resulta imprescindible.

Consultar una base de datos documental de tipo profesional y no obtener el resultado esperado y esperable es el camino más corto para abandonar el uso del sistema, lo que sería trágico para las expectativas de la empresa.

Por tanto, debéis valorar la puesta en marcha de un algoritmo similar al Page Rank de Google, de forma que todos los documentos estén jerarquizados a partir de una búsqueda dada.

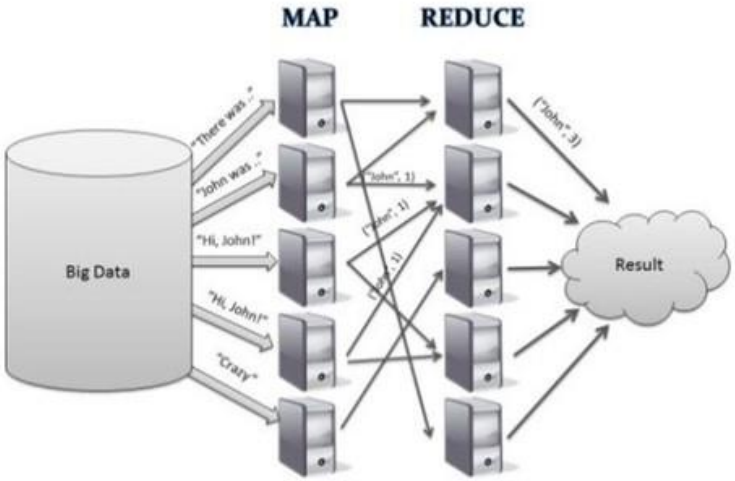
Aunque todavía no hayas decidido que algoritmo de ordenación se va a aplicar, es muy probable que el consumo de recursos sea muy elevado y debas apoyarte en Map Reduce para poder mantener unos tiempos de respuesta rápidos.

Lo primero que debemos pensar cuando hablamos de MapReduce es en big data y procesamiento paralelo de gran cantidad de información.

MongoDB nos proporciona soporte para esta gran herramienta, por lo que en este apartado estudiaremos de que trata MapReduce y cómo utilizarlo en nuestras bases de datos con MongoDB. Conoceremos también un poco sobre su historia y su arquitectura.

También revisaremos sus usos más comunes en la operativa convencional de sistemas de información.

MapReduce tiene ese nombre debido a los dos principales métodos que conforman este paradigma de programación, que son Map y Reduce. Este framework o enfoque fue diseñado para trabajar sobre múltiples volúmenes de datos de **manera paralela.**



Enfoque MapReduce
Fuente: http://blog.leonelatencio.com/tag/hadoop/

Existen implementaciones en diversos lenguajes de programación, tales como C++, Python, JAVA, etc.

MapReduce se realiza en dos fases, que son la de Map y la de Reduce. El Map se aplica en paralelo sobre todos los registros que queremos procesar, devolviéndonos los valores en formato clave valor. A continuación, en la "fase de Reduce", se devuelve una lista con los valores correspondientes a la clave deseada.

- **Historia**. El desarrollo de MapReduce fue liderado inicialmente por Yahoo!, pero en la actualidad se lleva a cabo dentro del proyecto Apache. Una de las principales compañías en potenciar el uso de MapReduce fue Google, ya que para el cálculo del PageRank (clasificación de páginas en las búsquedas), se necesita la multiplicación de grandes matrices.

- La **arquitectura** básica de MapReduce es de tipo maestro / esclavo, por lo que se compone de dos elementos, que son el servidor maestro y los servidores esclavos, uno por cada cluster donde se realizan todas las operaciones en paralelo.

  El servidor maestro es el punto de interacción entre el usuario y el framework MapReduce. Este espera la llegada de trabajos y los coloca por orden de recepción. Después los asigna a los servidores esclavos, para que realicen las tareas correspondientes.

- El **uso** de MapReduce está bastante relacionado con el manejo de grandes cantidades de datos, que requieren ser procesadas por varios nodos a la vez.

Los servicios de tipo cloud son un ejemplo de un uso práctico de MapReduce, ya que estos manejan cantidades gigantes de datos, además de ser tipo on demand (a petición), por lo que el manejo de las cargas va a necesitar que el procesamiento sea paralelo.

Google, como decíamos, para el cálculo del Page Rank, cuando un usuario hace una búsqueda, también utiliza MapReduce.

¿SABÍAS QUE…?

Existe una Implementación OpenSource de MapReduce llamada Hadoop

5.7.1 MapReduce en MongoDB

MongoDB, para hacer uso de MapReduce, nos proporciona la función db.collection.MapReduce(), la cual recibe tres parámetros, donde el primero sería una función que haría las veces de **map**, otra que hará de **reduce** y la tercera, que contendrá las opciones disponibles para la misma.

```
db.collection.MapReduce(
function (){},
function (key, values){},
{
query:{},
out: "Total"
}
)
```

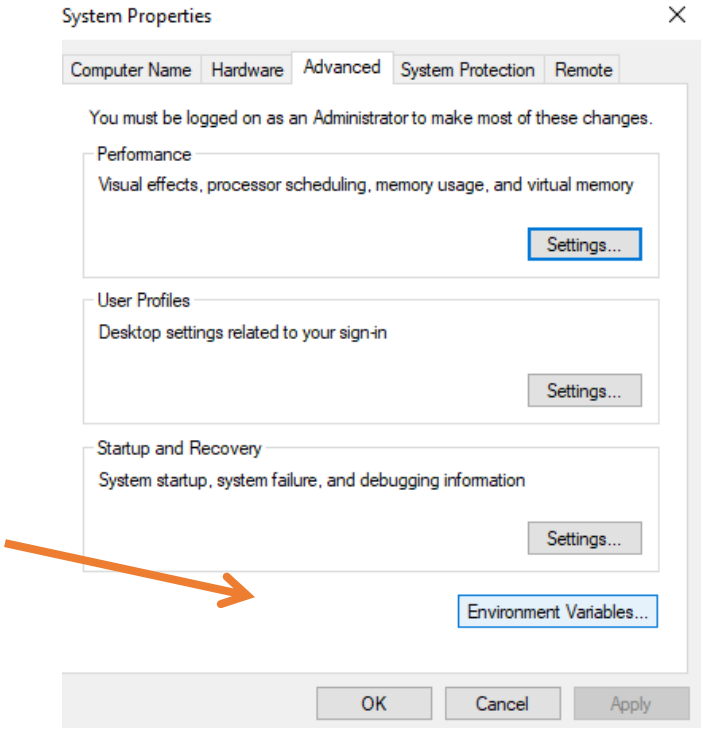
Todo el resultado de esta función se guardará en una nueva collection, cuyo nombre se lo asignamos en el parámetro **out** en las opciones.

Primero debemos configurar la Mongo Shell para trabajar con scripts en un editor, ya que es bastante engorroso trabajar con funciones de múltiples líneas en la consola. Para ello vamos a configurar la variable EDITOR en la consola e indicarle la ruta a nuestro editor preferido. En nuestro caso proponemos utilizar **sublime_text 3** en su versión más reciente, pero puede ser cualquier editor equivalente.

Tras descargar e instalar el editor, debemos agregar la ruta al path en las variables de entorno para Windows 10, que es donde estamos trabajado. En caso de encontrarnos en otra plataforma, deberemos cubrir las necesidades equivalentes.

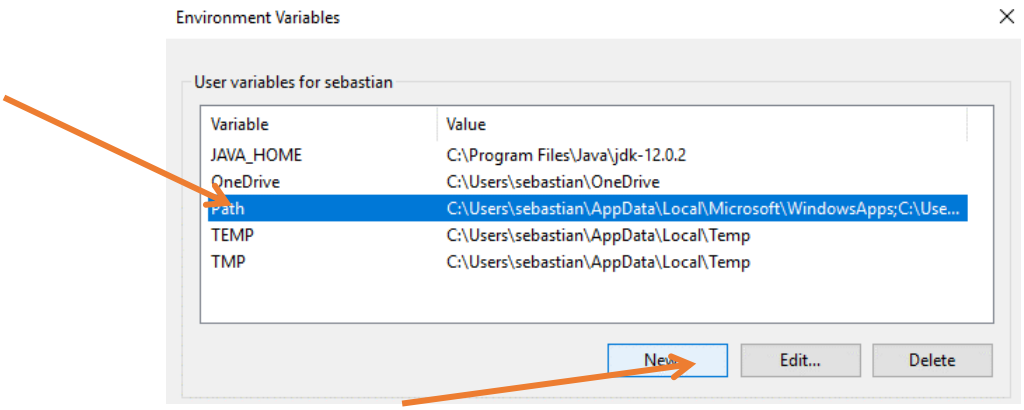
Los pasos serian:

1. Ir a la barra de menú, hacemos clic en buscar y colocamos environment.
2. En las propiedades del sistema, entramos en el botón variables de entorno (environment variables):



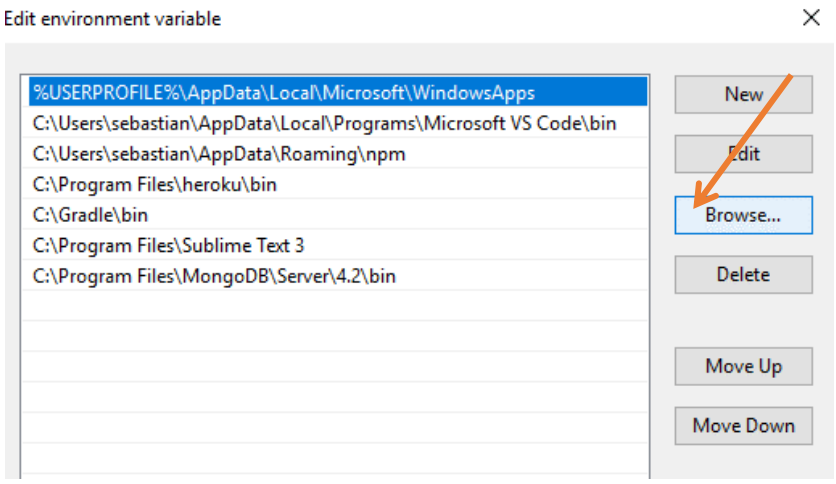
Fuente: Elaboración propia

3. Después, elegimos la variable path y editamos para agregar una nueva ruta.



Fuente: Elaboración propia

4. Presionamos el botón de selección o browse y buscamos la ruta hacia el ejecutable de nuestro editor elegido.



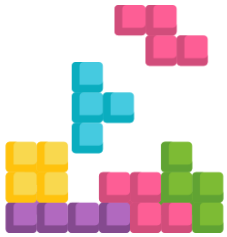
Fuente: Elaboración propia

5. Después le damos al botón Ok en todas las ventanas, para guardar los cambios realizados.
6. Ahora, debemos abrir nuestra consola nuevamente y abrir Mongo Shell.
7. A continuación, debemos escribir: EDITOR =”sublime_text.exe”
8. Y con esto tendremos configurado nuestro editor para esta sesión que hemos abierto.
9. Y ahora, para usar nuestro editor, solo hace falta teclear en la consola edit y el nombre la variable:
10. Primero creamos una variable con algún valor `Var miFuncion = function(){}`, y después `dit miFuncion`. Esto abrirá nuestro editor y, posteriormente, cuando guardemos los cambios y lo cerremos, este actualizará los valores de la variable en la consola. Ahora, que ya sabemos cómo usar un editor para facilitarnos la vida, veamos con un ejemplo práctico como hacer uso de MapReduce en nuestra base de datos de ejemplo.

ENLACE DE INTERÉS

En el siguiente enlace puedes descargar de forma gratuita Sublime Text 3.

https://www.sublimetext.com/3



EJEMPLO PRÁCTICO

Veamos si quisiéramos, por ejemplo, saber cuál es la cantidad de películas y agrupar todo en un solo resultado final por director.

Lo primero es crear la función de **map,** para ello escribiremos en la consola lo siguiente:

```
var mapFunc1 = function(){ emit(this.director, this.titulo)};
+ tecla Enter
```

Después escribiremos la función **reduce**

```
var reduceFunc1 = function(director,titulo){return titulo.length};
+ tecla Enter
```

A continuación, debemos ejecutar la función MapReduce():

```
db.peliculas.mapReduce(mapFunc1,reduceFunc1,{out:"example"})
```

Esto genera la siguiente salida en la consola:

```
> db.peliculas.mapReduce(mapFunc1,reduceFunc1,{out:"example"})
{
        "result" : "example",
        "timeMillis" : 630,
        "counts" : {
                "input" : 12,
                "emit" : 12,
                "reduce" : 4,
                "output" : 4
        },
        "ok" : 1
}
```

Lo que significa que se ejecutó todo correctamente y se creó una collection y que se redujo la entrada a 4, es decir, que agrupamos todas las películas por director.

Veamos la collection example, donde está almacenado el resultado de nuestro MapReduce.

```
> db.example.find()
{ "_id" : "Christopher Nolan", "value" : 3 }
{ "_id" : "James F Cameron", "value" : 4 }
{ "_id" : "Todd Phillips", "value" : 2 }
{ "_id" : "Wes Anderson", "value" : 3 }
```

Ya con esto hemos comprobado cómo utilizar satisfactoriamente MapReduce en MongoDB

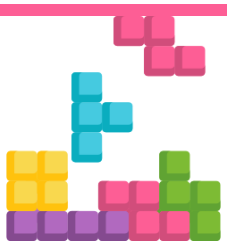
5.7.2 MapReduce Sharded Collections (Collections Fragmentadas) y MapReduce concurrencia

El **MapReduce Sharded Collections** es un método para distribuir datos haciendo uso de varias máquinas. Esto es algo que se usa para manejar grandes cantidades de información en MongoDB.

MapReduce proporciona soporte para este tipo de collections y proporcionará las consultas de forma paralela, optimizando así el tiempo y uso de los recursos.

Por otro lado, el MapReduce concurrencia, En MongoDB, en la fase de map y reduce, se generan collections temporales, para luego escribir la salida. Esto permite el trabajo en distintos hilos y generar una salida unificada.

Ahora, veamos un ejemplo más complejo, en el cual clasificaremos nuestra información según un criterio que elijamos. Esto ocurrirá en la función Reduce.



EJEMPLO PRÁCTICO

Supongamos que necesitamos saber la cantidad de las películas realizadas en cada siglo, por cada director (siglo XX y siglo XXI).

Lo primero sería hacer una función map, que mapee el año. Para ello tecleamos:
edit mapFunc1 + enter

para abrir nuestro editor y agregar nuestra función.

```
function(){
      emit(this.director,this.año)
}
```

Grabamos y cerramos el editor.
Después, tendríamos que hacer la lógica de nuestro ejemplo en el método reduce de igual manera:

edit reduceFunc1 + enter

Agregamos la función.

```
function(director,data){
 var sigloxx=0;
```

```
 var sigloxxi=0;
 for(i=0;i<data.length;i++){
      if (data[i] < '2000'){
            sigloxx++;
      }else{
            sigloxxi++;
      }
 }
 return { 'peliculas siglo XX':sigloxx, 'peliculas siglo
XI':sigloxxi };
}
```

Grabamos y cerramos el editor, para luego invocar la función mapReduce:

```
db.peliculas.mapReduce(mapFunc1,reduceFunc1,{out:"example"})
```

```
> db.peliculas.mapReduce(mapFunc1,reduceFunc1,{out:"example"})
{
        "result" : "example",
        "timeMillis" : 572,
        "counts" : {
                "input" : 12,
                "emit" : 12,
                "reduce" : 4,
                "output" : 4
        },
        "ok" : 1
}
```

Luego revisamos la collection example:

```
> db.example.find()
{ "_id" : "Christopher Nolan", "value" : { "peliculas siglo XX" : 0, "peliculas siglo XI" : 3 } }
{ "_id" : "James F Cameron", "value" : { "peliculas siglo XX" : 3, "peliculas siglo XI" : 1 } }
{ "_id" : "Todd Phillips", "value" : { "peliculas siglo XX" : 0, "peliculas siglo XI" : 2 } }
{ "_id" : "Wes Anderson", "value" : { "peliculas siglo XX" : 0, "peliculas siglo XI" : 3 } }
```

Y con esto, hemos usado satisfactoriamente el MapReduce().

5.8 MongoDB y el modelado de datos

Un proyecto tan complejo y ambicioso justifica, sin lugar a duda, dedicar mucho tiempo y recursos a realizar un modelado óptimo.

Lo que ya está claro es que la aplicación que permitirá a los usuarios realizar consultas a la biblioteca generará un gran número de consultas constantes y concurrentes. Por este motivo, indexar los documentos para mejor la velocidad de respuesta es algo innegociable.

Las colecciones tipo capped son una estrategia que sabes que no os aportaría ningún valor, puesto que en el mundo jurídico no es suficiente con realizar consultas sobre los documentos más recientes, sino que es necesario consultar todos ellos.

Otra cosa es que, en vuestro buscador, además de permitir la consulta de sentencias, leyes, etc., también os interese facilitar a los usuarios la consulta de noticias de actualidad en el sector. En ese caso, la aportación de las colecciones capped sí sería interesante.

El uso de índices no penalizará el rendimiento de vuestros servidores, porque las operaciones de escritura y actualización serán muy poco frecuentes. Lo que sí es posible es que, dada la gran cantidad de documentos, la variedad de idiomas y la amplitud de la semántica de los términos, el tamaño de los índices sea bastante grande en proporción al volumen de documentos, por lo que deberás tener en cuenta este aspecto en el momento de dimensionar la infraestructura.

El **modelado de los datos** es parte fundamental del diseño de cualquier sistema de información, programa, aplicación u otra estructura en la que debamos almacenar información. Hay que ser prudentes y cuidadosos en el momento de elegir la manera en cómo los datos serán almacenados, pensando en distintos aspectos que hay que tomar en consideración.

Entre dichos aspectos se encuentran el crecimiento a futuro de esos datos, el cómo será la mejor manera de consultar dichos datos y, en general, todos esos aspectos se deben revisar cuidadosamente durante la etapa de modelado, ya que, en el futuro, un mal modelado de datos, puede causar innumerables dolores de cabeza y hacernos la vida imposible, tanto en el mantenimiento de la aplicación como en su evolución.

MongoDB posee un esquema flexible y dinámico, por lo que saber utilizarlo con precisión nos ayudará a la hora de definir un modelo de datos eficiente.

El establecer como serán representadas las relaciones entre los elementos que serán almacenados es fundamental para esto, así como hacerlo de manera limpia y correcta para garantizar la calidad de los datos almacenados.

Hay que estar atentos con este esquema flexible porque puede ser un arma de doble filo si no se sabe usar y podemos terminar creando un desorden en nuestros datos.

Lo revisaremos en este módulo para lograr hacer un modelado de datos de manera exitosa que sea eficiente y aproveche todas las ventajas que nos da trabajar con MongoDB.

Cuando modelamos datos una de las primeras cosas que debemos tener en consideración es como las aplicaciones usarán la base de datos, por ejemplo:

- **¿Nuestra aplicación leerá constantemente datos?** Si es así, deberíamos pensar en indexar nuestros documentos para mejor la velocidad de respuesta.

- **¿Nuestra aplicación usara solo los documentos añadidos recientemente?** Si es así, deberíamos considerar el uso de Collections tipo Capped (analizadas anteriormente).

- Entendiendo lo que significa trabajar con un **esquema flexible.** La flexibilidad facilita el mapeo de documentos. En MongoDB no hace falta declarar la estructura de un documento antes de usarlo y esto significa que en una collection puede haber distintos tipos de documentos, inclusive el tipo de datos puede variar a lo largo de toda la collection. Para cambiar la estructura de un documento solo haría falta eliminar un campo o agregar uno nuevo.

- En MongoDB los documentos permiten que los **datos relacionados** puedan ser anidados o embebidos dentro del mismo documento, por ejemplo:

```
{
    _id: 'ObjectId(1asdasdadasd)',
    titulo: 'El Jocker',
    portada:{
                nombre: 'jocker',
                url:"../file/",
                extension: 'jpg'
            },
    comentarios:{[
                    {
                        usuario: "Juan Perez",
```

```
                        comentario:"Que buena pelicula",
                        puntuacion: 10
                    },
                    {
                        usuario: "Jose Ramirez",
                        comentario:"Buenisima, la
recomiendo!!",
                        puntuacion: 9
                    },
                    {
                        usuario: "Carlos Gonzalez",
                        comentario:"aburrida..",
                        puntuacion: 2
                    }
                    ]
                }
}
```

- Si nos fijamos en este documento de ejemplo, podemos ver que existe un documento de tipo **película,** el cual contiene el titulo como información y además de eso tiene dos documentos anidados, uno llamado **portada** y otro es un array de **comentarios.**

  Como ventaja del uso de este tipo de documentos embebidos o anidados es que nos va a reducir drásticamente el número de consultas a la base de datos.

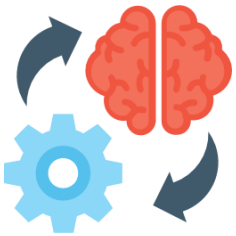
- **¿Cuándo usar documentos embebidos?**
  - Cuando se tienen relaciones de tipo "contenedoras", es decir relaciones uno a uno.
  - Cuando se tienen relaciones de uno a muchos.

- **Objetos referenciados.** También en MongoDB existe la opción de almacenar las relaciones entre los objetos de manera referencial, es decir, almacenar el objectId en los documentos hijos:

```
{
        _id: 'ObjectId(1asdasdadasd)',
        titulo: 'El Jocker',
    }
    {
        _id: 'ObjectId(1qw98jdwqd)',
        pelicula_id:'ObjectId(1asdasdadasd)',
        nombre: 'jocker',
        url:"../file/",
        extension: 'jpg'
    }

    {
        _id: 'ObjectId(dfsqd)',
        pelicula_id:'ObjectId(1asdasdadasd)',
        usuario: "Juan Perez",
```

```
        comentario:"Que buena pelicula",
        puntuacion: 10
    }
```

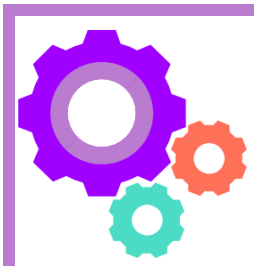


RECUERDA

El factor clave para el modelado de datos con MongoDB reside en la **estructura del documento y en cómo la aplicación representa la relación entre los datos.**

¿SABÍAS QUE…?

En general este tipo de documentos optimizan el proceso de lectura y hacen posible realizar actualizaciones a todos los datos relacionados en una sola operación.



PARA SABER MÁS

¿Cuándo podemos usar este tipo de modelado?

- Cuando el uso de documentos anidados represente una duplicación de datos, pero hay que tener en cuenta que no nos proporcionan suficiente rapidez en el momento de leer los datos.
- Para representar relaciones complejas como las de **muchos a muchos.**
- Para modelar largos conjuntos de datos en forma de herencia.

Como podemos apreciar, aquí tenemos representado el mismo modelo de datos anterior, pero ya no hay documentos anidados, si no que están asociados por el **pelicula_id.**

- **Atomicidad de las operaciones de escritura.** Debemos tener en cuenta que en MongoDB las operaciones de escritura son atómicas a nivel de cada documento, incluso las operaciones que actualizan documentos anidados dentro de otros.

  En un modelo de datos que no esté normalizado, es decir, que posea documentos anidados, combinará todos los datos relacionados en un solo documento, en lugar de realizar cambios en distintos documentos como se haría en una base de datos relacionada.

Esto disminuye el tiempo de procesamiento, porque siempre se realizará todo en una sola operación. También disminuye la cantidad de operaciones o peticiones al servidor.

- **Transacciones de múltiples documentos.** Con MongoDB en aquellos modelos donde se utilizarán referencias entre documentos y no es posible realizar transacciones atómicas, MongoDB soporta las transacciones múltiples a través de **replica-sets** y **sharded clusters**.

- **Sharding (fragmentación).** MongoDB ofrece sharding para proveer un escalado horizontal. Cuando se tiene gran cantidad de datos para procesar, MongoDB ofrece la posibilidad de fragmentar esta carga a lo largo de varias instancias o varios procesos, cada una de ellas llamadas **shards.**

  Para distribuir el tráfico de nuestros datos en MongoDB, debemos utilizar la **shard key.** El uso apropiado de una shard key puede tener implicaciones considerables en el rendimiento, pudiendo ser habilitada o no dependiendo de la ocasión. Es importante tener especial cuidado con el campo que se utilizará para esto.

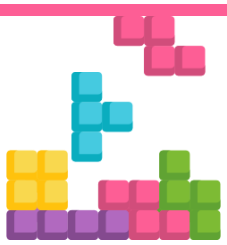
- **Índices.** Usa índices para mejorar el rendimiento en las consultas. Por defecto MongoDB crea un índice único "_id". Si se desea crear índices, se deben tener en cuenta las siguientes consideraciones:
  - Cada índice requiere al menos 8kb de espacio.
  - El uso de índices genera un impacto negativo a la hora de escribir los datos en lo inserts y updates.
  - Collections con gran demanda de lectura y escritura se beneficiarán del uso de los índices.

- **Alto número de Collections.** Muchas veces se necesita tener separada cierta información y aunque no lo parezca, es una buena práctica siempre y cuando el número de documentos en la collection no sea tan grande.

- **Gran número de pequeños documentos.** Si esto pasa, se debería considerar el uso de documentos anidados o embebidos, para optimizar nuestras consultas. Esto generará una gran potencia a la hora de hacer cambios en nuestros datos.

Estas son un pequeño resumen de las consideraciones más básicas que debemos tener cuando modelamos datos con MongoDB.

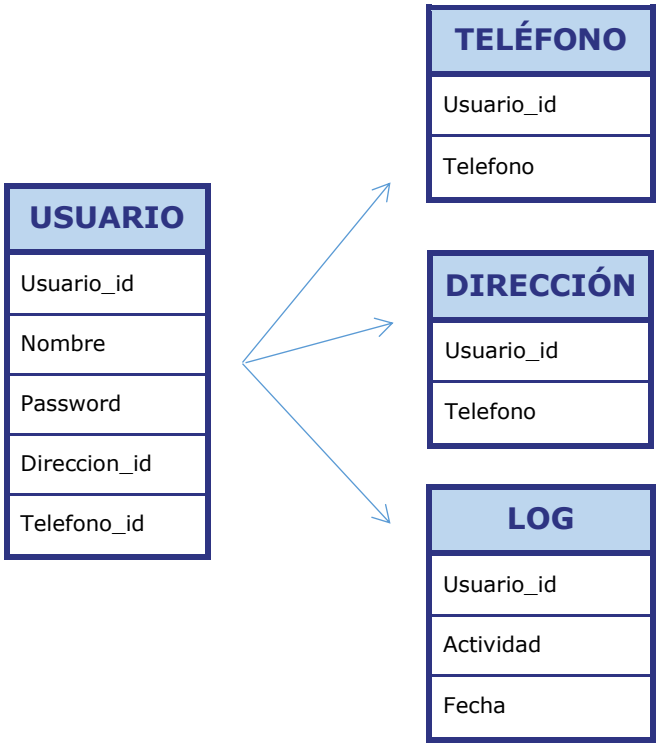
Ahora veamos todo más claramente aplicándolo en un ejemplo:



EJEMPLO PRÁCTICO

Supongamos que tenemos una base relacional normalizada y quisiéramos hacer una migración hacia MongoDB, buscando sacar el mejor provecho en cuanto a rendimiento y uso de los recursos.

Tenemos un registro de usuario en el cual se almacena información sobre este, tal como dirección, teléfono, nombre y, además, necesitamos almacenar un log de actividad. En un modelo entidad relación la cardinalidad sería más o menos así:



Nuestra tarea es llevar este modelo a un modelo alternativo más eficiente en MongoDB.

Lo primero que podemos observar es que este diseño sería ineficiente en MongoDB, ya que MongoDB por defecto indexa todos los documentos, y sería innecesariamente redundante el crear documentos separados en estas relaciones de uno a mucho.

Resultando un documento similar a este:

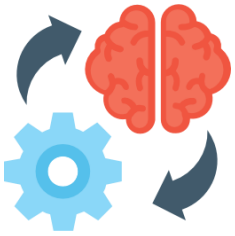
```
{
    nombre : 'usuario',
    password: 'xxxxx',
    direccion: {
```

```
        pais:'direccion',
        ciudad: 'ciudad',
        calle: 'calle',
    },
    telefono: {
        codigo: 'codigo',
        telefono: '123123',
    },
    log:{
        actividad:'Logasdad',
        fecha: 'xxxxxx',
    }
}
```

Ahora bien, podríamos incluso ir más allá y crear una capped collection para nuestro log, para que solo guarde los datos por un periodo de tiempo, para que no se nos agote el espacio en el disco.

En ese caso tendríamos dos collections, una donde estarán los datos del usuario y otra donde se guardarán todas las informaciones de log.

Haciendo uso de mongo, lo que en una base de datos relacional implicaría irremediablemente hacer varias consultas, aquí lo logramos hacer en una única, ahorrando tiempo y recursos.



RECUERDA

Es importante tener en cuenta que este tipo de transacciones en MongoDB conlleva un gran coste de recursos, y se recomienda minimizar el uso de este tipo de transacciones.

5.8.1 Validación de Schema (esquema)

Ahora revisaremos como hacer validaciones directamente sobre una collection. Podremos incluso añadir validaciones a una collection ya creada.

Además de esto tendremos dos parámetros, el **validationLevel** el cual nos indicará cuan estricto será MongoDB con las validaciones y **validationAction,** donde especificaremos que debe hacer MongoDB con los documentos erróneos.

El **validationLevel** acepta dos valores **strict (estricto),** el cual es el valor por defecto y aplica las reglas de validación a todas las inserciones y actualizaciones. También está el valor **moderated (moderado)** donde los

registros ya existentes en la collection no se verificarán, pero los que se inserten nuevos o actualicen si serán verificados.

La manera que usa MongoDB para especificar el Schema es el operador **$jsonSchema,** al cual debemos darle como valor un objeto Json, el cual tendrá todas las descripciones de nuestro esquema. Este mismo se lo pasaremos como parámetro dentro de un objeto validador a MongoDB, para que sea aplicada en nuestra collection.

Otras opciones que podríamos incorporar a nuestras validaciones es el comportamiento de MongoDB, con el **validationAction,** al cual podríamos situarle el valor **error,** que es su valor por defecto. Esto generará un rechazo en la inserción o actualización de los registros que violen las reglas de validación. También podríamos darle el valor **warn,** en el cual el comportamiento será distinto. Mongo permitirá la inserción, pero generará un log con las violaciones a las validaciones.

Este parámetro lo podemos situar en la definición de nuestro esquema de la siguiente manera:

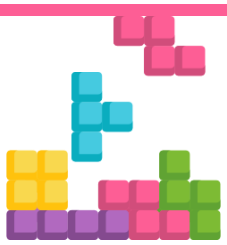
```
{
validator:{
      $jsonSchema:{
            bsonType:"Object",
            required:["titulo","director","año"],
            properties:{
                  titulo:{
                        bsonType:"string",
                        description:"El titulo debe ser un String"
                  },
                  director:{
                        bsonType:"string",
                        description:"El titulo debe ser un String"
                  },
                  año:{
                        bsonType:"number",
                        minimum:1900,
                        description:"El año debe ser mayor a 1900"
                  }
            }
      }
 },
validationAction: "warn"
}
```

Hacer esta configuración hará que en el monitor del servidor se muestren todas las violaciones a las validaciones, pero de igual manera se guardarán en nuestras Collections.

ENLACE DE INTERÉS

En el siguiente enlace puedes encontrar la documentación oficial de MongoDB al respecto del operador $jsonSchema:

https://docs.mongodb.com/manual/reference/operator/query/jsonSchema/#op._S_jsonSchema



EJEMPLO PRÁCTICO

Para agregar validaciones a collections nuevas debemos utilizar el método **db.createCollection().** Creemos una Collection y añadamos unas validaciones.

Lo primero sería construir un objeto json que describirá todas las validaciones que deseamos hacer sobre nuestra collection, que será así:

```
{
validator:{
      $jsonSchema:{
            bsonType:"Object",
            required:["titulo","director","año"],
            properties:{
                  titulo:{
                        bsonType:"string",
                        description:"El titulo debe ser un String"
                  },
                  director:{
                        bsonType:"string",
                        description:"El titulo debe ser un String"
                  },
                  año:{
                        bsonType:"number",
                        minimum:1900,
                        description:"El año debe ser mayor a 1900"
                  }
            }
      }
 }
}
```

Como podemos observar en el bsonType, especificamos el tipo de nuestro documento, que en este caso debe ser **Object.**

A continuación, tenemos la propiedad **required,** donde le diremos a MongoDB qué propiedades de nuestro objeto no pueden faltar, es decir, cuáles son requeridas.

Después tenemos properties, que nos permite decir a MongoDB todas las validaciones que debemos hacer sobre cada uno de nuestros campos, tales como tipo, que se lo decimos con el **bsontype** y la **description,** en la cual comentamos una descripción del campo y cómo debe ser.

Ahora abriremos una consola Mongo Shell y trabajaremos con el editor tal y como describimos ya en su configuración en apartados previos.
Escribimos

```
EDITOR="sublime_text.exe"+ enter
```

Luego creamos una nueva variable llamada param(se le puede indicar el nombre deseado) y le damos el valor de un objeto vacío "{}".

```
var param={} +enter
```

ahora abrimos nuestro editor escribiendo (pasándole a la variable que queremos editar):

```
editor param
```

```
{
    "validator" : {
        "$jsonSchema" : {
            "bsonType" : "object",
            "required" : [
                "titulo",
                "director",
                "año"
            ],
            "properties" : {
                "titulo" : {
                    "bsonType" : "string",
                    "description" : "El titulo debe ser un String"
                },
                "director" : {
                    "bsonType" : "string",
                    "description" : "El titulo debe ser un String"
                },
                "año" : {
                    "bsonType" : "number",
                    "minimum" : 1900,
                    "description" : "El año debe ser mayor a 1900"
                }
            }
        }
    }
}
```

Escribimos el objeto que mencionamos arriba.

A continuación, guardamos mediante **ctrl +s** y cerramos. Ahora, en la consola creamos nuestra nueva collection y añadimos sus validaciones:

```
db.createCollection("pelicula_validator",param) + enter
```

Si todo está bien, obtendremos la siguiente respuesta:

```
{ "ok": 1 }
```

Es el momento de probar cómo funcionan nuestras validaciones

```
db.pelicula_validator.insertOne({ titulo:"ejemplo", año:1960})
```

```
> db.pelicula_validator.insertOne({ titulo:"ejemplo", año:1960})
2020-03-09T09:29:24.244-0400 E  QUERY    [js] WriteError({
        "index" : 0,
        "code" : 121,
        "errmsg" : "Document failed validation",
        "op" : {
                "_id" : ObjectId("5e6644b412b517d17ebcdd70"),
                "titulo" : "ejemplo",
                "año" : 1960
        }
}) :
WriteError({
        "index" : 0,
        "code" : 121,
        "errmsg" : "Document failed validation",
        "op" : {
                "_id" : ObjectId("5e6644b412b517d17ebcdd70"),
                "titulo" : "ejemplo",
                "año" : 1960
        }
})
WriteError@src/mongo/shell/bulk_api.js:458:48
mergeBatchResults@src/mongo/shell/bulk_api.js:855:49
executeBatch@src/mongo/shell/bulk_api.js:919:13
Bulk/this.execute@src/mongo/shell/bulk_api.js:1163:21
DBCollection.prototype.insertOne@src/mongo/shell/crud_api.js:264:
@(shell):1:1
>
```

Como podemos observar, la validación falla, y así se evita la inserción de un objeto que no cumple con las normas de validación.

5.8.2 Actualización de esquemas ya existentes

¿Qué ocurre si ya tenemos un esquema creado y necesitamos añadir nuevas validaciones? Para eso debemos hacer uso del método **db.runCommand().** A este método le pasaremos un documento donde especificaremos el nombre de la collection con el parámetro **collMod,** lo que quedará de la siguiente manera:

```
{
      collMod:"pelicula_validator",
      validator:{
            "$jsonSchema" : {
                  "bsonType" : "object",
                  "required" : [
                        "titulo",
                        "director",
                        "año"
                  ],
                  "properties" : {
                        "titulo" : {
                              "bsonType" : "string",
                              "description" : "El titulo debe ser un
String"
                        },
                        "director" : {
                              "bsonType" : "string",
                              "description" : "El titulo debe ser un
String"
                        },
                        "año" : {
                              "bsonType" : "number",
                              "minimum" : 1900,
                              "description" : "El año debe ser mayor a
1900"
                        }
                  }
            }
      }
 }
```

De igual manera que en el anterior ejemplo, podemos guardar este valor en una variable y a continuación ejecutar:

```
db.runCommand("nuestravariable")
```

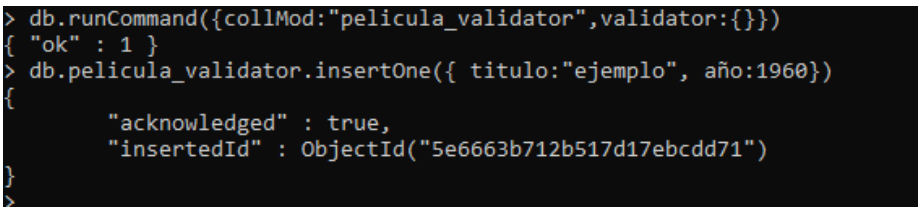
Esto será suficiente para lograr nuestro objetivo.

¿Como eliminar validaciones? La manera para eliminar las validaciones sería usar el comando db.runCommand() nuevamente, pero esta vez le

pasaríamos un objeto validador vacío y el nombre de la collection que deseamos modificar:

```
db.runCommand({collMod:"pelicula_validator",validator:{}})
```

Posteriormente, cuando intentamos insertar un objeto erróneo tendremos la siguiente respuesta, debido a que se ha desactivado la validación y lo inserta sin ningún inconveniente:



Fuente: Elaboración propia

5.9 Indexación

Dada la criticidad de los índices tu proyecto, es una cuestión sobre la que nunca te cansarás de pensar, analizar y comparar alternativas.

Probablemente el uso de índices compuestos en este caso sea una muy buena idea. Por ejemplo, en el caso de las sentencias, son varios los atributos candidatos a formar parte de ese índice compuesto.

Por un lado, se encontraría el título de la sentencia, que suele ofrecer una condensación bastante buena de los principales temas de interés que se tratarán en ella. Pero también hay otros atributos que se suelen utilizar mucho en la búsqueda y recuperación de este tipo de documentos, como son los nombres de los ponentes, es decir, del juez o magistrados que han emitido la resolución.

Los índices son especialmente eficientes cuando el número de documentos que se devuelven frente a una consulta es especialmente bajo en términos relativos respecto del total de información almacenada. En principio, todo hace prever que la biblioteca jurídica va a funcionar de ese modo por varios motivos.

En primer lugar, porque al tratarse de contenidos en todos los idiomas del mundo, al realizar una búsqueda, esta se efectuará en un idioma concreto, motivo por el cual, a pesar de que ese idioma sea muy popular como el inglés o el español, los resultados, como máximo, solo serán un porcentaje pequeño del total.

En segundo lugar, al introducir los términos de búsqueda se tenderá a introducir palabras o conceptos que segmenten muy bien la información a recuperar. Por ejemplo, si buscamos "orden civil" eso dejará fuera a todos los documentos legales que se refieran al orden penal, mercantil, laboral, social, etc.

En este apartado hablaremos sobre los **índices**, cómo funcionan, para qué sirven y cómo nos ayudarán en el momento de trabajar con nuestras bases de datos.

Los índices en general, al igual que el índice de un curso o de un libro, permite acceder a la información que necesitamos de marera directa, saltando todo lo que no estamos buscando.

Analizaremos a continuación:

- Qué son los índices y por qué deberíamos tener interés en utilizarlos.
- Cómo elegir un índice.
- Cómo reforzar y evaluar el uso de los índices.
- Cómo crear o eliminar un índice.

Como podemos observar, el uso de los índices es fundamental para el rendimiento de una base de datos.

En MongoDB los índices ayudan a no requerir de una búsqueda en todos los documentos de una collection para encontrar el resultado deseado.

5.9.1 Introducción a los índices en MongoDB

Como dijimos en la introducción, un índice es una manera de tomar un atajo para buscar alguna información en MongoDB. Esto quiere decir que consiste en buscar en una lista ordenada para llegar a la información deseada, lo cual permite que la búsqueda sea rápida.

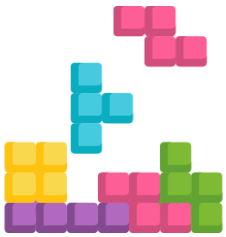
Una consulta que no utilice un índice deberá revisar cada uno de los elementos de una collection para poder encontrar lo que se está buscando.

En lo posible en nuestra base de datos debemos evitar hacer consultas de este tipo, porque en collections muy grandes es un proceso muy lento.

El uso de los índices puede producir cambios impresionantes a la hora de las consultas.

En general, lo que podría tomar más tiempo es decidir qué atributo usar como índice, ya que, dependiendo de los requerimientos, un índice será útil o no.

Veamos esto con un ejemplo práctico.



EJEMPLO PRÁCTICO

Vamos a generar una collection que tenga un alto número de elementos, para comprobar la utilidad de los índices.

Primero usaremos el siguiente script para generar estos registros

```
function (){
	for(i=0; i<= 100000;i++){
		db.usuarios.insertOne({
			"i": i,
			"nombre": "nombre"+i,
			"edad": Math.floor(Math.random()*120),
			"creado": new Date()
		})
	}
}
```

Con este script generaremos 100.000 registros en una collection llamada índice. Podemos ejecutar este script como mejor nos convenga, bien en la Mongo Shell directamente, o usando una variable y editándola en nuestro editor preferido, aunque recomendamos, por comodidad, el segundo método.

Ahora revisaremos el comportamiento de una consulta. Para ello utilizaremos el comando **explain,** un comando que nos será de mucha utilidad como podremos ver.
Ahora intentemos buscar un registro y para ellos usamos el siguiente comando:

```
db.usuarios.find({"nombre": "nombre500"}).explain("executionStats")
+tecla enter
```

Tras esto, obtendremos la siguiente salida por la pantalla

```
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 1,
                "executionTimeMillis" : 84,
                "totalKeysExamined" : 0,
                "totalDocsExamined" : 100001,
                "executionStages" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                                "nombre" : {
                                        "$eq" : "nombre500"
                                }
                        },
                        "nReturned" : 1,
                        "executionTimeMillisEstimate" : 0,
                        "works" : 100003,
                        "advanced" : 1,
                        "needTime" : 100001,
                        "needYield" : 0,
                        "saveState" : 781,
                        "restoreState" : 781,
                        "isEOF" : 1,
                        "direction" : "forward",
                        "docsExamined" : 100001
                }
        },
```

De momento solo prestaremos atención al documento anidado, llamado **executionStats,** donde vemos el total de documentos examinados y podemos observar que son bastantes en el parámetro **totalDocsExamined.** Hay otro parámetro más abajo llamado **nReturned,** el cual vendría a ser la cantidad de resultados arrojados por nuestra búsqueda y esto es correcto, porque según el script que hicimos debe existir solo un usuario con el nombre **nombre500.** Ahora, como podemos ver, **MongoDB** ha necesitado examinar cada uno de los documentos para poder encontrar lo que estábamos buscando. Eso se debe a que Mongo no sabe que el nombre es un parámetro único. Intentemos ahora crear un índice con nuestro parámetro nombre, para lo cual hacemos uso del método:

db.usuarios.createIndex({"nombre":1}) +tecla enter

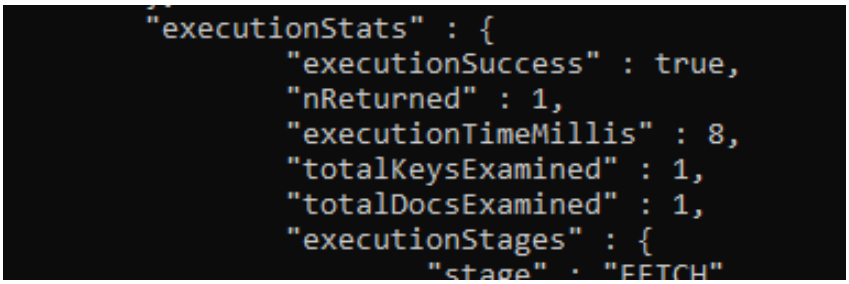
Ello nos proporcionará la siguiente salida:

```
> db.usuarios.createIndex({nombre:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

Ahora repitamos la consulta anterior, para comprobar qué sucede:

db.usuarios.find({"nombre": "nombre500"}).explain("executionStats") +tecla enter

Observamos en este momento una salida mas larga y compleja, pero comprobamos que presenta los mismos parametros de hace un momento



Como podemos observar, el cambio es bastante notable. Solamente se examinó un documento y el tiempo se redujo desde 84 a 8 milisegundos.

¿SABÍAS QUE…?

Los índices en MongoDB funcionan igual que en las bases de datos relacionales

5.9.2 Índices compuestos

En términos generales, el propósito de los índices es hacer las consultas lo más rápidas y eficientes posibles. Los índices compuestos no son más que el uso de más de un atributo en la collection para conformar un índice.

Siguiendo con nuestro ejemplo, ahora queremos hacer una consulta que nos retorne el resultado ordenado por edad. En este caso, nuestro índice **nombre** no nos será de utilidad, ya que por sí solo no nos aporta nada de información con respecto a la edad. Para que esto funcione debemos crear un índice compuesto entre **edad** y **nombre.**

1. Para crearlo usamos el siguiente comando:

```
db.usuarios.createIndex({edad:1,nombre:1}) +tecla enter
```

2. Lo que genera la siguiente salida

```
> db.usuarios.createIndex({edad:1,nombre:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 2,
        "numIndexesAfter" : 3,
        "ok" : 1
}
```

Fuente: Elaboración propia

3. Esto significa que ahora tenemos un índice que está organizado por edad y por nombre, y las búsquedas serán mucho más rápidas. Si queremos, por ejemplo, hacer una búsqueda de este tipo:

```
db.usuarios.find({edad:18})
```

```
> db.usuarios.find({edad:18})
{ "_id" : ObjectId("5e66b49d12b517d17ebd0527"), "i" : 10165, "nombre" : "nombre10165", "edad" : 18,
{ "_id" : ObjectId("5e66b49d12b517d17ebd058b"), "i" : 10265, "nombre" : "nombre10265", "edad" : 18,
{ "_id" : ObjectId("5e66b49d12b517d17ebd05f3"), "i" : 10369, "nombre" : "nombre10369", "edad" : 18,
{ "_id" : ObjectId("5e66b49d12b517d17ebd0679"), "i" : 10503, "nombre" : "nombre10503", "edad" : 18,
{ "_id" : ObjectId("5e66b49d12b517d17ebd06ea"), "i" : 10616, "nombre" : "nombre10616", "edad" : 18,
{ "_id" : ObjectId("5e66b49d12b517d17ebd0747"), "i" : 10709, "nombre" : "nombre10709", "edad" : 18,
{ "_id" : ObjectId("5e66b49d12b517d17ebd0789"), "i" : 10775, "nombre" : "nombre10775", "edad" : 18,
{ "_id" : ObjectId("5e66b49d12b517d17ebd07e6"), "i" : 10868, "nombre" : "nombre10868", "edad" : 18,
{ "_id" : ObjectId("5e66b49812b517d17ebce1c2"), "i" : 1104, "nombre" : "nombre1104", "edad" : 18, "c
{ "_id" : ObjectId("5e66b49e12b517d17ebd08bf"), "i" : 11085, "nombre" : "nombre11085", "edad" : 18,
{ "_id" : ObjectId("5e66b49e12b517d17ebd08fc"), "i" : 11146, "nombre" : "nombre11146", "edad" : 18,
{ "_id" : ObjectId("5e66b49e12b517d17ebd0916"), "i" : 11172, "nombre" : "nombre11172", "edad" : 18,
{ "_id" : ObjectId("5e66b49e12b517d17ebd0956"), "i" : 11236, "nombre" : "nombre11236", "edad" : 18,
{ "_id" : ObjectId("5e66b49e12b517d17ebd095f"), "i" : 11245, "nombre" : "nombre11245", "edad" : 18,
{ "_id" : ObjectId("5e66b49e12b517d17ebd09b7"), "i" : 11333, "nombre" : "nombre11333", "edad" : 18,
{ "_id" : ObjectId("5e66b49e12b517d17ebd09bc"), "i" : 11338, "nombre" : "nombre11338", "edad" : 18,
{ "_id" : ObjectId("5e66b49e12b517d17ebd09fb"), "i" : 11401, "nombre" : "nombre11401", "edad" : 18,
{ "_id" : ObjectId("5e66b49e12b517d17ebd0a33"), "i" : 11457, "nombre" : "nombre11457", "edad" : 18,
{ "_id" : ObjectId("5e66b49e12b517d17ebd0b0f"), "i" : 11677, "nombre" : "nombre11677", "edad" : 18,
{ "_id" : ObjectId("5e66b49e12b517d17ebd0b29"), "i" : 11703, "nombre" : "nombre11703", "edad" : 18,
Type "it" for more
```

Fuente: Elaboración propia

Nos fijamos que esta consulta no demoró prácticamente nada de tiempo. Eso se debe al modo en que MongoDB gestiona los índices. La comprensión de estos nos ayudara a optimizar las búsquedas y la obtención de los datos de manera eficiente.

PARA SABER MÁS

Los índices hacen posible la ejecución eficiente de las consultas en MongoDB porque cuando existe un índice adecuado para una consulta, MongoDB lo utiliza para limitar el número de documentos que debe inspeccionar. En el siguiente enlace puedes acceder a la documentación oficial sobre los índices en MongoDB para aprender más sobre este tema:

https://docs.mongodb.com/manual/indexes/

5.9.3 Índices en objetos y arrays

MongoDB permite que se pueda usar como índices objetos anidados y arrays. Estos pueden ser combinados para hacer uso de índices compuestos y en general se comportan de manera similar a otro tipo de atributos.

Veamos que para crear un índice sobre un documento u objeto anidado lo podemos hacer de manera similar a los otros, con la diferencia de que debemos hacer referencia a una de las propiedades de ese documento anidado, por ejemplo, supongamos que tenemos una collection llamada agenda con documentos con la siguiente estructura:

```
{
      nombre: "Maria Perez",
      direccion:{
            ciudad : "Valencia",
            pais: "España"
      } ,
      telefono:[
      {numero:”1234”,fecha:”01/01/2020”},
{numero:””1234523”,fecha:”10/10/2019”}
]
 }
```

1. Pudiéramos crear un índice con uno de los elementos de dirección:

   ```
   db.agenda.createIndex({dirección.pais : 1})
   ```

2. Debemos tener en cuenta que la creación de este índice tendrá un comportamiento distinto a que si creáramos un índice con el atributo **dirección.** Este índice sería solo de utilidad si quisiéramos hacer consultas basándonos en el documento anidado completo. Algo así:

   ```
   db.agenda.find({dirección:{ciudad:”Valencia”, pais:”España”}})
   ```

3. Ahora, con los arrays sucede de manera similar. Supongamos que tenemos un array con documentos anidados de tipo **teléfono** y quisiéramos indexar por el teléfono que tenga más antigüedad registrada, para lo cual deberíamos crear un índice más o menos así:

   ```
   db.agenda.createIndex({teléfono.fecha:1})
   ```

4. Esto nos permitiría tener un índice sobre la fecha del documento de teléfono. Si necesitásemos indexar sobre algún elemento en un array, también podríamos utilizar el valor de su índice

   ```
   db.agenda.createIndex({teléfono.1.fecha:1})
   ```

5.9.4 Cuando no usar índices

A pesar de ser una herramienta muy potente, que nos permite recuperar los datos de manera rápida, existen algunos casos en donde el **uso de índices no es recomendable**, por ejemplo, cuando las consultas cada vez necesitan traer un resultado de mayor tamaño de la collection.

Es decir, para traer pequeñas partes de la collection los índices son extremadamente eficientes. Aproximadamente, si una consulta devuelve alrededor del 30% de los datos, es cuando la velocidad puede disminuir y debemos pensar en utilizar otro método de consulta para recuperar esa cantidad de información.

5.9.5 Tipos de índices

A continuación, presentaremos una descripción de los tipos de índices más comunes.

- **Índices únicos.** Son aquellos donde su valor no podrá estar duplicado en la collection. Si quisiéramos insertar un documento con un índice repetido, MongoDB nos devolvería un error de duplicidad. El índice único por defecto que ya conocemos es "_id" (este índice no puede ser eliminado de una collection).

- **Índices únicos compuestos.** Estos son aquellos que están compuestos por más de un atributo. Dichos atributos pueden aparecer en varios documentos a lo largo de la collection, pero en conjunto crean una unicidad.

- **Índices parciales.** Estos índices pueden estar más de una vez en una collection y se usan para identificar grupos de datos dentro de la collection.

- **¿Cómo indicarle a MongoDB el tipo de un índice?** En el momento de la creación de un índice, el método **createIndex()** recibe como parámetros un segundo documento, el cual nos permite especificar el tipo. Sería como se indica a continuación:

  ```
  db.agenda.createIndex({nombre:1},{unique:true})
  ```

  Ahora, para crear los índices parciales debemos hacer uso de la opción **partialFilterExpression,** la cual lo que hace es crear un filtro basado en la expresión que recibe. Dicha expresión puede contener los siguientes operadores:

- `$eq` el operador de equivalencia
- `$exists` es el operador para comprobar la existencia o no
- `$gt, $gte, $lt, $lte` ya explicados anteriormente
- `$type` para comprobar el tipo.

Pudiéramos crear un subconjunto de datos de la manera siguiente:

```
db.agenda.createIndex({nombre:1},{partialFilterExpression:{direccion.
pais:{$eq: "España"}}})
```

Esto nos crearía un **índice** parcial **sobre** el campo país y validaría que solo los que tengan el valor determinado por la expresión estén dentro de ese subconjunto de datos.

5.9.6 Manejo de índices

Cuando trabajamos con **índices** debemos modificarlos, crearlos o eliminarlos. Para esas tareas tenemos a nuestra disposición **createIndex, createIndexes** y **dropIndexes.** También podemos consultar los índices que ya están disponibles en una collection con el método **getIndexes,** el cual nos arrojará un listado de todos los índices.

```
db.usuarios.getIndexes() + tecla Enter
```

Obtendremos la siguiente salida

```
> db.usuarios.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "test.usuarios"
        },
        {
                "v" : 2,
                "key" : {
                        "nombre" : 1
                },
                "name" : "nombre_1",
                "ns" : "test.usuarios"
        },
        {
                "v" : 2,
                "key" : {
                        "edad" : 1,
                        "nombre" : 1
                },
                "name" : "edad_1_nombre_1",
                "ns" : "test.usuarios"
        }
]
>
```

Fuente: Elaboración propia

Para eliminar algún índice del listado debemos utilizar su nombre, que se encuentra en el atributo **name.**

```
db.usuarios.dropIndex("edad_1_1nombre_1")
```

```
> db.usuarios.dropIndex("edad_1_nombre_1")
{ "nIndexesWas" : 3, "ok" : 1 }
```

Fuente: Elaboración propia

Como podemos observar, hemos eliminado uno de los índices que creamos anteriormente.

5.10 Consultas en MongoDB

En gran medida, el uso intensivo de los índices ya habrá resuelto gran parte de los problemas de eficiencia que podrían presentar las consultas que la aplicación de la biblioteca jurídica universal realizase sobre los documentos almacenados en MongoDB.

Como aspectos de mejor se podría pensar en las consultas cubiertas, es decir, además de permitir consultas a texto completo, totalmente libres y no estructuradas, tenemos la posibilidad de habilitar búsquedas exclusivamente por campos que han sido indexados. Por ejemplo, podríamos definir una opción de búsqueda que permitiese buscar solamente por el nombre de un juez y que nos devolvería todas las sentencias donde dicho juez ha participado. De este modo, un gran porcentaje de las consultas más frecuentes serían respondidas bajo el enfoque de consultas cubiertas y solamente aquellas más complejas presentarían un formato ampliado.

Otra posible mejora en el sistema de consultas de nuestra aplicación sería limitar el número de resultados en aquellas consultas que por ser demasiado vagas o genéricas devuelven un número muy grande de resultados. Por ejemplo, consultas donde los términos de búsqueda son interjecciones, artículos y otras partículas de los diferentes idiomas.

A medida que vamos desarrollando todo el sistema, siempre tendremos a disposición el método explain que nos permitirá ir pulsando cómo evoluciona la calidad de las consultas y tendremos información para tomar decisiones adecuadas y realizar cambios en la configuración.

5.10.1 Optimización de consultas

La **optimización** consiste en afinar o calibrar todos los componentes de un proceso para que este funcione bajo las mejores condiciones posibles, para de ese modo, entregar un resultado en el menor tiempo posible.

En MongoDB esto es posible realizarlo para que nuestras consultas sean lo más rápidas posible. Existen varias condiciones que podemos mejorar, ofreciéndonos así un menor tiempo de respuesta, lo que hace que también las aplicaciones desarrolladas funcionen mejor.

En este apartado analizaremos cómo lograr consultas más eficientes, valiéndonos de distintos métodos tales como el uso de índices, cuestión que ya introdujimos en el módulo anterior.

Como dijimos en la introducción, el uso de índices ayuda en la optimización de las consultas y, en general, podemos mejorar la eficiencia en todas las operaciones que realicen lecturas sobre nuestros datos, reduciendo en gran medida la cantidad de datos que se procesan.

¿Cómo crear un índice para ayudar a nuestras consultas? Si tenemos una búsqueda que requiera de buscar o clasificar los datos en función de un atributo o más, un índice en el atributo o un índice compuesto será la solución para evitar que MongoDB tenga que revisar toda la collection, comparar para posteriormente devolver el resultado deseado.

Como ya hemos mencionado, un índice puede evitar que Mongo tenga que revisar todos los elementos en la collection. Además de esto, los índices pueden ser ordenados para generar un mayor rendimiento. En general, los índices de un solo atributo no aplican esto, pero cuando hablamos de índices compuestos sí que funciona.

Además de esto, la búsqueda a través de los índices tiene soporte para el uso de operadores sobre ellos.

5.10.2 Selección de consultas

El uso de los índices en el momento de realizar una consulta no siempre va a ser el camino más rápido. Esto se puede determinar por el nivel de selección de una consulta.

Las consultas con mayor grado de selectividad retornan un menor porcentaje de resultados. Por ejemplo, si buscamos un resultado basándonos en _id, tendríamos un grado alto de selectividad, ya que como máximo devolverá un solo valor que coincida con este id.

Las consultas con un menor grado de selectividad no pueden usar eficazmente los índices.

5.10.3 Consultas cubiertas

Denominaremos este tipo de consultas a todas aquellas que pueden ser satisfechas por completo utilizando un índice, sin necesidad de examinar ningún otro documento.

Diremos que una consulta está cubierta cuando:

- Todos los campos en la consulta son partes de un índice.
- Todos los campos devueltos en los resultados están en el mismo índice.
  Ningún campo en la consulta es igual a nulo (es decir, `{"campo": nulo}` o `{"campo": {$ eq: nulo}})`.

El uso de este tipo de consultas es bastante recomendable, ya que son veloces. Si disponemos de un buen indexado de nuestros datos, podremos mejorar notablemente las consultas.

1. Supongamos que tenemos una base de datos con el siguiente tipo de documentos:

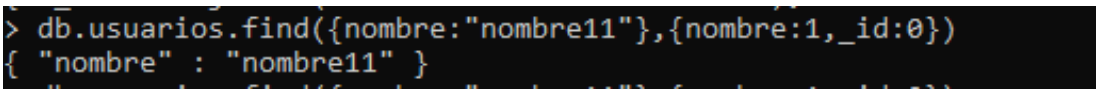
```
{
    _id: 0, usuario: {
        nombre: "juan";
    }
}
```

2. Tenemos el siguiente índice configurado:
   {"usuario.nombre":1}

3. Esta consulta estaría cubierta por este índice:
   db.usuarios.find({"usuario.nombre":"juan"},{"usuario.nombre":1, "_id":0})

4. Como vemos, debemos excluir el atributo **_id.** Esto se debe a que nuestro índice no incluye el **_id .** Un ejemplo de esto sería:

```
db.usuario.find({nombre:"nombre11"},{nombre:1,_id:0})
```



Fuente: Elaboración propia

5.10.4 Uso de explain

Para estudiar la cobertura de un índice podemos hacer uso del método explain(). Este proporciona la información sobre la consulta y lo podremos utilizas de la siguiente forma:

```
db.collection.find().explain()
```

A este método le podemos pasar un parámetro para configurar el nivel de explicación tal que así:

- "queryPlanner": es el valor por defecto que devuelve el plan de consulta generado por el optimizador de consultas.
- "executionStats": que proporciona un resumen de las estadísticas del optimizador de consultas, tras ser ejecutado.
- "allPlansExecution": que devuelve las estadísticas de todos los posibles planes contemplados por el planificador de consultas de MongoDB.

Probemos con un ejemplo partiendo del ejemplo anterior:

```
db.usuarios.find({nombre:"nombre11"}).explain("executionStats")
```

Esta consulta nos traerá como resultado las estadísticas de la ejecución como se observa a continuación:

Fuente: Elaboración propia

En la primera parte podemos observar la sección inicial "query planner" con las configuraciones de nuestra consulta. Más abajo encontramos el "winningPlan", donde podemos ver el nombre del índice y sus características, así como la dirección. Cabe destacar que, para MongoDB, en los índices de un solo campo, esta opción no genera ningún impacto en la búsqueda, ya que Mongo puede usar estos índices en ambas direcciones.

Posteriormente tenemos otra sección donde se encuentran las estadísticas de la ejecución y disponemos de ciertos parámetros que podemos tomar en cuenta a la hora de la optimización, tales como:

- **nReturned** : esta seria la cantidad de resultados devueltos.
- **executionTimeMillisEstimate**: es el tiempo de ejecución en milisegundos.
- **docsExamined**: es la cantidad de documentos examinados.

Fuente: Elaboración propia

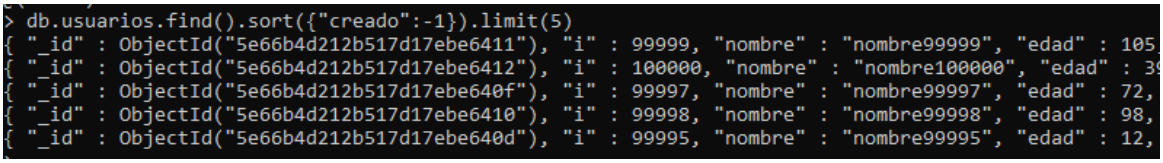
5.10.5 Limitando el número de resultados para reducir el tráfico en la red

MongoDB ofrece la posibilidad de limitar la cantidad de resultados que devuelve una consulta. Si ya sabemos la cantidad de resultados que necesitamos, podemos usar el método **limit()**.

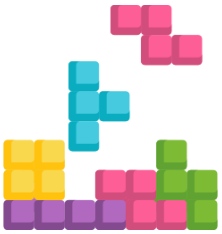
```
db.usuarios.find().sort({creado : -1}).limit(5)
```

Suponiendo que tengamos un campo fecha, esto nos entregaría los últimos cinco usuarios creados en nuestra collection. Por ejemplo, podríamos realizar la siguiente consulta:

```
db.usuarios.find().sort({creado:-1}).limit(5)
```



Fuente: Elaboración propia



EJEMPLO PRÁCTICO

Como hemos observado en la consulta anterior, se ha demorado un poco, debido a que no existe ningún índice. Veamos las estadísticas que nos ofrece esta consulta con explain.

```
db.usuarios.find().sort({ creado: -1
}).limit(5).explain("executionStats");
```

```
"queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "test.usuarios",
        "indexFilterSet" : false,
        "parsedQuery" : {

        },
        "winningPlan" : {
                "stage" : "SORT",
                "sortPattern" : {
                        "creado" : -1
                },
                "limitAmount" : 5,
                "inputStage" : {
                        "stage" : "SORT_KEY_GENERATO
                        "inputStage" : {
                                "stage" : "COLLSCAN"
                                "direction" : "forwa
                        }
                }
        },
        "rejectedPlans" : [ ]
```

Aquí podemos ver varias cuestiones interesantes. Primero, nos percatamos de que no existe ningún tipo de índice para esta búsqueda y en winnigPlan vemos que se realizara un SORT. Observamos también el tiempo que esto va a tomar en la sección de más abajo:

Vemos que la cantidad de documentos que debe revisar es la collection completa y el tiempo de ejecución será de 322 milisegundos. No parece mucho, pero cuando los datos crezcan esto puede aumentar exponencialmente y si tenemos procesos en cola y alto tráfico esto supone un potencial cuello de botella.

Veamos qué ocurre si creamos un índice para esta circunstancia:

```
db.usuarios.createIndex({creado:1})
```

Ahora, ejecutemos la misma búsqueda con el explain()

```
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "test.usuarios",
                "indexFilterSet" : false,
                "parsedQuery" : {

                },
                "winningPlan" : {
                        "stage" : "LIMIT",
                        "limitAmount" : 5,
                        "inputStage" : {
                                "stage" : "FETCH",
                                "inputStage" : {
                                        "stage" : "IXSCAN",
                                        "keyPattern" : {
                                                "creado" : 1
                                        },
                                        "indexName" : "creado_1",
                                        "isMultiKey" : false,
                                        "multiKeyPaths" : {
                                                "creado" : [ ]
                                        },
                                        "isUnique" : false,
                                        "isSparse" : false,
                                        "isPartial" : false,
                                        "indexVersion" : 2,
                                        "direction" : "backward",
                                        "indexBounds" : {
                                                "creado" : [
                                                        "[MaxKey, MinKey]"
                                                ]
                                        }
                                }
                        }
                },
                "rejectedPlans" : [ ]
        },
```


Lo primero que podemos observar en esta sección es el uso del índice que acabamos de crear automáticamente. El planificador de consultas de MongoDB elige el mejor plan para el uso de un índice.

A continuación, en las estadísticas podemos ver los cambios en la cantidad de documentos examinados y el tiempo de ejecución


```
        },
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 5,
                "executionTimeMillis" : 8,
                "totalKeysExamined" : 5,
                "totalDocsExamined" : 5,
                "executionStages" : {
                        "stage" : "LIMIT",
```


Vemos que ha bajado a 5 la cantidad de documentos examinados y a solamente 8 milisegundos el tiempo de ejecución.

5.10.6 Usando el parámetro projection para obtener solo los datos que necesitamos

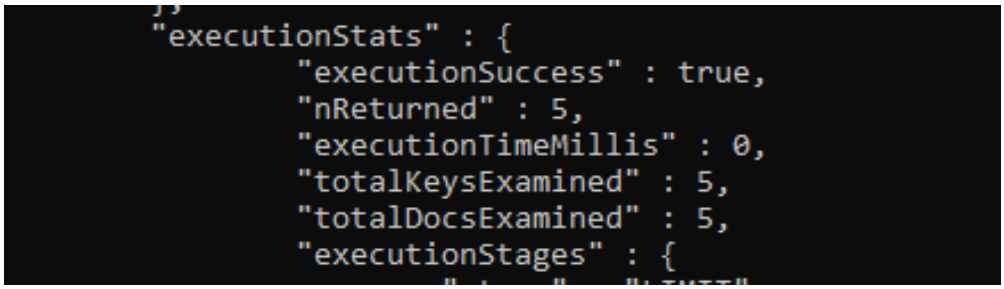
El método find() nos permite pasar un parámetro para especificar los campos que tendremos en nuestro resultado. Por ejemplo:

```
db.usuarios.find({},{nombre:1}).sort({fecha:-1})
```

En este caso debemos colocar el valor del campo y establecer 1 si queremos que esté en el resultado y 0 si no lo queremos.

Ahora, veamos cómo afecta a nuestra consulta anterior este parámetro:

```
db.usuarios.find({}, { nombre: 1 }).sort({ creado: -1
}).limit(5).explain("executionStats");
```



Fuente: Elaboración propia

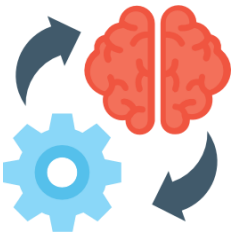
Como podemos ver, el tiempo de ejecución bajo de 8 a 0 milisegundos.

5.10.7 Uso de $hint

En la mayoría de los casos, el optimizador de consultas buscará usando el índice óptimo para la operación. En cualquier caso, también podemos forzar a MongoDB al uso de un índice específico.

Un ejemplo sería:

```
db.usuarios.find({},{nombre:1}).hint({"nombre_del_indice":1})
```



RECUERDA

MongoDB posee un planificador de consultas que implica un proceso de selección con el operador $hint. Podemos evitar esto y hacer una consulta saltándonos el proceso de selección, pero hay que tener cuidado y estar seguro de que es el mejor camino para realizar nuestra consulta.

5.11 Réplicas, clústers y balanceos de gargas en MongoDB

Un proyecto tan grande como el que estamos abordando debe tener todas las salvaguardas posibles para garantizar que no se pierda la información gestionada y que el tiempo de disponibilidad para los usuarios se acerque al 100%.

En cierta medida, las decisiones que tomemos al respecto de estas cuestiones en nuestro proyecto estarán influenciadas por el modo en que vayamos a desplegar la infraestructura, es decir, si decidimos implantar el sistema en una nube pública o en una nube privado o aplicamos arquitecturas más convencionales.

Utilizar replica set puede ser costoso, pero con la misma inversión estamos obteniendo dos beneficios. Por un lado, el de la seguridad, ya que se comportan como back ups, al estar los documentos replicados en dos o más servidores. De esta forma, fallos o caídas en parte del sistema no implican el cese de la prestación de servicios.

Pero también obtenemos el beneficio del rendimiento y eficiencia porque, recordemos, que nuestro sistema va a sufrir gran número de consultas concurrentes y gracias a los conjuntos de réplica, será posible que, en cada momento y a cada usuario, se les devuelva la misma información desde distintos servidores o nodos de la red.

Para el **desarrollo y pruebas** es interesante el uso de un servidor único donde realicemos todo el modelado de datos y demás tareas, pero a la hora de trasladar un sistema a producción disponer de un servidor único se vuelve peligroso. ¿Qué pasaría si el servidor se viniese abajo por algún motivo? Esto generaría retraso en todo. Pensemos incluso qué pasaría si el disco duro fallase y en el peor de los casos se perdiesen todos los datos.

Pero no hay porque preocuparse, dado que MongoDB ofrece la posibilidad de replicación, que es una manera de mantener copias exactas de los datos en distintos servidores. La replicación mantiene los datos a salvo y las aplicaciones en ejecución, incluso si algo le sucediese a más de uno de los servidores.

Si un servidor fallase, es posible seguir accediendo a los datos y si la información estuviera corrupta, podríamos hacer una nueva copia de los datos.

En este apartado haremos una introducción a los mecanismos de salvaguarda de MongoDB y su sistema de alta disponibilidad, llamado Replica Sets. Para ello abordaremos los siguientes temas:

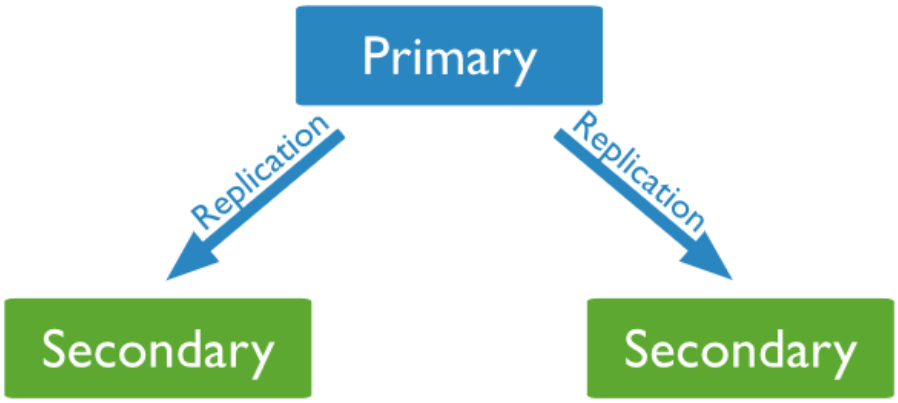
- ¿Qué son los replica sets?
- ¿Cómo configurarlos?

Además de esto abordaremos los **sharding clusters,** que no es más que dividir los datos en varios servidores, por lo que se puede usar el término particionado de datos o fragmentación y es un gran recurso para sacar el máximo provecho a los servidores. Sobre estas cuestiones vamos a analizar:

- ¿Qué es el particionado o sharding cluster, así como los elementos que conforman un cluster?
- ¿Cómo configurar el particionado?
- ¿Cómo se interacciona con este particionado?
- ¿Cómo balancear los datos?

Como dijimos anteriormente, la replicación consiste en tener copias exactas de nuestros datos repartidos en distintos servidores, lo que nos servirá de respaldo de la información y como mecanismo de seguridad para mantener la integridad de los datos y que siempre estén accesibles para conseguir un sistema de alta disponibilidad. En MongoDB, para tratar todos estos conceptos, hablamos de **replica sets.**

Un replica set no es más que un grupo de servidores en el cual uno de ellos será llamado **primario (primary) y** este servidor primario realizará acciones de escritura sobre los otros que llamaremos **secundarios (secondary),** que a su vez pueden ser múltiples. Estos secundarios mantendrán copias del **primario**, para si en alguno momento el **primario** falla, los **secundarios** podrán elegir un **nuevo primario** por sí solos.



Fuente: Elaboración propia

¿SABÍAS QUE…?

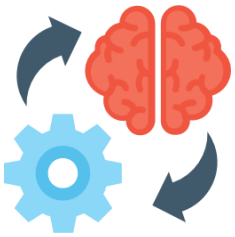
Para saber más sobre MongoDB y las herramientas que ofrece en la nube para experimentar con este tipo de configuraciones, accede al siguiente enlace.

https://www.mongodb.com/cloud/atlas

5.11.1 Cómo configurar un replica set

Ahora veremos cómo configurar un servidor con **replica sets.** En este apartado configuraremos un replica set de tres nodos en un mismo equipo, para que podamos experimentar y ver como es el funcionamiento de estas. También veremos las tareas de recuperación tras una pérdida de datos y otros aspectos interesantes.

Un dato importante que debemos tomar en cuenta cuando estemos en un entorno de producción es el uso de listas de conexión de semillas por DNS para la conexión de las réplicas, ya que podremos configurar la rotación sin necesidad de acceder a la configuración de cada servidor.



RECUERDA

En un entorno de producción siempre debemos tener las réplicas separadas en un servidor para cada una

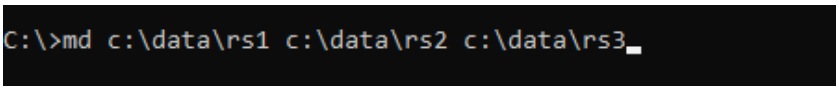
EJEMPLO PRÁCTICO

Dado que los clientes del servicio que presta tu empresa pagan una cantidad de dinero elevada por él, esperan obtener un servicio confiable.

Por este motivo es necesario crear uno o varios replica set en MongoDB, dado que es el mecanismo que esta base de datos proporciona para garantizar la alta disponibilidad

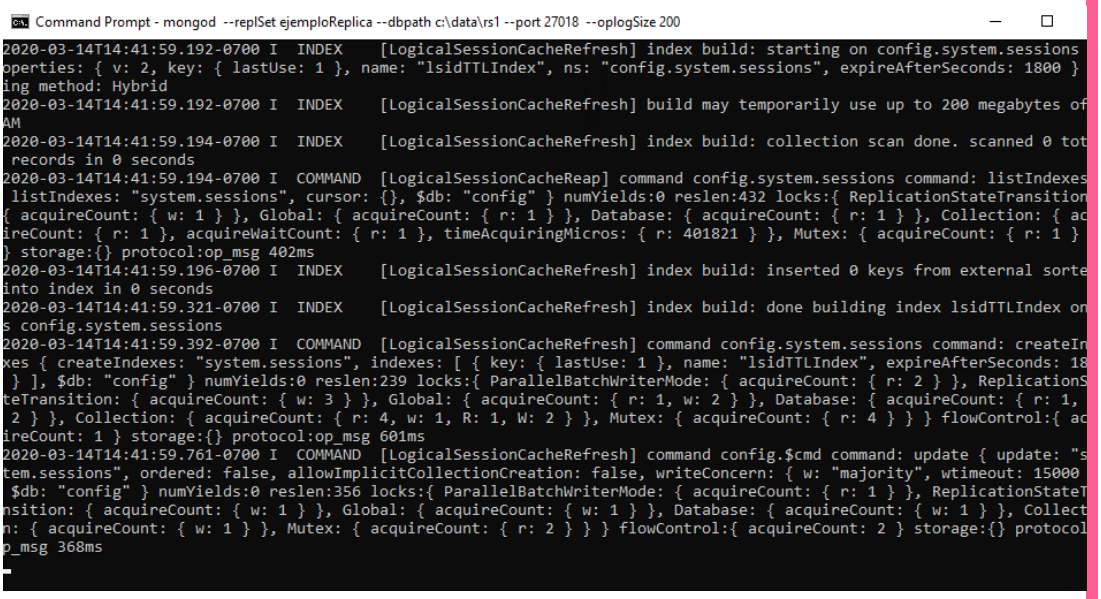
1. Lo primero será crear directorios separados para cada uno de nuestros nodos, para lo cual abrimos una consola y escribimos lo siguiente:

```
md c:\data\rs1 c:\data\rs2 c:\data\rs3 + enter
```



2. Tras esto abriremos 3 consolas separadas con el comando cmd en la pantalla de buscar y escribiremos en cada una el siguiente comando:

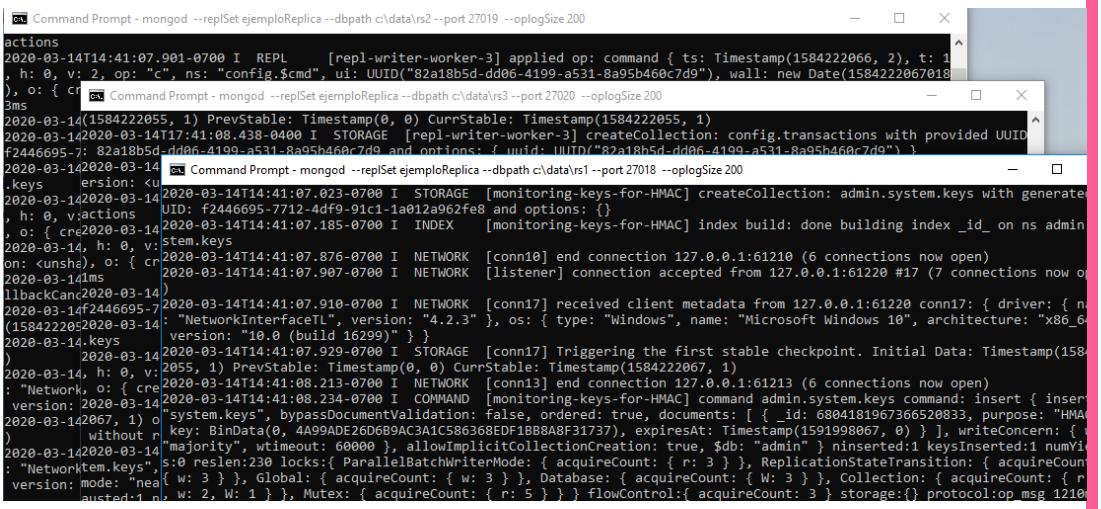
mongod --replSet ejemploReplica --dbpath c:\data\rs1 --port 27018 --oplogSize 200



Esto lo repetiremos para todas las réplicas:

mongod --replSet ejemploReplica --dbpath c:\data\rs2 --port 27019 --oplogSize 200

```
mongod --replSet ejemploReplica --dbpath c:\data\rs3 --port 27020  -
--oplogSize 200
```



Como podemos observar, en cada comando se cambia el p**uerto** y la **carpeta** donde almacenaremos los datos.

Ya con esto tendremos 3 instancias de MongoDB corriendo en nuestro servidor.

Cuando configuramos un replica set debemos recordar que en un ambiente de producción estarían en distintas máquinas, que tendrían diferentes direcciones ip y por eso debemos garantizar que estas se puedan comunicar entre sí. Para ello debemos agregar el parámetro **–bind_ip.** Por defecto MongoDB direcciona al localhost en busca de esas **replica set,** y esto lo deberíamos hacer en cada una de nuestras instancias, por ejemplo:

```
mongod --bind_ip localhost,192.51.100.1 --replSet ejemploReplica
    --dbpath  c:/data/rs1 --port 27017  --oplogSize 200
```

Ya teniendo nuestras **replica set** activas, podremos revisar el estatus de estas con el método **rs.status(),** el cual nos proporcionará alguna información sobre el estado de nuestra replica y otras cuestiones que es este momento no son necesarias tomar en cuenta. De momento solo observaremos la sección de miembros, donde están los datos tales como nombre del nodo, el estado y varias informaciones importantes.

```
        "members" : [
                {
                        "_id" : 0,
                        "name" : "localhost:27018",
                        "health" : 1,
                        "state" : 1,
                        "stateStr" : "PRIMARY",
                        "uptime" : 825,
                        "optime" : {
                                "ts" : Timestamp(1584222637, 1),
                                "t" : NumberLong(1)
                        },
                        "optimeDate" : ISODate("2020-03-
14T21:50:37Z"),
                        "syncingTo" : "",
                        "syncSourceHost" : "",
                        "syncSourceId" : -1,
                        "infoMessage" : "",
                        "electionTime" : Timestamp(1584222066, 1),
                        "electionDate" : ISODate("2020-03-
14T21:41:06Z"),
                        "configVersion" : 1,
                        "self" : true,
                        "lastHeartbeatMessage" : ""
                },
                {
                        "_id" : 1,
                        "name" : "localhost:27019",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 585,
                        "optime" : {
                                "ts" : Timestamp(1584222637, 1),
                                "t" : NumberLong(1)
                        },
                        "optimeDurable" : {
                                "ts" : Timestamp(1584222637, 1),
                                "t" : NumberLong(1)
                        },
                        "optimeDate" : ISODate("2020-03-
14T21:50:37Z"),
                        "optimeDurableDate" : ISODate("2020-03-
14T21:50:37Z"),
                        "lastHeartbeat" : ISODate("2020-03-
14T21:50:40.178Z"),
                        "lastHeartbeatRecv" : ISODate("2020-03-
14T21:50:39.917Z"),
                        "pingMs" : NumberLong(0),
                        "lastHeartbeatMessage" : "",
                        "syncingTo" : "localhost:27018",
                        "syncSourceHost" : "localhost:27018",
```

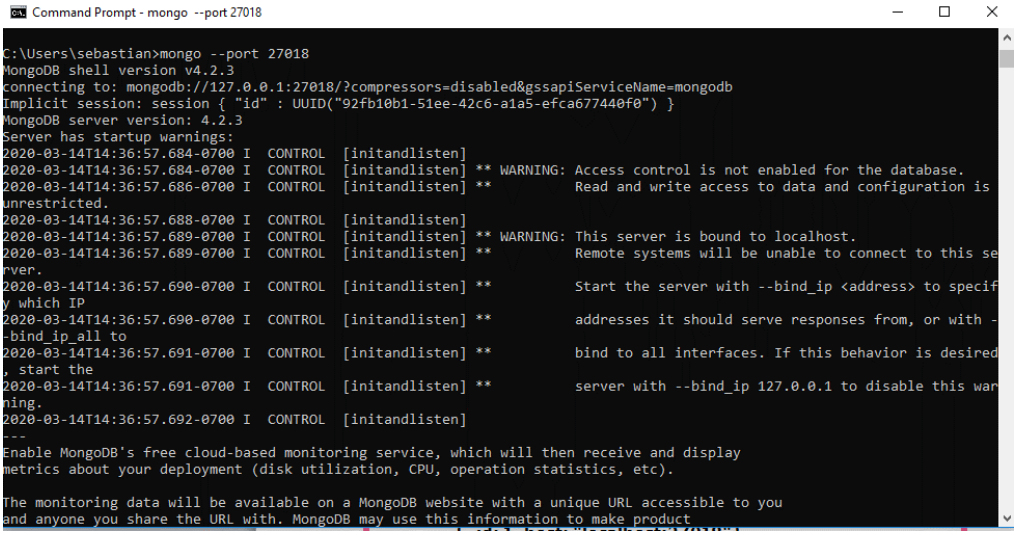
```
                        "syncSourceId" : 0,
                        "infoMessage" : "",
                        "configVersion" : 1
                },
                {
                        "_id" : 2,
                        "name" : "localhost:27020",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 585,
                        "optime" : {
                                "ts" : Timestamp(1584222637, 1),
                                "t" : NumberLong(1)
                        },
                        "optimeDurable" : {
                                "ts" : Timestamp(1584222637, 1),
                                "t" : NumberLong(1)
                        },
                        "optimeDate" : ISODate("2020-03-
14T21:50:37Z"),
                        "optimeDurableDate" : ISODate("2020-03-
14T21:50:37Z"),
                        "lastHeartbeat" : ISODate("2020-03-
14T21:50:40.178Z"),
                        "lastHeartbeatRecv" : ISODate("2020-03-
14T21:50:40.600Z"),
                        "pingMs" : NumberLong(0),
                        "lastHeartbeatMessage" : "",
                        "syncingTo" : "localhost:27019",
                        "syncSourceHost" : "localhost:27019",
                        "syncSourceId" : 1,
                        "infoMessage" : "",
                        "configVersion" : 1
                }
        ],
```

EJEMPLO PRÁCTICO

Continuemos con nuestro ejemplo donde ya hemos creado las 3 instancias o nodos. Ahora debemos configurar cada uno de estos para que funcione como deseamos.

Ahora debemos abrir una cuarta consola con la cual nos conectaremos a una de las otras tres instancias de MongoDB y haremos esto especificando el puerto con el cual nos queremos conectar:

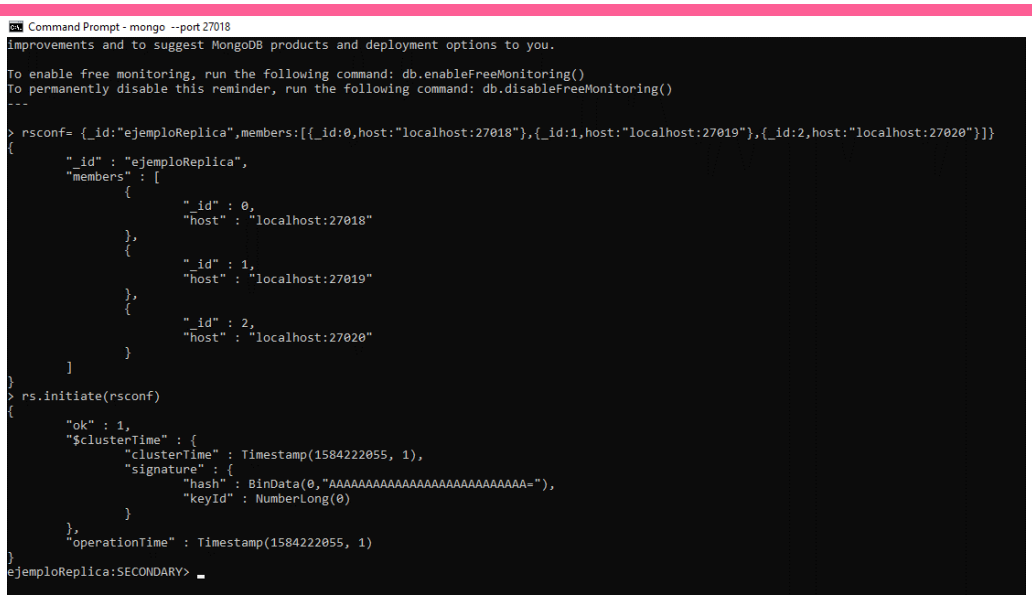
mongo –port 27018



Desde esta mongo Shell realizaremos la configuración de nuestro replica set a través de un **documento de configuración,** que luego se lo pasaremos al método **rs.initiate(),** el cual es un helper para hacer esta configuración:

```
rsconf = {
    _id: " ejemploReplica ",
    members: [
      {_id: 0, host: "localhost:27017"},
      {_id: 1, host: "localhost:27018"},
      {_id: 2, host: "localhost:27019"}
    ]
  }

rs.initiate(rsconf)
{ "ok" : 1, "operationTime" : Timestamp(1501186502, 1) }
```



Como podemos observar, el documento de configuración posee varios elementos. Tenemos el "_id", el cual debe coincidir exactamente con el nombre que hemos escrito cuando se lanzaron las réplicas anteriormente. En este caso es **mdbEjemplo.** El siguiente atributo es un array con los miembros que pertenecerán a este replica set, uno será un **_id** único y el otro elemento será la dirección del host y el puerto.

Después, cuando ejecutamos el método, **rs.initiate()** parseará el documento y avisará a todos los miembros del replica set. Se cargará la información y posteriormente, de forma automática, se elegirá un **nodo primario.** Se comenzará automáticamente a realizar todas las operaciones de escritura en todos los nodos.

Ya con esto tenemos la configuración básica de un sistema de replicación. Como podemos observar, debemos conectarnos al nodo primario como comprobamos en la salida de rs.status(), la que está corriendo en el puerto **27018** que es la primaria.

```
mongo port-27018
```

1. Tras conectarnos veremos que la consola indica: PRIMARY

```
ejemploReplica:PRIMARY>
```

2. Seleccionemos una **collection** de pruebas:`use prueba`

3. Ahora, insertemos unos documentos de prueba: `for (i=0; i<1000; i++) {db.coll.insert({count: i})}`

4. Y revisemos si estos se insertaron correctamente: `db.coll.count()`

```
ejemploReplica:PRIMARY> use prueba
switched to db prueba
ejemploReplica:PRIMARY> for (i=0; i<1000; i++) {db.coll.insert({count: i})}
WriteResult({ "nInserted" : 1 })
ejemploReplica:PRIMARY> db.coll.count()
1000
ejemploReplica:PRIMARY>
```

5. Tras esto usaremos el método: `db.isMaster()` para comprobar que el nodo al cual estamos conectados es el maestro.

```
ejemploReplica:PRIMARY> db.isMaster()
{
        "hosts" : [
                "localhost:27018",
                "localhost:27019",
                "localhost:27020"
        ],
        "setName" : "ejemploReplica",
        "setVersion" : 1,
        "ismaster" : true,
        "secondary" : false,
        "primary" : "localhost:27018",
        "me" : "localhost:27018",
        "electionId" : ObjectId("7fffffff0000000000000001"),
        "lastWrite" : {
                "opTime" : {
                        "ts" : Timestamp(1584224357, 1),
                        "t" : NumberLong(1)
                },
                "lastWriteDate" : ISODate("2020-03-14T22:19:17Z"),
                "majorityOpTime" : {
                        "ts" : Timestamp(1584224357, 1),
                        "t" : NumberLong(1)
                },
                "majorityWriteDate" : ISODate("2020-03-14T22:19:17Z")
        },
        "maxBsonObjectSize" : 16777216,
        "maxMessageSizeBytes" : 48000000,
        "maxWriteBatchSize" : 100000,
        "localTime" : ISODate("2020-03-14T22:19:23.012Z"),
        "logicalSessionTimeoutMinutes" : 30,
        "connectionId" : 24,
        "minWireVersion" : 0,
        "maxWireVersion" : 8,
        "readOnly" : false,
        "ok" : 1,
        "$clusterTime" : {
                "clusterTime" : Timestamp(1584224357, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        },
```

Fuente: Elaboración propia

6. Como podemos observar en la primera parte de la salida, el atributo **isMaster** es igual a **true,** eso quiere decir que nuestro nodo es el primario. Ahora para revisar los datos en los otros nodos abriremos una conexión directamente desde esta consola.

7. Crearemos un objeto tipo mongo al cual le pasaremos como parámetro el host al que nos queremos conectar:

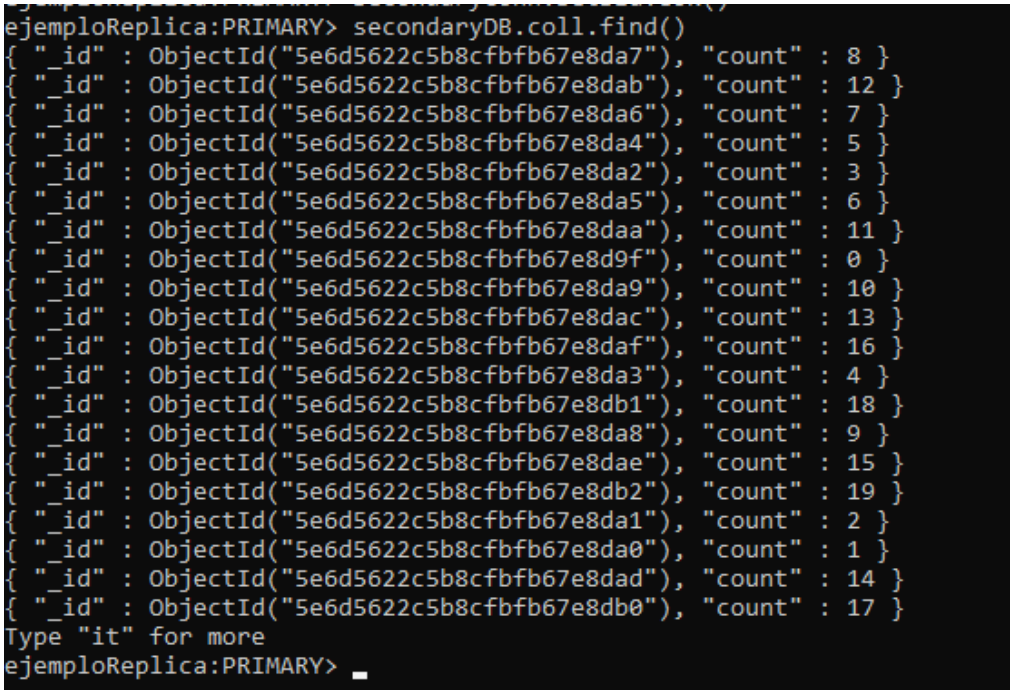
```
secondaryConn = new Mongo("localhost:27019");
```

8. Ahora obtendremos la base de datos con el siguiente método y lo guardaremos en una variable:

```
secondaryDB = secondaryConn.getDB("prueba");
```

9. Si todo está correcto, ya tendremos una instancia de la base de datos en la variable **secondaryDB.**

10. Para permitir la lectura de datos en esta base de datos debemos usar el método: `secondaryConn.setSlaveOk()` Ahora podemos intentar una consulta sobre la base de datos secundaria: `secondaryDB.coll.find()`



```
ejemploReplica:PRIMARY> db.isMaster()
{
        "hosts" : [
                "localhost:27018",
                "localhost:27019",
                "localhost:27020"
        ],
        "setName" : "ejemploReplica",
        "setVersion" : 1,
        "ismaster" : true,
        "secondary" : false,
        "primary" : "localhost:27018",
        "me" : "localhost:27018",
        "electionId" : ObjectId("7fffffff0000000000000001"),
        "lastWrite" : {
                "opTime" : {
                        "ts" : Timestamp(1584224357, 1),
                        "t" : NumberLong(1)
                },
                "lastWriteDate" : ISODate("2020-03-14T22:19:17Z"),
                "majorityOpTime" : {
                        "ts" : Timestamp(1584224357, 1),
                        "t" : NumberLong(1)
                },
                "majorityWriteDate" : ISODate("2020-03-14T22:19:17Z")
        },
        "maxBsonObjectSize" : 16777216,
        "maxMessageSizeBytes" : 48000000,
        "maxWriteBatchSize" : 100000,
        "localTime" : ISODate("2020-03-14T22:19:23.012Z"),
        "logicalSessionTimeoutMinutes" : 30,
        "connectionId" : 24,
        "minWireVersion" : 0,
        "maxWireVersion" : 8,
        "readOnly" : false,
        "ok" : 1,
        "$clusterTime" : {
                "clusterTime" : Timestamp(1584224357, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        },
```

Fuente: Elaboración propia

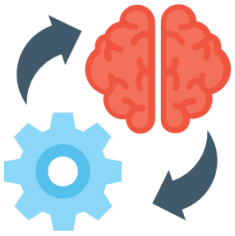
Como podemos ver, están los datos y nuestra replica funciona adecuadamente.

Ya con esto hemos cubierto lo básico de una configuración de replicación para cualquier necesidad que tengamos al respecto.

ENLACE DE INTERÉS

Para saber más sobre la replicación en MongoDb puedes visitar el siguiente enlace:

https://docs.mongodb.com/manual/replication/



RECUERDA

En cualquier momento, si por alguna razón, el nodo en el que estamos trabajando deja de ser maestro, podemos usar el método db.isMaster() para averiguarlo.

5.11.2 Sharding cluster

Sharding se traduce como particionado o fragmentación. En MongoDB consiste en dividir los datos en varios ordenadores. Albergando un fragmento de datos en cada máquina, podremos almacenar mayor cantidad de datos y gestionar mayores cargas sin que sea necesario cambiar a un hardware más potente. Este particionado también se puede utilizar para hacer que los datos más solicitados estén en el hardware con más recursos de nuestro particionado.

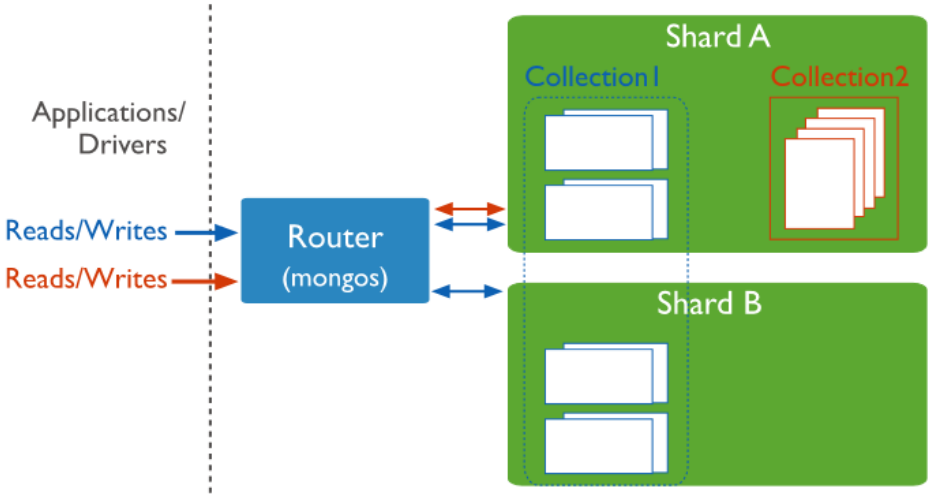
MongoDB soporta autosharding, lo que significa que existe una capa de abstracción la cual será transparente para la aplicación y no tendremos que lidiar manualmente con esto.

El sharding o particionado es de las configuraciones más complejas de MongoDB, desde el punto de vista operacional y de desarrollo. Para hacerlo correctamente debemos tener claro el uso de las réplicas y de MongoDB en general.

Primero revisemos los componentes de un cluster.

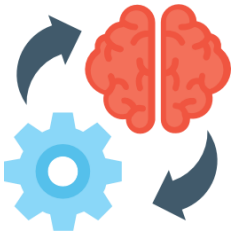
Para empezar, recordemos que cada particionado nos permite crear un cluster en distintas maquinas, esto hará que tengamos una porción de nuestra base de datos en cada una de estas, a diferencia de la réplica que son copias exactas. La meta del particionado será, por tanto, tener un cluster en cada máquina y que puedan ser tratados como si estuviera todo en una sola. Para poder lograr esto existen unos procesos de ruteo llamados **mongos**, donde cada uno de estos mongos tendrá -digamos- una "tabla de contenidos", que nos dirá que partición contiene cada dato.

Nuestra aplicación se conectará a un solo servicio de mongod, pero por debajo existirá un router que se encargará de saber a dónde va cada petición y se encargará de abstraer todo el trabajo difícil.



Conectarse a un clúster fragmentado
Fuente: http://informatica.gonzalonazareno.org/proyectos/2016-17/Cluster_Sharding_MongoDB_Carlos_Garcia.pdf

Ahora vamos a hacer pruebas con sharding o particionado para lo cual disponemos en MongoDB de una clase llamada **ShardingTest**



RECUERDA

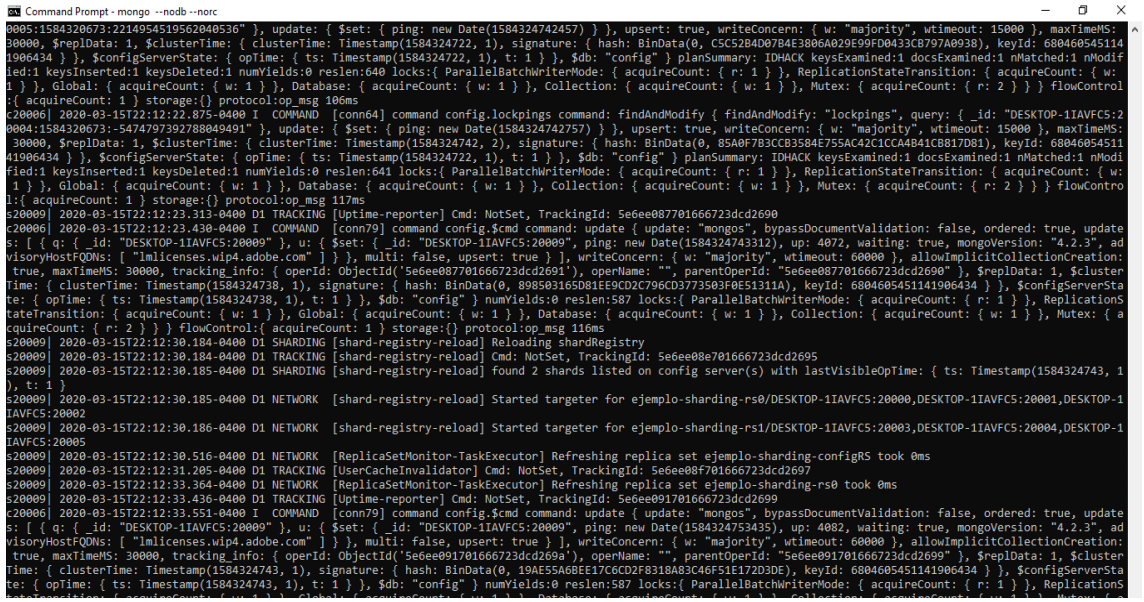
ShardingTest es una clase desarrollada para pruebas por los desarrolladores de MongoDB, por lo tanto, no tiene documentación para usuarios

5.11.3 ¿Cómo interactuaremos con este particionado?

Hagamos un particionado en una sola máquina para lo cual, primero abramos una consola de Mongo mediante el siguiente comando: **`mongo --nodb –norc`**

Ahora creamos una instancia de la clase **ShardingTest()**

```
st = ShardingTest({
    name: "ejemplo-sharding",
    chunkSize: 1,
    shards: 2,
    rs: {
        nodes: 3,
        oplogSize: 10,
    },
    other: {
        enableBalancer: true,
    },
});
```




```
ejemploReplica:PRIMARY> db.isMaster()
{
        "hosts" : [
                "localhost:27018",
                "localhost:27019",
                "localhost:27020"
        ],
        "setName" : "ejemploReplica",
        "setVersion" : 1,
        "ismaster" : true,
        "secondary" : false,
        "primary" : "localhost:27018",
        "me" : "localhost:27018",
        "electionId" : ObjectId("7fffffff0000000000000001"),
        "lastWrite" : {
                "opTime" : {
                        "ts" : Timestamp(1584224357, 1),
                        "t" : NumberLong(1)
                },
                "lastWriteDate" : ISODate("2020-03-14T22:19:17Z"),
                "majorityOpTime" : {
                        "ts" : Timestamp(1584224357, 1),
                        "t" : NumberLong(1)
                },
                "majorityWriteDate" : ISODate("2020-03-14T22:19:17Z")
        },
        "maxBsonObjectSize" : 16777216,
        "maxMessageSizeBytes" : 48000000,
        "maxWriteBatchSize" : 100000,
        "localTime" : ISODate("2020-03-14T22:19:23.012Z"),
        "logicalSessionTimeoutMinutes" : 30,
        "connectionId" : 24,
        "minWireVersion" : 0,
        "maxWireVersion" : 8,
        "readOnly" : false,
        "ok" : 1,
        "$clusterTime" : {
                "clusterTime" : Timestamp(1584224357, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        },
```

Fuente: Elaboración propia

Como vemos en la configuración, este **shard** estará compuesto por dos y el parámetro **rs** define que tendremos una **réplica** de 3 nodos, con una **oplogSize** de **10Mib**.

Con esta configuración es posible ejecutar un **mongod** independiente para cada **shard** o **partición**, lo que nos mostrará como como se presenta una arquitectura típica de un **sharded cluster**.

En el último parámetro le estamos indicando que habilitemos un balance y, por ende, todos los datos se distribuirán a lo largo de los **shards.**
Después de que tengamos nuestro servidor corriendo, ya nos podemos conectar:

1. Abrimos una nueva consola con el siguiente comando **`mongo –nodb`**. A continuación, debemos crear una nueva instancia de mongo de la siguiente manera:

```
db = new Mongo("localhost:20009").getDB("cuentas");
```

```
C:\Users\sebastian>mongo --nodb
MongoDB shell version v4.2.3
> db = (new Mongo("localhost:20009")).getDB("cuentas")
cuentas
mongos>
```

Fuente: Elaboración propia

```
ejemploReplica:PRIMARY> db.isMaster()
{
        "hosts" : [
                "localhost:27018",
                "localhost:27019",
                "localhost:27020"
        ],
        "setName" : "ejemploReplica",
        "setVersion" : 1,
        "ismaster" : true,
        "secondary" : false,
        "primary" : "localhost:27018",
        "me" : "localhost:27018",
        "electionId" : ObjectId("7fffffff0000000000000001"),
        "lastWrite" : {
                "opTime" : {
                        "ts" : Timestamp(1584224357, 1),
                        "t" : NumberLong(1)
                },
                "lastWriteDate" : ISODate("2020-03-14T22:19:17Z"),
                "majorityOpTime" : {
                        "ts" : Timestamp(1584224357, 1),
                        "t" : NumberLong(1)
                },
                "majorityWriteDate" : ISODate("2020-03-14T22:19:17Z")
        },
        "maxBsonObjectSize" : 16777216,
        "maxMessageSizeBytes" : 48000000,
        "maxWriteBatchSize" : 100000,
        "localTime" : ISODate("2020-03-14T22:19:23.012Z"),
        "logicalSessionTimeoutMinutes" : 30,
        "connectionId" : 24,
        "minWireVersion" : 0,
        "maxWireVersion" : 8,
        "readOnly" : false,
        "ok" : 1,
        "$clusterTime" : {
                "clusterTime" : Timestamp(1584224357, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        },
```

Fuente: Elaboración propia

2. Ahora, si nos fijamos, nuestra consola ha cambiado a **mongos** por lo que podremos pasarle peticiones normalmente como si fuese una base de datos al uso. Probemos a insertar unos registros para ver cómo se comporta:

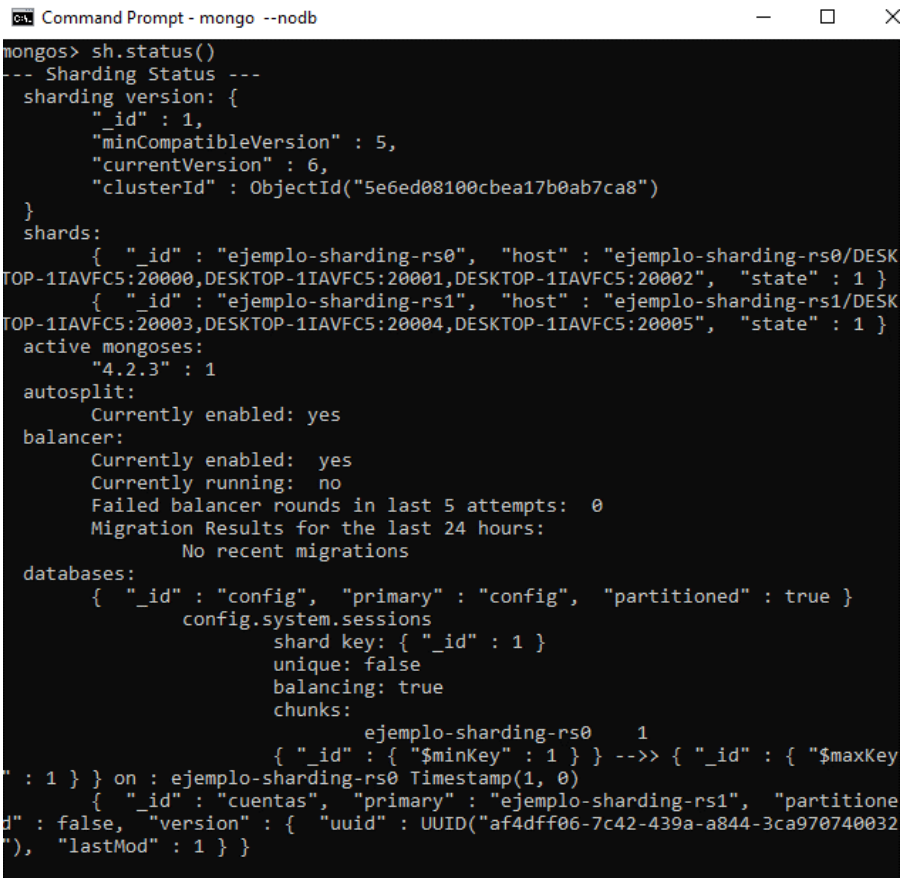
```
for (var i = 0; i < 100000; i++) {
    db.usuarios.insert({ nombre: "usuario" + i, created_at: new
Date() });
}
```

3. Tras ejecutar este script, podemos ver en nuestra otra consola como el log del servidor registra los inserts:

```
0000000000), keyId: 0 } }, $db: "cuentas" } nShards:1 ninserted:1 numYields:0 reslen:170 protocol:op_msg 0ms
s20009| 2020-03-15T22:30:50.054-0400 I  COMMAND  [conn23] command cuentas.usuarios appName: "MongoDB Shell" command: insert { insert: "usuarios", ordered: true, lsid
 id: UUID("a05753c3-f1af-45c8-8642-6e09be378a35") }, $clusterTime: { clusterTime: Timestamp(1584325850, 19), signature: { hash: BinData(0,
0000000000), keyId: 0 } }, $db: "cuentas" } nShards:1 ninserted:1 numYields:0 reslen:170 protocol:op_msg 0ms
s20009| 2020-03-15T22:30:50.056-0400 I  COMMAND  [conn23] command cuentas.usuarios appName: "MongoDB Shell" command: insert { insert: "usuarios", ordered: true, lsid
 id: UUID("a05753c3-f1af-45c8-8642-6e09be378a35") }, $clusterTime: { clusterTime: Timestamp(1584325850, 20), signature: { hash: BinData(0,
```

Fuente: Elaboración propia

4. Para ver un resumen de nuestro shard, podemos usar el siguiente comando: **`sh.status()`**



Fuente: Elaboración propia

5. Para obtener mas informacion de como usar **sh** podemos utilizar el comando
`sh.help()`. Ahora hemos configurado basicamente un cluster, pero toda nuestra informacion por defecto se almacena en un solo shard. Para poder distribuir nuestros datos debemos especificar a MongoDB como distribuirá todo. Para ello, en primer lugar debemos habilitar el sharding en nuestra base de datos:

`sh.enableSharding("cuentas")`

```
mongos> sh.enableSharding("cuentas")
{
        "ok" : 1,
        "operationTime" : Timestamp(1584379897, 4),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1584379897, 4),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
```

Fuente: Elaboración propia

6. Después crearemos un indice en la collection que queremos particionar con el siguiente comando:

```
db.usuarios.createIndex({ nombre: 1 });
```

```
mongos> db.usuarios.createIndex({nombre:1})
{
        "raw" : {
                "ejemplo-sharding-rs1/DESKTOP-1IAVFC5:20003,DESKTOP-1IAVFC5:20004,DESKTOP-1IAVFC5:20005" : {
                        "createdCollectionAutomatically" : false,
                        "numIndexesBefore" : 1,
                        "numIndexesAfter" : 2,
                        "ok" : 1
                }
        },
        "ok" : 1,
        "operationTime" : Timestamp(1584379746, 1),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1584379746, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
```

Fuente: Elaboración propia

7. Entonces debemos llamar al siguiente metodo:

```
sh.shardCollection("cuentas.usuarios",{nombre:1})
```

```
mongos> sh.shardCollection("cuentas.usuarios",{nombre:1})
{
        "collectionsharded" : "cuentas.usuarios",
        "collectionUUID" : UUID("233c525c-34d2-4533-a94b-2c91b34080bb"),
        "ok" : 1,
        "operationTime" : Timestamp(1584379902, 5),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1584379902, 5),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
```

Fuente: Elaboración propia

8. Con esto habilitamos el particionado para la collection **usuarios.** Veamos como ha quedado nuestro sharding con el siguiente comando: **`sh.status()`**

```
cuentas.usuarios
        shard key: { "nombre" : 1 }
        unique: false
        balancing: true
        chunks:
                ejemplo-sharding-rs0    6
                ejemplo-sharding-rs1    7
        { "nombre" : { "$minKey" : 1 } } -->> { "nombre" : "usuario17146" } on : ejemplo-sharding-rs0 Timestamp(2, 0)
        { "nombre" : "usuario17146" } -->> { "nombre" : "usuario24296" } on : ejemplo-sharding-rs0 Timestamp(3, 0)
        { "nombre" : "usuario24296" } -->> { "nombre" : "usuario31445" } on : ejemplo-sharding-rs0 Timestamp(4, 0)
        { "nombre" : "usuario31445" } -->> { "nombre" : "usuario38596" } on : ejemplo-sharding-rs0 Timestamp(5, 0)
        { "nombre" : "usuario38596" } -->> { "nombre" : "usuario45745" } on : ejemplo-sharding-rs0 Timestamp(6, 0)
        { "nombre" : "usuario45745" } -->> { "nombre" : "usuario52895" } on : ejemplo-sharding-rs0 Timestamp(7, 0)
        { "nombre" : "usuario52895" } -->> { "nombre" : "usuario60043" } on : ejemplo-sharding-rs1 Timestamp(7, 1)
        { "nombre" : "usuario60043" } -->> { "nombre" : "usuario67194" } on : ejemplo-sharding-rs1 Timestamp(1, 7)
        { "nombre" : "usuario67194" } -->> { "nombre" : "usuario74343" } on : ejemplo-sharding-rs1 Timestamp(1, 8)
        { "nombre" : "usuario74343" } -->> { "nombre" : "usuario81493" } on : ejemplo-sharding-rs1 Timestamp(1, 9)
        { "nombre" : "usuario81493" } -->> { "nombre" : "usuario88643" } on : ejemplo-sharding-rs1 Timestamp(1, 10)
        { "nombre" : "usuario88643" } -->> { "nombre" : "usuario95793" } on : ejemplo-sharding-rs1 Timestamp(1, 11)
        { "nombre" : "usuario95793" } -->> { "nombre" : { "$maxKey" : 1 } } on : ejemplo-sharding-rs1 Timestamp(1, 12)
```

Fuente: Elaboración propia

9. A diferencia de la primera vez que ejecutamos este comando, ahora vemos que nuestros datos están divididos en porciones (**chunks**) y organizados por el índice que creamos. Cada chunk almacena desde un mínimo hasta un máximo del índice.

   Sería algo así:

USUARIO 0			**USUARIO 400**
Usuario0... Usuario100	Usuario100... Usuario200	Usuario200... Usuario300	Usuario300... Usuario400

10. Es decir, nuestros datos han sido distribuidos a lo largo de los distintos pedazos o **chunks**.

    Cualquier búsqueda que hagamos se realizará sobre el chunk que necesitemos, a menos que queramos buscar todos los datos, por ejemplo:

Fuente: Elaboración propia

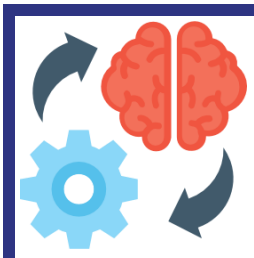
11. Vemos que se hace un **SHARD_MERGE.** Como hemos podido observar, el trabajo con particionado es complejo y necesita estudio y práctica.
    Ahora, cuando terminemos de experimentar, para cerrar nuestra consola de manera limpia volvamos a la primera consola y presionamos enter unas cuantas veces y escribimos:

    ```
    st.stop() + enter.
    ```

    Esto hará que todos los servicios se detengan. Luego bastará con introducir el comando **exit + enter.**

```
c20008| 2020-03-16T11:19:02.725-0700 I  CONTROL  [eventTerminate] now exiting
c20008| 2020-03-16T11:19:02.725-0700 I  CONTROL  [eventTerminate] shutting down with code:0
2020-03-16T14:19:02.736-0400 I  -        [js] shell: stopped mongo program on port 20008
ReplSetTest stop *** Mongod in port 20008 shutdown with code (0) ***
ReplSetTest stopSet stopped all replica set nodes.
ReplSetTest stopSet deleting all dbpaths
ReplSetTest stopSet deleting dbpath: /data/db/ejemplo-sharding-configRS-0
ReplSetTest stopSet deleting dbpath: /data/db/ejemplo-sharding-configRS-1
ReplSetTest stopSet deleting dbpath: /data/db/ejemplo-sharding-configRS-2
ReplSetTest stopSet deleted all dbpaths
2020-03-16T14:19:03.373-0400 I  NETWORK  [js] Removed ReplicaSetMonitor for replica set ejer
ReplSetTest stopSet *** Shut down repl set - test worked ****
ShardingTest stop deleting all dbpaths
*** ShardingTest ejemplo-sharding completed successfully in 14526.894 seconds ***
>
```

Fuente: Elaboración propia



RECUERDA

Para que este ShardedTest funcione, debe existir el directorio /data/db en la raíz de nuestro sistema.

5.12 Análisis de rendimiento

Durante el proceso de análisis previo a la construcción del sistema, además de una evaluación técnica, también es necesario estimar y anticipar el uso que se va a hacer del sistema para, desde las fases más tempranas de diseño, evitar la aparición de cuellos de botella.

Para un proyecto como este piensas que sería interesante para mejorar el rendimiento global del sistema el poder disponer de almacenamiento de estado sólido.

Probablemente poder desplegar la infraestructura de la biblioteca jurídica en la nube te aportará una serie de ventajas. Por un lado, la reducción de costes y problemas variados que siempre derivan de la administración de un sistema de estas características. Por otra parte, serás capaz de contratar más capacidad puntualmente en aquellos momentos donde el sistema reciba mayores exigencias.

Aparte del hardware, otras cuestiones que pueden determinar el surgimiento de cuellos de botella pueden ser las estrategias de acceso a la base de datos. En principio, dado que el sistema va a operar a nivel global, todo hace prever que será bastante constante el número de accesos por la diferencia horaria entre las jornadas laborales de los diferentes países y continentes.

También será determinante para la formación de cuellos de botella la estrategia de acceso a la base de datos, en este caso, como tienes claro, fundamentalmente de lectura.

Muchas veces podemos tener problemas en nuestras bases de datos, ya sea por un diseño que no se pensó adecuadamente para crecer y, con posterioridad se hace inmanejable, ya que la demanda los supera. En esos casos, un análisis oportuno y bien realizado a tiempo puede ayudarnos a tomar medidas preventivas o buscar soluciones de una manera inteligente.

Todas las bases de datos tienen debilidades, sean cual sean las razones. Aprender a detectar estos fallos es un trabajo que debemos aprender a realizar si queremos que nuestras aplicaciones funcionen de manera óptima.

Debemos tener una herramienta para medir que todo está funcionando lo mejor posible. Cabe destacar que lo mejor es disponer el diseño de nuestra base de datos lo mejor posible, para evitar cualquier tipo de inconveniente en el futuro.

En este apartado estudiaremos como realizar esos análisis sobre el rendimiento de nuestra base de datos, ubicar cuellos de botella, y algunos escenarios típicos, así como las estrategias para atacar estos problemas.

Cuando nos encontramos con una disminución en el rendimiento de nuestra base de datos es el momento para realizar un análisis de rendimiento y ver por qué esto está ocurriendo. Los puntos de error más comunes en los cuales se ve afectado el rendimiento son:

- La disponibilidad del hardware.
- La estrategia de acceso a la base de datos.
- El número de conexiones abiertas a la base de datos.

Todos estos factores son llamados también cuellos de botella y serán las principales causas de un bajo rendimiento en los procesos de nuestra base de datos. Mediante el análisis y las herramientas aquí mencionadas, podremos identificarlos y darles solución.

5.12.1 Búsqueda de cuellos de botella

Las principales **causas de este bajo rendimiento** pueden ser un diseño pobre o la mala implementación de los índices. Otras veces el bajo rendimiento puede ser por debido a alguna actividad inusual, por ejemplo, un incremento el tráfico. En general, un bajo rendimiento nos indica que la base de datos está trabajando al máximo de su capacidad física o lógica.

MongoDB posee un sistema de bloqueo para garantizar la integridad de los datos. Cuando una operación presenta una larga duración, esta bloquea todos los registros involucrados, colocando en cola las demás operaciones, lo que puede conllevar a un tiempo de espera inasumible.

Este tipo de retraso -por lo general- se presenta de manera intermitente. Para hacer un estudio de estas circunstancias, podemos valernos del comando ServerStatus.

Este comando se utiliza como sigue:

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

Esto nos devuelve una salida con una descripción bastante larga y detallada. Para nuestro interés en este momento, nos fijaremos en la sección **GlobalLock** y **lock**

La división de los parámetros **locks.timeAcquiringMicros** con **locks.acquireWaitCount** nos dará un promedio del tiempo de espera.

Si el parámetro **globalLock.currentQueue.total** tiene un valor constantemente alto, existe una posibilidad clara de que haya un número elevado de peticiones esperando por un bloqueo, lo que significa que debemos atacar la concurrencia de nuestra base de datos que está afectando el rendimiento.

Otro parámetro interesante de estudiar es **globalLock.totalTime,** que nos indica el tiempo total de bloqueo, y si su valor es alto en función del **uptime,** quiere decir que la base de datos ha estado en un bloqueo por un tiempo significativo.

El número de conexiones que tiene nuestra base de datos también es un parámetro importante de analizar a la hora de ver el rendimiento, dado que en ocasiones la cantidad de conexiones entre nuestras aplicaciones y nuestro servidor puede generar lentitud, ya que superan la capacidad del servidor.

Para analizar esto tenemos el documento **connections,** dentro de nuestro server status, el cual tiene dos atributos:

- **connections.current:** nos devuelve la cantidad de conexiones en el momento.
- **connections.available:** nos indicará el número de conexiones disponibles para nuevos clientes.

Si tenemos un número muy alto de concurrencia por parte de la aplicación, debemos aumentar la capacidad de nuestro desarrollo. Para las aplicaciones que tienen alta demanda de lecturas, podemos incrementar el número de **réplicas** y distribuir las operaciones de lectura entre los servidores secundarios. Para las aplicaciones que tiene alta demanda de escritura, se recomienda el uso de fragmentación o sharding, para distribuir la carga entre los distintos clusters.

¿SABÍAS QUE…?

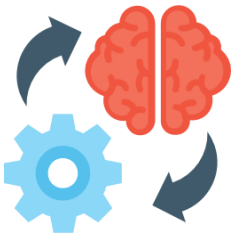
Las consultas largas pueden ser provocadas por deficiencias en el modelado de datos, mal uso de índices, pero también por falta de memoria RAM.

5.12.2 El perfil de la base de datos

MongoDB nos proporciona un perfil de la base de datos recolectando información detallada sobre los comandos realizados respecto de la instancia **mongod.** Esto incluye todas las operaciones de lectura, escritura y eliminaciones. Además de ello, también registra las operaciones administrativas y las almacena en una collection de tipo capped en la base de datos de administración **system.profile.**

La manera de usarlo sería a través del comando:`db.setProfillingLevel()` Este método acepta como parámetro el nivel de profilling como se presenta a continuación.

NIVEL	DESCRIPCIÓN
0	El perfil está desactivado. No se recolecta información y este es el valor por defecto, es decir está desactivado.
1	Se activa el perfil y se recolecta información de aquellas consultas que tarden más que el tiempo en la variable slowms.
2	Se recolecta la información de todas las operaciones.



RECUERDA

Activar el perfilado de la base de datos ocasiona grandes retardos en nuestra base de datos, ya que se debe registrar cada actividad que se realiza generando lentitud.

¿SABÍAS QUE…?

El perfil de la base de datos está desactivado por defecto y este puede ser activado solo para una instancia de mongod.

Por defecto, el valor de la variable slowms es de 100 milisegundos. Para cambiar ese valor tenemos varias opciones:

- Usando un documento de configuración, por ejemplo:
  `db.setProfilingLevel(1, { slowms: 20 })`

- Colocándolo como parámetro –slowms cuando iniciamos nuestra consola.
- Colocando un valor en **slowOpThresholdMs** en un archivo de configuración.

Para deshabilitar el perfil, basta con pasarle el parámetro 0 de la siguiente manera: `db.setProfilingLevel(0)`

1. Activar el perfilador para toda una instancia de MongoDB puede hacerse en entornos de desarrollo con propósitos de pruebas. Esto lo haríamos en el momento de levantar el servidor de la siguiente manera:

   ```
   mongod --profile 1 --slowms 15 --slowOpSampleRate 0.5
   ```

   Ello activará la recolección de información en el perfil de la base de datos en el nivel **1** a transacciones que duren más de 15 milisegundos y la tasa de muestreo es de 0.5.

2. Como hemos dicho antes, toda la información es almacenada en una collection llamada **profile**. Ahora veamos qué información se almacena aquí:

   Este sería un ejemplo de documento de tipo profile:

```
{
   "op" : "query",
   "ns" : "test.report",
   "command" : {
      "find" : "report",
      "filter" : { "a" : { "$lte" : 500 } },
      "lsid" : {
         "id" : UUID("5ccd5b81-b023-41f3-8959-bf99ed696ce9")
      },
      "$db" : "test"
   },
   "cursorid" : 33629063128,
   "keysExamined" : 101,
   "docsExamined" : 101,
   "fromMultiPlanner" : true,
   "numYield" : 2,
   "nreturned" : 101,
   "queryHash" : "811451DD",
   "planCacheKey" : "759981BA",
   "locks" : {
      "Global" : {
         "acquireCount" : {
            "r" : NumberLong(3),
            "w" : NumberLong(3)
         }
      },
      "Database" : {
         "acquireCount" : { "r" : NumberLong(3) },
         "acquireWaitCount" : { "r" : NumberLong(1) },
         "timeAcquiringMicros" : { "r" : NumberLong(69130694) }
      },
      "Collection" : {
         "acquireCount" : { "r" : NumberLong(3) }
      }
   },
   "storage" : {
      "data" : {
         "bytesRead" : NumberLong(14736),
         "timeReadingMicros" : NumberLong(17)
      }
   },
   "responseLength" : 1305014,
   "protocol" : "op_msg",
   "millis" : 69132,
   "planSummary" : "IXSCAN { a: 1, _id: -1 }",
   "execStats" : {
      "stage" : "FETCH",
      "nReturned" : 101,
      "executionTimeMillisEstimate" : 0,
      "works" : 101,
      "advanced" : 101,
      "needTime" : 0,
```

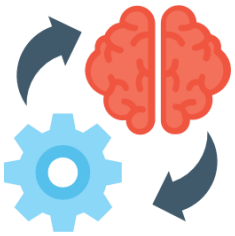
```
            "needYield" : 0,
            "saveState" : 3,
            "restoreState" : 2,
            "isEOF" : 0,
            "docsExamined" : 101,
            "alreadyHasObj" : 0,
            "inputStage" : {
               ...
            }
         },
         "ts" : ISODate("2019-01-14T16:57:33.450Z"),
         "client" : "127.0.0.1",
         "appName" : "MongoDB Shell",
         "allUsers" : [
            {
               "user" : "someuser",
               "db" : "admin"
            }
         ],
         "user" : "someuser@admin"
}
```

3. Como podemos ver, es una salida bastante detallada, en la cual sus elementos o subelementos dependerán del tipo de operación y estos pueden ser:
   - Command.
   - Count.
   - Distinct.
   - Geonear.
   - Getmore.
   - Group.
   - Insert.
   - Mapreduce.
   - Query.
   - Remove.
   - Update.

4. Este valor lo veremos en el atributo **profile.op.** También está **profile.command,** donde tendremos el nombre del comando utilizado con los parámetros usados, por ejemplo:

```
"command" : {
  "find" : "items",
  "filter" : {
    "sku" : 12312
  },
  ...
  "$db" : "peliculas"
}
```

5. Otro parámetro interesante es **profile.keysExamined** el cual nos indica la cantidad de índices examinados para realizar la operación.



RECUERDA

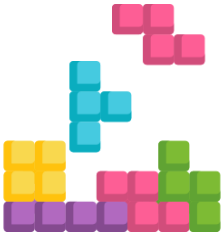
Si el parámetro **profile.keysExamined** es mucho mayor que **nreturned,** se debe considerar ajustar los índices para mayor rendimiento.

ENLACE DE INTERÉS

El parámetro **profile.docsExamined** será la cantidad de documentos examinados para llevar a cabo la operación solicitada.

Además de estos existen muchos más y los podemos ver en la siguiente dirección:

https://docs.mongodb.com/manual/reference/database-profiler/



EJEMPLO PRÁCTICO

El éxito que está teniendo la empresa está generando que cada mes se incremente un 25% el número de usuarios activos que se conectan al sistema.
La parte negativa es que el departamento de atención al cliente está recibiendo un número demasiado alto de quejas de usuarios que dicen percibir una pérdida de calidad, sobre todo por un incremento en los tiempos de respuesta lo que provoque que sean menos productivos a la hora de localizar la ley o la sentencia judicial que están buscando en cada momento

Son varias las herramientas que un administrador de MongoDb debe conocer y saber cómo usarlas para detectar cuellos de botellas. Aunque el diagnóstico parece claro y los síntomas son achacables al mayor número de usuarios conectados, no hay que descartar ninguna posibilidad.

Por este motivo, debemos analizar en detalle la situación para determinar cómo atacar algunos de los escenarios más comunes, usando las siguientes técnicas.

¿Tengo lentitud en mi base de datos?
Lo primero que debemos hacer es realizar un diagnóstico. Para ello podemos utilizar cualquiera de las herramientas mencionadas cuando nos hemos referido a la cuestión del análisis de rendimiento.

¿Tenemos sospecha de que se deba a una consulta?
Debemos entonces revisar el estatus del servidor y ver los tiempos de bloqueo debido a consultas

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

Este comando nos dará la información necesaria.
Hemos detectado que se trata de una consulta y ahora debemos optimizar dicha consulta. Para esto podemos utilizar el comando **explain** en la consulta y ahí podemos ver si necesitamos un índice, o tal vez optimizar nuestro documento de alguna manera.

¿Tengo lentitud y sospecho que puede ser por un incremento de tráfico en nuestro sistema?

En este caso también debemos revisar en el estatus de nuestro servidor, donde veremos las variables del documento **connections** y los elementos **current** y **available.** En este caso es posible comprobar si de verdad la demanda de conexiones es mayor a la disponibilidad.

También es factible dependiendo del caso, definir una fragmentación o usar mayor cantidad de réplicas.

¿Mi aplicación tiene alta demanda de escritura de datos nuevos?

En este caso usaremos **fragmentación** para dividir las cargas.

¿Mi aplicación tiene alta demanda de lectura de datos?
En este caso incrementaremos el número de **réplicas.**

Estos serían los casos más comunes y se podrían resolver solamente revisando el estatus del servidor. Ahora, si no sabemos qué tipo de operación está haciendo que nuestro servidor se comporte de forma lenta, podríamos intentar hacer un perfil de la **base de datos**.

Con el perfil podremos descubrir qué tipo de operación está tomando mayor cantidad de tiempo y se trataría de un estudio más profundo del funcionamiento de la base de datos.

VIDEOTUTORIAL

En el siguiente vídeo se muestra cómo utilizar el sistema gestor de bases de datos NoSQL MongoDb, desde su instalación hasta su administración.

https://vimeo.com/user64513894/review/488938351/f72e5561ce

RESUMEN FINAL

En esta unidad hemos aprendido qué es MongoDB, qué lo hace distinto a otras bases de datos, cuáles son sus principales virtudes y algunos de sus elementos componentes. Todo ello nos ha preparado para entender el funcionamiento de MongoDB y poder sacarle provecho utilizando completamente su potencial.

Hemos sabido que fue desarrollado en el año 2007 y desde entonces ha causado gran revuelo en el sector tecnológico. Está muy ligado a las nuevas tecnologías y stacks de desarrollo como MEAN y MENV. Actualmente, es una habilidad muy solicitada para los desarrolladores conocer MongoDB.

También hemos realizado una instalación en nuestra maquina local que nos ha servido a lo largo de todo el curso para hacer pruebas y ejemplos con MongoDB. Hemos aprendido el uso básico e intermedio de Mongo Shell y otros aspectos importantes para adentrarnos en la tecnología que nos ofrece MongoDB. Se ha presentado cómo acceder a la ayuda que nos ofrece mongo Shell. También hemos visto cómo usar algunos comandos básicos.

Se ha estudiado como hacer scripts sencillos, que nos harán la vida más cómoda con las tareas administrativas de la base de datos. Se han estudiado los tipos de datos que tenemos disponibles para almacenar nuestra información con una precisión y capacidad asombrosa todo ello aplicado a situaciones reales.

A medida que hemos avanzado en esta unidad, aprendimos las operaciones básicas sobre una base de datos en MongoDB, resumidas en el concepto CRUD (create, read, update y delete). Además, hemos visto un uso más profundo de la consola y cómo se comporta esta.

Se repasaron los distintos métodos y operadores necesarios para hacer que nuestra base de datos se comporte como esperamos. También como buscar la información que queremos y actualizarla. Estudiamos como hacer búsquedas con rangos y muchos otros operadores.

Vimos cómo podemos insertar datos de distintas maneras, también como eliminarlos. Aprendimos que los datos al ser eliminados no se pueden recuperar, a menos que se tenga un backup o una estrategia que lo habilite. Investigamos también la funcionalidad de las collections tipo capped, para qué se usan y cómo crearlas, incluso como convertir una collection normal a una de tipo capped, usando el metodo convertToCapped.

Aprendimos cómo hacer uso de MapReduce, cómo funciona y su principal utilidad y aplicación, que es en el procesamiento de grandes cantidades de datos. Su capacidad de procesamiento paralelo hace de esta herramienta un ayudante ideal para sacar provecho a la tecnología de datos fragmentados, reduciendo así también el tiempo de configuración y de procesamiento y todo ello de manera simple para el usuario.

Además de la funcionalidad de Mongo, conocimos un poco de su arquitectura y cómo funciona internamente. También el origen de su nombre y usos más comunes. Aprendimos como configurar nuestro editor preferido, en nuestra consola de Mongo Shell.

Conocimos que es usado actualmente por grandes empresas como Google, que maneja enormes cantidades de datos. También supimos de la existencia de una implementación de software libre llamada Hadoop, cuyo desarrollo está a cargo del proyecto apache.

Hemos presentado las sugerencias más pertinentes a la hora de modelar datos en MongoDB porque es de vital importancia tener claro muchos de estos conceptos para poder realizar exitosamente la implementación de una base de datos no relacional o, en algunos casos, migrar de un modelo menos eficiente a uno como este.

La reducción de la cantidad de operaciones para lograr los mismos resultados de manera rápida y segura, la división de cargas y el procesamiento en paralelo, son algunas de los aspectos que hacen de MongoDB una de las herramientas con mayor uso en la actualidad para el almacenamiento de datos.