versión: 2023.4
9 de marzo de 2023

Índice

1. Conceptos básicos

1.1. ¿Qué es MongoDB?

- MongoDB es un sistema de base de datos NoSQL multiplataforma de licencia libre (código abierto).
- MongoDB es un sistema de base de datos orientada a documentos: en lugar de guardar los datos en tablas, guarda los datos en documentos.
- Estos documentos son almacenados en BSON, que es una representación binaria de JSON:
  - Para el intercambio de datos para almacenamiento y transferencia de documentos en MongoDB usamos el formato BSON, (Binary JavaScript Object Notation). Se trata de una representación binaria de estructuras de datos y mapas, diseñada para ser más ligero y eficiente que JSON, (JavaScript Object Notation).
- Está orientado a documentos de esquema libre: cada registro puede tener un esquema de datos distinto. Los atributos no tienen que ser iguales en diferentes registros.
- MongoDB está pensado para mejorar la *escalabilidad horizontal*:
  - Es una propiedad deseable de un sistema que indica la capacidad para adaptarse sin perder calidad con el crecimiento del tamaño de la base de datos.
- Cada registro o conjunto de datos se denomina *documento*, que pueden ser agrupados en *colecciones*, (equivalente a las tablas de las bases de datos relacionales pero sin estar sometidos a un esquema fijo).

Servidor
Base de datos
Colección
Documentos
Campos

Estructura de la información en MongoDB.

- Principales herramientas para trabajar con MongoDB:
  1. *Mongod*: Servidor de bases de datos de MongoDB.
  2. *Mongo*: Cliente para la interacción con la base de datos MongoDB.
- Características:
  - Replicación:
    - MongoDB, es más flexible que las bases de datos relacionales, y por ello menos restrictivo, lo que puede presentar en ocasiones problemas de volatilidad.
    - MongoDB manda los documentos escritos a un servidor maestro, que sincronizado a otro u otros servidores mandará esta misma información replicada, a estos "esclavos".
  - Indexación: Cualquier campo en un documento de MongoDB puede ser indexado. Es posible hacer índices secundarios. El concepto de índices en MongoDB es similar al de datos relacionales.
  - Escalabilidad horizontal:
    - Capacidad de trabajar con varias máquinas de manera distribuida, almacenando en cada uno de los nodos cierta información que de una forma u otra debe estar comunicada con el resto de nodos que forman nuestro sistema.
    - Esto dota de mayor flexibilidad al sistema, ya que facilita la agregación de equipos en función de las necesidades.

- Sharding:
  - MongoDB utiliza el Sharding como método para dividir los datos a lo largo de los múltiples servidores de nuestra solución.
  - Las bases de datos relacionales también hacen tareas similares a ésta, si bien de forma diferente.
  - Tal vez el rasgo más destacable en MongoDB porque realiza estas tareas de manera automática.
- Balanceo:
  - El balanceador es un proceso de MongoDB para equilibrar los datos en nuestro sistema.
  - Mueve porciones de datos de un shard a otro, de manera automática.

- Algunos inconvenientes:
  - No implementa las propiedades ACID (atomicidad, consistencia, aislamiento y durabilidad): El no implementar las propiedades ACID genera que la base de datos no asegure la durabilidad, la integridad, la consistencia y el aislamiento requeridos obligatoriamente en las transacciones ("superables" en el futuro).
  - Tiene problemas de rendimiento cuando el volumen de datos supera los 100GB.
  - Otros problemas "menores" achacables a versiones concretas ("superables" en el futuro).
- Relación entre los conceptos de las bases de datos relacionales y MongoDB:

Bases de datos relacionales	MongoDB
Base de datos	Base de datos
Tabla	Colección
Tupla o fila	Documento
Columna	Campo

1.2. Documentos

- Los documentos son la unidad básica de organización de la información en MongoDB, y desempeñan el papel equivalente a una fila en las bases de datos relacionales.
- Un documento es un conjunto ordenado de claves que tienen asociados valores, y que se corresponden con algunas estructuras de datos típicas de los lenguajes de programación tales como tablas hash o diccionarios. En general los documentos contendrán múltiples pares clave-valor como: {`Nombre:Juan,Pais:''España''`}
- Características:
  - Las claves son cadenas, permitiéndose cualquier carácter.
  - Excepciones:
    - La clave no pueden contener el carácter nulo `\0`.
    - El punto (.) y el ($) están prohibidos.
  - MongoDB es sensitivo tanto a las mayúsculas/minúsculas como a los tipos de datos. Así por ejemplo los siguientes documentos se consideran distintos:
    {`Edad:3`} ,{`Edad: ''3''`},{`Edad:3`},{`edad:3`}
  - Los documentos no pueden tener claves duplicadas. Así por ejemplo el siguiente documento es incorrecto: {`edad:3,edad:56`}
  - Los pares clave-valor están ordenados en los documentos. Por ejemplo el documento {`x:3,y:5`} no es lo mismo que {`y:5,x:3`}.
  - Cuestión: ¿este orden es relevante para nosotros?.
  - Los valores de un documento pueden ser de diferentes tipos.

1.3. Tipos de datos

- Principales tipos de datos soportados por los documentos en MongoDB:
  - Nulo: Representa el valor nulo o bien un campo que no existe. Por ejemplo {`x:null`}. Nota: El uso de este valor va en contra de las ideas fundamentales de bases de datos semiestructuradas. Sólo utilizarlo en casos excepcionales.
  - Booleanos: Representa el tipo booleano, el cual puede tomar los valores de `true` o `false`. Por ejemplo {`x:true`}
  - Números: Distingue entre números reales como por ejemplo {`x:3.14`} y números enteros como por ejemplo {`x:45`}
  - Cadenas: Cualquier cadena de caracteres como por ejemplo {`x:''Ejemplo''`}
  - Fechas: Almacena la fecha en milisegundos. Por ejemplo: {`x:new Date()`}
  - Arrays: Se representa como un conjunto o lista de valores. Por ejemplo {`x:[''a'',''b'',''c'']`}
  - Documentos embebidos: Los documentos pueden contener documentos embebidos como valores de un documento padre. Por ejemplo {`x:{y:45}`}
  - Identificadores de objetos: Es un identificador de 12 bytes para un documento. Por ejemplo {`x: OjectId()`}
  - Datos binarios: Es una cadena de bytes arbitraria que no puede ser manipulada directamente desde el Shell y que sirve para representar cadenas de caracteres no UTF8.
  - Código Javascript: Los documentos y las consultas pueden contener código JavaScript. Por ejemplo {`x: function () { ...`}}
  - Fechas:
    - Para crear un objeto de tipo fecha se usa el comando `new Date()`. Sin embargo, si se llama sin *new*(solo *Date()*) entonces se retorna una cadena que representa la fecha. Y por tanto se trata de diferentes tipos de datos.
    - Las fechas en el Shell son mostradas usando la configuración local de la zona horaria, sin embargo la base de datos las almacena como un valor en milisegundos sin referencia a la zona horaria.
  - Sobre los arrays:
    - Pueden ser usados tanto en operaciones en las que el orden es importante tales como listas, pilas o colas como en operaciones en las que el orden no es importante tales como conjuntos.
    - Los arrays pueden contener diferentes tipos de valores como por ejemplo {`Cosas:[''edad'',45]`}. De hecho, soporta cualquiera de los tipos de valores soportados para los documentos, pudiéndose crear arrays anidados.
  - Una propiedad importante en MongoDB es que es reconoce la estructura de los arrays y permite navegar por el interior de los arrays para realizar operaciones sobre sus contenidos como consultas o crear índices sobre sus contenidos.
  - En el ejemplo anterior se podría crear una consulta para recuperar todos aquellos documentos donde 3.14 es un elemento del array "Cosas", y si por ejemplo esta fuera una consulta habitual entonces incluso se podría crear un índice sobre la clave "Cosas" y mejorar el rendimiento de la consulta.
  - Así mismo MongoDB permite realizar actualizaciones que modifican los contenidos de los arrays tales como cambiar un valor del array por otro.
- Documentos embebidos:
  - Los documentos pueden ser usados como valores de una clave, y en este caso se denominan "documentos embebidos". Se suelen usar para organizar los datos de una manera lo más natural posible.

- Por ejemplo si se tiene un documento que representa a una persona y se quiere almacenar su dirección podría crearse anidando un documento "dirección" al documento asociado a una persona como por ejemplo:

```
{
  nombre: "Juan",
  dirección: {
                calle:"Mayor,3",
                ciudad:"Madrid",
                 Pais:"España"
}}
```

- MongoDB es capaz de navegar por la estructura de los documentos embebidos y realizar operaciones con sus valores como por ejemplo crear índices, consultas o actualizaciones.

- Identificador de objetos:
  - Cada documento tiene que tener un clave denominada `_id`.
  - El valor de esta clave puede ser de cualquier tipo pero por defecto será de tipo *ObjectId*.
  - En una colección cada documento debe tener un valor único y no repetido para la clave `_id`, lo que asegura que cada documento en la colección pueda ser identificado de manera única.
  - Así por ejemplo dos colecciones podrían tener un documento con `_id` con el valor 123, pero en una misma colección no podría haber dos documentos con valor de `_id` de 123.
  - El tipo *ObjectId* es el tipo por defecto para los valores asociados a la clave `_id`. Es un tipo de datos diseñado para ser usado en ambientes distribuidos de manera que permita disponer de valores que sean únicos globalmente.
  - Cada valor usa 12 bytes lo que permite representar una cadena de 24 dígitos hexadecimales(2 digitos por cada byte). Si se crean múltiples valores del tipo *ObjectId* sucesivamente sólo cambian unos pocos dígitos del final y una pareja de dígitos de la mitad. Esto se debe a la forma en la que se crean los valores del tipo *ObjectIds*.
  - Observaciones:
    - Los primeros 4 bytes son un marca de tiempo en segundos que combinados con los siguientes 4 bytes proporciona unicidad a nivel de segundo y que identifican de manera implícita cuando el documento fue creado.
    - Por otro lado, a causa de que la marca de tiempo aparece en primer lugar, entonces los *ObjectIds* se ordenan obligatoriamente en orden de inserción lo que hace que la indexación sobre *ObjectIds* sea eficiente.
    - Los siguientes 3 bytes son un identificador único de la máquina que lo genera, lo que garantiza que diferentes máquinas no generan colisiones.
    - Para conseguir unicidad entre diferentes procesos que generan *ObjectIds* concurrentemente en una misma máquina se usan los siguientes 2 bytes que son tomados del identificador del proceso que genera un *ObjectID*.
  - Cuando un documento se va a insertar si no tiene un valor para la clave `_id` entonces es generado automáticamente por MongoDB.

1.4. Colecciones

- Una colección es un grupo de documentos, y desempeña el papel análogo a las tablas en las bases de datos relacionales.
- Las colecciones tienen esquemas dinámicos lo que significa que dentro de una colección puede haber cualquier número de documentos con diferentes estructuras.
- Por ejemplo en una misma colección podrían estar los siguientes documentos diferentes: {edad:34},{x:"casa"} que tienen diferentes claves y diferentes tipos de valores.

- Dado que cualquier documento se puede poner en cualquier colección y dado que no es necesario disponer de esquemas distintos para los diferentes tipos de documentos, entonces surge la pregunta de por qué se necesita usar más de una colección y tener que separar los documentos mediante colecciones separadas:
  - Cuando se crean índices se impone cierta estructura a los documentos (especialmente en los índices únicos). Estos índices están definidos por colección de forma que poniendo documentos de un solo tipo en la misma colección entonces se podrán indexar las colecciones de una forma más eficiente.
  - En general, por eficiencia, es razonable crear un esquema y agrupar los tipos relacionados de documentos juntos aunque MongoDB no lo imponga como obligatorio.
- Una colección se identifica por su nombre, el cual es una cadena con las siguientes restricciones:
  - La cadena vacía no es un nombre válido para una colección.
  - Los nombres de las colecciones no pueden contener el carácter nulo `\0` pues este símbolo se usa para indicar el fin del nombre de una colección.
  - No se debe crear ninguna colección que empiece con "system" dado que es un prefijo reservado para las colecciones internas. Por ejemplo, la colección *system.users* contiene los usuarios de la base de datos, la colección *system.namespaces* contiene información acerca de todas las colecciones de la base de datos.
  - Las colecciones creadas por los usuarios no deben contener el carácter reservado $ en su nombre.
- Una convención para organizar las colecciones consiste en definir subcolecciones usando espacios de nombres separados por el carácter ".".
- Por ejemplo, una aplicación que contuviese un blog podría tener una colección denominada *blog.posts* y otra colección denominada *blog.autores* con un propósito organizativo y que, sin embargo, ni exista la colección *blog* y en caso de existir no exista una relación entre la colección padre *blog* y las subcolecciones.
- Las colecciones se agrupan en bases de datos, de manera que una única instancia de MongoDB puede gestionar varias bases de datos cada una agrupando cero o más colecciones.
- Bases de datos. Algunas observaciones:
  - Cada base de datos tiene sus propios permisos y se almacena en ficheros del disco separados.
  - Una buena regla general consiste en almacenar todos los datos de una aplicación en la misma base de datos.
  - Las bases de datos separadas son útiles cuando se almacenan datos para aplicaciones o usuarios diferentes que usan el mismo servidor de MongoDB.
- Las bases de datos se identifican mediante nombres que son cadenas con las siguientes restricciones:
  - La cadena vacía no es un nombre válido para una base de datos.
  - El nombre de una base de datos no puede contener ninguno de los siguientes caracteres: `\,/,.,'',*,<,>,:,|,?,$`,espacio o `\0`(valor nulo)
  - Los nombres de las bases de datos son sensitivos a mayúsculas y minúsculas incluso sobre sistemas de archivos que no lo sean. Una regla práctica es usar siempre nombres en minúscula.
  - Los nombres están limitados a un máximo de 64 bytes.
- Existen nombres que no pueden usarse para las bases de datos por estar reservados:

- *admin*. Es el nombre de la base de datos "root" en términos de autenticación. Si un usuario es añadido a esta base de datos entonces el usuario hereda los permisos para todas las bases de datos. Existen determinados comandos que solo pueden ser ejecutados desde esta base de datos tales como listar todas las bases de datos o apagar el servidor.
- *local*. Esta base de datos nunca será replicada y sirve para almacenar cualquier colección que debería ser local a un servidor.
- *config*. Cuando en MongoDB se usa una configuración con sharding se usa esta base de datos para almacenar información acerca de los fragmentos o shards que se crean.

2. CRUD

Operaciones CRUD: creación, lectura, actualización y borrado.

2.1. Inserción

- Creación de bases de datos y colecciones:
  - `use mi_bbdd`: crea o abre *mi_bbdd*.
  - `db.createCollection(''mi_colección'')`: crea *mi_colección*.
- Para insertar un documento en una colección se usa el método `insertOne`. Ejemplo:

```
db.prueba.insertOne({Titulo: "El_Quijote"})
```

  Esta acción añadirá al documento el campos `_id` en caso de no existir en el documento, y almacenará el mismo en MongoDB.

- Cuando es necesario insertar un conjunto de documentos, se puede pasar como parámetro un array con el conjunto de documentos que deben ser insertados y utilizar el método `insertMany`. Ejemplo:

```
db.prueba.insertMany([{Titulo: "Otro"},{Titulo: "Otro_mas"}])
```

- Se pueden insertar mediante un array múltiples documentos siempre que se vayan almacenar en una única colección, en caso de varias colecciones no es posible.
- Observación: Cuando se inserta usando un array, si se produce algún fallo en algún documento, se insertan todos los documentos anteriores al que tuvo el fallo, y los que hay a continuación no se insertan. Este comportamiento se puede cambiar usando la opción `continueOnError` que en caso de encontrarse un error en un documento lo salta, y continua insertando el resto de documentos. Esta opción no está disponible directamente en la shell, pero si en los drivers de los lenguajes de programación.
- Actualmente existe un límite de longitud de 48 MB para las inserciones realizadas usando un array de documentos.
- Cuando se inserta un documento MongoDB realizan una serie de operaciones con el objetivo de evitar inconsistencias tales como:
  - Se añade el campo `_id` en caso de no tenerlo.
  - Se comprueba la estructura básica. En particular se comprueba el tamaño del documento(debe ser más pequeño de 16 Mb). Para saber el tamaño de un documento se puede usar el comando `Object.bsonsize(doc)`.
  - Existencia de caracteres no válidos.

2.2. Consultas

- El método `find` se utiliza para realizar consultas en MongoDB, las cuales retornan un subconjunto de documentos de una colección (desde ningún documento hasta todos los documentos de la colección).
- El primer argumento especifica las condiciones que deben cumplir los documentos que se quieren recuperar.
- Una condición de búsqueda vacía ({}) encaja con todos los documentos de la colección.
- En caso de no especificar ninguna condición entonces se toma por defecto la condición vacía({}). Por ejemplo la consulta `db.c.find()` recupera todos los documentos de la colección `c`.
- Cuando se añaden pares clave/valor a las condiciones de búsqueda se restringe la búsqueda. Esto funciona directamente para la mayoría de los tipos: números coinciden con números, booleanos con booleanos, cadenas con cadenas...
- Si se quiere consulta un tipo simple entonces basta especificar el valor que se está buscando.
- Por ejemplo si se quiere encontrar todos los documentos dónde el valor de la edad es 27, entonces se añade como condición de búsqueda el par clave/valor {edad:27} a la condición de búsqueda:

```
db.users.find({edad:27})
```

- Cuando se quieren usar múltiples condiciones juntas se añaden los pares clave/valor que sean necesarias, las cuales serán interpretadas como `'Condición1 AND Condición2 AND...AND Condición N'`.
- Por ejemplo, si se quieren recuperar todos los usuarios con 27 años y que se llamen "Isabel" se realizaría la siguiente consulta:

```
db.users.find({nombre: "Isabel",edad:27})
```

- A veces cuando se recupera un documento no es necesario recuperar todos los campos del documento. Para ello se puede pasar un segundo argumento al método `find` para especificar que campos se quieren recuperar.
- Por ejemplo, si se tiene una colección de usuarios y solo se quiere recuperar el nombre del usuario y el email se podría realizar la siguiente consulta:

```
db.users.find({},{nombre: 1,email: 1})
```

- Siempre que se recupera un documento, por defecto se recupera el campo `_id`.
- También es posible especificar explícitamente que pares clave/valor no se quieren recuperar en la consulta.
- Por ejemplo, se puede tener una colección que tenga documentos con diferentes claves pero en todos ellos no se quiere recuperar la clave "teléfono", entonces se podría realizar la siguiente consulta:

```
db.users.find({},{telefono: 0})
```

- También podría usarse para evitar recuperar la clave `_id`:

```
db.users.find({},{nombre: 1, _id:0})
```

- Existe un método similar a `find()` que es `findOne()` que permite recuperar un único documento cumpliendo las condiciones especificadas.

- Operadores condicionales:
  - Los operadores "$lt", "$lte","$gt" y "$gte" corresponden a los operadores de comparación <,<=,> y >= respectivamente, y pueden combinarse para buscar rangos de valores.
  - Por ejemplo si se quiere buscar los usuarios que tienen una edad entre 18 y 30 años se puede hacer de la siguiente manera:

```
db.usuarios.find({edad : {$gte : 18, $lte : 30}})
```

  - Este tipo de consultas son muy útiles para realizar consultas sobre las fechas. Por ejemplo para encontrar las personas que se registraron antes del 1 de Enero de 2007 se puede hacer de la siguiente manera:

```
> fecha = new Date("01/01/2007")
> db.usuarios.find({registrados : {$lt : fecha}})
```

  - Una coincidencia exacta sobre la fecha es menos útil puesto que las fechas son sólo almacenadas con precisión de milisegundos, y con frecuencia lo que se busca es comparar un día, semana o mes entero haciendo necesario una consulta sobre rangos.
  - También puede ser útil consultar los documentos en los que el valor de una clave no es igual a cierto valor, para lo cual se usa el operador "$ne" que representa "no igual". Por ejemplo, si se quieren recuperar los usuarios que no tienen por nombre "Pablo", se podrían consultar de la siguiente manera:

```
db.users.find({username : {$ne : "joe"}})
```

  - Observa que el operador "$ne" puede ser usado con cualquier tipo.
- Consultas de tipo `OR`:
  - Existen dos posibilidades para realizar una consulta "OR":
    - El operador "$in" que puede ser usado para consultar sobre una variedad de valores para una clave dada. Este operador interpreta un "OR" como una pertenencia a un conjunto.
    - El operador "$or" que puede ser usado para consultar sobre un conjunto de valores dados sobre múltiples claves dadas.
  - Si se tiene más de un posible valor a encajar sobre una clave dada, es mejor usar un "$in" sobre un array con los valores.
  - Por ejemplo, si se quieren recuperar los documentos de personas que tienen un dni autorizado (sean 725,542 y 390 los dnis autorizados) se podría hacer de la siguiente manera:

```
db.autorizados.find({dni : {$in : [725, 542, 390]}})
```

  - "$in" es más flexible y permite especificar criterios sobre diferentes tipos y valores.
  - Por ejemplo, supongamos una base de datos donde se pueden usar tanto nombres de usuario como identificadores numéricos de usuario, entonces se podría realizar una consulta de la siguiente manera:

```
db.usuarios.find({user_id : {$in : [12345, "Pablo"]})
```

  - Si el operador "$in" aparece con un array con un único valor, entonces se comporta intentado hacer coincidir el valor. Por ejemplo {`dni : {$in : [725]}`} es equivalente a {`dni : 725`}.
  - El operador opuesto a "$in" es "$nin", el cual retorna documentos que no coinciden con ninguno de los criterios dados en el array de valores.

- Por ejemplo si se quieren recuperar todos los documentos de personas que tienen un dni que no está autorizado se podría hacer de la siguiente manera:

```
db.autorizados.find({dni : {$nin : [725, 542, 390]}})
```

- "$or" toma un array con posibles criterios de coincidencia. Por ejemplo, si se quieren recuperar los documentos de personas que tiene 30 años de edad ó que no son fumadores se podría hacer de la siguiente manera:

```
db.personas.find({$or : [{edad :30},{fumador:false}]})
```

- "$or" también puede contener otros condicionales. Por ejemplo, si se quieren recuperar los documentos de personas que tienen 30, 34 y 35 años o que no son fumadores se podría hacer de la siguiente manera:

```
db.personas.find({$or : [{edad :{$in:[30,34,35]}},{fumador:false}]})
```

- El operador "$or" siempre funciona pero siempre que se pueda es mejor usar "$in" dado que es más eficiente.

- El "$not":
  - El operador "$not" es un metacondicional, es decir que puede ser aplicado sobre otros criterios.
  - "$not" puede ser útil en conjunción con expresiones regulares para encontrar todos los documentos que no encajan con un determinado patrón.

- Observar que no existe el operador "$eq" pero se puede simular con el operador "$in" con un único valor.

- Observaciones:
  - Se pueden expresar múltiples condiciones sobre una clave dada. Por ejemplo, si se quieren encontrar todos los usuarios que tienen una edad entre 20 y 30, se podrían usar los operadores `$gt` y `$lt` sobre la clave "edad" de la siguiente manera:

```
db.usuarios.find({edad : {$lt : 30, $gt : 20}})
```

- Consultas sobre arrays:
  - Las consultas sobre elementos de un array están diseñada para comportarse de la misma forma que sobre valores escalares.
  - Por ejemplo, si se tiene un array que representa una lista de frutas como por ejemplo:

```
db.comida.insertOne({fruta : ["manzana", "platano", "melocoton"]})
```

    entonces la consulta:

```
db.comida.find({fruta : "platano"})
```

    entonces tendrá éxito sobre el documento que se ha insertado. La consulta es equivalente a si se tuviera un documento de la forma:

```
{fruta : "manzana", fruta : "platano", fruta : "melocoton"}
```

    Es decir, esta consulta determina si `platano` pertenece al conjunto [manzana", "platano", "melocoton"]".
  - En cambio , la consulta

```
db.comida.find({fruta : ["manzana","platano"]})
```

no devuelve nada ya que interpreta que exigimos una igualdad entre conjuntos ([manzana", "platano"]ƨ [manzana", "platano", "melocoton"]").

- El operador `$all`:
  - Cuando se quieren encajar todos los valores de un array se puede usar el operador `$all`.
  - Por ejemplo, si se tuviera la siguiente colección de documentos:

```
> db.comida.insertOne({_id : 1, fruta : ["manzana", "platano", "melocoton"]})
> db.comida.insertOne({_id : 2, fruta : ["manzana", "pera", "naranja"]})
> db.comida.insertOne({_id : 3, fruta : ["cereza", "platano", "manzana"]})
```

  - Se podría buscar todos los documentos que tienen a la vez "manzana" y "platano" mediante una consulta de la forma:

```
db.comida.find({fruta : {$all : ["manzana", "platano"]}})
```

    La presencia del operador `$all` hace que la búsqueda se interprete como una relación ente conjuntos. Ahora se exige que el conjunto [manzana", "platano"].[es]té contenido en alguno de los conjuntos de la clave `fruta`.
  - El orden no importa en la consulta. Así, por ejemplo, en el segundo resultado aparece "platano" antes de "manzana".
  - Cuando se usa `$all` con un array de un solo elemento entonces es equivalente a no usar `$all`. Por ejemplo:

```
{fruta : {$all : ['manzana']}
```

    es equivalente a:

```
{fruta : 'manzana'}
```

  - Se puede realizar una consulta para buscar una coincidencia exacta usando el array entero. Sin embargo, la coincidencia exacta no encajará un documento si alguno de los elementos no se encuentran o sobran.
  - Por ejemplo, la siguiente consulta recupera el primer documento:

```
db.comida.find({fruta : ["manzana", "platano", "melocoton"]})
```

    sin embargo las consultas:

```
db.comida.find({fruta : ["manzana", "platano"]})
db.comida.find({fruta : ["platano", "manzana", "melocoton"]})
```

    no recuperan el primer documento.
  - También es posible consultar un elemento específico de un array usando la notación indexada *clave.índice* como por ejemplo con:

```
db.comida.find({"fruta.2" : "melocoton"})
```

  - Observa que los índices empiezan a contar desde cero por lo que la consulta anterior buscaría que el tercer elemento del array tome el valor de "melocoton".
- El operador `$size`:
  - Este operador permite consultar arrays de un tamaño dado. Por ejemplo, la siguiente consulta:

```
db.comida.find({fruta : {$size : 3}})
```

- Un uso normal consiste en recuperar un rango de tamaños. El operador `$size` no puede ser combinado con otro operador condicional pero se puede añadir un clave "size" al documento, de manera que cada vez que se añade un elemento al array, entonces se incrementa el valor de la clave "size".

- El operador `$slice`:
  - Este operador puede ser usado para retornar un subconjunto de elementos de un clave que tiene por valor un array.
  - Por ejemplo, supongamos que se tiene un documento sobre un post de un blog y se quiere recuperar los 10 primeros comentarios entonces se podría hacer de la siguiente manera :

```
db.blog.posts.findOne(criterio, {comentarios : {$slice : 10}})
```

  - Y si se quieren recuperar los 10 últimos comentarios entonces se podría hacer de la siguiente manera:

```
db.blog.posts.findOne(criteria, {comentarios : {$slice : -10}})
```

  - El operador `$slice` también puede retornar elementos concretos de los resultados para lo cual es necesario especificar un valor que indica desde que elemento se empieza a recuperar y otro valor que indica cuántos se van a considerar a partir del indicado.
  - Así por ejemplo, la consulta:

```
db.blog.posts.findOne(criteria, {comentarios : {$slice : [23, 10]}})
```

    se salta los 23 primeros elementos y recupera del 24 al 33. Si hubiera menos de 33 elementos en el array entonces se recuperan tantos como sea posible.
  - Observa que a menos que se indique lo contrario, todas las claves de un documento son recuperadas cuando se usa el operador `$slice`. Este comportamiento es diferente con respecto a otros especificadores de clave que suprimen las claves que no se mencionan que deban recuperarse.
  - Por ejemplo, si se tuviera el siguiente documento:

```
>db.blog.posts.findOne()
{
    _id : ObjectId(...),
    titulo: "un_buen_post",
    contenido: "...",
    comentarios = [
        {
            nombre : "juan",
            email: "juan@ejemplo.com",
            contenido : "estupendo"
        },
        {
            nombre : "isabel",
            email: "isabel@ejemplo.com",
            contenido : "excelente"
        }
    ]
}
>
```

  - Si se usa el operador `$slice` para conseguir el último comentario se haría de la siguiente manera:

```
>db.blog.posts.findOne({},{comentarios : {$slice : -1}})
{
    _id : ObjectId(...),
    titulo: "un_buen_post",
```

```
    contenido: "...",
    comentarios = [
        {
            nombre : "isabel",
            email: "isabel@ejemplo.com,
____________contenido_:_"excelente"
________}
____]
}
>
```

- Se puede observar que se recupera además del comentario, también la clave "titulo" y "contenido" aunque no se haya especificado.
- A veces se desconoce el índice del elemento que se quiere recuperar. En estos casos se puede utilizar el operador $.
- En el ejemplo anterior si se quiere recuperar el comentario que ha realizado "juan" se podría hacer de la siguiente manera:

```
db.blog.posts.find({comentarios.nombre : "juan"}, {comentarios.$ : 1})
```

- La única limitación de esta técnica es que solo recupera la primera coincidencia, de manera que si hubiera más comentarios de "juan" en este post no serían retornados.

- Arrays y rangos:
  - Los valores escalares(no arrays) en los documentos deben coincidir con cada claúsula que aparece en la consulta. Así por ejemplo en la consulta:

```
{x : {$gt : 10, $lt : 20}}
```

    debería cumplirse a la vez que x es más grande que 10 y más pequeño que 20.
  - Sin embargo, si en un documento x fuera un array entonces el documento encajaría si existe un elemento de x que encaja con cada uno de los criterios que aparecen en la consulta pero cada criterio puede encajar con un elemento diferente del array. Así por ejemplo si se tuvieran los siguientes documentos en una colección:

```
{x : 5}
{x : 15}
{x : 25}
{x : [5, 25]}
```

    Si se quieren encontrar todos los documentos donde x está entre 10 y 20, entonces se podría construir la siguiente consulta:

```
db.test.find({x : {$gt : 10, $lt : 20}})
```

    esperando que recuperase como resultado el documento {`x : 15`} sin embargo recupera dos documentos:

```
{x : 15}
{x : [5, 25]}
```

    Aunque ni 5 ni 25 están entre 10 y 20, sin embargo 25 encaja con la condición de ser más grande que 10 y 5 encaja con la condición de ser más pequeño que 20. Es por ello que las consultas sobre rangos en arrays no son muy útiles pues un rango encajará con cualquier array multielemento.

    Sin embargo existen varias formas de conseguir el comportamiento esperado:

    - Se puede usar el operador `$elemenMatch` para forzar que MongoDB compare cada condición con cada elemento del array. Sin embargo este operador no encajará con elementos que no sean arrays.

- Así por ejemplo

```
db.test.find({x : {$elemMatch : {$gt : 10, $lt : 20}})
```

no devuelve ningún resultado cuando el documento {x : 15} cumpliría el criterio. El problema es que en ese documento x no es un array.

2.2.1. Consultas sobre documentos embebidos

- Existen dos caminos para consultar sobre documentos embebidos:
  1. Consultar sobre el documento entero.
  2. Consultar sobre pares individuales clave-valor.
- La consulta sobre un documento embebido entero funciona de la misma forma que una consulta normal.
- Sin embargo, una consulta sobre un subdocumento entero debe encajar exactamente con el subdocumento. Por ejemplo, si se añadiera un nuevo campo en el subdocumento o se cambiara el orden de los campos entonces ya no coincidirían con la búsqueda.
- En general, es mejor realizar consultas sobre claves específicas de un documento embebido de manera que si se producen cambios en la estructura de los documentos, las consultas no se vean afectadas por los cambios.
- Para consultar sobre claves específicas de documentos embebidos se usa la notación "." como por ejemplo:

```
db.prueba.find({"nombre.nombre" : "Javier", "nombre.apellido" : "Sanz"})
```

- Observa que en los documentos de consultas el uso de la notación dot tiene como significado alcanzar el interior del documento embebido. Es por ello que no se permita usar el carácter "." dentro de los documentos que se van a insertar(por ejemplo existen problemas cuando se quieren almacenar URLS).
- Una forma de resolverlo consiste en realizar un reemplazamiento global antes de insertar o después de recuperar, sustituyendo un carácter que no es legal en una URL por el carácter "."
- Las coincidencias con documentos embebidos puede complicarse cuando la estructura del documento se hace compleja.
- Por ejemplo, supongamos que se están almacenando posts de un blog y se quieren recuperar comentarios de Javier que fueron puntuados con al menos un 5, entonces se podría modelar el post de la siguiente manera:

```
>db.blog.posts.findOne()
{
    _id : ObjectId(...),
    contenido: "...",
    comentarios = [
        {
            autor : "javier",
            puntuacion: 3,
            comentario : "bonito_post"
        },
        {
            autor : "maria",
            puntuacion: 6,
            comentario : "terrible_post"
        }
    ]
}
>
```

Ahora bien con esta estructura no se puede consultar usando la expresión:

```
db.blog.find({comentarios : {autor : "javier", puntuacion : {$gte : 5}}})
```

dado que los documentos embebidos deben encajar el documento entero y éste no encaja con la clave "comentario".

- Tampoco funcionaría la expresión:

```
db.blog.find({"comentarios.autor" : "javier", "comentarios.puntuacion" : {$gte : 5}})
```

dado que la condición del autor podría encajar con un comentario diferente al comentario que encajaría con la condición de la puntuación.

- Esta consulta devolvería el documento anterior dado que encajaría autor "Javier" con el primer comentario y puntuación "6" con el segundo comentario.
- Para agrupar correctamente los criterios de búsqueda sin necesidad de especificar cada clave se puede usar `$elemMatch`. Este operador permite especificar parcialmente criterios para encajar con un único document embebido en un array. Así la consulta correcta sería:

```
db.blog.find({comentarios:{$elemMatch:{autor:"javier",puntuacion:{$gte:5}}}})
```

- Por tanto, `$elemMatch` será útil cuando exista más de una clave que se quiere encajar en un documento embebido.

2.2.2. Límites, saltos y ordenaciones

- Las opciones más comunes sobre las consultas es limitar el número de resultados recuperados, saltarse un número de resultados o bien la ordenacion de los resultados. Todas estas opciones deben ser añadidas antes de que la consulta sea enviada a la base de datos.
- Para conseguir limitar los resultados se encadena la función `limit()` sobre la llamada a `find()`. Por ejemplo, para retornar solo 3 resultados se haría de la siguiente manera:

```
db.c.find().limit(3)
```

- Si existen menos de 3 documentos que encajan con la consulta entonces solo se retornan los documentos que encajan (`limit` solo establece un límite superior pero no un límite inferior).
- La función `skip()` funciona de una manera similar `db.c.find().skip(3)`. Este ejemplo se saltará los 3 primeros documentos que encajen y retornará el resto de resultados. Si existen menos de 3 documentos , no retornará ningún documento.
- La función `sort()` toma un objeto formado por pares clave-valor donde las claves son nombres de claves y los valores indican un sentido de la ordenación: 1(ascendente) o -1(descendente). Cuando existen más de una clave, los resultados son ordenados en ese orden.
- Por ejemplo, para ordenar los resultados en orden ascendente de "nombre" y en orden descendente de "edad" se haría de la siguiente manera:

```
db.c.find().sort({username : 1, age : -1})
```

- Estos métodos se pueden combinar. Por ejemplo, es útil en la paginación de resultados. Supongamos que se está consultando una base de datos de libros y se van a mostrar los resultados por páginas con un máximo de 50 resultados por página ordenados por precio de mayor a menor, entonces se puede hacer lo siguiente:

```
db.stock.find({desc : "libros"}).limit(50).sort({price : -1})
```

- Cuando la persona hace click sobre la página siguiente para ver más resultados entonces se puede añadir un `skip()` a la consulta de manera que se salten los primeros 50 resultados:

```
db.stock.find({desc : "libros"}).limit(50).skip(50).sort({price : -1})
```

- A veces se puede dar el caso de tener un clave con multiples tipos, de manera que si se aplica la función `sort()` entonces se ordenarán de acuerdo a un orden predefinido. Los valores del más pequeño al más grande son:

  1. Minimum value
  2. Null
  3. Numbers: integers, floats, doubles
  4. Strings
  5. Objects/document
  6. Array
  7. Binary data
  8. Object ID
  9. Boolean
  10. Date
  11. Timestamp
  12. Regular expression
  13. Maximum value

2.3. Actualizaciones

Existen dos tipos de actualizaciones en MongoDB:

- Reemplazamiento: se sustituye un documento (o varios) previamente seleccionado por otro documento.
  - Si se seleccionan varios documentos para ser sustituidos por otro puede dar error al haber varios documentos con la clave `_id` duplicada.
- Modificaciones: se cambia el contenido del valor de una clave de un documento previamente seleccionado por otro.

2.3.1. Reemplazamiento

- Con la instrucción `replaceOne` podemos sustituir un documento por otro.
- Sintaxis: `replaceOne(criterio,nuevo documento)`
  - *criterio* determina qué elemento va a ser sustituido.
  - *nuevo documento* especifica el contenido del nuevo elemento
- Ejemplo:

```
use academia

db.createCollection('alumnos')

db.alumnos.insertMany([{codigo:111,nombre:'pepe',edad:22},{codigo:222,nombre:'ana'}])
```

```
db.alumnos.find()
{
  _id: ObjectId("63ff72b9db530f89fc947269"),
  codigo: 111,
  nombre: 'pepe',
  edad: 22
}
{
  _id: ObjectId("63ff72b9db530f89fc94726a"),
  codigo: 222,
  nombre: 'ana'
}
```

- La instrucción siguiente:

```
db.alumnos.replaceOne({codigo:111},{codigo:333,nombre:'juan',direccion:{calle:'xxx',numero:3}})
```

cambia el documento cuyo código es 111 por otro documento:

```
db.alumnos.find()
{
  _id: ObjectId("63ff72b9db530f89fc947269"),
  codigo: 333,
  nombre: 'juan',
  direccion: {
    calle: 'xxx',
    numero: 3
  }
}
{
  _id: ObjectId("63ff72b9db530f89fc94726a"),
  codigo: 222,
  nombre: 'ana'
}
```

2.3.2. Modificadores

- En muchas ocasiones el tipo de actualización que se quiere realizar consiste en añadir, modificar o eliminar claves, manipular arrays y documentos embebidos, etc. Para estos casos se utilizan los operadores de modificación.
- Métodos de actualización:
  - `updateOne(criterio,acualizacion)`: actualiza *un* documento que cumple el criterio.
  - `updateMany(criterio,acualizacion)`: actualiza *todos* los documentos que cumplen el criterio.
- Es crucial saber cuántos documentos estarán afectados por el criterio: uno o más de uno. Es decir, hay que saber si las claves involucradas en el criterio juegan el papel de 'clave' o no.

$inc

- Este operador permite cambiar el valor numérico de una clave que ya existe incrementando su valor por el especificado junto al operador, o bien, puede crear una clave que no existía inicializándola al valor dado.
- Por ejemplo, supongamos que queremos almacenar el número de visitas al campus virtual a través de una clave llamada `visitas`. Podemos asignar a todos los alumnos de la colección `alumnos` un valor para esa clave:

```
db.alumnos.updateMany({},{$inc:{visitas:0}})
```

- Cada vez que alguien visita el campus se actualiza el valor de la clave `visitas` de la siguiente forma:

```
db.alumnos.updateOne({nombre:'ana'},{$inc:{visitas:1}})
```

- También sería posible incrementar el valor por un valor mayor que 1:

```
db.alumnos.updateOne({nombre:'ana'},{$inc:{visitas:15}})
```

- De la misma forma se podría decrementar usando números negativos:

```
db.alumnos.updateOne({nombre:'ana'},{$inc:{visitas:-5}})
```

- Cuando se usan operadores de modificación el valor del campo `_id` no puede ser cambiado (en cambio cuando se reemplaza un documento entero si es posible cambiar el campo `_id`). Sin embargo los valores para cualquier otra clave incluyendo claves indexadas únicas si pueden ser modificadas.
- Este operador solo puede ser usado con números enteros, enteros largos o double, de manera que si se usa con otro tipo de valores (incluido los tipos que algunos lenguajes tratan como números tales como booleanos, cadenas de números, nulos,...) producirá un fallo.

$set y $unset

- Este operador establece un valor para un campo dado y, si el campo dado no existe, lo crea. Es útil para modificar el esquema de un documento o añadir claves definidas por el usuario.
- Por ejemplo, a todos los alumnos de la academia les podemos asociar al centro de Madrid (representado con la clave valor: `centro:'Madrid'`) de la siguiente forma:

```
db.alumnos.updateMany({},{$set:{centro:"Madrid"}})
```

- Podemos modificar este valor para uno o varios alumnos (documentos):

```
db.alumnos.updateMany({nombre:"juan"},{$set:{centro:"Madrid"}})
```

- Podemos utilizar `$set` para dar valores no atómicos a una clave. Por ejemplo, podemos asignar a una clave un array:

```
> db.alumnos.updateOne({nombre:"juan"},{$set:{materias:["matematicas","quimica"]}})
```

  o un valor que sea otro documento:

```
> db.alumnos.updateOne({nombre:"ana"},{$set:{direccion:{calle:"yyy",numero:"29"}}})
```

- Existe un operador denominado `$unset` que permite eliminar campos de un documento. En el ejemplo anterior si se quiere eliminar el campo `centro`:

```
> db.alumnos.updateMany({},{$unset:{centro:1}})
```

- Observa que no es necesario tener ninguna clave a `null`. Si el valor de una clave no se conoce no se debe incluir en el documento. Con los operadores `$inc`, `$set` y `$unset` se pueden gestionar en todo momento el contenido de las claves.

2.3.3. Modificadores de arrays

- Adición de elementos:
  - El modificador `$push` añade elementos al final del array si existe o bien crea uno nuevo si no existe.
  - Por ejemplo, queremos gestionar notas de alumnos. Insertamos un nuevo alumno:

```
db.alumnos.insertOne({codigo:555,nombre:"carlos",notas:[7,9]})
```

    comprobamos la inserción:

```
db.alumnos.find({nombre:"carlos"},{nombre:1,notas:1,_id:0})
{
  nombre: 'carlos',
  notas: [
    7,
    9
  ]
}
```

    podemos añadir una nueva nota de la siguiente manera:

```
db.alumnos.updateOne({nombre:"carlos"},{$push:{notas:5}})
```

    comprobamos la inserción:

```
db.alumnos.find({nombre:"carlos"},{nombre:1,notas:1,_id:0})
{
  nombre: 'carlos',
  notas: [
    7,
    9,
    5
  ]
}
```

    observa que el valor se ha añadido al final del array.
  - Es posible añadir un conjunto de valores al array (en lugar de uno solo) combinando los operadores `$push` y `$each`. Ejemplo:

```
db.alumnos.updateOne({nombre:"carlos"},{$push:{notas:{$each:[3,3,3]}}})
```

    comprobamos la inserción:

```
db.alumnos.find({nombre:"carlos"},{nombre:1,notas:1,_id:0})
{
  nombre: 'carlos',
  notas: [
    7,
    9,
    5,
    3,
    3
  ]
}
```

  - También es posible limitar la longitud hasta la que puede crecer un array usando el operador `$slice` junto al operador `$push`. Ejemplo: añadimos tres notas más y limitamos la longitud del array:

```
db.alumnos.updateOne({nombre:"carlos"},{$push:{notas:{$each:[8,8,8],$slice:-5}}})
```

    comprobamos la inserción:

```
db.alumnos.find({nombre:"carlos"},{nombre:1,notas:1,_id:0})
{
  nombre: 'carlos',
  notas: [
    3,
    3,
    8,
    8,
    8
  ]
}
```

Observaciones:

- Con un valor negativo para `$slice`, los valores que no quepan se pierden en el principio del array. Con un valor positivo, se pierden al final.
- La limitación del tamaño del array se produce en la actualización.

- Por último el operador `$sort` permite ordenar los elementos indicando el campo de ordenación y el criterio en forma de 1(ascendente) o -1(descendente).

```
db.alumnos.updateOne({nombre:"carlos"},{$push:{notas:{$each:[8,8,8],$slice:-5,$sort:-1}}})
```

- Tanto `$slice` como `$sort` deben ir junto a un operador `$each` y no pueden aparecer solos con un `$push`.
- Observa que el contenido del array son valores numéricos. En caso de que el contenido del array fuera documentos, sería posible realizar una ordenación similar. Por ejemplo, si tuviéramos un array de documentos con un campo `valoracion`, se puede realizar la ordenación especificando el campo por el cual ordenar de la siguiente forma: `$sort:{valoracion:-1}`

- Usando arrays como conjuntos:

  - Los arrays se pueden tratar como un conjunto añadiendo valores solo si no estaban ya. Para ello se usa el operador `$addToSet`. Ejemplo:

  ```
  db.alumnos.updateOne({nombre:'juan'},{$addToSet:{notas:10}})
  ```

  Si el valor no está en el array `notas` lo inserta. Si no está no hace nada.

- Borrado de elementos:

  - Existen varias formas de eliminar elementos de un array dependiendo de la forma en la que se quieran gestionar. Si se quiere gestionar como si fuera una pila o una cola entonces se puede usar el operador `$pop` que permite eliminar elementos del final del array(si toma el valor 1) o bien del principio del array (si toma el valor -1):

  ```
  db.alumnos.updateOne({nombre:'juan'},{$pop:{notas:-1}})

  db.alumnos.updateOne({nombre:'juan'},{$pop:{notas:1}})
  ```

  - Otra forma alternativa de eliminar elementos es especificando un criterio en vez de una posición en el array usando el operador `$pull`. El operador `$pull` elimina todas las coincidencias que encuentre en los documento no solo la primera coincidencia.

  ```
  db.alumnos.updateOne({nombre:'juan'},{$pull:{notas:9}})
  ```

- Modificaciones posicionales en un array:

  - Las manipulaciones de un array se convierten en algo complejo cuando se tienen múltiples valores y se quieren modificar solo algunos de ellos. En este sentido existen dos caminos para manipular valores de un array.

- Mediante su posición. En este caso sus elementos son seleccionados como si se indexaran las claves de un documento.

```
db.alumnos.updateOne({nombre:'juan'},{$inc:{"notas.0":2}})
```

  Hemos incrementado la primera nota en dos puntos.

- En algunas ocasiones no se sabe en qué posición se encuentra el valor a modificar. En tales casos es posible indicar el valor que tiene que ser sustituido (junto con el nuevo)

```
db.alumnos.updateOne({nombre:'juan',notas:6},{$set:{"notas.$":7}})
```

- Observa que solo se actualiza la primera coincidencia.

2.4. Borrado

- Funciones para eliminar documentos:
  - `deleteOne(criterio)`: borra el primer documento que satisface el criterio.
  - `deleteMany(criterio)`: borra todos los documentos que satisfacen el criterio.
  - `remove(criterio)`(obsoleta): borra todos los documentos que satisfacen el criterio.
- Ejemplos:

```
db.alumnos.deleteMany({nombre:"juan"})
```

  borra todos los documentos que poseen una clave `nombre` que tiene el valor `juan`.

```
db.alumnos.deleteMany()
```

  borra todos los documentos de la colección `alumnos`.

- A veces si se van a borrar todos los documentos es más rápido eliminar toda la colección en vez los documentos. Para ello se usa el método `drop`:

```
db.alumnos.drop()
```

3. Índices

3.1. Introducción

- Un índice es una estructura administrada en segundo plano que permite que las consultas y ordenaciones sean más rápidas.
- Podemos imaginar un índice sobre una colección como una lectura de todos los documentos de esa colección ordenada según cierto campo, o conjunto de campos. Los resultados de esta lectura se almacenan.
- Una vez creado un índice, las consultas que involucren a ese campo serán más rápidas.
- Por ejemplo, consideremos la siguiente consulta:

```
db.alumnos.find().sort({nombre:1})
```

  Muestra todos los documentos de la colección `alumnos` ordenados por nombre. Ahora la consulta:

```
db.alumnos.find({nombre:"juan"})
```

es más eficiente. Ya no necesita recorrer de forma completa toda la colección. En cuanto lea un documento con un nombre posterior a *juan* finalizará (porque está ordenados por nombre).

- Un índice permite realizar una consulta como la anterior y almacenarla.
- Podemos clasificar los problemas de los índices en dos tipos: dificultad para su actualización y dificultad para su almacenamiento.
- Dificultad para su actualización:
  - La estructura generada en segundo plano debe ser actualizada. Si se realizan inserciones, borrados o modificaciones en documentos, la información del índice también debe modificada.
  - Cuando se crea un índice aumenta la velocidad de las consultas, pero se reduce la velocidad de las inserciones y las eliminaciones. Hay que actualizar el contenido de la colección y el contenido del índice asociado.
  - Es mejor añadir índices en las colecciones cuando el número de lecturas es mayor que el número de escrituras. Si hay más escrituras que lecturas entonces los índices pueden ser contraproducentes.
- Dificultad para su almacenamiento:
  - Los índices ocupan espacio. Si tenemos una colección y creamos tres índices necesitaremos cuatro veces el espacio de la colección.
- Por otra parte, cuando se tienen índices de vez en cuando hay que borrar algunos índices o reconstruirlos debido a varias razones:
  - Limpiar algunas irregularidades que aparecen en los índices.
  - Aumento del tamaño de la base de datos.
  - Espacio excesivo ocupado por los índices. Solo se pueden definir como máximo 40 índices por colección.

3.2. Gestión de índices. Índices simples

- La información sobre los índices asociados a una colección se puede consultar con `db.coleccion.getIndexes()`. Ejemplo:

```
db.alumnos.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
]
```

- Siempre existe un índice sobre el campo `_id` por defecto. Son creados y borrados automáticamente por el sistema cada vez que se crea o se borra una colección.
- Creación de un índice simple:
  - Para añadir nuevos índices a una colección se usa la función `createIndex()`.
  - Esta función primero chequea si ya se ha definido un índice con la misma especificación, en cuyo caso devuelve el índice, y en caso contrario lo crea.
  - La función toma como parámetros el nombre de una clave de uno de los documentos que se usará para crear el índice, y un número que indica la dirección de ordenación del índice: 1 almacena los ítems en orden ascendente y -1 almacena los ítems en orden descendente.
  - El comando asegura que el índice se creará para todos los valores de la clave indicada para todos los documentos de la colección.

- Ejemplo:

```
> db.alumnos.createIndex({nombre:1})
>db.alumnos.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { nombre: 1 }, name: 'nombre_1' }
]
```

3.3. Índices compuestos

- En muchas ocasiones el criterio de ordenación es complejo e involucra varios campos. En estos casos es posible crear índices compuestos.
- Por ejemplo, en una colección que almacene puntuaciones en una competición se puede ordenar los documentos por las puntuaciones obtenidas. En caso de que dos participantes obtengan la misma puntuación se ordena por nombre. Es decir, el criterio de ordenación es *(puntuación,nombre)*.
- Observa que el orden de los campos cambia la ordenación. No es lo mismo ordenar por *(puntuación, nombre)* que por *(nombre, puntuación)*.
- Existen dos formas de crear índices compuestos:
  - Crear un índice de subdocumentos.
  - Crear un índice definido por el usuario.

3.3.1. Índices de subdocumentos

- Se puede crear un índice compuesto usando un subdocumento entero, de manera que cada elemento del documento embebido se convierte en parte del índice.
- Por ejemplo, supongamos que un documento para la clave `nombreCompleto` su valor es un documento embebido formado las claves `nombre` y `apellidos`.
- Podemos crear un índice compuesto mediante la instrucción:

```
db.alumnos.createIndex({nombreCompleto:1})
```

  de esta forma se ordenan todos los documentos según el criterio *nombre,apellidos*.

- Limitaciones:
  - El orden de las claves en el subdocumento es el criterio de ordenación.
  - Si en las lecturas cambiamos el orden de las claves es como si el índice no existiera. Por ejemplo, `db.alumnos.find().sort(nombreC.apellidos:1,nombreC.nombre:1)` no sería más rápida debido al índice.
  - No podemos combinar criterios ascendentes con descendentes.

3.3.2. Índices definidos por el usuario

- Un índice compuesto definido por el usuario se crea especificando cada uno de los campos del criterio de ordenación.
- Por ejemplo, podemos crear un índice compuesto similar al anterior mediante la instrucción:

```
db.alumnos.createIndex({nombreCompleto.nombre:1,nombreCompleto.apellidos:1})
```

  o bien:

```
db.alumnos.createIndex({nombreCompleto.apellidos:1,nombreCompleto.nombre:1})
```

- Podemos cambiar el orden de las claves que hay en el subdocumento para crear el criterio.
- Además, podemos combinar criterios ascendentes con descendentes:

```
db.alumnos.createIndex({nombreCompleto.apellidos:1,nombreCompleto.nombre:-1})
```

Con los índices con subdocumentos no es posible esta combinación.

3.4. Índices únicos

- En algunas circunstancias se exige que no haya en la misma colección dos documentos con el mismo valor de una clave. Es decir, se utiliza esa clave 'primaria'.
- Con la opción *unique* se puede crear un índice donde todas las claves deben ser diferentes.
- De manera que el sistema retornará un error si se intenta insertar un documento donde clave del índice coincide con la clave de un documento existente.
- Es útil para campos donde se quiere asegurar que no se repiten valores.
- Sin embargo si se quiere añadir un índice único a una colección ya existente con datos, hay que asegurarse de que no existen duplicaciones en las claves, pues de lo contrario fallará si cualquiera de las claves no son únicas.
- En el caso de los índices compuestos, el sistema fuerza a la unicidad sobre la combinación de los valores en vez del valor individual para alguno o todos los valores de la clave.
- Si un documento es insertado con un campo que falta y especificado como una clave única, entonces automáticamente se inserta el campo con el valor a `null`.
- Esto significa que solo se puede insertar un documento en el que falte un campo pues nuevos valores nulos harán que se considere que la clave no es única.
- Ejemplo:

```
> db.alumnos.createIndex({codigo:1},{unique:true})
```

- Si se quiere crear un índice único para un campo donde se conoce que existe valores duplicados, entonces se puede usar la opción `dropdups`, que elimina los documentos que causan que falle la creación de un índice único.
- Se mantendrá el primer documento que se encuentre en la ordenación natural de la colección y se eliminará cualquier otro documento que se encuentre y viole la condición de creación del índice.
- Ejemplo:

```
> db.alumnos.createIndex({codigo:1},{unique:true,dropdups:true})
```

3.5. Borrado de índices

- Para borrar un índice se utiliza el método `dropIndex`.
- Sintaxis: `db.coleccion.dropIndex(especificacion del indice)`
- Ejemplo:

```
db.alumnos.dropIndex({nombre:1})
{ nIndexesWas: 2, ok: 1 }
db.alumnos.getIndexes()
[ { v: 2, key: { _id: 1 }, name: '_id_' } ]
```

3.6. Reindexaciones

- Los índices son estructuras complejas que se actualizan cada vez que se añaden, eliminan o modifican documentos.
- Con su uso los índices se pueden 'dañar', es decir, su información no es del todo coherente.
- En tales casos se puede reindexar una colección. Se eliminan todos los índices de esa colección y se vuelven a crear.
- El método `reIndex()` realiza esta operación. Ejemplo:

```
db.alumnos.reIndex()
{
  nIndexesWas: 2,
  nIndexes: 2,
  indexes: [
    { v: 2, key: [Object], name: '_id_' },
    { v: 2, key: [Object], name: 'nombre_1' }
  ],
  ok: 1
}
```

3.7. Selección de índices

- Cuando se quiere ejecutar una consulta, entonces el sistema crea un *plan de ejecución* que es una lista de los pasos que debe ejecutar para llevar a cabo la consulta.
- Cada consulta tiene múltiples planes que producirán el mismo resultado.
- Sin embargo cada plan puede tener elementos que son más costosos de ejecutar que otros.
- Si hay varios índices, MongoDB elige el que cree más óptimo. Sin embargo, es posible forzar a que utilice un índice con él métod `hint()`.
- Ejemplo:

```
db.alumnos.find({nombreC.nombre:"pepe",nombreC.apellidos:"garcia"}).hint({nombreC.nombre:1})
```

Obligamos al sistema a aplicar el índice `nombreC.nombre:1`.

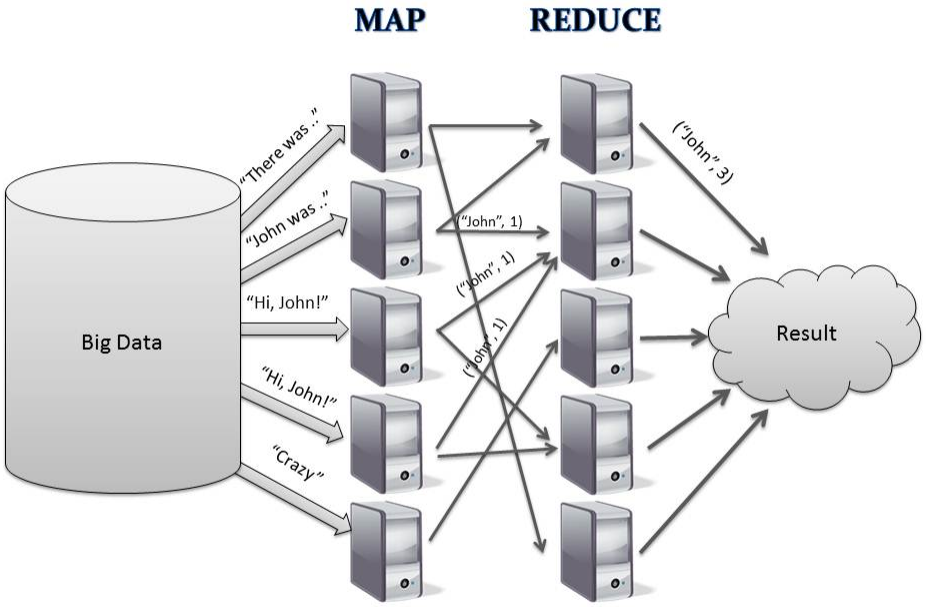

Figura 1: Esquema de MapReduce

4. *MapReduce*

- ¿Qué es MapReduce?:
  - Map-reduce es un paradigma de programación orientado al procesamiento paralelo de grandes volúmenes de información.
  - No todos los procesos pueden ser abordados mediante MapReduce. Concretamente son abordables sólo aquellos que se pueden separar en operaciones de *map* y de *reduce*.
  - La función *map* se ejecuta de forma distribuida a lo largo de varias máquinas. Los datos de entrada, procedentes por regla general de un gran archivo, se dividen en un conjunto de m particiones de entrada (generalmente 16 a 64 megabytes). Estas particiones pueden ser procesadas en diversas máquinas.
  - La función *reduce* se aplica en paralelo para cada grupo creado por la función *map*.
- MongoDB puede utilizar comandos *mapReduce*.
- Sintaxis de los comandos básicos de *mapReduce* en MongoDB:

```
>db.collection.mapReduce(
    function() {emit(key,value);}, //map function
    function(key,values) {return reduceFunction}, { //reduce function
        out: collection,
        query: document,
        sort: document,
        limit: number
    }
)
```

- Comentarios:

- *map* es una función javascript función que asigna un valor a una clave.
- *reduce* es una función javascript función que agrupa todos los documentos por su clave. Algunos parámetros:
  - *out* especifica la colección que almacena el resultado de la consulta *map-reduce*.
  - *query* especifica el criterio, opcional, para la selección de documentos.
  - *sort* especifica el criterio, opcional, para la ordenación del resultado
  - *limit* especifica el máximo número de elementos devuleltos.

- Consideremos la siguiente estructura de documento que almacena usuarios. El documento almacena `user_id` de un usuario y el estatus de "post".

```
{
"post_text": "tutorialspoint is an awesome website for tutorials",
"user_id": "mark",
"status":"active"
}
```

- La función *map* asigna a cada clave (*autor* en este caso) el valor 1.
- La función *reduce* relaciona todos los valores generados para una clave (en este caso los suma todos).
- Usamos una función *mapReduce* en nuestra colección de "posts" para seleccionar todos los "posts" activos, agruparlos por `user_name` y contar el número de "posts" por cada usuario:

```
>db.posts.mapReduce(
    function() { emit(this.user_id,1); },
    function(key, values) {
    var count = 0;
    for (var i = 0; i < values.length; ++){
    count += values[i];
    }
    return count}, {
        query:{status:"active"},
        out:"post_total"
}
)
```

- Para ver el resultado de esta consulta *mapReduce* se utiliza el operador `find`:

```
>db.post_total.find()
```

- La consulta da el siguiente resultado que indica que ambos usuarios tienen dos "posts" activos:

```
{ _id : "tom", value : 2 }
{ _id : "mark", value : 2 }
```

- De forma similar, las consultas MapReduce se pueden utilizar para construir consultas de agregación complejas. El uso de funciones Javascript functions hace que el uso de MapReduce sea muy flexible.

5. MongoDB y PHP

La interfaz de PHP a MongoDB es muy sencilla. La veremos a través de ejemplos.
Conexión a una base de datos:

```
<?php
    // connect to mongodb
    $m = new MongoClient();
    echo "Connection to database successfully";
    // select a database
    $db = $m->mydb;
    echo "Database mydb selected";
?>
```

Si no existe la base de datos se crea.

Creando una colección:

```
<?php
    // connect to mongodb
    $m = new MongoClient();
    echo "Connection to database successfully";
    // select a database
    $db = $m->mydb;
    echo "Database mydb selected";
    $collection = $db->createCollection("mycol");
    echo "Collection created successfully";
?>
```

Insertando un documento:

```
<?php
    // connect to mongodb
    $m = new MongoClient();
    echo "Connection to database successfully";
    // select a database
    $db = $m->mydb;
    echo "Database mydb selected";
    $collection = $db->mycol;
    echo "Collection selected successfully";
    $document = array(
        "title" => "MongoDB",
        "description" => "database",
        "likes" => 100,
        "url" => "http://www.tutorialspoint.com/mongodb/",
        "by", "tutorials point"
    );
    $collection->insertOne($document);
    echo "Document inserted successfully";
?>
```

Seleccionando todos los documentos de una colección:

```
<?php
    // connect to mongodb
    $m = new MongoClient();
    echo "Connection to database successfully";
    // select a database
    $db = $m->mydb;
    echo "Database mydb selected";
    $collection = $db->mycol;
    echo "Collection selected successfully";
    $cursor = $collection->find();
    // iterate cursor to display title of documents
    foreach ($cursor as $document) {
        echo $document["title"] . "\n";
    }
?>
```

Actualizando un documento:

```
<?php
    // connect to mongodb
    $m = new MongoClient();
    echo "Connection to database successfully";
    // select a database
    $db = $m->mydb;
    echo "Database mydb selected";
    $collection = $db->mycol;
    echo "Collection selected succsessfully";
    // now update the document
    $collection->update(array("title"=>"MongoDB"),
        array('$set'=>array("title"=>"MongoDB Tutorial")));
    echo "Document updated successfully";
    // now display the updated document
    $cursor = $collection->find();
    // iterate cursor to display title of documents
    echo "Updated document";
    foreach ($cursor as $document) {
        echo $document["title"] . "\n";
```

```
    }
?>
```

Borrando datos de un documento:

```
<?php
    // connect to mongodb
    $m = new MongoClient();
    echo "Connection to database successfully";
    // select a database
    $db = $m->mydb;
    echo "Database mydb selected";
    $collection = $db->mycol;
    echo "Collection selected succsessfully";
    // now remove the document
    $collection->remove(array("title"=>"MongoDB Tutorial"),false);
    echo "Documents deleted successfully";
    // now display the available documents
    $cursor = $collection->find();
    // iterate cursor to display title of documents
    echo "Updated document";
    foreach ($cursor as $document) {
    echo $document["title"] . "\n"; }
?>
```