CURSO DE MONGODB

¡HOLA!

Soy Pablo Campos

Co-founder y CTO en Secmotic Innovation (@secmotic) y un loco de la innovación y las nuevas tecnologías.
@PabGallagher

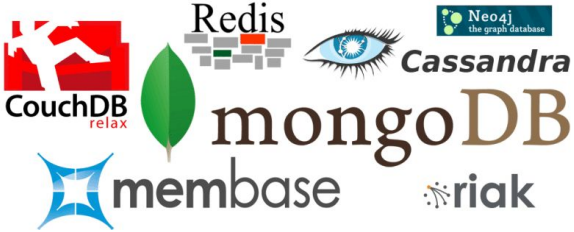
1.

INTRODUCCIÓN

Primero lo primero

EL CONCEPTO NOSQL

- No solo SQL
- Escalabilidad
- Versatilidad
- Agilidad
- Velocidad

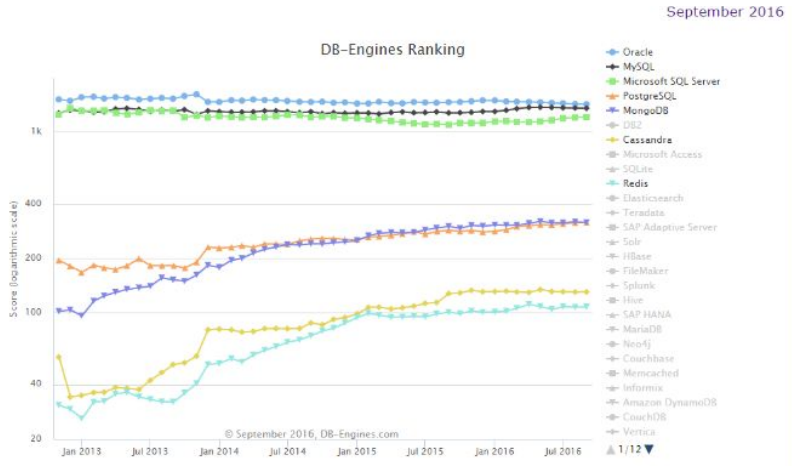


CONTEXTUALIZANDO SQL Y NOSQL

NoSQL	SQL
Base de datos	Base de datos
Colección	Tabla
Documento	Fila
Campos	Columna

ORIGEN Y EVOLUCIÓN DE MONGO

- 10GEN detecta la necesidad en 2007
- Se lanza el proyecto open source en 2009



MONGODB

- Alto rendimiento
- Lenguaje de consultas avanzado
- Alta disponibilidad
- Escalabilidad horizontal
- Consola en JS

INSTALANDO MONGODB

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
EA312927

echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.2
multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list

sudo apt-get update

sudo apt-get install -y mongodb-org

sudo service mongod start
```

2.

MÉTODOS CRUD

Create
Read
Update
Delete

CONTEXTUALIZANDO SQL Y NOSQL

NoSQL	SQL
Base de datos	Base de datos
Colección	Tabla
Documento	Fila
Campos	Columna

BASES DE DATOS

- Activamos el servidor mongod

```
sudo service mongod start
```

- Abrimos el cliente mongoDB

```
mongo
```

- Creamos la base de datos

```
use nuestraBaseDeDatos
```

- *Eliminamos la base de datos

```
use nuestraBaseDeDatos
db.dropDatabase()
```

COLECCIONES

Existen dos métodos para crear una colección

db.createCollection()
En este caso definimos el nombre de la colección así como ciertas propiedades al crearla

db.ejemplo.insert()
Para este caso directamente damos un nombre *ejemplo* a la colección e insertamos un documento

OPCIONES PARA LA CREACIÓN DE COLECCIONES

- *capped*: Booleano que junto a la opción *size* permite definir una *capped collection* (tamaño máximo en Bytes).
- *autoIndex*: Booleano para especificar el uso o no del campo *_id*
- *max*: Entero para definir el máximo número de documentos permitidos en una colección
- *validator*: Documento que define las validaciones de cada campo

OPCIONES PARA LA CREACIÓN DE COLECCIONES

- *validationLevel*: Nivel de restricción en las validaciones -> moderado (sólo aplica a aquellos documentos que cumplan la validación) o estricto (aplica a todos los documentos)

```
db.createCollection( "contacts",
   { validator: { $or:
      [
         { phone: { $type: "string" } },
         { email: { $regex: /@mongodb\.com$/ } },
         { status: { $in: [ "Unknown", "Incomplete" ] } }
      ]
   }
} )
```

LOS DOCUMENTOS

- No tienen estructura definida
- Tienen formato JSON
- Se almacenan en colecciones
- Son las “filas” de SQL
- Pares key - value
- La clave primaria se por defecto en la clave *_id*

LOS DOCUMENTOS

```
{
   _id: ObjectId(7df78ad8902c)
   title: 'Introduccion a MongoDB',
   description: 'MongoDB es una base de datos no sql',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100,
   comments: [
      {
         user:'usuario1',
         message: 'Mi primer comentario',
         dateCreated: new Date(2011,1,20,2,15),
         like: 0
      },
      {
         user:'usuario2',
         message: 'Mi segundo comentario',
         dateCreated: new Date(2011,1,25,7,45),
         like: 5
      }
   ]
}
```

LOS DOCUMENTOS - CREATE

- Para crear entradas en una base de datos usamos el método insert
- Previamente debemos definir la base de datos en la que queremos insertarlo
- Es necesario especificar en qué colección se creará

LOS DOCUMENTOS - CREATE

```
use openWebinars

db.articulos.insert({
      Título: ‘Aprendiendo MongoDB',
      Contenido : ‘Este es el contenido del post',
      Tags : [‘eLearn' , 'MongoDB']
});
```

LOS DOCUMENTOS - READ

- Para leer entradas en una base de datos usamos el método read
- Previamente debemos definir la base de datos en la que queremos insertarlo
- Es necesario especificar en qué colección se creará
- Es posible filtrar y aplicar operaciones post-query

LOS DOCUMENTOS - READ

MongoDB permite

- Proyectar solo algunos de los campos de un documento
- Aplicar expresiones regulares en nuestras búsquedas
- Ordenar la búsqueda
- Limitar el número de resultados
- “Clarificar” la salida

LOS DOCUMENTOS - READ

```
use openWebinars

db.articulos.find()

db.articulos.find({},{titulo:1, _id:0})

db.artigulos.find({titulo:{$regex:".*ngod*."}})

db.articulos.find().sort({titulo:1})   *o -1

db.articulos.find().limit(5)

db.articulos.find().pretty()
```

LOS DOCUMENTOS - DELETE

- Para eliminar entradas en una base de datos usamos el método remove o drop
- Previamente debemos definir la base de datos en la que queremos insertarlo
- Es necesario especificar en qué colección se creará
- Es posible eliminar documentos o colecciones completas

LOS DOCUMENTOS - DELETE

```
use openWebinars

db.articulos.remove({titulo:”El que quiero eliminar”})

db.articulos.drop()

db.articulos.remove()
```

LOS DOCUMENTOS - UPDATE

- Para actualizar entradas en una base de datos usamos el método update
- Previamente debemos definir la base de datos en la que queremos insertarlo
- Es necesario especificar en qué colección se creará
- Las opciones multi y upsert permiten modificar más de un documento si cumpliera el filtro y crear el documento si no hay resultados respectivamente

LOS DOCUMENTOS - UPDATE

MongoDB permite muchos operadores para las actualizaciones de documentos

- $inc: Incrementar el valor de un campo numérico
- $rename: Renombrar el nombre de una clave
- $set: Definir la clave que se actualizará (o creará si no existiera)
- $unset: Definir la clave del documento que se eliminará

LOS DOCUMENTOS - UPDATE

- $pull: Eliminar los valores de un array que cumplan el filtro
- $pullAll: Elimina los valores especificados de un array
- $pop: Elimina el primer o último valor de un array
- $push: Añade un elemento a un array
- $addToSet: Añade elementos a un array si no existen
- $each: Se emplea con $addToSet y $push para permitir la agregación múltiple de elementos a un array

LOS DOCUMENTOS - UPDATE

```
use openWebinars

db.articulos.update({
       título:‘Aprendiendo MongoDB',
},{
      $push: {tags :{$each:[‘eLearn' , 'MongoDB‘, ‘mongo’]}}});

db.artículos.update({
       título:‘Aprendiendo MongoDB',
},{
      $push: {tags:’bases de datos’}});

db.artículos.update({
       título:‘Aprendiendo MongoDB',
},{
      $set: {comentarios:’23’}});
```

EL MÉTODO SAVE

- Es útil para crear un documento si no existe o actualizarlo si existe

```
db.artículos.save({
      título: ‘Aprendiendo a usar save',
});

db.artículos.find({titulo:‘Aprendiendo a usar save'})
Apuntamos el _id del documento

Db.artículos.save({
      _id: ObjectId(`Número apuntado`),
      título: ‘Aprendiendo a usar save',
      Contenido : ‘En este curso vamos a aprender las funcionalidades
básicas de mongoDB',
      tags : [‘eLearn' , 'MongoDB']
})
```

3.

USUARIOS Y ROLES

El conocimiento es una herramienta, y como todas las herramientas, su impacto está en manos del usuario.

LA SEGURIDAD EN MONGODB

- MongoDB es una **base de datos insegura por defecto**
- Debemos activar el control de accesos
  - mongod --auth
- Pueden definirse roles a los que se les permita realizar determinadas acciones sobre determinados recursos

CREACIÓN DE USUARIOS

- Es requisito acceder a la base de datos de administración

```
use admin
```

- Creamos el usuario especificando qué roles o permisos adquiere en según qué BBDD o colección

```
db.createUser({
  user: "myUserAdmin",
  pwd: "abc123",
  roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
 })
```

- Nos autenticamos

```
db.auth(“user”,”pass”)
```

USUARIOS Y ROLES

Roles pre-definidos más empleados:

- Usuario db
  - read - readWrite
- Administrador db
  - dbAdmin – userAdmin - dbOwner
- Cluster admin
  - clusterAdmin - clusterManager
- Todas las bases de datos
  - readWriteAnyDatabase – userAdminAnyDatabase - dbAdminAnyDatabase
- Root

USUARIOS Y ROLES

- Los privilegios de cada rol se definen explícitamente o se heredan
- Un privilegio se define como las acciones permitidas sobre un determinado recurso
- Un recurso es una base de datos, colección, grupo de colecciones o un cluster
- La definición de un rol puede incluir a uno o más roles distintos para especificar que se heredan sus privilegios
- Los privilegios de un rol pueden verse con el comando rolesInfo

ROLES - RECURSOS

- Colección de una base de datos
  { db: “dispositivos", collection: “wearable" }
- Base de datos
  { db: “dispositivos", collection: “" }
- Colección recurrente en bases de datos
  { db: “", collection: “users" }
- Todas las colecciones y bases de datos (que no sean propias del sistema)
  { db: “", collection: “" }
- Cluster
  { cluster : true }

USUARIOS Y ROLES

```
use admin
db.createRole(
   {
     role: "manageOpRole",
     privileges: [
       {  resource: { db: "", collection: "" },
          actions: [ "changePassword"] }
     ],
     roles: [ ]})
```

USUARIOS Y ROLES

```
db.revokeRolesFromUser(
    "reportsUser",
    [
      { role: "readWrite", db: "accounts" }
    ]
)

db.grantRolesToUser(
    "reportsUser",
    [
      { role: "read", db: "accounts" }
    ]
)

db.changeUserPassword("reporting", "SOh3TbYhxuLiW8ypJPxmt1oOfL")
```

USUARIOS Y ROLES - SEGURIDAD BÁSICA

- KISS - Keep It Simple Stupid!
- La seguridad es tan fuerte como lo sea el eslabón más débil de la cadena
- Mantener siempre los mínimos permisos posibles para la operativa normal de cada usuario

4.

DISEÑO Y ARQUITECTURA

OpenWebinars

A MONGO LE VALE TODO

Una de las ventajas de MongoDB es la agilidad y flexibilidad

TIPOS DE DATOS

- String
- Integer
- Boolean
- Double
- Timestamp
- Date
- Null
- Array
- Object
- ObjectId
- Binarios
- Javascript

¿EMBEBER O REFERENCIAR?

- Embeber: Añadir documentos enteros dentro de documentos
- Referenciar: Añadir _id (“*primary key*”) dentro de documentos que referencien a otro documento

¿EMBEBER O REFERENCIAR?

La decisión depende del tipo de relación y del crecimiento de las mismas, en definitiva, de la aplicación.

DE SQL A NOSQL - 1 A 1

Queremos relacionar un artículo del blog al autor del mismo:

```
Articulo = {
      _id: ‘ID_DEL_ARTICULO_1’,
      título: ‘Relaciones 1 a n en SQL para NoSQL’,
      Contenido: ‘Vamos a hacer una relación 1 a n en mongoDB’,
      tags: [‘Relaciones’, ‘1 a n’],
      autor: {
            usuario: ‘pcampos’,
            fechaDeRegistro: ’10/10/10’,
            articulosEscritos: ‘16’
      }
}
```

DE SQL A NOSQL - 1 A N

Queremos relacionar un artículo del blog a sus comentarios:

```
Articulo = {
      _id: ‘ID_DEL_ARTICULO_1’,
      titulo: ‘Relaciones 1 a n en SQL para NoSQL’,
      Contenido: ‘Vamos a hacer una relación 1 a n en mongoDB’,
      tags: [‘Relaciones’, ‘1 a n’],
      comentarios: [{
            usuario: ‘pcampos’,
            comentario: ‘Gran blog!’
      },{
            usuario: ‘smario’,
            comentario: ‘Mamma mia!’
      },{
            usuario: ‘Paco’,
            comentario: ‘El link está roto!’
      }]
}
```

DE SQL A NOSQL - 1 A N

Queremos relacionar un artículo del blog a sus comentarios:

```
articulo = {
      _id: ‘ID_DEL_ARTICULO_1’,
      titulo: ‘Relaciones 1 a n en SQL para NoSQL’,
      Contenido: ‘Vamos a hacer una relación 1 a n en mongoDB’,
      tags: [‘Relaciones’, ‘1 a n’],
      comentarios: [‘ID_COMENTARIO_1’,
                    ’ ID_COMENTARIO_2’,
                    ’ ID_COMENTARIO_3’]}

comentario = {
      _id: ‘ID_DEL_COMENTARIO’,
      idDelArticulo: ‘ID_DEL_ARTICULO_1’,
      usuario: ‘pcampos’,
      comentario: ‘Buen articulo!’}
```

DE SQL A NOSQL - N A N

Queremos relacionar varios artículos del blog a sus coautores:

```
Articulo = {
      _id: ‘ID_DEL_ARTICULO_1’,
      titulo: ‘Relaciones 1 a n en SQL para NoSQL’,
      Contenido: ‘Vamos a hacer una relación 1 a n en mongoDB’,
      tags: [‘Relaciones’, ‘1 a n’],
      autores: [‘ID_DEL_AUTOR_1’, ‘ID_DEL_AUTOR_2’, ‘ID_DEL_AUTOR_3’]
}

Autor = {
      _id: ‘ID_DEL_AUTOR_1’,
      usuario: ‘pcampos,
      artículos: [‘ID_DEL_ARTICULO_1’, ‘ID_DEL_ARTICULO_2’]
}
```

CONSIDERACIONES GENERALES

- Añade a un documento toda la información que puedas, ahorrarás *"JOINS"*
- Si un *subdocumento* se repite en nuestra aplicación, es mejor referenciar
- Si una lista de *subdocumentos* va a crecer indiscriminadamente, mejor referenciar
- Los documentos están limitados a 16MB
- Si se anida mucha información en un mismo documento, la búsqueda es más ineficiente
- La atomicidad de las operaciones es a nivel de documento en MongoDB

CONSIDERACIONES GENERALES PRE-LOCALIZACIÓN

- Cuando conocemos la estructura final que tendrá un documento (*time-series*) podemos pre-localizar el documento y evitar la relocalización en disco de la información. Únicamente será necesario actualizar nuestro documento.

```
{
  timestamp_hour: ISODate("2013-10-10T23:00:00.000Z"),
  type: “memory_used”,
  values: {
    0: { 0: 999999, 1: 999999, …, 59: 1000000 },
    1: { 0: 2000000, 1: 2000000, …, 59: 1000000 },
    …,
    58: { 0: 1600000, 1: 1200000, …, 59: 1100000 },
    59: { 0: 1300000, 1: 1400000, …, 59: 1500000 }}}
```

CONSIDERACIONES GENERALES ATOMICIDAD

- Todas las operaciones en mongo son atómicas a nivel de documento
- Si es necesario modificar un documento y otro documento al que este haga referencia, las operaciones deben hacerse por separado (*two phase commit*)
- Siempre que sea posible debe guardarse toda la información relevante y sujeta de ser modificada dentro de un documento, si la aplicación soporta actualizaciones no atómicas, la información puede separarse en dos documentos

CONSIDERACIONES GENERALES
COLECCIONES VOLÁTILES

- Índices TTL -> se explicarán en el módulo de índices
  Capped collections -> Colecciones con un tamaño máximo definido
- Mantienen el orden de inserción
  Funciona como una cola FIFO
  Si van a actualizarse documentos en una capped collection, es necesario definir un índice para no escanear la colección entera
  No pueden “shardearse”

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

5.

ÍNDICES

¿QUÉ ES UN ÍNDICE?

- Los índices son una estructura especial de datos que almacenan una pequeña porción de los datos de una manera fácil y rápida de recorrer.
- Un índice almacena el valor de un campo específico de cualquier colección.
- Los índices se almacenan en un espacio de memoria de rápido acceso.

Los índices apoyan la resolución eficiente de las queries

¿CÓMO USAR LOS ÍNDICES?

- Los índices son capaces de aumentar el rendimiento de resolución de queries en gran medida, pero abusar de estos mermará el rendimiento en memoria de la máquina que albergue los procesos
- Es necesario encontrar una solución de compromiso, definir nuestra estrategia de indexamiento

¿CÓMO USAR LOS ÍNDICES?

- La mejor estrategia es definir un índice para cada campo de nuestras colecciones que usemos frecuentemente en nuestra aplicación o servicio
- Si un índice no se usa a menudo, es mejor eliminarlo

¿QUÉ TIPOS DE ÍNDICES EXISTEN?

- _id: Se genera automáticamente por parte de MongoDB para cualquier documento
- Sencillo: Se definen sobre un único campo de cualquier documento
- Compuesto: Se definen sobre varios campos de un documento
- Multillave: Se emplean cuando el campo a indexar pertenece a un subdocumento dentro de un array del documento padre

¿QUÉ TIPOS DE ÍNDICES EXISTEN?

- Geoespacial: Se definen para indexar campos relativos a coordenadas de tipo GeoJSON
- Texto: Se definen para indexar el contenido de los campos y poder buscar ágilmente dentro del contenido de la base de datos
- Hash: Se emplean para almacenar los valores hasheados del campo a indexar cuando aplicamos sharding sobre las bases de datos

¿QUÉ PROPIEDADES TIENEN LOS ÍNDICES? - UNICIDAD

- Se define sobre índices que queremos que funcionen como una *primary key*
- No puede aplicarse sobre una colección que ya contenga datos que violen la norma
- Pueden crearse índices únicos: Simples o compuestos
- Podrán insertarse documentos en la colección si no contienen el campo al que el índice se refiera

```
db.usuarios.createIndex({ “email” : 1 },{ unique : true })
```

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

¿QUÉ PROPIEDADES TIENEN LOS ÍNDICES? - SPARSE

- Los índices pueden ser dispersos (*sparse*) para permitir filtrar los resultados que se obtienen buscando por un índice y no mostrando aquellos que no tienen el campo

```
{ “_id” : “1”, “user_id” : “antonio”, “horas” : 16}
{ “_id” : “2”, “user_id” : “juan”, “horas” : 12}
{ “_id” : “3”, “user_id” : “pablo”}

db.developers.ensureIndex( { horas : 1 }, { sparse : true } )

db.developers.find( { horas : { $lt : 16 } } )

db.developers.find().sort({ horas : -1 })
db.developers.find().sort({ horas : -1 }).hint({ horas : 1 })
```

¿QUÉ PROPIEDADES TIENEN LOS ÍNDICES? - ÍNDICES PARCIALES

- Se define una condición y solo se indexa la parte de la muestra que la cumpla
- Es incompatible con la propiedad sparse

```
db.delanteros.createIndex({
      nombre : 1, equipo : 1 },{
      partialFilterExpression : { goles : { $gt : 10 } }})
```

¿Qué ocurre si filtramos para delanteros con más de 12 goles?
¿Qué ocurre si filtramos para delanteros con menos de 15 goles?
¿Si buscamos delanteros del Betis con más de 12 goles usamos el índice?
¿Si buscamos a Messi del Barcelona usamos el índice?

¿QUÉ PROPIEDADES TIENEN LOS ÍNDICES? - ÍNDICES PARCIALES

- La condición de ser único solo debe cumplirse para aquellos documentos que cumplan con el filtro parcial

```
db.delanteros.createIndex({
      nombre : 1 },{
      unique : true,
      partialFilterExpression : { goles : { $gt : 10 } }})
```

```
{ “_id” : “1”, “nombre” : “Rubén”, “goles” : 16}
{ “_id” : “2”, “nombre” : “Lionel”, “goles” : 12}
{ “_id” : “3”, “nombre” : “Cristiano”, “goles” : 11}
```

¿Qué ocurre si introducimos a Rubén con 19 goles?
¿Qué ocurre si introducimos a Cristiano con 10 goles?
¿Qué ocurre si introducimos a Rubén sin campos goles?

¿QUÉ PROPIEDADES TIENEN LOS ÍNDICES? - TTL

- Se asigna un campo de *expireAfterSeconds* o *expireAt* y el propio motor de mongo eliminar el documento transcurrido ese tiempo

```
db.eventsLog.ensureIndex( { “createdAt” : 1 }, { expireAfterSeconds : 18000 } )
db.eventsLog.insert({
      “createdAt” : new Date(),
      “evento” : 4})
db.runCommand({"collMod":"log_events","index":{keyPattern:{createdAt:1},expireAft
erSeconds:5}})

db.eventsLog.ensureIndex( { “expireAt” : 1 }, { expireAfterSeconds : 0 } )
db.eventsLog.insert({
      “expireAt” : new Date(‘December 25, 2016 15:00:00’),
      “evento” : 4})
```

ÍNDICES SIMPLES

Los índices simples se definen sobre un único campo de un documento

```
Artículo = {
      _id: ‘ID_DEL_ARTICULO_1’,
      titulo: ‘Relaciones 1 a n en SQL para NoSQL’,
      Contenido: ‘Vamos a hacer una relación 1 a n en mongoDB’,
      tags: [‘Relaciones’, ‘1 a n’],
      comentarios: [{
            usuario: ‘pcampos’,
            comentario: ‘Gran blog!’},
      {     usuario: ‘smario’,
            comentario: ‘Mamma mia!’},
      {     usuario: ‘Paco’,
            comentario: ‘El link está roto!’
      }],
      shares: 19}
```

ÍNDICES SIMPLES

Para crear un índice usamos el comando ensureIndex

```
db.artículos.ensureIndex({ shares : 1 })

db.artículos.find({ shares : {$gt:10} })
```

¿Y si queremos indexar un campo embebido?
¿Y si queremos indexar un documento embebido?
¿Y si queremos ordenar (sort) usando índices?

ÍNDICES COMPUESTOS

- Para crear un índice usamos el comando ensureIndex
- Está limitado por mongoDB a 31 campos
- Para buscar y ordenar, el orden de los campos altera el rendimiento de la query

```
db.artículos.ensureIndex({ titulo : 1, contenido : 1 })

db.artículos.find({ titulo : “LOQUESEA” })
db.artículos.find({ titulo : “LOQUESEA”, contenido : “ELQUESEA” })
db.artículos.find({ contenido : “LOQUESEA”, titulo : “ELQUESEA” })
```

ÍNDICES COMPUESTOS

- Los prefijos se definen como partes del todo de un índice compuesto completo
- El orden en el que se defina el índice es relevante y de gran importancia para el rendimiento de la base de datos

```
db.artículos.ensureIndex({ stock : 1, udsVendidas : 1, precio : 1 })
Prefijo 1 -> { stock : 1}
Prefijo 2 -> { stock : 1, udsVendidas : 1}
```

ÍNDICES COMPUESTOS

- Usando índices compuestos pueden realizarse operaciones de ordenación usando prefijos completos o partes de prefijos
- Si se usa el prefijo completo, debe especificarse en el orden que se declaró para que sea usado

```
db.artículos.find().sort({ stock:1, udsVendidas:1 })
```

- Si se usa parte de un prefijo, será necesario realizar alguna operación de igualdad para las claves que precedan a la parte del prefijo

```
db.artículos.find({stock:20}).sort({udsVendidas:1})
```

ÍNDICES MULTILLAVE

- Para crear un índice usamos el comando ensureIndex
- Puede construirse sobre arrays que contengan cadenas, números u otros documentos
- No es posible construir un índice compuesto de índices multillave, pero si un índice compuesto de uno simple y uno multillave

```
db.artículos.ensureIndex({ tags : 1 })

db.artículos.find({tags:'eLearn'})
```

ÍNDICES MULTILLAVE

- Pueden realizarse queries para buscar un array específico pero el índice multillave no será usado en su totalidad

```
db.películas.find( { ratings: [ 5, 9 ] } )
```

- Pueden definirse índices multillave para campos contenidos en un objeto dentro de un array

```
{ _id: ID,
  item: “NOMBRE",
  stock: [
      { size: “TALLA", color: “COLOR", quantity: CANTIDAD },
      { size: “TALLA", color: “COLOR", quantity: CANTIDAD },
      { size: “TALLA", color: “COLOR", quantity: CANTIDAD },  ]}
```

ÍNDICES DE TEXTO

- Facilitan y optimizan las búsquedas basadas en texto en gran medida
- Se declaran igual que el resto de índices con la particularidad de no ponerse a uno el valor, si no a “text”
- Pueden indexarse varios campos con un índice de texto
- Solo puede declararse un índice text por colección

```
db.artículos.createIndex({ titulo : “text” })

db.artículos.createIndex({ titulo : “text”, contenido : “text” })
```

ÍNDICES DE TEXTO

Para búsquedas de texto podemos definir un peso relativo a cada uno de los campos de texto que se indexan, de esta forma se ordenará la búsqueda según nuestro criterio.

```
db.artículos.createIndex({
      titulo : “text”, contenido : “text”
},{
      weights: { titulo : 10, contenido : 5 },
      name: “TextIndex”
})
```

ÍNDICES DE TEXTO

Si se quiere indexar todo el contenido de texto de una colección se emplea el wildcard specifier.
Pueden declararse pesos para cualquier campo indexado

¿Por qué no declara siempre el wildcard specifier e indexar todo el texto?

```
db.collection.createIndex( { "$**": "text" })
```

ÍNDICES DE TEXTO

- La tokenización de las palabras está resuelta incluso con símbolos (desde la versión 3 de mongo)
- Mongo cuenta con varios lenguajes predefinidos, así como stop words
- Los índices de texto son sparse por defecto

```
db.artículos.createIndex({
      contenido : “text”
},{
      default_language: “spanish”})

db.artículos.createIndex({
      contenido : “text”
},{
      language_override: “idioma”})
```

ÍNDICES DE TEXTO

- La tokenización de las palabras está resuelta incluso con símbolos (desde la versión 3 de mongo)
- Mongo cuenta con varios lenguajes predefinidos, así como stop words
- Los índices de texto son sparse por defecto

```
{
  $text:
    {
      $search: <string>,
      $language: <string>,
      $caseSensitive: <boolean>,
      $diacriticSensitive: <boolean>
    }
}
```

ÍNDICES DE TEXTO

- Buscar una palabra
- Buscar por varias palabras
- Buscar una frase
- Buscar por una palabra excluyendo otra
- Buscar por idioma
- Buscar con sensibilidad a mayúsculas y minúsculas
- Buscar y recuperar la puntuación del resultado

ÍNDICES DE TEXTO

Si son tan útiles, ¿por qué no empleamos siempre la wildcard e indexamos todo?

- Son extensos
- Son lentos de construir
- Son costosos en memoria
- La búsqueda de frases es más efectiva sin índices

ÍNDICES - ALGO MÁS SOBRE ELLOS

- Para no dejar sin servicio a la aplicación, cuando construir un índice vaya a ser muy costoso, hacerlo en background

```
db.COLECCION.createIndex( { CAMPO: 1}, {background: true} )
```

- Es posible analizar el rendimiento de nuestras queries y compararlas empleando el comando explain() con la opción executionStats

```
db.inventory.find(
   { quantity: { $gte: 100, $lte: 200 } }
).explain("executionStats")
```

ÍNDICES - ALGO MÁS SOBRE ELLOS

- Debemos intentar usar las **covered queries** siempre que sea posible: Son queries dónde **todos los campos de filtro son un índice** y donde solo se proyecta en el resultado dichos campos.

ÍNDICES - ALGO MÁS SOBRE ELLOS

- Es posible analizar el rendimiento de nuestras queries y compararlas empleando el comando explain() con la opción executionStats

```
{ "_id" : 1, "item" : "f1", type: "food", quantity: 500 }
{ "_id" : 2, "item" : "f2", type: "food", quantity: 100 }
{ "_id" : 3, "item" : "p1", type: "paper", quantity: 200 }
{ "_id" : 4, "item" : "p2", type: "paper", quantity: 150 }
{ "_id" : 5, "item" : "f3", type: "food", quantity: 300 }
{ "_id" : 6, "item" : "t1", type: "toys", quantity: 500 }
{ "_id" : 7, "item" : "a1", type: "apparel", quantity: 250 }
{ "_id" : 8, "item" : "a2", type: "apparel", quantity: 400 }
{ "_id" : 9, "item" : "t2", type: "toys", quantity: 50 }
{ "_id" : 10, "item" : "f4", type: "food", quantity: 75 }
```

¿Qué índices usarías para buscar tipo food de entre 100 y 300 de calidad?¿Por qué?¿Estabas en lo cierto?

6.

AGREGACIÓN

AGREGACIÓN

- La agregación permite realizar queries “anidadas” que incluyan operaciones
- Se desarrollan como la tunelación el Linux, siguiendo el orden especificado
- Si se usa MongoDB como motor de base de datos, es más eficiente emplear los métodos de agregación por tunelación que mapReduce
- Map-Reduce se ejecuta partido en dos fases
  - Mapeo de todos los documentos para emitir los objetos deseados a la siguiente fase
  - Fase de “reducción” que se encarga de operar con los datos recibidos de la primera fase

AGREGACIÓN

- Operaciones de agregación simple
  - Count: Devuelve un entero que indica el número de documentos que cumplesn con la query realizada
  - Group: Agrupa todos los documentos de una colección según el campo que se indique… Igual que GROUP BY
  - Distinct: Devuelve los distintos campos únicos que existan para el que se especifique

```
db.COLECCION.count(QUERY)
db.COLECCION.distinct(“CAMPO”)
db.COLECCION.group{key, reduce, initial} ()
```

AGREGACIÓN

- Mongo aprovecha los índices durante las operaciones de agregación para los operadores $match y $sort
- MongoDB optimiza solo las operaciones de agregación para que sean más eficientes cuando detecta un error común

OPERACIONES

- $project -> Proyecta como salida los campos especificados de todos los obtenidos en la query anterior
- $match -> Filtra documentos según el criterio que se especifique, como en un find()
- $limit -> limita los resultado al número de documentos especificado
- $skip -> Se salta el número de documentos indicados del resultado de la operación anterior

OPERACIONES

- $unwind -> Deconstruye un el array de un documento y construye muchos documentos que tienen un solo valor en el campo en el que había un array
- $group -> Agrupa el resultado en base al campo que se indique
- $sort -> Ordena los resultados
- $lookup ->Permitie realizar operaciones JOIN SQL en base de datos mongoDB

OPERACIONES

- $out -> Debe ser el último operador y permite especificar una colección en la que guardar el resultado obtenido
- $and, $or y $not -> Operadores booleanos
- $cmp -> 0 si los dos valores son iguales y 1 en caso contrario
- $eq -> true si los valores son equivalentes
- $gt, $gte, $lt, $lte, $ne -> Mayor que, mayor que o igual…
- $in, $nin -> contenido en, no contenido en

OPERACIONES

- $sum -> Suma los valores numéricos del campo especificado (usarlo en group)
- $avg -> Valor medio del campo especificado (group)
- $first, $last, $max, $min
- Operadores aritméticos: $divide, $sqrt, $exp…
- Operadores de fechas: $week, $hour, $year…

OPTIMIZACIÓN

- $match siempre antes que $sort para minimizar el número de objetos que ordenar
- Cuando ponemos $skip antes de $limit, Mongo los reordena al revés (cambiando los valores $limit=$limit + $skip) para permitir la fusión de los comandos $sort y $limit
- Cuando Mongo detecta que dos comandos pueden fusionarse, lo hace automáticamente de manera interna y transparente

OPTIMIZACIÓN

- Cuando ponemos $project seguido de $skip o $limit, mongo adelanta a los segundos para que sea una proyección de menos resultados o para que si hay un $sort delante pueda fusionarse
- Mongo fusiona dos $limit seguidos
- Mongo fusiona dos $skip seguidos

OPTIMIZACIÓN

- Cuando ponemos dos $match seguidos, es más eficaz realizar la misma operación en uno solo contando con el operador lógico $and
- Cuando ejecutemos un $lookup seguido de un $unwind podemos optimizarlo poniendo el $unwind en el propio $lookup

```
{
  $lookup: {
    from: "otherCollection",
    as: "resultingArray",
    localField: "x",
    foreignField: "y"
  }
},
{ $unwind: "$resultingArray"}
```

```
{
  $lookup: {
    from: "otherCollection",
    as: "resultingArray",
    localField: "x",
    foreignField: "y",
    unwinding: { preserveNullAndEmptyArrays:
false }
  }
}
```

MAR-REDUCE

- Menos eficiente que aggregate()
- Se define la fase de mapeo, la de reducción, la posible query y la posible colección de salida para el resultado

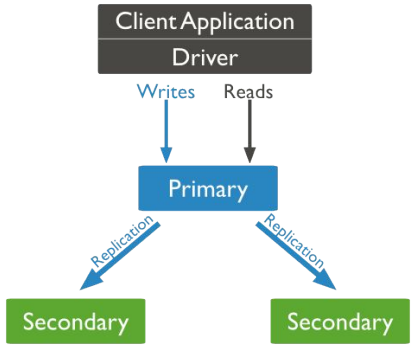
```
db.pedidos.mapReduce(
      function(){ emit( this._id, this.precio ) },
      function(key, values ){ return Array.sum( value ) },
      {
            query: { categoría : “limpieza” },
            out: “totales”
      }
)
```

7.

RÉPLICA SETS

REPLICA SET

- Provee “sólo” de redundancia y disponibilidad
- Nada de balanceo de carga
- Es un grupo de instancias mongod configuradas para replicar el contenido de determinadas bases de datos según las entradas del log de un servidor primario



REPLICA SET

- Solo el nodo primario recibe las operaciones de escritura
- Los nodos secundarios replican las operaciones que realiza el nodo primario cuando este las anota en el log de operaciones de mongo
- Si un nodo primario cae, los nodos secundarios deciden quien juega el papel de primario desde ese momento

REPLICA SET

- Pueden definirse instancias de mongod como árbitros que no replican información, únicamente tienen voto a la hora de elegir un nuevo nodo primario
- Los nodos árbitro nunca cambian de estado hacia otro tipo de nodo

REPLICA SET

- Si el nodo primario del cluster se cae o pierde la comunicación durante 10 segundos, se le considera caído y se pasa a buscar un nuevo nodo primario
- El proceso de caída/reelección dura aproximadamente 1 minuto
  - 10-30 segundos caída del nodo primario y comunicación de la misma
  - 10-30 segundos elecciones

REPLICA SET

- Por defecto se realizan las operaciones de lectura sobre el nodo primario
- Puede configurarse un cluster con un readConcern que especifique que se lea de nodos secundarios cuando sea necesario
  - Primary
  - primaryPreferred
  - Secondary
  - secondaryPreferred
  - Nearest
- CUIDADO con la replicación asíncrona

```
db.getMongo().setReadPref('primaryPreferred')
```

REPLICA SET

- Miembros con prioridad 0: sets de réplica secundarios que no pueden ser elegidos como primarios
  - Aceptan operaciones de lectura
  - Mantienen una copia de los datos
  - Votan en las elecciones
  - No pueden ser elegidos
  - No pueden lanzar unas elecciones
  - Pueden configurarse en stand-by

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

REPLICA SET

- Miembros ocultos: sets de réplica secundarios que no pueden ser elegidos como primarios ni recibir operaciones de lectura
  - No aceptan operaciones de lectura
  - Mantienen una copia de los datos
  - Votan en las elecciones
  - No pueden ser elegidos
  - No pueden lanzar unas elecciones

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

REPLICA SET

- Miembros “retrasados”: sets de réplica secundarios que guardan el estado del nodo primario con un retraso de tiempo definido
  - No aceptan operaciones de lectura
  - Mantienen una copia de los datos
  - Votan en las elecciones si se especifica
  - No pueden ser elegidos
  - No pueden lanzar unas elecciones
  - Deben ser hidden y priority 0
  - Su “retraso” debe ser más pequeño que la capacidad del opLog

```
{
   "_id" : <num>,
   "host" : <hostname:port>,
   "priority" : 0,
   "slaveDelay" : <seconds>,
   "hidden" : true
}
```

REPLICA SET

- El log de operaciones es una capped collection que mantiene un registro de las operaciones de la BBDD
- 5% de la capacidad libre del disco {990MB-50GB}
- CUIDADO con las actualizaciones “masivas”
- CUIDADO con hacer muchas pequeñas actualizaciones

```
rs.printReplicationInfo()
```

REPLICA SET

- Sincronización inicial
  - Clona todas las bases de datos y construye los índices _id
  - Refleja los nuevos cambios del opLog a la nueva instancia
  - Construye el resto de índices
  - La instancia pasa a ser una instancia secundaria
- Se sincronizan las réplicas por batches
- Los miembros secundarios que tengan voto nunca copian de un miembro que no tenga voto
- Los miembros secundarios nunca copian de un miembro “retrasado”
- Pueden darse rollbacks de miembros primarios que reconectan tras una caída

REPLICA SET

- Un set de réplicas sólo puede tener 7 miembros con voto
- Un set de réplicas sólo puede tener 50 miembros en total
- SIEMPRE es recomendable configurar un número impar de miembros con voto para evitar los empates
  - En caso de tener un número par de votantes, mejor añadir un miembro árbitro (No requiere mucho espacio)

REPLICA SET

- Considerar la tolerancia al fallo
  - Diferencia entre el número de miembros y el número de votos requeridos para elegir un primario
- Balancea la carga de lecturas si el número de operaciones es alto

¿Qué posibilidades hay para montar un set de 3 miembros?

8.

SHARDING

¿QUÉ ES?

- Método que permite distribuir datos a través de múltiples máquinas
- Se emplea para arquitecturas y aplicaciones consolidadas, cuando el tamaño de la base de datos y el volumen de operaciones crece demasiado para mantenerlo en una sola máquina (CPU y RAM)
- El escalado vertical puede ser útil si la máquina que alberga la instancia mongoDB no era ya demasiado potente
- El escalado horizontal (sharding) permite dividir el sistema en varios servidores y balancear la carga de trabajo

¿QUIÉNES ACTÚAN?

- Un cluster de instancias mongoDB fragmentado siempre tiene implicado a tres tipos de actores:
  - Fragmentos (*shards*): Cada fragmento contiene parte del total de datos que almacena la base de datos y puede desplegarse como un set de réplicas a su vez
  - Routers (*mongos*): Son la interfaz que une a los clientes con los distintos fragmentos
  - Servidores de configuración: Servidores que almacenan los metadatos necesarios para que el cluster opere con normalidad

¿CÓMO FUNCIONA?

- Entendiendo el sharding con una baraja de cartas
  - 48 cartas - 4 palos
  - Cuatro jugadores y un repartidor
  - Cada jugador tendrá un máximo de 12 cartas y el repartidor mirará cada carta antes de repartirla

- Los jugadores son los shards
- Las cartas son documentos
- El repartidor es la instancia mongos (router)
- La shard key será el número de carta

¿CÓMO FUNCIONA?

- El criterio que define la distribución de los documentos entre los distintos shards lo define la shard key
- La shard key es uno o varios campos que existen en TODOS los documentos de la colección
- La shard key se especifica antes de fragmentar la colección y no puede cambiarse después

¿CÓMO FUNCIONA?

- Para hacer un sharding de colecciones con datos ya insertados es imprescindible tener un índice definido cuyo primer campo sea la shard key
- La elección de la mejor shard key posible es imprescindible para el buen rendimiento del cluster
- Cada “rango” de shard keys ubicadas en un fragmento se denomina chunk

¿CÓMO FUNCIONA?

- El sharding balancea la carga de lecturas/escrituras y distribuye el trabajo de la base de datos por todo el cluster
- La capacidad de almacenamiento de mongoDB se distribuye en el cluster y el crecimiento del mismo es muy fácilmente implementable
- Si un shard se cayera, las operaciones de lectura/escritura de los shards que sigan activos seguirán realizándose

¿CÓMO FUNCIONA?

- El sharding requiere un estudio previo de necesidades para ser eficientes en la elección de la shard key así como la configuración del cluster
- Si se realiza una query a una colección fragmentada y se filtra por un campo que no esté incluido en la shard key (o en su índice compuesto), el mongos realizará una query broadcast a todos los shard
- La elección de la shard key es de vital importancia puesto que no puede modificarse una vez escogida

¿CÓMO FUNCIONA?

- Para una misma base de datos pueden tenerse colecciones fragmentadas y no fragmentadas
- En cualquier caso será necesario acceder a través de ellas a través de un mongos, aunque no esté fragmentada

ESTRATEGIAS DE DISTRIBUCIÓN

- MongoDB admite dos estrategias de distribución de datos basado en shard keys
  - Hashed keys: La distribución por shards se realiza en base al hash del campo que se escoja
  - Ranged sharding: La distribución se realiza en base a rangos definidos en base al índice escogido

ESTRATEGIAS DE DISTRIBUCIÓN

- La arquitectura mínima recomendada para producción con sharding incluye:
  - 3 servidores de configuración como set de réplicas
  - Cada shard como un set de réplicas de 3 miembros cada uno
  - Un router mongos como mínimo

SHARD PRIMARIO

- Cada base de datos que implemente sharding tiene un shard primario definido en el que se almacenan las colecciones no fragmentadas

```
db.runCommand({movePrimary:<databaseName>,to:<newPrimaryShard>})

sh.status()
```

SERVIDORES DE CONFIGURACIÓN

- Los servidores de configuración almacenan metadatos relativo a los chunks almacenados en cada shard así como la distribución de shard keys
- Cada sharded cluster requiere de su propio servidor de configuración, no puede definirse uno genérico
- Se escribe información en ellos cuando se produce una migración o una división de chunks
- Se lee de ellos cuando se inicia un nuevo mongos (por primera vez o restart) y cuando cambian sus metadatos
- Los metadatos se almacenan en la base de datos config

MONGOS

- Se encarga de enrutar las queries de lectura/escritura al shard que corresponda
- Los router cachean los metadatos acerca de la distribución de los chunk sobre los shards para enrutar
- Primero determina los shard que deben recibir la query y después establece un cursor apuntando al resultado de cada query
- Operaciones como la agregación o la ordenación se ejecutan en el primary shard (u otro) antes de devolver el resultado completo

```
db.isMaster()
```

PUESTA EN MARCHA

- Se define una shard key con el siguiente comando:

```
sh.shardCollection( namespace, key )
```

- Imprescindible definir previamente un índice si ya existen datos insertados
- Para colecciones fragmentadas solo pueden ser índices únicos el _id y la propia shard key
- Pueden ser un índice simple, compuesto (que empiece con la shard key) o hasheado
- Son inmutables
- Tres parámetros clave para elegir clave: Cardinalidad, frecuencia y monotonía.

PUESTA EN MARCHA

- Si definimos la shard key con un índice hasheado
  - Ganamos en balanceo de carga de datos
  - Perdemos en la eficiencia de las queries
  - Son la solución idónea para keys de gran cardinalidad o muy monótonos en su crecimiento (_id u ISODates)

```
sh.shardCollection( "database.collection", { <field> : "hashed" } )
```

PUESTA EN MARCHA

- Si definimos la shard key con un índice por rangos
  - Los datos se distribuyen de manera menos uniforme
  - Las queries son más óptimas
- Son óptimos para keys distribuidas uniformemente y con mucha cardinalidad

```
sh.shardCollection( "database.collection", { <shard key> } )
```

9.

CLIENTES GRÁFICOS

CLIENTES GRATUITOS

- Robomongo
- Mongoclient
- Mongo-express
- Mongobooster

* Mi recomendación: Aprender con la consola

GRACIAS!

Preguntas?

Podéis encontrarme en @PabGallagher &
pcamposred@gmail.com