

NoSQL

(Not Only SQL)

Introducción a las BBDD noSQL

La forma en que las aplicaciones web tratan los datos, ha cambiado de forma significativa durante la última década.

Cada vez se recopilan más datos y cada vez son más los usuarios que acceden a estos datos al mismo tiempo.

Esto significa que la escalabilidad y el rendimiento se han convertido en auténticos retos para las bases de datos relacionales basadas en esquemas.

Introducción a las BBDD noSQL

Cuando en la década de 1990 los comerciales de bases de datos relacionales vendían sus sistemas, acostumbraban a decir que eran "escalables" en el sentido de que serían capaces de manejar los datos del censo de todo un país, y que aun con esa cantidad ingente de datos se podría consultar la información que se deseara muy rápidamente.

Tenían razón, pero el problema es que seguían pensando en términos de los primeros ordenadores en cuanto al tamaño.

Introducción a las BBDD noSQL

¿Es que es posible imaginar una cantidad de bytes mayor que el censo de un país?

En la época la respuesta era *NO*, pero hoy en día, los teléfonos móviles de una gran compañía generan más datos por minuto que el censo de un país como Estados Unidos (solo los datos de localización ya superan esta cifra, sin contar llamadas, mensajes, etc).

Y esto no una vez cada cierto número de años, como el censo, sino constantemente. Datos que interesa almacenar y tratar, ya que son una valiosa fuente de información.

Introducción a las BBDD noSQL

Otra fuente similar de datos es internet. Incluso las estimaciones más conservadoras asumen que la web contiene actualmente más de 1 Yotta-byte en total: 1.000.000.000.000.000.000.000.000 Bytes!

Por eso cuando las grandes compañías de internet como Google se encontraron con el problema de indexar una cantidad ingente de páginas que además eran modificadas continuamente, los "grandes números" de las bases de datos relacionales se les quedaron muy cortos. Incluso compañías como Twitter se encuentran con alrededor de **500 millones de tweets nuevos al día**. Mensajes que además no son homogéneos ya que pueden incluir contenido multimedia. Este es el mundo Big Data.

Introducción a las BBDD noSQL

Las limitaciones del modelo relacional ha venido desde dos fuentes:

1.- Como hemos visto al hablar de Big Data, las aplicaciones modernas necesitan procesar gran cantidad de datos a gran velocidad. Para colmo puede tratarse de datos heterogéneos, es decir con estructura diferente. El modelo relacional no fue pensado para este contexto: trata de adaptarse, pero no es lo suyo.

Introducción a las BBDD noSQL

2.- Cambios en el hardware

- Aumento del nivel de paralelismo: ordenadores con más núcleos
- Aparición de la llamada "commodity computing era"
- La nube (Cloud)

El primer punto es el "problema", el segundo abre nuevas posibilidades que en particular proporcionan nuevos modos de almacenar y gestionar grandes cantidades de datos. Es decir estas nuevas posibilidades para tener bases de datos escalables sin gran coste. Se trata del **escalado horizontal** o "scale-out" que propone añadir más y más nodos (equipos) según aumentan la cantidad de datos va creciendo

Introducción a las BBDD noSQL

La ventaja de el escalado horizontal es sobre todo el precio: salen más baratos 10 servidores de bajo coste que uno 10 veces más potentes (escalado vertical). Pero hay que dejar claro que el escalado horizontal también tiene sus problemas:

- Cuantos más equipos, más posibilidades de que uno falle en cualquier momento. Esto obliga a replicar la información.
- Además en el clúster todos los equipos deben estar coordinados, esto complica el software y a menudo significa pérdida de tiempo en transferencia de datos entre nodos.

Introducción a las BBDD noSQL

En primer lugar, había tipos de bases de datos NoSQL (de origen cerrado), desarrolladas por grandes empresas para satisfacer sus necesidades específicas, como BigTable de Google, que se cree es el primer sistema NoSQL y DynamoDB de Amazon.

El éxito de estos sistemas patentados, inició el desarrollo de varios sistemas de bases de datos de código abierto y de propietarios similares siendo los más populares Hypertable, Cassandra, MongoDB, DynamoDB, HBase y Redis

Introducción a las BBDD noSQL

Los sistemas de bases de datos NoSQL crecieron con las principales redes sociales, como Google, Amazon, Twitter y Facebook.

Con el crecimiento de la web en tiempo real existía una necesidad de proporcionar información procesada a partir de grandes volúmenes de datos que tenían unas estructuras horizontales más o menos similares. Estas compañías se dieron cuenta de que el rendimiento y sus propiedades de tiempo real eran más importantes que la coherencia, en la que las bases de datos relacionales tradicionales dedicaban una gran cantidad de tiempo de proceso

Introducción a las BBDD noSQL

SQL → lenguaje estructurado de consultas... (MySQL, ORACLE, PostGreSQL)

clave: un Id, un DNI, ... Es un dato que nos identifica quién es, y nos facilita su relaciones.

BBDD no Relacionales: colecciones de datos que se parecen entre sí, pero que no son necesariamente iguales.

clave: podemos tener, pero no la necesitamos.

Introducción a las BBDD noSQL

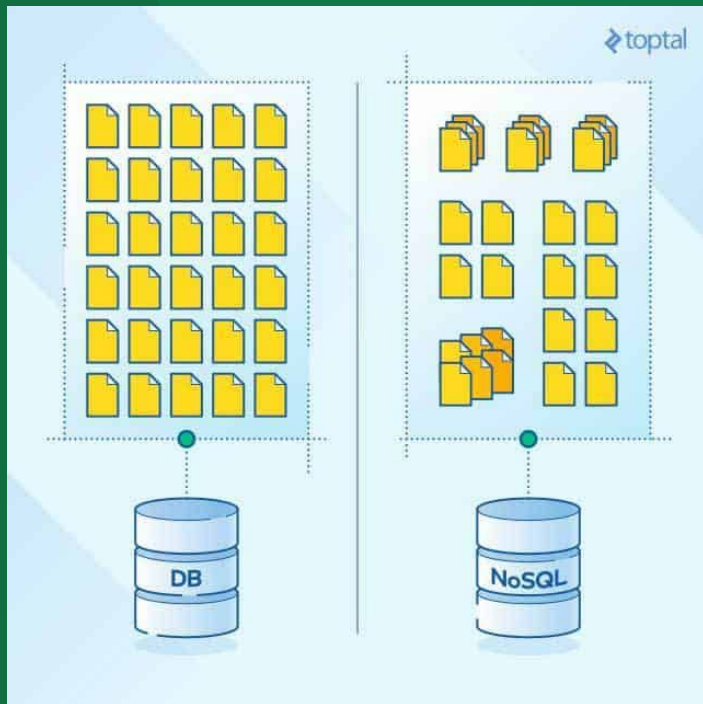
Bajo la denominación de NoSQL se engloba un conjunto de bases de datos que han sido construidas para soportar entornos de aplicación en donde las bases de datos relacionales no constituyen la mejor la solución.

Estos entornos de aplicación tienen dos características principales:

Introducción a las BBDD noSQL

- En primer lugar, se trata de entornos que necesitan disponer de esquemas de datos más flexibles, es decir, se trata de entornos que no se ajustan a la estructuración rígida en forma tabla que ofrecen las bases de datos relacionales.
- Y en segundo lugar, estas BD se utilizan en entornos de aplicación altamente distribuidos que necesitan estar siempre operativos (en línea) y que necesitan gestionar un volumen importante de datos, posiblemente heterogéneos.

Introducción a las BBDD noSQL



Una diferencia clave entre las bases de datos de NoSQL y las bases de datos relacionales tradicionales, es el hecho de que NoSQL es una forma de almacenamiento estructurado pero no fijo.

NoSQL no tiene una estructura de tabla fija como las que se encuentran en las bases de datos relacionales.

Introducción a las BBDD noSQL

Las bases de datos NoSQL están altamente optimizadas para las operaciones recuperar y agregar, y normalmente no ofrecen mucho más que la funcionalidad de almacenar los registros (p.ej. almacenamiento clave-valor).

La pérdida de flexibilidad en tiempo de ejecución, comparado con los sistemas SQL clásicos, se ve compensada por ganancias significativas en escalabilidad y rendimiento cuando se trata con ciertos modelos de datos.

Introducción a las BBDD noSQL

Las implementaciones típicas de SGBDR se han afinado o bien para una cantidad pequeña pero frecuente de lecturas y escrituras o para un gran conjunto de transacciones que tiene pocos accesos de escritura.

Por otro lado NoSQL puede servir gran cantidad de carga de lecturas y escrituras.



50 TB de la búsqueda de la bandeja de entrada de Facebook

Introducción a las BBDD noSQL

Bastantes sistemas NoSQL emplean una arquitectura distribuida, manteniendo los datos de forma redundante en varios servidores.

De esta forma, el sistema puede realmente escalar añadiendo más servidores, y el fallo en un servidor puede ser tolerado.

Ventajas NoSQL

- A diferencia de las bases de datos relacionales, las bases de datos NoSQL están basadas en key-value pairs
- Algunos tipos de almacén de bases de datos NoSQL incluyen por ejemplo el almacenamiento de columnas, de documentos, de key value store, de gráficos, de objetos, de XML , ...
- Las bases de datos NoSQL de código abierto tienen una implementación rentable. Ya que no requieren las tarifas de licencia y pueden ejecutarse en hardware de precio bajo.

Ventajas NoSQL

- La expansión es más fácil y más barata que cuando se trabaja con bases de datos relacionales.



Se realiza un **escalado horizontal** (aumento de máquinas o nodos) y se distribuye la carga por todos los nodos. En lugar de realizarse una escala vertical, más típica en los sistemas de bases de datos relacionales (aumento de capacidad de disco o cambio por un procesador más rápido).

Ventajas NoSQL

- Pueden manejar enormes cantidades de datos.
- No generan cuellos de botella.
- Diferentes DBs NoSQL para diferentes proyectos

Desventajas NoSQL

La mayoría de las bases de datos NoSQL no admiten funciones de fiabilidad, que son soportadas por sistemas de bases de datos relacionales. Estas características de fiabilidad pueden resumirse en: “atomicidad, consistencia, aislamiento y durabilidad.” (ACID)



Las bases de datos NoSQL, que no soportan esas características, ofrecen consistencia para el rendimiento y la escalabilidad.

Desventajas NoSQL

Falta de madurez de la mayoría de los sistemas NoSQL y los posibles problemas de inestabilidad → desconfianza

La falta de experiencia y falta de profesionales

Problemas de compatibilidad.- Las bases de datos relacionales, comparten ciertos estándares, **PERO** las bases de datos NoSQL tienen pocas normas en común. Cada base de datos NoSQL tiene su propia API, las interfaces de consultas son únicas y tienen peculiaridades. Esta falta de normas significa que es imposible cambiar simplemente de un proveedor a otro, por si no quedara satisfecho con el servicio.

Desventajas NoSQL

Con el fin de apoyar las características de fiabilidad y coherencia, los desarrolladores deben implementar su propio código, lo que agrega más complejidad al sistema.

Esto podría limitar el número de aplicaciones en las que podemos confiar para realizar transacciones seguras y confiables, ¿cuáles por ejemplo?

**Sistemas de
reservas**

**Sistemas
bancarios**

**Seguridad Social y
Admon en general**

Desventajas NoSQL

No usan SQL como lenguaje principal de consultas → se necesita un lenguaje de consulta manual, haciendo los procesos mucho más lentos y complejos.

Los datos almacenados no requieren estructuras fijas como tablas, normalmente no soportan operaciones JOIN

Funcionalidades NoSQL vs. BBDDRR

| Feature | NoSQL Databases | Relational Databases |
|--------------|-------------------------|---------------------------|
| Performance | High | Low |
| Reliability | Poor | Good |
| Availability | Good | Good |
| Consistency | Poor | Good |
| Data Storage | Optimized for huge data | Medium sized to large |
| Scalability | High | High (but more expensive) |

Esta tabla muestra una comparación a nivel de la base de datos, no sobre los diversos sistemas de gestión de bases de datos que implementan ambos modelos. Estos sistemas proporcionan *sus propias técnicas patentadas* para superar los problemas y deficiencias encontradas en el sistema, además de intentar mejorar significativamente el rendimiento y la fiabilidad.

Tipos de almacenamiento de datos NoSQL

Key Value Store: se utiliza una tabla hash en la que una clave única apunta a un elemento.

Todo lo que se necesita para hacer frente a los elementos almacenados en la base de datos: es la clave. Los datos se almacenan en forma de una cadena, JSON.

USO: Aplicaciones que sólo utilizan consulta de datos por un solo valor de la clave

DynamoDB de Amazon

Tipos de almacenamiento de datos NoSQL

Almacenes de documentos: son similares a los almacenes de valores clave, porque no tienen un esquema y se basan en un modelo de nombreCampo:Dato

- “Intuitivo” → representa la información tal cual.
- Manera natural de modelar datos cercana a la programación orientada a objetos
- Flexibles, con esquemas dinámicos
- Reducen la complejidad de acceso a los datos.

Tipos de almacenamiento de datos NoSQL

```
{  
  título: "Doctor",  
  Nombre: "Pedro",  
  Edad: 37,  
  City: "Madrid",  
  PerfilFB: "https://fb.com",  
  Amigos: [ {nombre: "Maria", edad: 28}, {nombre: "Luis", alias: "ito"} ],  
  Aficiones: [ "leer", "tennis", "viajar" ]  
}
```

Tipos de almacenamiento de datos NoSQL

En el almacén de documentos, los valores (documentos) proporcionan codificación XML, JSON o BSON (JSON codificado binario) para los datos almacenados.

USO: Se pueden utilizar en diferentes tipos de aplicaciones debido a la flexibilidad que ofrecen

MongoDB y Apache CouchDB

Tipos de almacenamiento de datos NoSQL

Almacenamiento en columnas: en lugar de almacenarse en filas, (BBDDRR).

Un almacén de columnas está compuesto por una o más familias de datos que se agrupan de forma lógica en determinadas columnas en la base de datos. Una clave se utiliza para identificar y señalar a un número de columnas en la base de datos. Cada columna contiene filas de nombres o tuplas, y valores, ordenados y separados por comas.

Tipos de almacenamiento de datos NoSQL

ejemplo simple de una base datos de 4 columnas y 3 filas que contiene los siguientes datos:

ID apellido nombre bono

1 González Manuel 6000

2 Martínez Antonio 3000

3 Gutiérrez Alberto 2000

En un sistema de gestión de base de datos orientado a filas los datos se almacenarán de la siguiente manera: 1, González, Manuel, 6000; 2, Martínez, Antonio, 3000; 3, Gutiérrez, Alberto, 2000;

En un sistema de gestión de base de datos columnar los datos se almacenan de la siguiente manera: 1, 2, 3; González, Martínez, Gutiérrez; Manuel, Antonio, Alberto; 6000, 3000, 2000;

Tipos de almacenamiento de datos NoSQL

Tienen acceso rápido de lectura y escritura a los datos almacenados. En un almacén de columnas, las filas que corresponden a una sola columna se almacenan como una sola entrada de disco, lo cual facilita el acceso durante las operaciones de lectura y escritura.

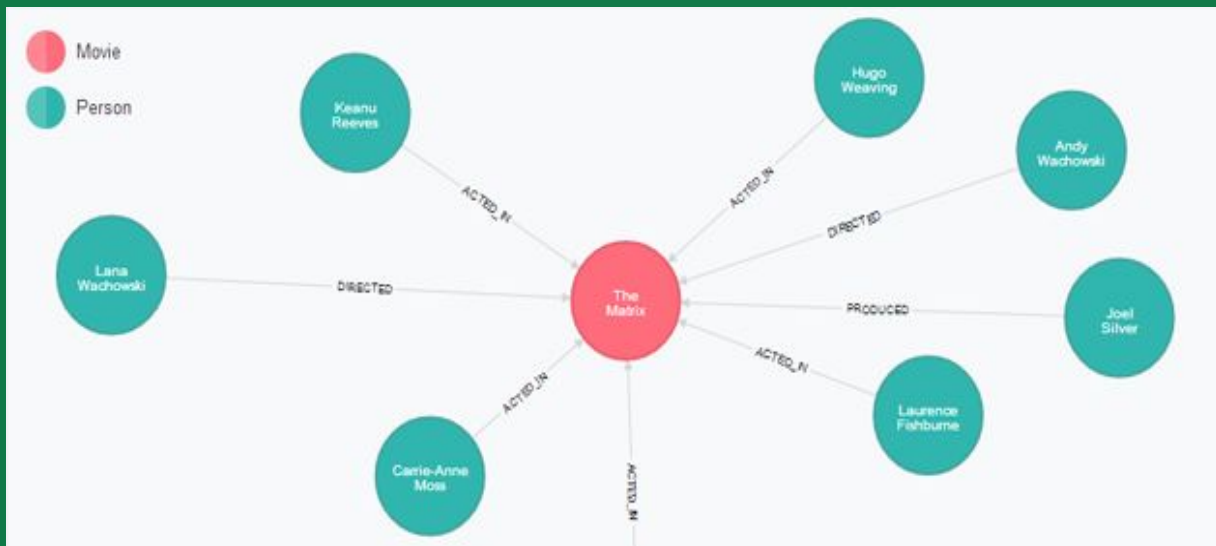
Los datos pueden ser altamente comprimidos. La compresión permite que las operaciones columnares como MIN, MAX, SUM, COUNT y AVG se realicen muy rápidamente.

Como es auto indexable, utiliza menos espacio en disco que un sistema de gestión de base de datos relacional que contenga los mismos datos.

Google BigTable, HBase y Cassandra (desarrollada por FaceBook).

Tipos de almacenamiento de datos NoSQL

Orientada a grafos: se utiliza una estructura de grafos para representar los datos. El gráfico está compuesto por aristas (relaciones) y nodos (información).



Tipos de almacenamiento de datos NoSQL

- Los datos se modelan como un conjunto de relaciones entre elementos específicos.
- Flexibles, atributos y longitud de registros variables
- Permite consultas más amplias y jerárquicas.

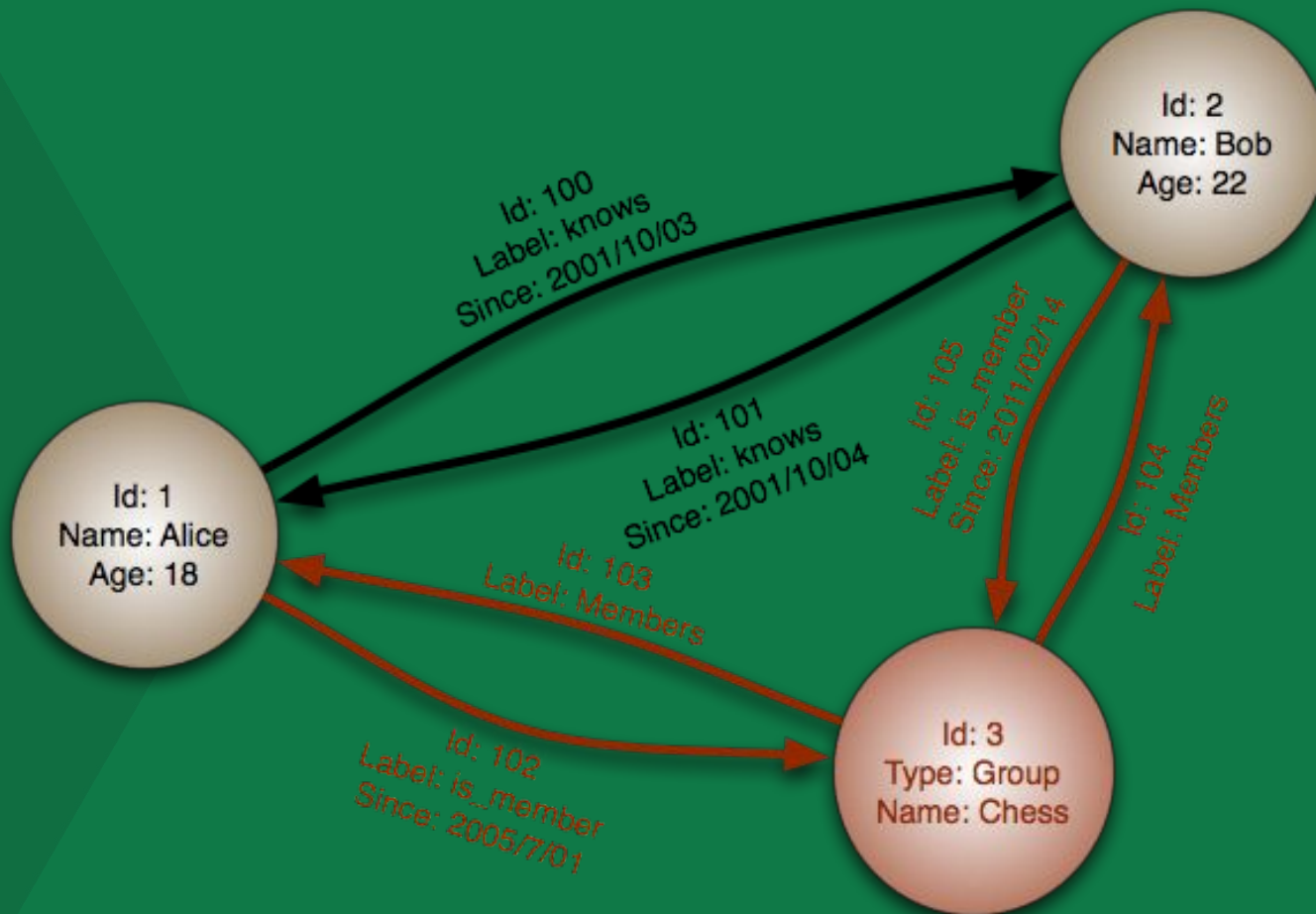
USO: Redes sociales, software de recomendación, geolocalización, topologías de red, ...

InfoGrid, InfiniteGraph y Neo4j

Tipos de almacenamiento de datos NoSQL

Ejemplos de APLICACIÓN de grafos.- permite conectar de forma eficaz a las personas con nuestros productos y servicios, en función de la información personal, sus perfiles en redes sociales y su actividad online reciente. En este sentido, las bases de datos orientadas a grafos son interesantes porque son capaces de conectar personas e intereses. Con esa información, una empresa puede ajustar sus productos y servicios a su público objetivo y personalizar las recomendación en función de los perfiles. Eso es lo que permite que se aumente la precisión comercial y el compromiso del cliente.

Cómo se relacionan las personas que trabajan en una empresa y el conocimiento que poseen, la relación que existe entre lectores de blogs y el contenido de los artículos que leen o incluso cómo interactúan los clientes con los productos de una compañía. Al igual que en ámbitos más científicos, entendiendo las relaciones entre los genes del ADN o entre las partículas que forman la materia.



Modelo de consultas de datos NoSQL

Cada aplicación tiene unos requisitos distintos. En algunos casos, es suficiente tener un modelo de consultas básico en el que la aplicación acceda a los registros basándose en una clave primaria.

Sin embargo, en la mayoría de aplicaciones, es necesario poder ejecutar consultas basándose en varios valores distintos para cada uno de los registros.

Modelo de consultas de datos NoSQL

- **Bases de datos orientadas a documento:** Este tipo de bases de datos proporcionan la posibilidad de ejecutar consultas en base a cualquier tipo de campo dentro del documento.
 - a. También permiten hacer consultas basadas en índices secundarios. Esto permite actualizar registros incluyendo uno o más campos dentro del documento.

Modelo de consultas de datos NoSQL

- **Bases de datos orientadas a grafo:** El almacenamiento en este tipo de bases de datos está optimizado para ejecutar la navegación entre nodos (traversals). Por este motivo, las bases de datos orientadas a grafo son eficientes para realizar consultas en las que existan relaciones de proximidad entre datos, y no para ejecutar consultas globales.

Modelo de consultas de datos NoSQL

- **Bases de datos clave-valor y orientadas a columna:** Este tipo de sistemas permiten obtener y actualizar datos en base a una clave primaria. Las bases de datos clave-valor y orientadas a columna ofrecen un modelo de consultas limitado que puede imponer costes de desarrollo y requisitos a nivel de aplicación para ofrecer un modelo de consultas avanzado. Un ejemplo de esto son los índices, que deben ser gestionados por el propio usuario.

Modelo de consistencia de datos NoSQL

Los sistemas NoSQL típicamente mantienen varias copias de los datos para proporcionar escalabilidad y disponibilidad. Estos pueden utilizar dos tipos de consistencia distinta entre los datos de sus múltiples copias: consistencia y consistencia eventual.

En los **sistemas consistentes** se garantiza que las escrituras sean inmediatamente visibles para las consultas posteriores. Este tipo de sistemas son especialmente útiles para aplicaciones en las que se hace indispensable que los datos sean siempre coherentes. Los sistemas consistentes proporcionan ventajas en las escrituras, aunque, por contra, las lecturas y las actualizaciones son más complejas.

Modelo de consistencia de datos NoSQL

En los **sistemas eventualmente consistentes**, existe un periodo durante el que no todas las copias de los datos están sincronizadas. El hecho de no tener que comprobar la consistencia de los datos en cada una de las operaciones supone una mejora importante en el rendimiento y disponibilidad del sistema, aunque para ello se sacrifique la coherencia de los datos.

Estos tipos de sistemas son especialmente útiles para datos que NO cambian a menudo, como archivos históricos o logs.

Modelo de consistencia de datos NoSQL

Las bases de datos orientadas a documento o grafo existentes pueden ser consistentes o eventualmente consistentes, mientras que las bases de datos clave-valor y orientadas a columna son típicamente eventualmente consistentes.

SQL → NoSQL

SQL



Web



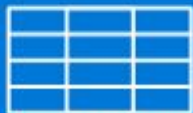
Mobile



Enterprise



Data mart



Relational table storage



Relationships use joins

NoSQL



Gaming



Social



IoT



Web



Mobile



Enterprise



Key/value store

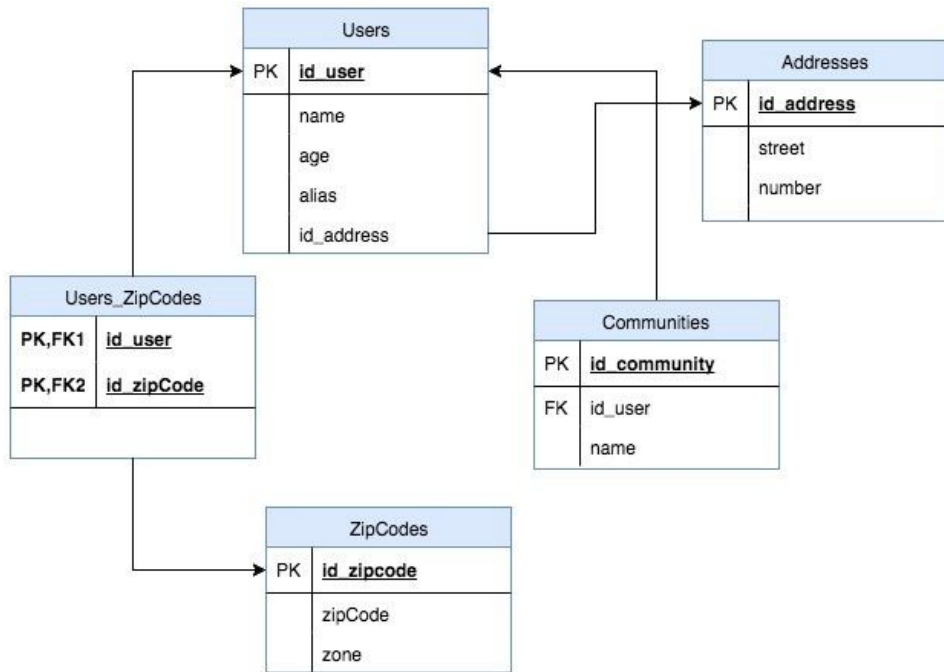


Document database



Column family store

SQL → NoSQL



Users <collection>

```
{
  name : "Marcela Sena",
  age  : 29,
  alias : "MarceStarlet",
  memberOf : ["TechWo", "JavaGDL"],
  address : { street : "Main Boulevard", number : 234 },
  zipCodes : [
    { zipCode : 1234, zone: "Guadalajara" },
    { zipCode : 4321, zone: "arajaladaug" }
  ]
}
```

SQL → NoSQL

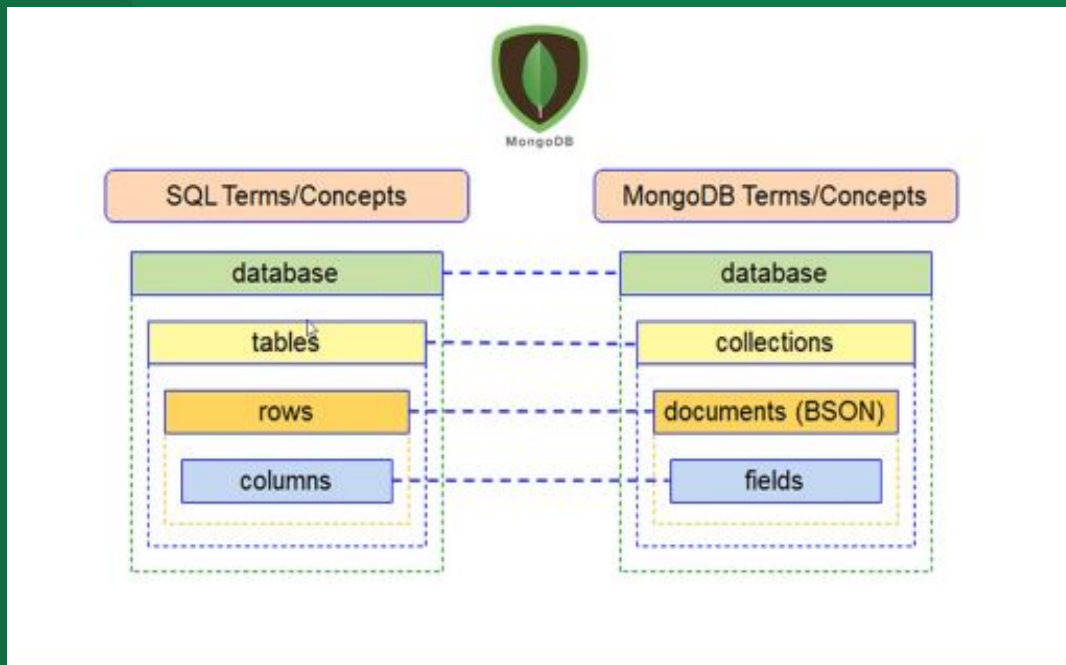
La diferencia sustancial sería la abstracción completa de todos los datos definidos en un solo documento y no en distintas tablas relacionadas.

Las restricciones y la existencia de “ids” de otras tablas como parte de nuestros registros ya no son relevantes pues lo que nos interesa es la información como tal.

SQL → NoSQL

MONGO DB

MySQL → MongoDB



El concepto de *database* en ambos modelos es el mismo.

En MongoDB se le llama *collection* a la agrupación de *documents* (BSON — binary JSON) que se almacenan en una database.

Por lo tanto, las operaciones de lectura y escritura y las consultas que haremos serán sobre una database y sus collections,

SQL → NoSQL

```
> use GenteDB
```

```
> db.createCollection("usuarios")
```

o

```
> db.usuarios.insertOne(...)
```

¿¿¿¿¿
nos suena de
algo
?????

MongoDB

MongoDB es una base de datos libre de esquemas, orientada a documentos, escrita en C ++. La base de datos está basada en el almacén de documentos, lo que significa que almacena valores (denominados documentos) en forma de datos codificados.

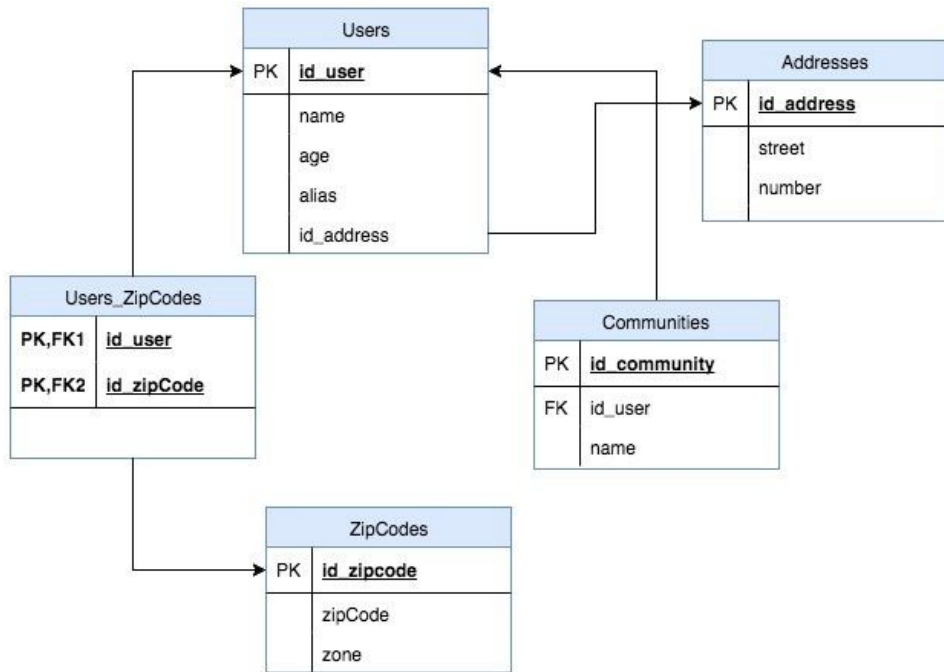
La elección del formato codificado en MongoDB es JSON. Es muy potente, porque incluso si los datos están anidados dentro de los documentos JSON, seguirá siendo consultable e indexable.

MongoDB

MongoDB tiene un sistema flexible de almacenamiento de esquemas. Lo que significa que los objetos almacenados no tienen que tener la misma estructura o los mismos campos.

MongoDB también tiene algunas características de optimización, que distribuye las colecciones de datos, mejorando el rendimiento y consiguiendo un sistema más equilibrado.

SQL → NoSQL



Users <collection>

```
{
  name : "Marcela Sena",
  age  : 29,
  alias : "MarceStarlet",
  memberOf : ["TechWo", "JavaGDL"],
  address : { street : "Main Boulevard", number : 234 },
  zipCodes : [
    { zipCode : 1234, zone: "Guadalajara" },
    { zipCode : 4321, zone: "arajaladaug" }
  ]
}
```

Estructura...

Las ventajas de tener este tipo de estructura de datos como registros es que principalmente podemos reducir las “relaciones” entre tablas que comúnmente usamos en bases de datos relacionales cuando hacemos consultas con los famosos “joins” o uniones de datos, ya que resulta muy costoso.

Otra ventaja es que de cierta forma esta estructura se corresponde con tipos de datos en varios lenguajes de programación lo cual resulta conveniente para desarrollar código de forma más ágil.

Además, también nos permite guardar grandes volúmenes de datos sin tanto coste.

MongoDB

MongoDB promete un alto desempeño en tareas de I/O con el uso de modelos de datos embebidos e índices, además de alta disponibilidad haciendo réplicas y reduciendo la redundancia de datos; en MongoDB es posible escalar horizontalmente agregando clusters de máquinas donde los datos se distribuyen por medio de sharding. Las ventajas de tener este tipo de estructura de datos como registros es que principalmente podemos reducir las “relaciones” entre tablas que comúnmente usamos en bases de datos relacionales cuando hacemos consultas con los famosos “joins” o uniones de datos, ya que resulta muy costoso. Otra ventaja es que de cierta forma esta estructura se corresponde con tipos de datos en varios lenguajes de programación lo cual resulta conveniente para desarrollar código de forma más ágil. Además, también nos permite guardar grandes volúmenes de datos sin tanto costo.

MongoDB JSON

Todos los datos que se guardan en MongoDB, son almacenados en documentos JSON.

JSON es un formato estándar para datos que destaca por ser ligero y rápido (por tanto muy útil para desarrollos web).

Los datos en formato JSON pueden ser utilizados por prácticamente todos los lenguajes de programación (como Java, C#, C, C++, PHP, JavaScript, Python, etc.)

MongoDB JSON

Los archivos JSON son simples archivos de texto con extensión .json y se puede crear con cualquier editor de texto plano.

Los documentos JSON se componen de dos estructuras básicas: **arrays** y **diccionarios**.

Los arrays son un conjunto de valores que se encuentran encerrados entre corchetes [] y los diccionarios son mapas asociativos representados por clave:valor que se encuentran encerrados entre llaves { }

MongoDB JSON

Cada una de estas estructuras pueden contener a la otra o a sí misma.

El concepto para los diccionarios de clave:valor se refiere a que podemos insertar valores con algún identificador para reconocerlo más fácilmente, como por ejemplo si tuviéramos un diccionario con datos personales sería algo como esto:

```
{"nombre" : "Jason", "edad" : 20}
```

Un diccionario puede tener varios valores, éstos deben separarse por una coma, además los strings siempre deberían estar entre comillas,

MongoDB JSON

Un diccionario puede contener otros diccionarios y arrays dentro.

Cómo ingresar datos dentro de un array y de otro diccionario:

```
{  
  "nombre" : "Jason",  
  "edad" : 20,  
  "email" : {  
    "email personal" : "personal@email.com",  
    "email corporativo" : "corporativo@email.com"  
  },  
  "frutas favoritas" : [pera, manzana, ciruela]  
}
```



más fácil organizar varios datos de una persona en un mismo documento, haciendo (caso de MongoDB) que no se necesiten de otras tablas adicionales para ingresar varios datos personales a una persona, como lo son en este caso las email y frutas favoritas.

MongoDB JSON.- ejemplo de array de diccionarios

```
{  
  "marcadores": [  
    {  
      "latitude": 40.416875,  
      "longitude": -3.703308,  
      "city": "Madrid",  
      "description": "Puerta del Sol"  
    },  
  ],  
}
```

```
{  
  "latitude": 40.417438,  
  "longitude": -3.693363,  
  "city": "Madrid",  
  "description": "Paseo del Prado"  
},  
]  
}
```

MongoDB JSON.- ejemplo de conjunto de valores directo

```
{  
  "título": "Doctor",  
  "Nombre": "Pedro",  
  "Edad": 37,  
  "City": "Madrid"  
  "facebook": "https://fb.com"  
}
```

MongoDB - Características

Shards / Fragmentos

Sharding es la partición y distribución de datos a través de múltiples máquinas (nodos). Un fragmento, es una colección de nodos MongoDB.

En MongoDB es posible escalar horizontalmente agregando clusters de máquinas donde los datos se distribuyen horizontalmente a través de múltiples nodos. En el caso de que haya una aplicación que utilice un único servidor de base de datos, se puede convertir en clúster fragmentado, con muy pocos cambios en el código de la aplicación original.

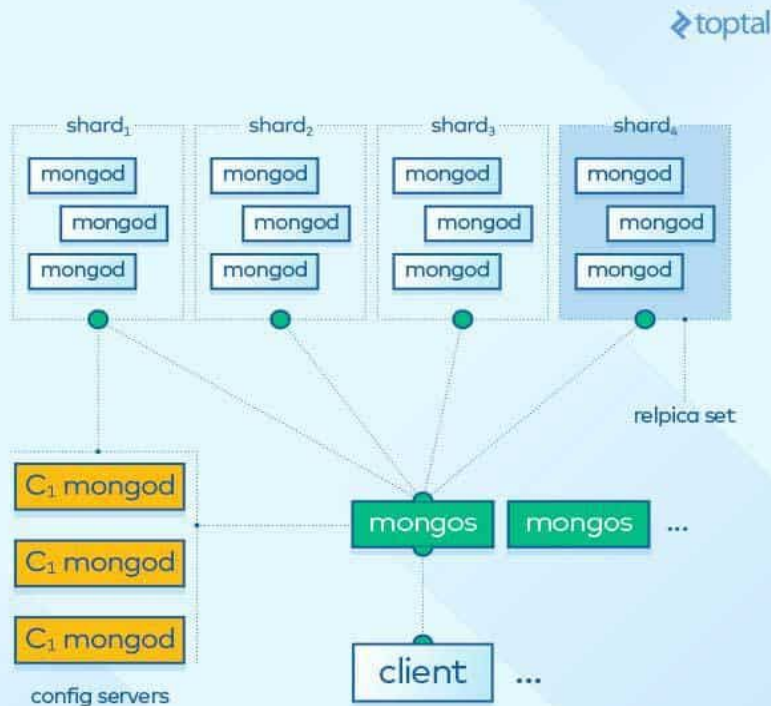
MongoDB - Características

Lenguaje de consulta Mongo

Como se mencionó anteriormente, MongoDB utiliza una API RESTful. Para recuperar ciertos documentos de una colección db, se crea un documento de consulta que contiene los campos que deben coincidir con los documentos deseados. JavaScript.

MongoDB soporta un lenguaje de consultas enriquecido para hacer operaciones de escritura y lectura (CRUD)

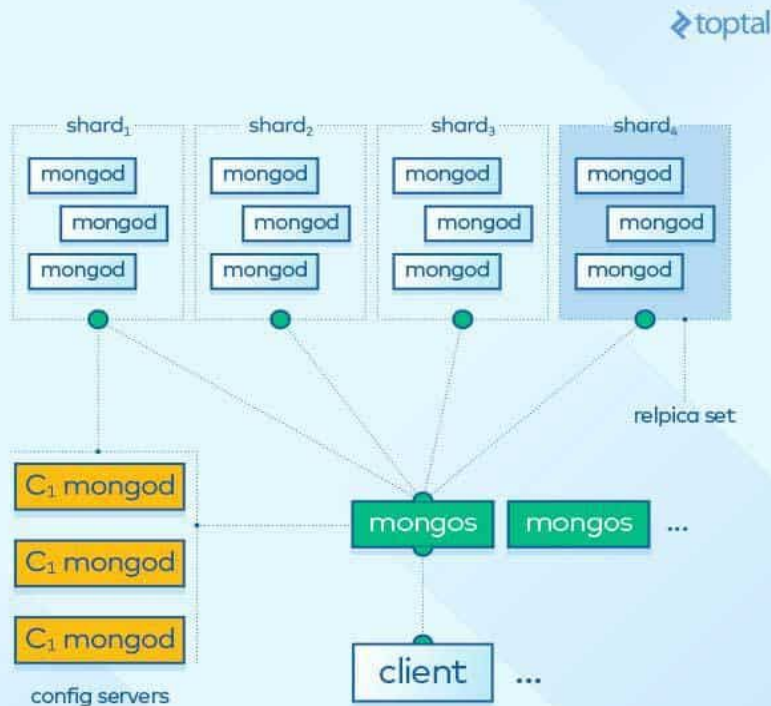
Arquitectura MongoDB



En MongoDB, hay un grupo de servidores llamados enrutadores. Cada uno actúa como un servidor para uno o más clientes. Del mismo modo, el clúster contiene un grupo de servidores denominados servidores de configuración. Cada uno contiene una copia de los metadatos que indican qué fragmento contiene qué datos. Las acciones de lectura o escritura se envían desde los clientes a uno de los servidores de enrutador del clúster y son encaminadas automáticamente por ese servidor, a los fragmentos adecuados que contienen los datos con la ayuda de los servidores de configuración.

servidores del enrutador en verde, los servidores de configuración en amarillo y los fragmentos que contienen los nodos MongoDB en azul.

Arquitectura MongoDB



Cabe señalar que el sharding (o compartir los datos entre fragmentos) en MongoDB es completamente automático, lo que reduce la tasa de fallos y hace MongoDB un sistema de gestión de base de datos altamente escalable.

Escalabilidad .- Propiedad de aumentar la capacidad de trabajo o de tamaño de un sistema sin comprometer el funcionamiento y calidad normales del mismo.

Documentos

En MongoDB un documento es un registro compuesto por pares “campo : valor”.

```
{  
  name : "Marcela Sena",  
  age  : 29,  
  alias : "MarceStarlet",  
  memberOf : ["TechWo", "JavaGDL"],  
  address : { street : "Main Boulevard", number : 234 },
```

```
  zipCodes : [  
    { zipCode : 1234, zone: "Guadalajara" },  
    { zipCode : 4321, zone: "arajaladauG" }  
  ]  
}
```

Como vemos, los valores de los campos pueden ser otros documentos, arrays y arrays de documentos.

Instalación y CRUD

INSTALACIÓN

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

Descargar paquete MSI

Click para iniciar el wizard de instalación

Visitar:

<https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/>

<https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/>

INSTALACIÓN

Crearse en el directorio raíz C, una carpeta “data” y dentro, dos subcarpetas: “db” y “log”

Acceder a Archivos de Programa/mongoDB/Server/4.0/bin/
y ejecutar:

1. mongod
2. mongo

Comparativa CRUD

Podemos comenzar a manipular datos con las operaciones comunes CREATE, READ, UPDATE, DELETE; y para migrar de SQL a NoSQL con Mongo, lo haremos paso a paso con una comparativa entre sentencias en uno y otro lenguaje.

Comparativa CRUD

<https://docs.mongodb.com/manual/reference/sql-comparison/>

Comparativa CRUD

CREATE (base de datos) en SQL

```
CREATE DATABASE alumnos;
```

CREATE (base de datos) en
Mongo

```
use peliculas
```



Comparativa CRUD

CREATE (tablas) en SQL

```
CREATE TABLE usuarios (  
  id MEDIUMINT NOT NULL AUTO_INCREMENT,  
  nombre Varchar(30),  
  edad Number,  
  alias Varchar(20),  
  PRIMARY KEY (id)  
)
```

CREATE (colección) en Mongo

```
db.createCollection("películas")
```

o también implícitamente:

```
db.películas.insertOne(  
  {  
    titulo : "Titanic",  
    año : 1979,  
    pPal : "DiCaprio"  
  }  
)
```

Comparativa CRUD

ALTER en SQL

```
ALTER TABLE usuarios ADD nacimiento DATETIME
```



ALTER en Mongo

```
db.peliculas.updateMany(  
  {},  
  { $set: { f_revision: new Date() } }  
)
```

```
ALTER TABLE usuarios DROP COLUMN nacimiento
```



```
db.users.updateMany(  
  {},  
  { $unset: { "f_revision": "" } }  
)
```

Comparativa CRUD

Los métodos **insertOne(<document>)** e **insertMany([<doc1>,<doc2>...])** implícitamente crean una colección si no existe e insertan uno o muchos documents .

No tuvimos que especificar el “id” de nuestro documento, si no definimos un campo “_id”, Mongo lo crea automáticamente.

El método **updateMany(<filter>,<update>)** actualiza múltiples documentos dentro de una collection basado en el filtro que recibe como argumento, pero además puede alterar los documentos con operaciones de **\$set** (establecer) y **\$unset** (des-establecer) para agregar o borrar campos.

La forma de alterar información en Mongo no se hace a nivel de colecciones ya que no es una modificación estructural sino de documentos (registros de nuestra colección).

Comparativa CRUD

INSERT en SQL

```
INSERT INTO usuarios(nombre, edad, alias)
VALUES ("Pepe Reina", 29, "Speaker"),
("Julián Contreras", 17, "ErJuli")
);
```

INSERT en Mongo

```
db.users.insertMany(
  {
    nombre : "Pepe Reina",
    edad  : 29,
    alias : "Speaker"
  },
  {
    nombre : "Julián Contreras",
    edad  : 17,
    alias : "ErJuli"
  })
```


Comparativa CRUD

UPDATE y DELETE en SQL


```
UPDATE usuarios  
SET alias = "Chavales"  
WHERE age > 25;
```

```
DELETE FROM usuarios  
WHERE alias = "erJuli" ;
```

UPDATE y DELETE en Mongo



```
db.usuarios.updateMany(  
  { age: { $gt: 25 } },  
  { $set: { alias: "Majetes" } }  
)
```



```
db.usuarios.deleteMany(  
  { alias: "erJuli" }  
)
```

Comparativa CRUD

SELECT en SQL

```
SELECT *FROM usuarios;
```

```
SELECT nombre, alias FROM usuarios
```

```
SELECT nombre, edad FROM usuarios  
WHERE alias = "Speaker"
```

SELECT en Mongo

```
db.usuarios.find()
```

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


```
db.usuarios.find(  
  { alias: "Speaker" },  
  { nombre: 1, edad: 1, _id: 0 })
```

Comparativa CRUD

SELECT en SQL


```
SELECT * FROM usuarios  
WHERE alias = "erJuli"  
ORDER BY nombre DESC
```

SELECT en Mongo



```
db.usuarios.find (  
  { alias: "erJuli" }  
) .sort( { nombre: -1 } )
```

```
SELECT * FROM clientes  
WHERE nombre = "Pedrooooooo"  
ORDER BY nombre DESC
```



```
db.Clientes.find (  
  { Nombre: "Pedroooooooo" } )
```

Comparativa CRUD

SELECT en SQL

```
SELECT age, name, uz.zone
FROM users u, users_zipCodes uz
WHERE u.id_user = uz.id_user
GROUP BY age, name
ORDER BY age, name
```



SELECT en Mongo

```
db.users.aggregate(
[
  { $unwind: "$zipCodes" },
  { $group:
    { _id:
      {
        age: "$age",
        name: "$name",
        zone: "$zipCodes.zone"
      }
    }
  },
  { $sort: { _id: 1 } }
]
)
```


Comparativa CRUD

DROP en SQL

```
DROP TABLE IF EXISTS usuarios;
```

DROP en Mongo

```
db.usuarios.drop ( )
```

```
DROP DATABASE IF EXISTS Juegos;
```

```
use Juegos  
db.dropDatabase ( )
```

Comparativa CRUD .- tipos de datos

MongoDB, a través de JSON, puede utilizar los siguientes tipos:

- **String:** guardados en UTF-8. Van siempre entre dobles comillas.
- **Number:** números. Al guardarse en BSON pueden ser de tipo byte, int32, int64 o double.
- **Boolean:** con valor true o false.
- **Array:** van entre corchetes [] y pueden contener de 1 a N elementos, que pueden ser de cualquiera de los otros tipos.
- **Documentos:** un documento en formato JSON puede contener otros documentos embebidos que incluyan más documentos o cualquiera de los tipos anteriormente descritos.
- **Null.**

Ejemplo de tipo de datos

```
{
  "People":
    [
      {
        "_id": ObjectId("51c420ba77edcdc3ec709218"),
        "nombre": "Manuel",
        "apellidos": "Pérez",
        "fecha_nacimiento": "1982-03-03",
        "altura": 1.80,
        "activo": true,
        "intereses":["fútbol","tenis"],
        "tarjeta_credito": null,
        "dni":
          {
            "numero":"465464646J",
            "caducidad":"2013-10-21"
          }
      },
    ]
}
```

```
{
  "_id": ObjectId("51c420ba77ed1dc3ec705289"),
  "nombre": "Sara",
  "apellidos": "Ruano",
  "fecha_nacimiento": "1985-12-03",
  "altura": 1.65,
  "activo": false,
  "intereses":["moda","libros","fotografía","política"],
  "tarjeta_credito": null
}
]
```

Campo `_id` → `ObjectId()`

- Es un campo requerido en cada documento de MongoDB. El campo `_id` debe tener un valor único. Pensaremos en el campo `_id` como la clave principal del documento (por lo que estará indexado automáticamente).
- Si crea un nuevo documento sin un campo `_id`, MongoDB crea automáticamente el campo y asigna un `ObjectId` BSON único
- Es un tipo especial de BSON de 12 bytes que garantiza la singularidad dentro de la colección.
- El `ObjectId` se genera en función de la marca de tiempo, la ID de la máquina, la ID del proceso y un contador incremental local del proceso.



Campo `_id` → `ObjectId()`

- En definitiva los nueve primeros bytes nos garantizan un identificador único por segundo, máquina y proceso. Los tres últimos bytes, nos garantizan que cada segundo podemos insertar $2^{24} = 16.777.216$ documentos con un identificador distinto. Aunque técnicamente un `_id` podría repetirse, en la práctica es un número tan alto que es muy difícil que eso suceda.
- MongoDB utiliza los valores de `ObjectId` como valores predeterminados para los campos `_id`.
- Que el `ObjectId` esté compuesto de esa manera, nos da algunas funcionalidades muy útiles. La primera es que nos puede dar una indicación de el orden de creación de los documentos. No es algo del todo fiable si estamos tratando con documentos creados en el mismo margen de tiempo, pero sí en tramos de tiempo más largos.

Campo `_id` → `ObjectId()`

- También nos sirve para obtener la fecha de creación del documento. Por ejemplo, vamos a crear un *ObjectId* y vamos a extraer su fecha de creación desde la consola:

```
> var myObjectId = new ObjectId();
```

```
> myObjectId.getTimestamp();
```

```
ISODate("2014-02-08T23:23:41Z")
```

En el ejemplo vemos, que no solo **MongoDB** puede crear campos *ObjectId*. Nosotros también podemos hacerlo usando el *new ObjectId()*. Además con el método *getTimestamp()* podemos extraer la fecha en la que se creó el documento.

Ejercicio 0.- CRUD alumnos

1. Crear un base de datos de personas .
2. Crear una colección de alumnos.
3. Insertar diez documentos (registros con campos comunes y no comunes)
 - a. dos con nombre Pepe y tres con nombre María.
4. Listar todos los documentos que componen la colección estudiantes.
5. Listar todos los campos de aquellos documentos que contengan el nombre Pepe.
6. Listar todos los campos de aquellos documentos que contengan el nombre María y que tengan 20 años.
7. Mostrar los apellidos de los alumnos que se llamen María.
8. Saber el número de alumnos en total.
9. Saber el número de alumnos que se llaman Pepe.
10. Obtener sólo el apellido del estudiante que tenga como mail: pp@soypepe.com.

Ejercicio CRUD alumnos .- SOLUCIÓN

use personas

```
db.createCollection("Alumnos")
```

```
db.Alumnos.insertMany([
  { nombre: "María", apellidos: "Potosí", edad: 36,
    titulación: "Ingeniera" },
  { nombre: "María", edad: 25, ciudad: "Madrid" }
])
```

```
db.Alumnos.insertMany([
  { nombre: "Pepe", apellidos: "Bermejo Casariego", edad: 21,
    titulación: "Graduado" },
  { nombre: "Pepe", edad: 25, ciudad: "Madrid" }
])
```

```
db.Alumnos.insertMany([
  { _id: "01", nombre: "Juan", ciudad: "Madrid" },
  { nombre: "Penelope", CursosMatriculados: 3 },
  { nombre: "Santos", pais: "España" }
]);
```


Ejercicio CRUD alumnos .- SOLUCIÓN

```
db.Alumnos.find()
```

```
db.Alumnos.find({nombre:"Pepe"})
```

```
db.Alumnos.find({nombre:"María", edad:20})
```

```
db.Alumnos.find({nombre:"María"},{apellidos:true,_id:false})
```

```
db.Alumnos.count()
```

```
db.Alumnos.count({nombre:"Pepe"})
```

```
db.Alumnos.find({email:"pp@soypepe.com"},{apellidos:1, _id:0})
```

Introducción a sentencias

CREATE

`db.createCollection(nombre)`

```
db.createCollection("Alumnos")
```

recordamos que con `db.nombColec.insertOne(..)` también se creaba de forma implícita

DROP

db.dropDatabase()

- Elimina la base de datos actual, eliminando los archivos de datos asociados.
- Bloqueará otras operaciones hasta que se complete.

```
use temp
```

```
db.dropDatabase()
```

DROP

`db.nombColección.drop()`

- Elimina la Colección nombColeccción, eliminando todos los documentos que dependan de ella.
- Bloqueará otras operaciones hasta que se complete.

```
use temp
```

```
db.archivosTemporales.drop()
```

Inserciones

`.insert()`

`.insertOne()`

`.insertMany()`

Si la colección no existe, se crea.

Si el documento no especifica un campo `_id`, MongoDB agregará el `_id` campo y asignará un único Object Id para el documento antes de insertarlo.

Si el documento contiene un campo `_id`, el valor debe ser único dentro de la colección para evitar un error de clave duplicado.

Los valores `Object_Id` son específicos de la máquina y la hora en que se ejecuta la operación. Como tal, sus valores pueden diferir en cada ejemplo.

INSERT

```
db.nombcollection.insert(  
  <document or array of documents>,  
  {  
    ordered: <boolean>  
  }  
)
```

```
db.alumnos.insert( { name: "Amanda", status: "Updated" } )
```

INSERT

Varios documentos a la vez (sin obligar a una misma estructura en todos)

```
db.productos.insert(  
  [  
    { _id: 11, elemento: "raqueta", qty: 50, tipo: "no.2" },  
    { elemento: "overgrip", qty: 20 },  
    { elemento: "dumper", qty: 25 }  
  ]  
)
```

Los documentos en la matriz no necesitan tener los mismos campos

Por defecto, MongoDB realiza una inserción *ordenada*

Con las inserciones *ordenadas*, si se produce un error durante la inserción de uno de los documentos, MongoDB devuelve un error **sin procesar los documentos restantes en la matriz.**

INSERT

desordenada....

```
db.productos.insert(  
  [  
    { _id: 20, elemento: "zapatillas", qty: 50, type: "clay" },  
    { _id: 21, elemento: "zapatillas", qty: 20, type: "hard" },  
    { _id: 22, elemento: "calcetines", qty: 100 }  
  ],  
  { ordered: false }  
)
```

Con inserciones *desordenadas*, si ocurre un error durante una inserción de uno de los documentos, MongoDB continúa insertando los documentos restantes en la matriz.

```
db.products.find({qty:{$gt:25}})
```

db.collection.insertOne ()

Sin valor id

```
db.productos.insertOne( { elemento: "cajón pelotas", qty: 15 } );
```

Debido a que los documentos no incluyeron `_id`, mongod crea y agrega el campo `_id` y le asigna un valor ObjectId único

Con valor id

```
db.productos.insertOne( { _id: 10, elemento: "bote", qty: 20 } );
```

La inserción de un valor duplicado para cualquier clave que forme parte de un índice único , como por ejemplo `_id`, lanza una excepción.

db.collection.insertMany ()

Inserta múltiples documentos en una colección.

Sin valor id

```
db.productos.insertMany( [  
  { elemento: "camiseta M", qty: 15 },  
  { elemento: "camiseta W", qty: 20 },  
  { elemento: "camiseta Jr" , qty: 30 }  
] )
```

Por defecto los documentos se insertan en orden.

Con `ordered` a `false`, la operación de inserción continuará con los documentos restantes, si ocurriera un error en la inserción actual.

Con valor id

```
db.productos.insertMany( [  
  { _id: 10, elemento: "pantalón M", qty: 20 },  
  { _id: 11, elemento: "pantalón W", qty: 55 },  
  { _id: 12, elemento: "pantalón Jr", qty: 30 }  
] );
```

La inserción de un valor duplicado para cualquier clave que forme parte de un índice único , como por ejemplo `_id`, lanza una excepción.

Borrados : `deleteOne()` , `DeleteMany()`

Borrado de documentos de una colección. Se identifican criterios/filtros para para identificar el documento a borrar

db.collection.deleteOne ()

Sólo borra el primer documento que coincide con el filtro

```
db.collection.deleteOne(  
  <filter>  
)
```

```
db.Amigos.deleteOne(  
  { _id: "CTY23a" },  
)
```

```
db.Amigos.deleteOne(  
  {  
    nombre : "Irene",  
    apellidos : "Juarez Trujillo"  
  },  
)
```

db.collection.deleteMany ()

Cambia TODOS los documentos en una colección, que cumplen con un criterio.

```
db.comunidades.deleteMany(  
  { casas: { $lt: 2 }, codPostal: "54021" }  
)
```

Actualizaciones

Cambia de documentos de una colección. Se identifican criterios/filtros para para identificar el documento a actualizar

```
.updateOne()  
.updateMany()  
.replaceOne()
```

db.collection.updateOne ()

Sólo actualiza el primer documento que coincide con el filtro

```
db.collection.updateOne(  
  <filter>,  
  <update>,  
  {   upsert: <boolean> }  
)
```

upsert: TRUE → si no existe el documento, se CREARÁ.

```
db.Amigos.updateOne(  
  { nombre : "Paola" },  
  { $set: { telefono : "2334455667" } }  
)
```

```
db.Amigos.updateOne(  
  { nombre : "Carlos" },  
  { $set: { _id : 290, telefono: "72345678" } },  
  { upsert: true }  
)
```


db.collection.updateMany ()

Cambia TODOS los documentos en una colección, que cumplen con un criterio.

```
db.familia.updateMany(  
  { casas: { $gt: 1 } },  
  { $set: { AvisarVerano : true } }  
);
```

db.collection.replaceOne ()

Sólo actualiza el primer documento que coincide con el filtro

```
db.collection.replaceOne(  
  <filter>,  
  <replacement>,  
  {upsert: <boolean>  
})
```

```
db.Amigos.replaceOne(  
  { "nombre" : "JoseAntonio" },  
  { "nombre" : "Pepe", "deporte" : "Tenis" }  
)
```

Método `.find()` en detalle

`.distinct()`

`.limit()`

`.skip()`

`.sort()`

`.findOne()`

`.findOneAndReplace()`

`.findOneAndUpdate()`

db.collection.distinct()

Supongamos:

```
{ "_id": 1, "dept": "A", "item": { "sku": "111", "color": "red" }, "sizes": [ "S", "M" ] }  
{ "_id": 2, "dept": "A", "item": { "sku": "111", "color": "blue" }, "sizes": [ "M", "L" ] }  
{ "_id": 3, "dept": "B", "item": { "sku": "222", "color": "blue" }, "sizes": "S" }  
{ "_id": 4, "dept": "A", "item": { "sku": "333", "color": "black" }, "sizes": [ "S" ] }
```

devolviendo valores de un campo: `db.inventario.distinct("dept")`

devolviendo valores de un campo array: `db.inventario.distinct("sizes")`

devolviendo valores de un campo embebido: `db.inventario.distinct("item.sku")`

con condición: `db.inventario.distinct("item.sku", { dept: "A" })`

.limit(n) y .skip(n)

1. Consulta de datos

```
db.amigos.find({}, {Nombre: 1})
```

Personas mayores de 25 años

```
db.amigos.find({"Edad": {$gt: 25}})
```

2. Mostrar proyección

Mostrar nombre y apellidos de las 3 primeras Marisas encontradas

```
db.amigos.find({Nombre: "Marisa"}, {Nombre: 1, Apellidos: 1}).limit(3)
```

Mostrar nombre y apellidos de las Marisas encontradas, empezando por el cuarto documento

```
db.amigos.find({Nombre: "Marisa"}, {Nombre: 1, Apellidos: 1}).skip(3)
```

Ordenaciones

A la hora de realizar una consulta podemos encontrarnos la necesidad de realizar una ordenación de los datos. Ya que si no decimos nada las consultas en MongoDB devolverán los datos tal cual se encuentren almacenados en la base de datos, es decir, con su orden de inserción.

Así si queremos realizar ordenaciones en MongoDB deberemos de utilizar el método **.sort()**. La sintaxis del método MongoDB **.sort()** es la siguiente:

```
db.coleccion.find(filtro).sort(ordenacion)
```

Indicaremos el campo o los campos por los cuales queremos ordenar y el tipo de ordenación que queremos: ordenación ascendente u ordenación descendente.

Ordenaciones

La estructura de la ordenación es la siguiente:

```
.sort({campo1:tipoOrdencion1, campo2:tipoOrdenacion2,..., campoN:tipoOrdenacionN});
```

Dónde los tipos de ordenación y sus valores son:

- **Ascendente:** 1
- **Descendente:** -1

```
db.alumnos.find().sort(apellido:1)
```

db.collection.findOne ()

Devuelve un solo documento que satisface los criterios de consulta especificados en la colección. Si varios documentos satisfacen la consulta, este método devuelve el primer documento según el orden natural que refleja el orden de los documentos en el disco

Si ningún documento satisface la consulta, el método devuelve un valor nulo.

```
db.collection.findOne( consulta , campos_a_mostrar )
```


db.collection.findOne ().- Ejemplos

```
db.gerentes.findOne ()
```

```
db .gerentes. findOne (  
  {},  
  { nombre : 1 , Svariable : 1 }  
)
```

```
db .gerentes. findOne (  
  {  
    $ or : [  
      { nombre.nombre : "Pepe" },  
      { fechaIngreso : { $ lt : new Date ( '01 / 01/1945 ' ) } }  
    ]  
  }  
)
```

```
db.gerentes.findOne (  
  { contribuciones : 'POO' },  
  { _id : 0 , 'nombre.nombre' : 0 , fechaIngreso : 0 }  
)
```

db.collection.findOneAndUpdate() v3.2

```
db.collection.findOneAndUpdate(  
  <filter>,  
  <update>,  
  {  
    sort: <document>,  
    upsert: <boolean>,  
    returnNewDocument: <boolean>,  
  }  
)
```

Buscará documentos que coincidan con los criterios de búsqueda, y actualizará el primero de ellos, según lo que se especifique en Update

db.collection.findOneAndUpdate() .- Ejemplo

```
db.clasificacion.findOneAndUpdate(  
  { "nombre" : "C. Menendez" },  
  { $inc : { "puntos" : 5 } },  
  { sort : { "puntos" : 1 }, upsert : true }  
)
```

Se ordenan aquellos documentos con el campo “nombre” con valor “C. Menéndez”, ascendentemente según el campo “Puntos”, y se actualiza (con un incremento de cinco unidades), por tanto el documento con menos cantidad de puntos.

Se devuelve el documento PREVIO a la actualización!

db.collection.findOneAndUpdate().- Ejemplo II

```
db.clasificacion.findOneAndUpdate(  
  { "nombre" : "Luco" },  
  { $set: { "nombre" : "Lucio Constantino", "prueba" : 5}, $inc : { "puntos" : 5 } },  
  { sort: { "puntos" : 1 }, upsert:true, returnNewDocument : true }  
)
```

Busca los documentos que coincidan con ese nombre, los ordena ascendentemente por su campo PUNTOS, actualiza según los datos expuestos ("nombre" y "prueba") e incrementa cinco unidades sus puntos.

Si no existiera el documento LO CREA, y devuelve, como fin último de la operación, el nuevo documento

db.collection.findOneAndReplace() v3.2

Modifica y reemplaza un solo documento basado en los criterios

```
db . colección . findOneAndReplace (
  < filter > ,
  < replacement > ,
  {
    projection : < document > ,
    sort : < document > ,
    upsert : < boolean > ,
    returnNewDocument : < boolean > ,
  }
)
```

db.collection.findOneAndReplace() v3.2

```
db.collection.findOneAndReplace(  
  <filter>,  
  <replacement>,  
  {  
    projection: <document>,  
    sort: <document>,  
    upsert: <boolean>,  
    returnNewDocument: <boolean>,  
  }  
)
```

Busca un documento (según unos criterios especificados) y lo reemplaza por otro.

No se podrá especificar un campo `_id`, que sea diferente al del documento reemplazado.

No se pueden utilizar operadores de actualización (`$inc`, `$rename`, `$set`, `$unset`, ...)

db.collection.findOneAndReplace().- Ejemplo

```
db.clasificacion.findOneAndReplace(  
  { "calificacion" : { $lt : 100 } },  
  { "nombre" : "Lau Zam", "puntuacion" : 100 })
```

Encontrará el primer documento
con “puntuación” menor que 100
y lo cambiará por lo
especificado en la siguiente
línea.

Se devolverá el documento ORIGINAL que ha sido reemplazado...

```
{ "_id" : 2512, "nombre" : "NurLop", "puntuacion" : 92 }
```

Si, returnNewDocument está configurado como TRUE, también se devolverá el documento reemplazado,

db.collection.findOneAndReplace().- Ejemplo II

```
db.clasificacion.findOneAndReplace(  
  { calificacion : { $lt : 200 } },  
  { nombre : "PaoMay", calificacion : 200 },  
  { sort: { calificacion : 1 },  
    returnNewDocument : 1,  
    project : {nombre:1,calificacion:1,_id:0}  
  }  
)
```

primero ordena todos los documentos por el campo "Calificacion", luego selecciona el primero que tenga un score menor que 200, y lo reemplaza por esos valores...

Devolverá el documento anterior al reemplazado, pero sólo los campos "nombre" y "calificacion"

db.collection.findAndModify() v3.6

```
db.collection.findAndModify({  
  query: <document>,  
  sort: <document>,  
  remove: <boolean>,  
  update: <document>,  
  new: <boolean>,  
  fields: <document>,  
  upsert: <boolean>,  
});
```

- Se modificará el primero de los encontrados.
- Se indica cuales son los criterios para identificar documentos, y si, el resultado de esa búsqueda, requiere alguna ordenación.
- O Remove o Update tiene que especificarse.
- Opcionalmente, se indica si se quiere devolver el nuevo documento (en un Update), o los campos que se quieren devolver, o si se quiere insertar si no existiera coincidencia con los criterios de búsqueda.
- En un Remove, devolverá el documento eliminado

db.collection.findAndModify().- Ejemplo

```
db.alumnos.findAndModify({  
  query: { nombre: "Helena", estado: "activa", calificacion: { $gt: 10 } },  
  sort: { calificacion: 1 },  
  update: { $inc: { puntos: -6 } }  
})
```

Ordena, todos los documentos que cumplan esas tres condiciones, por el campo “calificacion”; luego selecciona el primero y lo decrementa en seis unidades.

db.collection.findAndModify().- Ejemplo II

```
db.alumnos.findAndModify({
  query: { nombre: "Andrés" },
  sort: { calificacion: 1 },
  update: { $mul: { puntos: 2 } },
  upsert: true,
  new: true,
  fields: { nombre: 1, calificacion: 1 }
})
```

buscará todos aquellos documentos cuyo valor para el campo “nombre”, coincida con Andrés, los ordenará ascendentemente por su valor de campo “Rating”, actualizará el documento incrementando el campo “score” por el doble de su valor actual; si no existiera ningún documento con “nombre = Andrés”, lo crearía, y en todos los casos devolvería el nombre y el rating del documento actualizado

diferencias

`findOneAndUpdate()`: Buscará documentos que coincidan con los criterios de búsqueda, y actualizará el primero de ellos, según lo que se especifique en Update

`findOneAndReplace()`: Busca un documento (según unos criterios especificados) y lo reemplaza por otro. No se podrán utilizar operadores de Actualización (`$mul`, `$inc`, `$min`, `$set`, `$rename`,...)

```
db.trabajadores.updateMany({},{$rename:{'apodo':'alias','teléfono móvil':'Móvil'}})
```

`findAndModify()`: Modifica el primero de los encontrados tanto para update como para remove.

db.collection.findOneAndDelete()

```
db.collection.findOneAndDelete(  
  <filter>,  
  {  
    projection: <document>,  
    sort: <document>,  
  }  
)
```

```
db.puntuaciones.findOneAndDelete(  
  { "nombre" : "G.Banderas" },  
  {projection:{nombre:1, puntos:1}}  
)
```

Se devolverá el documento que ha sido borrado...

```
{ _id: 71, nombre: "G.Banderas", "puntos" : 8 }
```

db.collection.findOneAndDelete()

La diferencia con `db.collection.deleteOne()` → se borra directamente por coincidencia de un campo. Y sólo se muestra que la operación se realizó con éxito: `{ "acknowledged" : true, "deletedCount" : 1 }`

Y en este puedes ordenar (por si coincidieran varios) y eliminar el primero. Y además puedes elegir qué campos del documento eliminado mostrar para informar al usuario.

db.collection.renameCollection()

```
db.rrecord.renameCollection("record")
```

db.collection.count()

Devolverá el número total de documentos de una colección.

`db.orders.count()` es igual que : `db.orders.find().count()`

`db.orders.count({ ord_dt: { $gt: 3 } })`

igual que:

`db.orders.find({ ord_dt: { $gt: 3 } }).count()`

**ejercicio practicando estos últimos comandos
sobre los datos del ejercicio anterior de alumnos**

IMPORTAR un documento JSON

1.- Abrir el servidor y el cliente de mongo

2.- Situarse **en CMD** en la ruta:
Program Files\MongoDB\Server\4.0\bin>

3.- Ejecutar en CMD:
mongoimport.exe --db NombBD --collection NombColec --drop --file
"<Ruta_del_archivo_json_a_insertar>/NombArchivo.json"

IMPORTAR un documento JSON

1.- Situarse en CMD en la ruta:

```
Program Files\MongoDB\Server\4.0\bin>
```

2.- Ejecutar:

```
mongoimport.exe --db ciudades --collection ciudadesUSA --drop --file  
"C:\Users\buzon\Desktop\BootCamp  
UpgradeHub\MongoDB\ROBERTO\ciudadesUSA.json"
```

3.- Ejecutar:

```
mongoimport.exe --db restaurantes --collection restaurantesUSA --drop  
--file "C:\Users\buzon\Desktop\BootCamp  
UpgradeHub\MongoDB\ROBERTO\restaurants.json"
```

Operadores Relacionales

\$eq

\$gt

\$lt

\$in

\$ne

\$gte

\$lte

\$ni

Operadores Relacionales

“ Igual que ”: Función **\$eq**, se usa de la siguiente manera:

```
db.puntuaciones.find( { puntos : { $eq : 95 } } )
```

“ Distinto de ”: Función **\$ne**:

```
db.puntuaciones.find( {puntos: { $ne : 95 } } ).limit(10)
```

Operadores Relacionales

“ Mayor que ”: Función **\$gt**, se usa de la siguiente manera:

```
db.puntuaciones.find( { puntos : { $gt : 95 } } )
```

“ Mayor o igual que ”: Función **\$gte**:

```
db.puntuaciones.find( { puntos : { $gte : 95 } } )
```

Operadores Relacionales

“ Menor que ”: Función **\$lt**:

```
db.puntuaciones.find( { puntos : { $lt : 95 } } )
```

“ Menor o igual que ”: Función **\$lte**

```
db.puntuaciones.find( { puntos : { $lte : 95 } } )
```

Operadores Relacionales.- Ejemplos

```
db.puntuaciones.find( {puntos : { $gte : 90, $lt : 95} } )
```

¿Qué devuelve?

Devolverá todos los documentos que tengan puntajes entre 90 y 94 (incluyendo el número 90 y excluyendo el número 95)

Operadores Relacionales

Éstas funciones también aplican para letras, por ejemplo si se quieren los nombres de personas que empiecen por a, b y c es posible escribiendo el comando:

```
db.directores.find( { nombre: { $lte : "C" } } )
```

Operadores pertenencia

\$in, y \$nin : devuelven el conjunto de documentos (registros) que pertenecen (o no) al array impuesto.

Operadores de pertenencia

Función **\$in**: selecciona los documentos donde el valor de un campo es igual a cualquier valor en la matriz especificada

`db.tienda.find({ qty: { $in: [5, 15] } })` selecciona aquellos que tengan cantidad 5 o 15

Función **\$ni**: selecciona documentos en los que el valor del campo, no esté especificado en el array o documentos donde el campo no existe

`db.tienda.find({ qty: { $nin: [5, 15] } })` selecciona aquellos que no tengan ni 5 ni 15 unidades

Operadores pertenencia

```
db.medicamentos.find( { cantidad: { $in: [ 0, 5 ] } })
```

```
db.medicamentos.update(  
  { tags: { $in: ["analgesia", "resfriado"] } },  
  { $set: { venta:true } }  
)
```

Operadores - Ejemplo en JS

en JavaScript...

```
1. conn = new MongoClient();
2. db = conn.getDB("demografia");
3. cursor = db.ciudades.find({habitantes:{$gt:1000000}});
4. while (cursor.hasNext()) {
    printjson(cursor.next());
}
5. conn.close()
```

Operadores Relacionales - Ejemplo

```
db.ciudades.find({ciudad:{$ne: "Madrid"}})
```

¿Qué devuelve?

Operadores Relacionales - Ejercicio T^{co}

Vamos a buscar ciudades que tengan más de un millón de habitantes.

1.- Así el documento que componamos será:

```
{ $gt: 1000000 }
```

2.- Que se anidará al documento con el campo sobre habitantes:

```
{ habitantes: { $gt: 1000000 } }
```

3.- Sólo nos quedará poner las consultas mayor que en MongoDB dentro del método .find().

```
db.ciudades.find({ habitantes: { $gt: 1000000 } }, { NombCiudad: 1, _id: 0 } )
```

Filtros

Una de las maneras más importantes para filtrar información son con los comandos:

\$exists

\$type

\$regex

Filtros - Exists

Si tuviéramos una colección donde no todos los registros tengan determinado campo, podríamos filtrar la información de dos maneras, traer los datos que NO tengan "teléfono", o traer los que SI tengan, ambas opciones se harían de la siguiente manera:

```
db.personas.find( { "telefono" : { $exists : true } } )
```

```
db.personas.find( { "telefono" : { $exists : false } } )
```

Filtros - Type

Suponiendo que tenemos un campo que puede ser tanto como numérico, como tipo string, podemos filtrar la información con el comando **\$type** y se usa de la misma manera que el anterior filtro, con la diferencia de que en vez de poner valores true o false se pone el tipo de la variable que se requiere.

```
db.personas.find( { telefono : { $type : "string" } } )
```

Ejemplo:

```
db.notas.find( { "mediaClase" : { $type : [ "string" , "double" ] } }, { nombre:1, mediaClase:1, _id:0 } )
```

y devolverá tanto los registros que tengan el campo media: 9.33333333333 o "5.5", pero no devolverá las notas con formato INT, como: 6

Filtros

Tipos:

- double,
- string,
- object,
- array,
- objectid,
- bool,
- null,
- date,
- int,
- decimal,
- number (double, 32-bit integer, 64-bit integer, decimal)
- ...

Filtros - Regex

Por último **\$regex** nos ayuda a filtrar información que termine o contenga algún carácter en específico:

```
db.alumnos.find( { nombre: { $regex : "e" } } )
```

Retornará todas las personas que tengan un nombre que contenga la letra "e"

```
db.alumnos.find( { apellido : { $regex : "^R" } } )
```

 devolverá aquellos que su apellido empiece con "R"

```
db.alumnos.find( { apellido : { $regex : "z$" } } )
```

 Retornará todas las personas que tengan un apellido que termine con la letra "z"

Filtros - Regex

```
db.productos.find( { codigo: { $regex: /s/ } } )
```

Devolverá aquellos documentos que contengan una “s” formando parte de su campo “código”

```
db.productos.find( { descripcion: { $regex: /^S/} } )
```

y también funcionará:

```
db.productos.find({descripcion:/^M/})
```

Devolverá aquellos documentos en los que el campo descripción empiece con la letra S

```
db.productos.find( { codigo: { $regex: /s3$/ } } )
```

Devolverá aquellos documentos que contengan un “s3” finalizando su campo “código”

Operadores lógicos

OR

AND

NOT

NOR

Operadores lógicos: OR

Se usa el comando **\$OR** cuando se quiera filtrar por una u otra razón, pero que sólo es necesario que se cumpla una de las razones para que se imprima ese resultado, por ejemplo:

```
db.inventario.find( { $or: [ { qty: { $lt: 20 } }, { precio: 10 } ] } )
```

Esto mostrará aquellos documentos que cumplan que , o bien su stock está por debajo de 20 unidades o su precio es 10, o ambas.

Operadores lógicos: AND

El comando `$and` se usa cuando se quieren que ambas condiciones se aplican

```
db . inventario . find ( { $and : [ { precio : { $ ne : 1.99 } }, { precio : { $ exists : true } } ] } )
```

Esta consulta seleccionará todos los documentos en la inventory colección donde:

- el valor del campo “Precio” no es igual 1.99 **y**
- el campo “Precio” **existe**

lo mismo que: `db . inventario . find ({ precio : { $ ne : 1.99 , $exists : true } })`

OR y AND

El comando `$and` se usa cuando se quieren que ambas condiciones se aplican

```
db.people.find( { $and : [ { name : { $regex : "a$" } }, { name : $gt : "C" } ] } )
```

- Esto mostrará las personas que tengan un nombre que termine en "a" y empiecen por las letras "a", "b" o "c"

```
db.people.find( { name : { $regex : "a$", $gt : "C" } } ] } )
```

- Esto mostrará las personas que tengan un nombre que termine en "a" y empiecen por las letras "a", "b"

Operadores lógicos: NOT

```
db . inventario . find ( { precio : { $ not : { $ gt : 1.99 } } } )
```

Esta consulta seleccionará todos los documentos en la colección donde:

- el valor del campo “Precio” es menor o igual que 1.99 o
- el campo “Precio” no existe

Mientras que `db . inventario . find ({ precio : { $lte: 1.99 } })` devolvería sólo aquellos documentos con el precio menor de 1.99

Operadores lógicos: NOR

Realiza la operación lógica en un array de una o más expresiones de consulta y selecciona los documentos que **fallan en** todas las expresiones de consulta en el array

```
db.inventario.find( { $nor: [ { precio: 1.99 }, { oferta: true } ] } )
```

Devolverá Todos aquellos documentos que:

- contengan el campo “precio” con valor distinto a 1,99 y que contenga el campo “oferta” distinto de TRUE, **Ó**
- contengan el campo “precio” con valor distinto a 1,99 PERO no contengan el campo “oferta”, **Ó**
- no contengan el campo “precio”, PERO contengan el campo “oferta” distinto de TRUE **Ó**
- no contengan ni el campo “precio” ni el campo “oferta”

EJERCICIOS

Ejercicio 1.-

1.- insertar en la colección ciudades

```
{ "ciudad" : "Madrid", "habitantes" : 3233527, capital: "sí" }  
{ "ciudad" : "Barcelona", "habitantes" : 1620943 }  
{ "ciudad" : "Valencia", "habitantes" : 797028 }  
{ "ciudad" : "Sevilla", "habitantes" : 702355 }  
{ "ciudad" : "Zaragoza", "habitantes" : 679624 }
```

2.- ordenar por ciudad, ascendente, y sólo obtener los cuatro primeros documentos

3.- ordenar por población descendente

4.- obtener las ciudades que empiecen por “M”

5.- ordenar , a la vez, primero por nombre (desc) y luego por número de habitantes (asc)

6.- Obtener aquellas ciudades que sean capital, y mostrarlo.

7.- Mostrar el nombre de las ciudades que superen un millón de habitantes

8.-¿qué devolverá la consulta: db.ciudades.find({ciudad:{\$in:['Avila','Zamora','Madrid']}})?

9.- ¿Número de ciudades que componen la colección?

10.- Ciudades que empiecen por “B” o acaben por “z”

Ejercicio 1.- SOLUCIÓN

1.- `db.ciudades.insertMany([{ciudad:"Madrid",habitantes:3233527, capital:"sí"}, {ciudad:"Barcelona",habitantes:1620943}, {ciudad:"Valencia",habitantes:797028}, {ciudad:"Sevilla",habitantes:702355}, {ciudad:"Zaragoza",habitantes:679624}])`

2.- `cursor = db.ciudades.find({}, {_id:0}).sort({ciudad:1}).limit(4)`

3.- `cursor = db.ciudades.find({}, {_id:0}).sort({habitantes:-1})`

4.- `cursor = db.ciudades.find({ciudad:/^M/}).sort({ciudad:-1})`

5.- `cursor = db.ciudades.find().sort({ciudad:-1,habitantes:1})`

6.- `db.ciudades.find({ "capital" : { $exists : true } }, {_id:0})`

7.- `db.ciudades.find({ habitantes : { $gt : 1000000 } }, {ciudad:1,_id:0})`

8.- devolvería sólo los datos del documento con valor → `ciudad:"Madrid"`

9.- `db.ciudades.count()` o `db.ciudades.find().count()`

10.- `db.ciudades.find({$or:[{ciudad:/^M/},{ciudad:/ona$/}]}, {_id:0})`

Eso sí, la final no se te olvide recorrer el cursor para mostrar el contenido del mismo:

Modelado de datos

MongoDB Data Model

Quizás la gran duda a la hora de definir un esquema es:

¿Embeber subdocumentos, o hacer relaciones con otros documentos/colecciones?

MongoDB Data Model

```
{
  "_id": ObjectId("528f4e630fe5e6467e58ae7b"),
  "user_id": "user1",
  "contraseña": "1a2b3c",
  "sexo masculino",
  "edad": 17,
  "date_of_join": "16/10/2010",
  "educación": "mca",
  "profesión": "CONSULTOR",
  "Interés": "MÚSICA",
  "extra": {
    "nombre de la comunidad" : [
      "MÚSICA MODERNA",
      "MÚSICA CLÁSICA",
      "MÚSICA DEL OESTE"
    ]
    "community_moder_id": [
      "MR. Alex",
      "MR. Dang",
      "MR Haris"
    ]
  }
}

"Miembros de la comunidad" : [
  700,
  200,
  1500
]
"amigos" : {
  "valued_friends_id": [
    "kumar",
    "acosar",
    "anand"
  ]
  "ban_friends_id": [
    "Amir",
    "Raja",
    mont
  ]
}
```

MongoDB Data Model

Documentos Embebidos

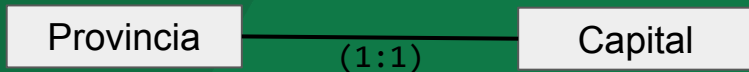
Se deberían usar cuando:

- No tienen suficiente entidad propia, es decir, carecen de sentido sin la información del documento padre.
- Son datos principalmente de lectura, no sufrirán muchos cambios.

MongoDB Data Model

Documentos Embebidos

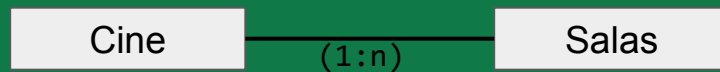
Relación 1:1



En las relaciones 1:1 no tiene sentido que cada registro se guarde en un documento aparte.

En estos casos el documento principal será el registro con más entidad, y el embebido el que tenga menos importancia.

Relación 1:n con mucha dependencia



Cuando se da una relación 1:n se embeberán los registros del lado "muchos" cuando no sean suficientemente importantes como para estar en un documento por sí solos.

MongoDB Data Model

```
{
  _id: ObjectId("11"),
  nombreProv: "Badajoz",
  población: 150000,
  gentilicio: "Pacense",
  provincia: {
    nombre: "Badajoz",
    capital: "Badajoz",
    siglas: "BA"
  }
}
```

Embebemos la información básica de la Provincia que se necesitamos usar.

La cantidad de información a guardar dependerá de nuestra aplicación y el tipo de consultas a hacer.

MongoDB Data Model

```
{
  _id: ObjectId("123"),
  empresa: "Kinépolis",
  ciudad: "Pozuelo de Alarcón",
  nombre: "Kinépolis Ciudad de la Imagen",
  salas: [
    {
      nombre: "Sala 1",
      capacidad: 204,
      pantalla: "6x14m"
    },
    {
      nombre: "Sala 25",
      capacidad: 1016,
      pantalla: "10x25m"
    }
  ]
}
```

La información de las salas debería ser la mínima necesaria que nos sea útil.

Si se mete mucha información corremos el riesgo de alcanzar el tamaño máximo del documento (16MB).

En caso de necesitar tener mucha información habría que considerar usar referencias.

MongoDB Data Model

MongoDB, al igual que las bases de datos documentales en general, es muy flexible a la hora de modelar la estructura de datos. Sin embargo no deja de ser una tarea compleja encontrar la estructura idónea para nuestro proyecto.

Como cualquier base de datos, hay que tener en cuenta dos factores:

- **LECTURAS.** Hay que optimizar las operaciones de lecturas más frecuentes, no sólo con índices, también organizando los datos para que sean fácilmente accesibles.
- **ESCRITURAS.** Intentar agilizar las escrituras más frecuentes para que sean lo menos bloqueante posible. Tener siempre en mente el mantenimiento de los índices.

MongoDB Data Model

Documentos Relacionados

Se deberían usar cuando:

- Cuando el mismo dato esté repartido por varios documentos (habría varios documentos embebidos iguales en varios documentos -> problemas en actualizaciones.
- Para representar relaciones más complejas.
- Cuando el subdocumento es grande (existe limitación en el tamaño de un documento a 16MB).

MongoDB Data Model

Documentos Relacionados

Referencias Manuales: cuando sólo se trabaja con otra colección, o se sabe a ciencia cierta hacia dónde va la referencia.

Se guardará el campo `_id` de un documento como referencia en otro documento.

```
{id_movie: ObjectId("345")}
```

Suponemos que la colección destino será **Movies** y busquemos en ella el identificador "345"

MongoDB Data Model

Documentos Relacionados

DBRef: cuando hay que resolver la referencia dinámicamente. Es un documento embebido compuesto por:

\$ref: colección destino

\$id: identificador del documento destino

\$db: (opcional) base de datos destino

Se crea:

```
{
  student: {
    $ref: "students",
    $id: "9999",
    $db: "college"
  }
}
```

Se muestra "la llamada":

```
{
  student: DBRef("students", "999")
}
```

MongoDB Data Model

Documentos Relacionados - EJEMPLO

```
{
  "_id": ObjectId("514d920b44ae16d201a3ff51"),
  "nombre": "Capuchino",
  "precio": 20,
  "marca": "Nescafé",
  "comentarios":
  [
    DBRef("Comentario", ObjectId("514d91a644ae16d201a3ff50"))
  ]
}
```

Documento de PRODUCTO con referencia a un documento de COMENTARIO

MongoDB Data Model

```
db.actors.insertOne({name: "qwerty usa",  
  pais: "USA", _id: ObjectId("2233")})
```

```
db.movies.insert(  
  {actor:  
    {  
      "$ref": "actors",  
      "$id": ObjectId("2233"),  
      "$db": "upflix"  
    }  
  }  
)
```

```
db.movies.find(  
  {  
    "actor.$id": ObjectId("2233")  
  }  
)
```

El DBRef se crea así, después se muestra como:

DBRef("actors", ObjectId("2233"), "upflix")

El orden es importante. Siempre tiene que ser (\$ref, \$id, \$db). \$db es opcional, por defecto es la misma base de datos.

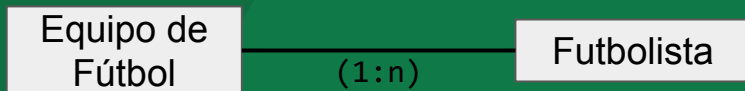
Así se realiza una búsqueda sobre un campo del DBRef.

Se están buscando todas las películas en donde aparezca el actor con identificador "2233"

MongoDB Data Model

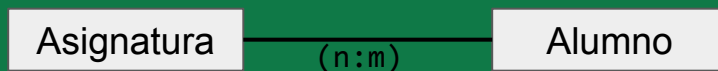
Documentos Relacionados

Relación 1:n



Cuando el campo con múltiple valores pueda tener muchos tipos de consultas y actualizaciones, es mejor separarlo en un documento aparte, y referenciarlo donde se necesite.

Relación n:n



En las relaciones “muchos a muchos” se suelen usar siempre referencias, ya que ambas entidades seguramente serán objeto de todo tipo de consultas.

MongoDB Data Model

Colección Equipos:

```
{
  _id: ObjectId("11"),
  name: "Extremadura UD",
  league: "La Liga 123",
  stadium: "Francisco de la Hera",
  players: [
    DBRef("players", "22"),
    DBRef("players", "25"),
    DBRef("players", "26")
  ]
}
```

Listado de referencias a los futbolistas.

Con este esquema, aquí sólo habría que añadir o quitar referencias, todos los datos del futbolista estarían en su colección particular.

MongoDB Data Model

Se pueden hacer referencias cruzadas si fuera necesario

colección Asignaturas / Subjects

```
{
  _id: ObjectId("99"),
  name: "Bases de Datos",
  centre: "Escuela Politécnica",
  career: "Grado en Ing del Software",
  course: 2,
  students:[
    DBRef("students","622"),
    DBRef("students","650"),
    DBRef("students","712"),
    DBRef("students","720")
  ]
}
```

colección Alumnos / Students

```
{
  _id: ObjectId("622"),
  name: "Juan López",
  birthday: ISODate("30/02/1998"),
  career: "Grado en Ing del Software",
  subjects:[
    DBRef("subjects","78"),
    DBRef("subjects","98"),
    DBRef("subjects","99"),
    DBRef("subjects","100"),
    DBRef("subjects","102")
  ]
}
```

MongoDB Data Model

Replicación de datos

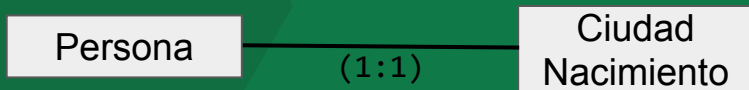
- En general no es una práctica recomendable
- Se trata de duplicar datos a lo largo de la base de datos
- Esta técnica se usa para optimizar las búsquedas: el objetivo ideal es que un documento tenga toda la información necesaria
- Sólo hacerlo en datos con pocas inserciones y muy raras actualizaciones

No es siempre tan sencillo, hay que pensar qué tipo de operaciones se realizarán, y optimizarlas.

MongoDB Data Model

¿Cuándo y cómo replicamos los datos?

Son datos con poca entidad por sí misma, y van a tener pocas o ninguna actualización.

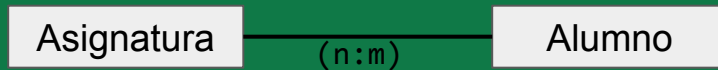


Ejemplo: Cada persona tiene un subdocumento con su ciudad de nacimiento con los campos:

Cod_Ciudad, Nombre_Ciudad, Provincia.

Estos datos estarán duplicados (en una ciudad nacen muchas personas), pero normalmente esos datos no se modificarán.

No es necesario replicar todos los datos, se podría replicar sólo los necesarios, y referenciar al documento original para obtener el resto.



Ejemplo: Cada alumno tendrá un listado de asignaturas con datos básicos: **Nombre_Asign, Calificación, Curso, Referencia_Asignatura.**

Para sacar el resto de datos de la asignatura cuando sea necesario, se tiene que resolver la referencia.

MongoDB Data Model

- La mezcla de Documentos embebidos y Referencias no es exclusiva en datos replicados
- Se puede usar una estructura normal de datos.

Si quisiéramos saber más de alguno de los jugadores, deberíamos buscar según su `_id` devuelto en la tabla jugadores

colección de equipos

```
{
  _id: ObjectId("1234"),
  "name": "C.D. Badajoz",
  "league": "Segunda División B",
  "stadium": "Nuevo Vivero",
  "players": [
    {
      "player_id": ObjectId("11"),
      "shortName": "Guzmán",
      "position": "Extremo derecho"
    },
    {
      "player_id": ObjectId("22"),
      "shortName": "César Morgado",
      "position": "Defensa Central"
    }
    ...
  ]
}
```

MongoDB Data Model

- La mezcla de Documentos embebidos y Referencias no es exclusiva en datos replicados
- Se puede usar una estructura normal de datos.

colección de jugadores/player

```
{
  _id: ObjectId("11"),
  "shortName": "Guzmán",
  "fullName": "Guzmán Casaseca Lozano",
  "position": "Extremo Derecho",
  "birthDate": ISODATE("1993-01-26"),
  "statistics": {
    "matches": 13,
    "goals": 0,
    "assists": 1
  }
},
{
  _id: ObjectId("22"),
  "shortName": "César Morgado",
  "fullName": "César Morgado Ortega",
  "position": "Defensa Central",
  "birthDate": ISODATE("1984-12-26"),
  ...
},
```

MongoDB Data Model

Por lo general, para embeber o referenciar puedes tener en cuenta estas pautas para tomar la decisión correcta:

Embeber es mejor para:

- Pequeños subdocumentos.
- Datos que no cambian regularmente.
- Cuando la consistencia final es aceptable.
- Documentos que crecen en pequeñas cantidades.
- Datos que necesitarás a menudo para realizar una segunda consulta para obtener lecturas rápidas.

MongoDB Data Model

Referenciar es mejor para:

- Subdocumentos grandes.
- Datos volátiles.
- Cuando es necesaria la consistencia inmediata.
- Documentos que crecen una gran cantidad.
- Datos que a menudo excluye de los resultados.
- Escrituras rápidas.

MongoDB Data Model

Ejemplo: Diseño de un Blog - Funcionalidades

- Un usuario se puede darse de alta, modificar sus datos, y darse de baja.
 - Un usuario puede escribir tantos Posts como desee.
 - Cada Post tiene información básica: título, texto e imágenes.
 - Los Posts se categorizan con tags.
 - Existirá una sección con los Posts más leídos.
 - Existirá una sección con los tags más populares (que tengan los Posts más leídos).
 - Existirá una sección con los Usuarios más valorados (con mejores Posts).
- Un usuario puede suscribirse a otro Usuario para recibir notificaciones de sus nuevos Posts.
 - Un usuario puede suscribirse a un tag.
 - Un usuario puede buscar Posts por su título.
 - La *home* del usuario será una combinación personalizada de Posts según sus suscripciones a tags y/o usuarios.
 - Un usuario puede comentar un Post.
 - Existirá una sección con los Posts más comentados.

MongoDB Data Model

Conclusiones

- Hay una colección clara: Usuarios. Para empezar tendrá los datos básicos del usuario.
- ¿Qué hacemos con los Posts?
 - Es una relación 1:n con **Usuarios**
 - Vamos a tener que mostrarlos de forma independiente al Usuario
 - Sobre todo tendrá lecturas, pero puede ser modificado
 - **Conclusión:** los Posts van en una colección separada
- Por lo tanto, los usuarios tendrán un array con referencias a sus Posts; y cada post, una referencia a su autor
- ¿Cómo gestionamos los Tags?
 - Tienen una relación n:m con los Posts
 - En principio una vez creados no se modificarán
 - Los usuarios se pueden suscribir a los tags.
 - **Posible solución:** array de subdocumentos en Posts y Usuarios, con los datos de sus tags (los del Post, y los suscritos)
- Comentarios: no tienen identidad por sí mismo. Van embebidos en el Post.

MongoDB Data Model

```
{
  "_id": ObjectId("1"),
  "name": "John Doe",
  "photo": "image.jpg",
  "description": "Lorem ipsum dolor...",
  "posts": [
    {postId: ObjectId("1")},
    {postId: ObjectId("2")}
  ],
  "userFollows": [
    {userId: ObjectId("2")}
  ],
  "tagsLike": [
    {
      "idTag": 111,
      "tagName": "Tag 1"
    }
  ]
}
```

Colección USUARIOS

Datos del usuario

Referencias a Posts

o también: DBRef("posts", "89"),

Referencias a otros Usuarios

Conjunto de Tags

MongoDB Data Model

```
{
  "_id": ObjectId("1"),
  "title": "Sample Title",
  "photo": "image.jpg",
  "body": "Lorem ipsum dolor...",
  "userId": ObjectId("1"),
  "comments": [
    {
      userId: ObjectId("2"),
      comment: "..."
    }
  ],
  "tags": [
    {
      "idTag": 111,
      "tagName": "Tag 1"
    }
  ]
}
```

Colección POSTS

Datos de un Post

Referencia al autor

Conjunto de Comentarios

Conjunto de Tags

MongoDB Data Model

Operaciones a realizar sobre las colecciones

- CRUD de Usuarios: escritura/lectura en colección de Usuarios.
- Suscribirse a tags/usuario: añadir referencia al array correspondiente
- Crear un Post: añadir entrada en Post, y añadir referencia en Usuario
- Añadir Comentario: añadir entrada en array de Comentarios del Post
- Mostrar últimos posts: últimos n Posts ordenados por fecha
- Últimos posts filtrando por tag y/o usuario
- Posts más vistos de un usuario: ordenar los posts por visitas

MongoDB Data Model

CRUD de usuarios

```
{
  "_id": ObjectId("1"),
  "name": "John Doe",
  "photo": "image.jpg",
  "description": "Lorem ipsum
dolor...",
  "posts": [
    {post_Id: ObjectId("1")},
    {post_Id: ObjectId("2")}
  ],
  "userFollows": [
    {user_Id: ObjectId("2")}
  ],
  "tagsLike": [
    {
      "Tag_id": 111,
      "tagName": "Tag 1"
    }
  ]
}
```

```
db.usuarios.insert({name: "...", photo:"...",
description: "..."});
```

```
db.usuarios.find({name: "John Doe"});
```

```
db.usuarios.update({_id: ObjectId("1")},
{$set:{description: "..."}});
```

```
db.usuarios.remove({_id: ObjectId("1")});
```

MongoDB Data Model

```
{
  "_id": ObjectId("1"),
  "name": "John Doe",
  "photo": "image.jpg",
  "description": "Lorem ipsum
dolor...",
  "posts": [
    {post_Id: ObjectId("1")},
    {post_Id: ObjectId("2")}
  ],
  "userFollows": [
    {user_Id: ObjectId("2")}
  ],
  "tagsLike": [
    {
      "Tag_id": 111,
      "tagName": "Tag 1"
    }
  ]
}
```

Suscribirse y desuscribirse a tags/usuarios

1. Añade un nuevo seguidor

```
db.usuarios.update({_id: ObjectId("1")}, {$addToSet:
{userFollows: ObjectId("3")} });
```

2. Elimina un tag al que está suscrito

```
db.usuarios.update({_id: ObjectId("1")}, {$pull:
{tagsLike: {Tag_id: 111}} });
```

3. Añade dos tags y elimina a todos los usuarios a los que sigue

```
db.usuarios.update({_id: ObjectId("1")}, {$addToSet:
{tagsLike: { $each: [{Tag_id: 222, tagName:
"Tag2"},{idTag: 333, tagName: "Tag3"}]}},
$set:{userFollows: []} });
```

MongoDB Data Model

```
{
  "_id": ObjectId("1"),
  "title": "Sample Title",
  "photo": "image.jpg",
  "body": "Lorem ipsum dolor...",
  "userId": ObjectId("1"),
  "comments": [
    {
      userId: ObjectId("2"),
      comment: "..."
    }
  ],
  "tags": [
    {
      "idTag": 111,
      "tagName": "Tag 1"
    }
  ]
}
```

Crear un Post nuevo del User1

Creación del propio post:

```
db.posts.insert({title: "Título del nuevo artículo",
photo:"nueva_foto.png", body:"Este es el contenido
del artículo", userId: ObjectId("1"), tags: [{idTag:
111, tagName: "Tag 1"}, {idTag: 333, tagName: "Tag
3"}]});
```

Actualización de la referencia al post en la colección usuario.

```
db.usuarios.update({_id: ObjectId("1")},
{$addToSet:{posts: ObjectId("2")}});
```

Suponemos que el usuario que crea el Post es el usuario 1, y que el identificador del propio Post una vez creado será el 2

MongoDB Data Model

```
{
  "_id": ObjectId("1"),
  "title": "Sample Title",
  "photo": "image.jpg",
  "body": "Lorem ipsum dolor...",
  "userId": ObjectId("1"),
  "comments": [
    {
      userId: ObjectId("2"),
      comment: "..."
    }
  ],
  "tags": [
    {
      "idTag": 111,
      "tagName": "Tag 1"
    }
  ]
}
```

Añadir un comentario

```
db.posts.update({_id: ObjectId("1")},
{$push:{comments: {userId: ObjectId("2"), comment:
"Esto es un nuevo comentario del usuario 2 al
artículo del usuario 1"}}});
```

MongoDB Data Model

```
{
  "_id": ObjectId("1"),
  "title": "Sample Title",
  "photo": "image.jpg",
  "body": "Lorem ipsum dolor...",
  "userId": ObjectId("1"),
  "createdAt":
    ISODate("01-12-2018:19:00:00"),
  "comments": [
    ...
  ],
  "tags": [
    {
      "idTag": 111,
      "tagName": "Tag 1"
    }
  ]
}
```

Mostrar últimos Posts

últimos 10 post creados

```
db.posts.find().sort({createdAt:-1}).limit(10);
```

últimos 10 post creados por un usuario

```
db.posts.find( {userId: ObjectId("1")} ).sort(
  {createdAt:-1} ).limit( 10 );
```

últimos 10 post creados de una determinada temática

```
db.posts.find( {"tags.idTag": ObjectId("111")}
  ).sort( {createdAt:-1} ).limit( 10 );
```

MongoDB Data Model

```
{
  "_id": ObjectId("1"),
  "title": "Sample Title",
  "photo": "image.jpg",
  "body": "Lorem ipsum dolor...",
  "userId": ObjectId("1"),
  "createdAt":
    ISODate("01-12-2018:19:00:00"),
  "views": 99,
  "comments": [
    ...
  ],
  "tags": [
    ...
  ]
}
```

Mostrar Posts más vistos

5 últimos posts

```
db.posts.find().sort({views:-1}).limit(5);
```

en la condición del find, podremos filtrar por **tags** o por **usuario**

Los 10 Posts más vistos del usuario "1" con el tag "111"

```
db.posts.find({userId: ObjectId("1"), "tags.idTag":
111}).sort({views:-1}).limit(10);
```

MongoDB Data Model

Ejercicio: Diseñar la estructura de datos de Twitter

- Vamos a centrarnos en cómo representar un Tweet
 - Un tweet consta básicamente de:
 - Texto, multimedia, fecha de creación
 - Tiene que tener información del usuario que escribió el tweet: nombre, id, foto, descripción
 - Almacenará el número de retweets y favoritos que tiene.
- Tendrá el listado de menciones (respuestas):
 - Cada mención será otro tweet hecho por otro usuario
 - El tweet tendrá que saber si “él mismo” es una mención a otro tweet, o no.
 - Si es una mención tendrá que saber a qué tweet está contestando
 - Los multimedia tendrán cierta información
 - tipo (imagen, video...), url, id, dimensiones

MongoDB Data Model

Conclusiones

- El tweet tendrá información básica: id, texto, numRetweets, numFavorites, fecha de creación.
- Podrá tener o no elementos multimedia. Como hay que almacenar varios datos, tendrá que ser un documento embebido.
- Para guardar los datos del usuario, podríamos suponer que está en una colección aparte, pero por eficiencia lo guardamos como documento embebido.
- Las menciones serán un array de objetos. En cada objeto se guardará información básica de cada tweet: texto, multimedia, usuario y a su vez el listado de sus menciones; y por último una referencia al tweet.
 - La complicación de esto sería mantener actualizado las menciones: merece la pena porque hay muchas más lecturas que escrituras
- Para saber si el tweet es, a su vez, una mención hay que guardar una referencia al tweet origen, y un campo booleano.

MongoDB Data Model

```
{ _id: ObjectId("111"),  
  FechaCreacion: ISODate("01-12-2018:19:00:00")  
  texto: "...",  
  numLikes: 20,  
  numRetweets: 200,  
  numFavs: 5,  
  usuario: {  
    "id": "3322", foto: " ", desc: " "  
  },  
  multimedia: {  
    tipo: "...", url: "...",  
    dimensiones: { alto: 11, ancho: 22 }  
  },  
  esMencion: true,  
  idOrigenMencion: ObjectId("100"),  
  menciones: [  
    {  
      id: ObjectId(), texto: "", media: {},  
      menciones: [{}, {}]  
    }  
  ]  
}
```

Esquema de cada Tweet

Body del tweet y resto de campos

Datos del Usuario que escribe el tweet

Subdocumento con los elementos multimedia

¿es mención de otro tweet? ¿de cuál?

Listado de menciones o respuestas a este propio Tweet

Índices

MongoDB Índices

Al igual que sucede en las bases de datos relacionales, es conveniente crear índices para optimizar las consultas que sean más habituales y que puedan resultar costosas.

Al ejecutar una consulta sin índices, MongoDB tiene que recorrer toda la colección para filtrar los datos a mostrar, lo cual será más lento cuanto mayor sea el número de documentos de la colección.

Los índices almacenan la información de forma que sea más rápidamente accesible. Los datos se ordenarán según el orden especificado.

Esa estructura de datos no resulta gratis: se tiene que actualizar por cada inserción. Ocupa SU espacio. No hay que crear índices indiscriminadamente.

MongoDB Índices

Identificador de la colección

Por defecto MongoDB crea un identificador de la colección: el campo **_id**

_id es un identificador único, que se crea automáticamente en base a la hora, un valor aleatorio y un contador incremental.

El valor de _id es un objeto, del tipo ObjectId.

```
db.coll.findOne(ObjectId("52f602d787945c344bb4bda5"));  
db.coll.find({_id:ObjectId("52f602d787945c344bb4bda5")})
```

Estas dos consultas son inmediatas: usan el índice _id para obtener el documento buscado

MongoDB Índices

Índice de un sólo campo

- Es el índice más simple. Se usa cuando se hacen consultas sobre un campo, o al ordenar sobre ese campo.
- El valor numérico indica el orden del índice: 1 para ascendente, -1 para descendente.
- El índice se puede hacer de un campo normal, un campo dentro de un objeto, o incluso de un objeto entero.

```
db.product.createIndex({type:1})  
db.product.createIndex({"manufacturer.name":-1})
```

crea dos índices dentro de la colección "product".

- Uno para el campo "type",
- Otro para el subcampo "manufacturer.name"

MongoDB Índices

Ejemplos de sentencias y uso de índices

```
db.product.find({type:"book"});  
db.product.find({type:"book", price:{$gt:100}})  
db.product.find({}).sort({book:1});
```



Se utiliza el índice "type"

```
db.product.find({"manufacturer.name":"Dell"});  
db.product.find({price:100})  
    .sort({"manufacturer.name":1});
```



Se utiliza el índice
"manufacturer.name"

```
db.product.find({  
    "manufacturer.name":"LG",  
    "id": {$gt: 100}  
});  
db.product.find({"manufacturer.name":"HP"})  
    .sort({_id:1});
```



En estos caso se utiliza sólo uno de los
índices. MongoDB calcula cuál es el
óptimo de los dos.

MongoDB Índices

Índice compuesto

- Los índices se pueden crear de tantos valores como se necesite. Siempre hay que especificar el orden que tendrá el campo.
- Este índice se aplica cuando se haga consulta por el primer campo, o por los dos primeros, o por los tres primeros... así hasta completar todos.
- En la ordenación sólo se aplicará cuando coincidan con cómo se ha creado el índice.

```
db.product.createIndex({type:1, price:-1})
```

Crea un único índice por dos campos, “type” de forma ascendente y “price” de forma descendente.

Actuará cuando busquemos por ambos campos o cuando sólo busquemos por “type”

MongoDB Índices

Índice compuesto

`{type:1, price:-1}`

book, 120
book, 100
book, 95
book, 60
laptop, 1500
laptop, 1000
laptop, 800
shoe, 90
shoe, 85
watch, 120

Internamente los índices compuestos se almacenan de esta manera.

La ventaja de estos índices es que también se puede aprovechar los subíndices de izquierda a derecha. Es decir, crear un índice de 4 campos es como si también se creara uno de 3, otro de 2 y otro de 1.

`(w, x, y, z)`

`(w, x, y)`

`(w, x)`

`(w)`

Pero NO existe índice de (x) o de (y) individualmente

MongoDB Índices

Ejemplos de sentencias y uso de índices

```
db.product.find({type:"book", price:60});  
db.product.find({type:"laptop"}).sort({price:-1})  
db.product.find({price:100}).sort({type:1})
```

```
db.product.find({}).sort({type:-1, price:1});
```

```
db.product.find({}).sort({type:-1});  
db.product.find({type:"book"});
```

```
db.product.find({}).sort({type:1, price:1});
```

```
db.product.find({}).sort({price:-1});  
db.product.find({price:100});
```

→ **Aprovecha el índice al usar ambos campos en la búsqueda y/o en la ordenación**

→ **También lo aprovecha porque el orden es justo el inverso**

→ **Al usar sólo el primer campo, también aprovecha el índice**

→ **NO lo aprovecha porque el orden no coincide**

→ **Como sólo usa el segundo campo, NO aprovecha el índice**

MongoDB Índices

Índice de arrays

En MongoDB también se permite crear índices para campos que son un array de valores.

Estos campos se pueden combinar con campos normales para crear índices compuestos. Sin embargo no se puede crear un índice compuesto con dos o más campos que sean arrays.

Internamente MongoDB crea una copia del documento con cada valor del array por separado, para optimizar la búsqueda. Esto implica que éste sea un índice costoso de mantener y no sea muy recomendable usar.

MongoDB Índices

Índice de arrays

```
{
  type: "shoe",
  name: "Reebok Classic",
  colors: ['green', 'red', 'blue'],
  price: 90
},
{
  type: "shoe",
  name: "Adidas Original",
  colors: ['black', 'red'],
  price: 100
}
```

```
db.product.createIndex({colors:1})
```

Crearé un índice para:

- black
- blue
- green
- red

MongoDB Índices

Índice de texto

- En el caso de cadenas de texto, los índices normales son útiles cuando se busca la cadena completa del texto. Sin embargo, cuando se busca usando una expresión regular el índice normal no se aplica.
- Para búsquedas de textos parciales es necesario crear un índice específico de texto.
- Una colección puede tener varios campos con índice de texto, y estos índices se pueden combinar con los índices normales para crear índices compuestos.

```
db.coll.createIndex({NomCampo:"text", NomCampo2:"text"})
```

MongoDB Índices

Índice de texto

Como estos índices son un caso especial, las búsquedas que se hacen para poder aprovecharlo son algo diferentes. Sólo se puede crear un índice de tipo texto en una colección, por ese motivo en las búsquedas no se especifica sobre qué campo se va a buscar. La forma de buscar es usando las funciones **`{ $text: { $search: "texto a buscar" } }`**

```
db.coll.find({$text:{$search:  
    "palabra1 -palabra2 \"una frase literal\""}})
```

MongoDB Índices

Índice de texto

```
db.coll.find({$text:{$search:  
  "palabra1 -palabra2 \"una frase literal\" "}})
```

En este caso se da por supuesto que la colección “coll” tiene un índice de texto, sobre uno o varios campos, y se realiza una búsqueda sobre dicho índice. No se especifica en ningún momento el campo en el que se busca porque sólo puede haber un índice de texto.

La búsqueda que se se hace en el ejemplo es: dame todos los documentos que contengan en el texto al menos una de estas palabras (o frases): “palabra1”, “una frase literal”, y que no incluya “palabra2”

MongoDB Índices

Índice de texto

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

Usando la anotación especial "\$**" MongoDB creará un índice de texto por cada campo que sea texto. Esto permitirá realizar búsquedas de texto sobre todo el documento.

MongoDB Índices

Restricciones de índice de texto

- Sólo puede haber un índice de texto por colección.
- No se aplica en las ordenaciones.
- No se puede crear un índice compuesto en donde se combine un índice de texto y uno de array.

Idioma del índice

- Se puede especificar un idioma para el índice.
- También se puede especificar un idioma para cada documento en concreto, que sobrescribe el del índice.
- El idioma se usa para optimizar la búsqueda: ignora las palabras más comunes del idioma para que sea más rápido.

MongoDB Índices

**¿Cómo sabemos si estamos
aprovechando un índice?**



MongoDB Índices

Explain

Con el comando `explain` podemos obtener información de cómo se ha hecho la consulta:

- Número de documentos escaneados y números de documentos devueltos
- Tiempo que ha tardado la consulta en ejecutarse
- Listado de posibles índices a usar
- De todas las posibilidades, el índice que realmente ha usado

MongoDB Índices

Explain

- `db.products.explain().count({ quantity: { $gt: 50 } })`
- `db.products.explain("executionStats").find({ quantity: { $gt: 50 }, category: "apparel" })`
- `db.products.explain("allPlansExecution").update({ quantity: { $lt: 1000 }, category: "apparel" }, { $set: { reorder: true } })`
- <https://docs.mongodb.com/manual/reference/command/explain/index.html>
- <https://docs.mongodb.com/manual/reference/explain-results/>

MongoDB Índices

Ejemplo

Supongamos que tenemos la colección de emails de Enron, y hacemos la siguiente consulta:

```
db.messages.find({"headers.From":"eric.bass@enron.com",  
"headers.Date":{"$gt":  
ISODate("2001-04-07T00:00:00")}}).sort({filename:1});
```

MongoDB Índices

```
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 13,
  "executionTimeMillis" : 115,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 120477,
  "executionStages" : {
    "stage" : "SORT",
    "nReturned" : 13,
    "executionTimeMillisEstimate" : 100,
    "memUsage" : 23730,
  }
}
```

Al hacer el **explain("executionStats")** de esa consulta devuelve, de forma resumida, esta información.

- Ha tardado 115 milisegundos
- Ha devuelto 13 documentos
- Ha examinado 120477 documentos
- Ha consumido 23Kb de memoria

MongoDB Índices

```
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 13,
  "executionTimeMillis" : 115,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 120477,
  "executionStages" : {
    "stage" : "SORT",
    "nReturned" : 13,
    "executionTimeMillisEstimate": 100,
    "memUsage" : 23730,
  }
}
```

El problema es que presumiblemente esta colección irá creciendo con el tiempo:

- El número de documentos a recorrer cada vez será mayor
- Por lo tanto la consulta cada vez será más lenta
- Además, la memoria que consume poco a poco irá creciendo

MongoDB Índices

¡¡SOLUCIÓN!!

```
db.messages.createIndex({"headers.From": 1, filename:1});
```

... y repetimos la consulta:

```
db.messages.find({"headers.From":"eric.bass@enron.com",  
"headers.Date":{"$gt":  
ISODate("2001-04-07T00:00:00")}}).sort({filename:1});
```


MongoDB Índices

```
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 13,
  "executionTimeMillis" : 0,
  "totalKeysExamined" : 160,
  "totalDocsExamined" : 160,
  "executionStages" : {
    "stage" : "FETCH",
    "filter" : {
      "headers.Date" : {
        "$gt" : ISODate("2001-04-07")
      }
    },
    "nReturned" : 13,
    "executionTimeMillisEstimate" : 0,
  }
}
```

Al hacer nuevamente el **explain("executionStats")** después de crear el índice, el resultado es este.

- Ha tardado 0 milisegundos
- Ha devuelto 13 documentos
- Ha examinado 160 documentos
- No ha consumido memoria extra

MongoDB Índices

```
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 13,
  "executionTimeMillis" : 0,
  "totalKeysExamined" : 160,
  "totalDocsExamined" : 160,
  "executionStages" : {
    "stage" : "FETCH",
    "filter" : {
      "headers.Date" : {
        "$gt" : ISODate("2001-04-07")
      }
    },
    "nReturned" : 13,
    "executionTimeMillisEstimate" : 0,
  }
}
```

Ahora da igual cuánto crezca la colección, que ésta consulta siempre va a ser prácticamente inmediata.

En este caso ha devuelto 160 documentos porque son los que coinciden con la comprobación del email, el resto se filtra por la fecha (no tenemos indexado esto).