
MANUAL DE MONGODB

1.1 ¿Qué es NoSQL?

NoSQL o "No solamente SQL" (Not Only SQL) es un término acuñado por Carlo Strozzi en 1998 y nuevamente retomado por Eric Evans en 2009 y se refiere a un conjunto de bases de datos que se diferencian en gran parte de las bases de datos convencionales, en características tanto de uso como de implementación; estos tipos de bases de datos no usan SQL o al menos no como lenguaje predeterminado para realizar las consultas. Las bases de datos NoSQL, no soportan totalmente ACID, esto lo explica el teorema del profesor Eric Brewer, Teorema CAP (2000), donde menciona:

Es imposible para un sistema distribuido garantizar simultáneamente las siguientes tres características:

- **Consistency** (Consistencia): todos los nodos ven los mismos datos al mismo tiempo.
- **Availability** (Disponibilidad): una garantía de que todas las consultas recibirán siempre una respuesta válida (independientemente de si el requerimiento fue exitoso o fallido).
- **Partition Tolerance** (Tolerancia a Particiones): el sistema continúa operando a pesar de la pérdida arbitraria de mensajes, o ante fallos de parte del sistema.

Aunque a priori puede suponer una desventaja, realmente esto permite que los motores de bases de datos no relacionales se puedan escalar fácilmente de manera horizontal. Para subsanar el problema de ACID, nuevamente el profesor Brewer ideó BASE (Basically Available, Soft-state, Eventually consistent) que lo conforman los siguientes puntos:

- **Disponibilidad Básica:** para cada solicitud se garantiza una respuesta satisfactoria o un error en la ejecución.
- **Estado "Soft":** El estado del sistema puede cambiar con el tiempo, en ocasiones incluso aunque no se hayan incluido nuevas inserciones en el sistema.
- **Consistencia Eventual:** la base de datos puede ser en un momento inconsistente pero acabará siendo consistente con el tiempo.

El lenguaje SQL no es un lenguaje predominante entre los distintos tipos de bases de datos no relacionales, por lo general cada motor tiene su propio lenguaje de consultas. Cabe destacar que la información no se almacena con un esquema fijo (pero si usando almacenamiento estructurado), aunque sí existe un esquema que el ABD (Administrador de Bases de Datos) o el DBD (Desarrollador de Bases de Datos) propone con anterioridad de manera virtual, es decir, no se crea en el motor antes de utilizar la base de datos sino al almacenar el primer valor.

1.2 ¿Qué NO es NoSQL?

El término NoSQL no es una base de datos y tampoco un tipo de base de datos, sino una definición que engloba un conjunto de tipos de bases de datos que difiere con las bases de datos relacionales.

1.3 Tipos de bases de datos NoSQL

En el mundo de las bases de datos no relacionales nos encontramos con distintos modelos o tipos, que se desempeñan mejor en algunos ambientes específicos; esas distintas facetas no se ven en las bases de datos relacionales. En este manual se exponen los tipos más comunes.

1.3.1 Bases de datos documentales

Las bases de datos orientadas a documentos o también denominadas como Bases de Datos Documentales, trabajan bajo el marco de la definición de un "*Documento*", donde cada motor que usa esta definición difiere en los detalles, pero la mayoría concuerda en cómo se almacena la información con algún formato estándar. Los formatos más utilizados por los motores más populares son: JSON y BSON.

Cada documento, es muy similar a un registro en una base de datos relacional, donde se puede observar un esquema parecido aunque no rígido. Es decir, dos documentos no tienen porqué tener el mismo esquema, aunque sean de una misma colección de datos.

```
{
  _id : 1 ,
  nombre : "MongoDB" ,
  url : " http://www.mongodb.org",
  tipo : "Documental"
}
```

Ejemplo de un documento en Formato JSON.

Este ejemplo demuestra la sencillez de un documento. Concretamente, se observa un modelo al estilo **clave : valor**. Una analogía con las bases de datos relacionales sería: Clave = Campo y Valor = Dato del campo.

1.3.2 Bases de datos clave-valor

Este tipo de bases de datos es muy similar a las bases de datos documental en el concepto de guardar la información con el modelo **clave : valor**, la diferencia radica en que un documento se almacena en una clave; esta definición puede parecer algo abstracta. Esto se explica mejor con un ejemplo donde se transforma el modelo documental anterior a un esquema clave-valor.

```
mongodb => {
  _id : 1 ,
  nombre : "MongoDB" ,
  url : "http://www.mongodb.org" ,
  tipo : "Documental "
```

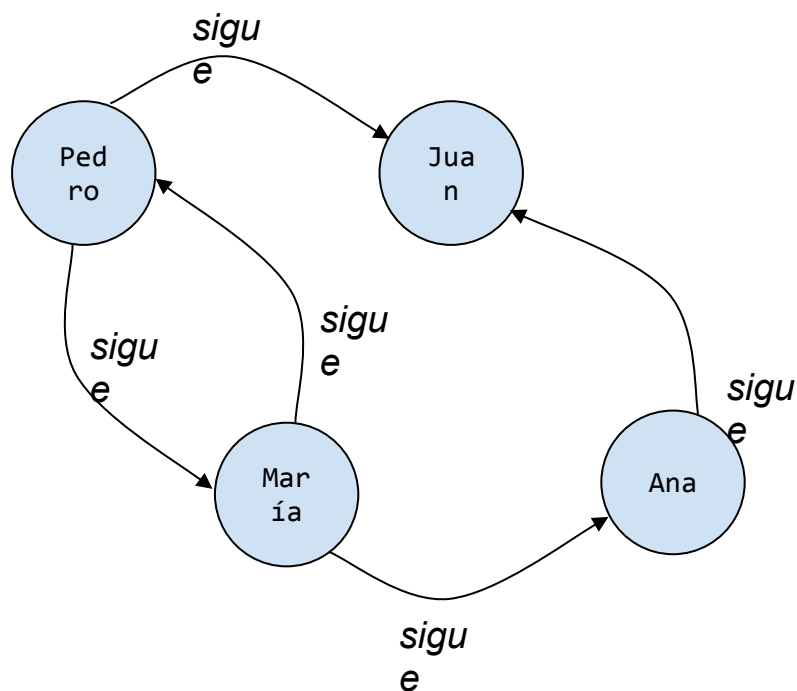
}

Ejemplo de una entrada en una base de datos clave-valor.

La clave en este caso es 'mongodb' y su contenido es el mismo documento de la sección anterior. Esto hace que varíe la forma de recuperar la información con respecto a las bases de datos basadas en documentos. Algo muy interesante de este tipo de bases de datos es que permite ser utilizado junto a bases de datos orientadas a documentos, lo que origina motores de bases de datos híbridos.

1.3.3 Bases de datos orientadas a grafos

Este tipo difiere completamente a los tipos antes mencionados, y trata la información de una manera peculiar usando grafos y teoría de grafos. Cada nodo debe contener una sola columna, por lo tanto se deben normalizar completamente las bases de datos.



Ejemplo de una base de datos orientada a grafos

1.4 ¿Por qué usar NoSQL?

En la actualidad se generan cantidades enormes de datos desestructurados. En este contexto, las bases de datos relacionales empiezan a mostrar deficiencias tanto a nivel de almacenamiento como de ejecución de operaciones. Concretamente, esta última es una de las principales razones de impulsar el uso de bases de datos no relacionales. Muchas personas se quejan del movimiento NoSQL, más que nada por una resistencia al cambio, que por los propios contras de este tipo de bases de datos. Concretamente NoSQL permite gestionar fácilmente una cantidad de datos gigantesca. Esto no es tan sencillo si

piensas en estructuras y bases de datos relaciones. Otra de las razones relevantes es la arquitectura, que permite escalar horizontalmente de manera sencilla sin tantos problemas de rendimiento.

La velocidad de desarrollo y la velocidad de la base de datos son puntos a favor para las bases de datos no relacionales, reduciendo el tiempo de desarrollo evitando complejas sentencias SQL y además aumentando la velocidad de respuestas para los clientes.

2.1 MongoDB

MongoDB es un sistema de bases de datos no relacionales, multiplataforma e inspirado en el tipo de bases de datos documental y clave/valor, su nombre proviene del término en inglés "humongous". Está liberada bajo licencia de software libre, específicamente GNU AGPL 3.0. MongoDB usa el formato BSON (JSON Compilado) para guardar la información, dando la libertad de manejar un esquema. Este motor de bases de datos es uno de los más conocidos y usados, pudiéndose comparar en popularidad con MySQL en el caso de las bases de datos relacionales.

El desarrollo de MongoDB comenzó en el año 2007 por la empresa 10gen3, publicando la primera versión comercial en 2009.

2.2 Instalación de MongoDB

En esta guía se tratará la instalación de MongoDB usando contenedores Docker. Concretamente, se usará la herramienta docker-compose que permitirá gestionar rápidamente la instalación de MongoDB (servidor) y MongoDB-PHP-GUI (cliente web). El fichero docker-compose incluido a continuación incluye la configuración necesaria para instalar ambos servicios en una distribución Ubuntu Linux.

```
version: '3'

services:
  mongo-db:
    image: mongo:latest
    ports:
      - 27017:27017
    volumes:
      - mongo_data:/data/db
  phpmongo-db:
    image: samuelstallet/mongodb-php-gui
    ports:
      - 5000:5000
    depends_on:
      - mongo-db
volumes:
  mongo_data:
```

Adicionalmente, se puede usar (si se prefiere) el cliente por defecto de mongodb (*mongo*), para ello, hay instalar el paquete *mongodb-clients* tal y como se muestra a continuación:

```
sudo apt install mongodb-clients
```

Para la instalación del cliente y servidor MongoDB incluido en el fichero *docker-compose.yml* simplemente hay que ejecutar el siguiente comando en el *path* donde se encuentre ubicado el fichero docker-compose:

```
sudo docker-compose up -d
```

Una vez ejecutado el comando es importante verificar que los dos servicios (contenedores) se han creado y están siendo ejecutados. Esto se puede comprobar mediante:

```
sudo docker ps
```

CONTAINER ID	IMAGE	CREATED	STATUS	PORTS
9f245aa95541	samueltalet/mongodb-php-gui	12s ago	Up 10s	5000->5000
15efc8a59a7d	mongo:latest	11s ago	Up 11s	27017->27017

Como se puede observar en la salida del comando *docker ps* el contenedor que contiene el servidor MongoDB (ID: 15efc8a59a7d) está escuchando en local (127.0.0.1) en el puerto 27017 mientras que el cliente (ID: 9f245aa95541) está escuchando en el puerto 5000 de la máquina local (127.0.0.1).

Es importante tener en cuenta que si se realiza la instalación en una máquina virtual, es necesario habilitar la redirección de puertos para poder acceder al cliente web tal y como se indica en la siguiente imagen.



La primera regla se encarga de redireccionar el puerto 22 de la máquina invitada al puerto 2222 de la máquina anfitrión con el fin permitir las conexiones por SSH al la máquina invitada. Por otro lado, la segunda regla es la encargada de permitir la conexión al cliente MongoDB-PHP-GUI que está alojado en el contenedor de la máquina remota desde la máquina local (mapeo 5000:5000).

2.3 Conceptos básicos entorno a MongoDB

Es importante conocer los conceptos básicos sobre los que se sustenta MongoDB así como su correspondencia con los SGBDR tradicionales.

2.3.1 JSON - JavaScript Object Notation

JSON es un formato compacto de representación de objetos. Las especificaciones las publicó Douglas Crockford en el documento RFC 46274. JSON es un formato independiente del lenguaje, aunque su uso extendido hasta hace poco era en el lenguaje Javascript. Actualmente se usa JSON en gran cantidades de sistemas para intercambiar información por su simplicidad en comparación con XML.

Este formato soporta gran cantidad de tipos de datos, lo que lo hace atractivo para un uso generalizado, y cada vez más lenguajes de programación dan soporte a este formato. El ejemplo del capítulo anterior, donde se mostraba un "documento", no es más que un fichero que sigue la estructura de JSON.

2.3.2 Documento

Un documento es un conjunto de datos estructurados (sin un esquema definido), que contiene pares clave/valor, y usa BSON (JSON Binario) como formato para almacenar los documentos. Un documento puede ser comparado con una fila o registro en una base de datos relacional.

2.3.3 Colección

Es un conjunto de documentos, similar a una tabla en las bases de datos relacionales.

2.5 La consola interactiva MongoDB

Una vez instalado MongoDB, podemos acceder a su consola interactiva y realizar nuestras primeras interacciones con MongoDB usando el comando *mongo*.

01	xdr@xdr:~\$ mongo
02	MongoDB shell version v3.6.8
03	connecting to: mongodb://127.0.0.1:27017
04	MongoDB server version: 5.0.6
05	>

Al iniciar la consola se conecta automáticamente a la base de datos "test", y a partir de ahí, podemos realizar consultas sobre esa base de datos. La consola es importante para administrar MongoDB.

Para acceder a la ayuda de MongoDB en la consola, se utiliza el comando *help* y nos muestra el contenido básico de la ayuda.

```

> help
    db.help()           help on db methods
    db.mycoll.help()    help on collection methods
    sh.help()           sharding helpers
    rs.help()           replica set helpers
    help admin          administrative help
    help connect        connecting to a db help
    help keys           key shortcuts
    help misc           misc things to know
    help mr             mapreduce

    show dbs            show database names
    show collections    show collections in current database
    show users          show users in current database
    show profile        show most recent system.profile entries with time >= 1ms
    show logs           show the accessible logger names
    show log [name]     prints out the last segment of log in memory, 'global' is
default
    use <db_name>       set current database
    db.foo.find()        list objects in collection foo
    db.foo.find( { a : 1 } ) list objects in foo where a == 1
    it                  result of the last line evaluated; use to further iterate
    DBQuery.shellBatchSize = x set default number of items to display on shell
    exit                quit the mongo shell
  
```

2.4.1 Conectando a una base de datos

La conexión a bases de datos desde la consola o a través de algún lenguaje es muy sencilla; aún así en MongoDB **no** se crean las bases de datos antes de usarlas. En bases de datos relacionales, se debe crear toda una estructura inicial para poder almacenar la información, sin embargo, en MongoDB no es necesario. Para crear una base de datos en MongoDB primero se debe seleccionar la bases de datos y luego almacenar un documento creando una colección de documentos.

2.4.2 Seleccionando la base de datos

Antes de seleccionar una base de datos, uno tiene la opción de ver el listado de bases de datos que existen en el sistema, con el comando *show dbs*.

```

01 > show dbs
02 admin    0.000GB
03 config  0.000GB
04 local    0.000GB
05 >
  
```

Tal y como se puede observar, la salida de este comando nos muestra el nombre y el tamaño de la base de datos. Es importante mencionar que MongoDB al crear el primer documento reserva espacio en disco, como mínimo 200 mb.

Una vez que se sabe la lista de bases de datos existente, se debe seleccionar una (no es obligatorio que esté en la lista), para seleccionar la base de datos se hace uso del comando *use <basededatos>*, permitiendo que se pueda trabajar con la base de datos existente u operar para crear una nueva.

```
01 > use ejemplo
02 switched to db ejemplo
03 >
```

2.4.3 Manejo de la base de datos

MongoDB permite trabajar con varios tipos de documentos, para el curso se trabajará con documentos que vienen a ser objetos de JS. Después de tener el cliente conectado al servidor creamos nuestro primer documento:

```
01 documento = {nombre: "Eduardo"}
```

Cada documento va a estar entre llaves, dentro del cual estará el atributo (en este caso nombre), seguido de dos puntos (:) y su valor (Eduardo) entre comillas.

```
01 > documento = {
02 ... nombre: "Eduardo"
03 ... }
04 { "nombre" : "Eduardo" }
05 >
```

Los documentos pueden tener atributos con distintos tipos de datos, cada atributo de un documento se separa con comas (,).

```
01 > documento = {
02 ... nombre: "Eduardo",
03 ... edad: 22,
04 ... altura: 1.85,
05 ... casado: true,
06 ... fechaAlta: new Date()
07 ... }
08 {
09     "nombre" : "Eduardo",
10     "edad" : 22,
11     "altura" : 1.85,
12     "casado" : true,
13     "fechaAlta" : ISODate("2022-04-22T12:21:10.128Z")
14 }
```

A los documentos también se los puede guardar en variables:

```
01 > var documento = {
02 ... nombre: "Eduardo",
03 ... edad: 22,
```



```

04 ... altura: 1.85,
05 ... casado: true,
06 ... fechaAlta: new Date()
07 ... }

```

Tal y como se puede ver, la principal diferencia es que si se almacena en una variable no se muestra el contenido del documento una vez creado.

Otra ventaja que tiene MongoDB es que permite el uso de acentos en los caracteres de texto.

```

01 > documento2 = {
02 ... nombre: "Ángel",
03 ... edad: 18,
04 ... altura: 1.62,
05 ... casado: false,
06 ... fechaAlta: new Date()
07 ... }
08 {
09     "nombre" : "Ángel",
10     "edad" : 18,
11     "altura" : 1.62,
12     "casado" : false,
13     "fechaAlta" : ISODate("2022-04-22T12:25:10.128Z")
14 }

```

MongoDB permite al usuario conocer en todo momento la BD en la que está trabajando mediante el comando *db*.

```

01 > use ejemplo
02 switched to db ejemplo
03
04 > db
05 ejemplo

```

La imagen superior muestra un fragmento de código compuesto por dos comandos. El primero se encarga de seleccionar la BD sobre la que se quiere trabajar. Si no está creada, el comando *use* no crea la base de datos, simplemente indica cuál será el nombre de la base de datos que se va a usar.

Inserciones en MongoDB

Para crear la base de datos es necesario insertar al menos un documento en una colección. Para ello, crearemos un nuevo documento (usuario) que almacenará la información de una persona y que a su vez, se guardará en la colección (usuarios). A continuación se muestra un fragmento de código que ejemplifica dicho procedimiento:

```

01 usuario1 = {

```

```

02 ... nombre: "Eduardo",
03 ... edad: 22,
04 ... altura: 1.85,
05 ... casado: true,
06 ... fechaAlta: new Date()
07 ... }
08 {
09     "nombre" : "Eduardo",
10     "edad" : 22,
11     "altura" : 1.85,
12     "casado" : true,
13     "fechaAlta" : ISODate("2022-04-22T12:25:10.128Z")
14 }
15 > db.usuarios.insert(usuario1)
16 WriteResult({ "nInserted" : 1 })

```

Donde:

db = base de datos en la que estamos.
 usuarios = nombre de la colección que guardará los documentos
 insert = operación de inserción
 usuario = documento que se va a insertar

Hay que tener en cuenta que si la colección no ha sido creada anteriormente (como en el caso que nos ocupa), MongoDB la crea automáticamente en el momento de ejecutar el comando. En todo caso se puede verificar las colecciones creadas en una BD usando el comando *show collections*.

```

01 > show collections
02 usuarios

```

Búsquedas en MongoDB

Por otro lado, para consultar los registros (documentos) guardados en la BD, MongoDB dispone del comando *.find()*.

```

01 > db.usuarios.find()
02 { "_id" : ObjectId("6262a0ccb2a119d14cdadcfc"), "nombre" :
03 "Eduardo", "edad" : 22, "altura" : 1.85, "casado" : true,
04 "fechaAlta" : ISODate("2022-04-22T12:21:10.128Z") }

```

Tal y como se puede deducir del fragmento de código anterior, el comando *.find()* tiene un comportamiento similar al *SELECT ** de las BD relacionales. Por otro lado, es importante tener en cuenta que si no se asigna manualmente, MongoDB se encarga de generar automáticamente un identificador (o clave primaria) que permita identificar unívocamente cada uno de los documentos.

```

01 usuario2 = {
02 ... nombre: "María",

```

```

03 ... apellido1: "Pérez",
04 ... apellido2: "Rodríguez",
05 ... edad: 32,
06 ... altura: 1.74,
07 ... fechaAlta: new Date() }
08 {
09     "nombre" : "María",
10     "apellido1" : "Pérez",
11     "apellido2" : "Rodríguez",
12     "edad" : 32,
13     "altura" : 1.74,
14     "fechaAlta" : ISODate("2022-04-22T13:11:45.128Z")
15 }
16 > db.usuarios.insert(usuario2)
17 WriteResult({ "nInserted" : 1 })

```

Tal y como se comentó anteriormente, el comando `.find()` permite visualizar si el nuevo usuario se ha almacenado correctamente en la base de datos.

```

01 > db.usuarios.find()
02 { "_id" : ObjectId("6262a0ccb2a119d14cdadcfc"), "nombre" :
03 "Eduardo", "edad" : 22, "altura" : 1.85, "casado" : true,
04 "fechaAlta" : ISODate("2022-04-22T12:21:10.128Z") }
05 { "_id" : ObjectId("4545a4fea3b331d41dfddea"), "nombre" : "María",
06 "apellido1" : "Pérez", "apellido2" : "Rodriguez", "edad" : 32,
07 "altura" : 1.74, "fechaAlta" : ISODate("2022-04-
08 22T13:11:45.128Z") }

```

En la imagen anterior podemos apreciar la gran ventaja que ofrece las bases de datos NoSQL, ya que el primer documento insertado tiene cinco atributos (*nombre*, *edad*, *altura*, *casado*, *fechaAlta*) mientras que el segundo documento tiene seis atributo (*nombre*, *apellido1*, *apellido2*, *edad*, *altura* y *fechaAlta*).

A parte del método `find`, explicado anteriormente, existe el método `findOne` que permite encontrar el primer registro o documento dentro de la colección:

```

01 > db.usuarios.findOne()
02 {
03     "_id" : ObjectId("6262a0ccb2a119d14cdadcfc"),
04     "nombre" : "Eduardo",
05     "edad" : 22,
06     "altura" : 1.85,
07     "casado" : true,
08     "fechaAlta" : ISODate("2022-04-22T12:21:10.128Z")
09 }

```

Adicionalmente, MongoDB permite almacenar el resultado en una variable si se desea trabajar con el después.

```

01 > var test = db.usuarios.findOne()
02 > test
03 {
04     "_id" : ObjectId("6262a0ccb2a119d14cdadcfc"),
05     "nombre" : "Eduardo",
06     "edad" : 22,
07     "altura" : 1.85,
08     "casado" : true,
09     "fechaAlta" : ISODate("2022-04-22T12:21:10.128Z")
10 }

```

Para restringir las búsquedas (haciendo analogías con el WHERE en BD relacionales) dentro de los paréntesis del *.find()* abrimos llaves y e indicamos el parámetro de búsqueda tal y como se ejemplifica en el fragmento de código indicado a continuación:

```

01 > var test_dos =db.usuarios.find( { nombre : "María" } )
02 { "_id" : ObjectId("62668373b2a119d14cdadcfd"), "nombre" : "María",
03   "apellido1": "Pérez", "apellido2": "Rodriguez", "edad" : 32,
04   "altura" : 1.74, "fechaAlta" : ISODate("2022-04-
05   22T13:11:45.128Z") }

```

En este caso particular el método *.find()* solo devuelve un documento ya que no hay más que cumplan la condición de búsqueda. Sin embargo, si hubiese más documentos que cumplen el criterio de búsqueda el comando *.find()* los devolvería todos. A continuación se muestra un ejemplo de un comando *.find()* con múltiples registros.

```

01 usuario3 = {
02     ... nombre: "Concha",
03     ... apellido1: "López",
04     ... apellido2: "Arias",
05     ... edad: 22,
06     ... altura: 1.66 }
07 {
08     "nombre" : "Concha",
09     "apellido1" : "López",
10     "apellido2" : "Arias",
11     "edad" : 22,
12     "altura" : 1.66
13 }
14
15 > db.usuarios.insert(usuario3)
16 WriteResult({ "nInserted" : 1 })
17 > db.usuarios.find( { edad: 22 })
18
19 { "_id" : ObjectId("6262a0ccb2a119d14cdadcfc"), "nombre" :
20   "Eduardo", "edad" : 22, "altura" : 1.85, "casado" : true,
21   "fechaAlta" : ISODate("2022-04-22T12:21:10.128Z") }
22 { "_id" : ObjectId("626683c8b2a119d14cdadcfe"), "nombre" :

```

23	"Concha", "apellido1" : "López", "apellido2" : "Arias", "edad" :
24	22, "altura" : 1.66 }

Tal y como se puede observar en la línea 17 del fragmento de código anterior, se ha usado el código *.find()* indicando entre paréntesis el valor del atributo a buscar (edad : 22). En las líneas 19-24 se muestra como el método ha obtenido dos resultados que satisfacen los criterios de búsqueda. Es decir, tanto Eduardo como Concha tienen 22 años de edad.

Siguiendo con el método *.find()*, cabe destacar que permite obtener documentos que cumplan múltiples condiciones de búsqueda. Para ilustrar este caso, el fragmento siguiente incluirá el código que permitirá buscar los documentos cuyos usuarios tengan 22 años y una altura de 1.85.

01	> db.usuarios.find({ edad: 22, altura: 1.85 })
02	
03	{ "_id" : ObjectId("6262a0ccb2a119d14cdadcfc"), "nombre" :
04	"Eduardo", "edad" : 22, "altura" : 1.85, "casado" : true,
05	"fechaAlta" : ISODate("2022-04-22T12:21:10.128Z") }

Tal y como se puede observar en la línea 01, en MongoDB la coma representa el AND en BD relacionales en la realización de búsquedas.

Por otro lado, MongoDB permite buscar resultados cuando la consulta es falsa. Para ello se usa la palabra reservada *\$ne* (No Equals) con el valor entre llaves. A continuación se muestra un ejemplo donde se pretenden obtener los documentos de los usuarios que tengan una edad diferente a 22.

01	> db.usuarios.find({ edad: { \$ne: 22 } })
02	
03	{ "_id" : ObjectId("62668373b2a119d14cdadcfd"), "nombre" : "María",
04	"apellido1": "Pérez", "apellido2": "Rodriguez", "edad" : 32,
05	"altura" : 1.74, "fechaAlta" : ISODate("2022-04-
06	22T13:11:45.128Z") }

Además, MongoDB permite realizar búsquedas mixtas en las que se combinan indistintamente cláusulas afirmativas con otras negadas. Concretamente, se pueden obtener los documentos que contengan información de las personas que no tienen 32 años y estén casados (casado : true).

01	> db.usuarios.find({ edad: { \$ne: 32 }, casado : true })
02	
03	{ "_id" : ObjectId("6262a0ccb2a119d14cdadcfc"), "nombre" :
04	"Eduardo", "edad" : 22, "altura" : 1.85, "casado" : true,
05	"fechaAlta" : ISODate("2022-04-22T12:21:10.128Z") }

En este caso concreto, hay dos documentos que satisfacen la condición de la edad (el de Eduardo y el de Concha). Sin embargo, Concha no tiene definido el campo casado por lo que el documento con el id=6262a0ccb2a119d14cdadcfc es el único que satisface las dos condiciones.

Finalmente, cabe destacar que los métodos de búsqueda incorporan el método `.pretty()` que se encarga de mostrar los resultados de una manera formateada con el fin de facilitar su visualización. A continuación se muestra la salida del ejemplo anterior usando el método `.pretty()`

```

01 > db.usuarios.find( { edad: {$ne: 32}, casado : true }).pretty()
02
03 {
04     "_id" : ObjectId("6262a0ccb2a119d14cdadcfc"),
05     "nombre" : "Eduardo",
06     "edad" : 22,
07     "altura" : 1.85,
08     "casado" : true,
09     "fechaAlta" : ISODate("2022-04-22T12:21:10.128Z")
10 }
```

Como se puede observar, los resultados de la búsqueda se formatean mostrando cada par campo-valor en una línea.

Sin embargo, si se quieren insertar documentos que tengan el mismo identificador el método `.insert()` muestra un error de clave duplicada y no inserta el documento en la base de datos. Para evitar este error, MongoDB dispone del comando `.save()` que se encarga de actualizar los documentos que tengan el mismo identificador que los que se quieren insertar.

```

01 > db.usuarios.save( { "_id" : ObjectId("6262a0ccb2a119d14cdadcfc"),
02     "nombre" : "Eduardo", "edad" : 22, "altura" : 1.90,
03     "casado" : true, "fechaAlta" : new Date() })
04 WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 0 })
05
06 > db.usuarios.find({nombre:"Eduardo"}).pretty()
07 {
08     "_id" : ObjectId("62668c7ebd92fbd18b5b6ca2"),
09     "nombre" : "Eduardo",
10     "edad" : 23,
11     "altura" : 1.85,
12     "casado" : true,
13     "fechaAlta" : ISODate("2022-04-25T12:01:35.020Z")
14 }
```

Hasta este momento se han abordado los comandos `.find()` y `.findOne()` para la realización de consultas simples sobre MongoDB. Sin embargo MongoDB permite realizar operaciones complejas mediante el uso de operadores de comparación, comprobaciones de existencia de campos o consultas sobre elementos complejos (como arrays y subdocumentos).

Operadores de comparación genéricos

En las bases de datos relacionales, es muy típico filtrar los resultados según el valor de un determinado campo. Por ejemplo si $X > 0$. Para ello, MongoDB dispone de los operadores

\$gt (>), \$gte (>=), \$lt (<), \$lte (<=), \$ne (<>), \$eq (=). Para ilustrar este caso, asumamos que queremos obtener de la BD los usuarios que tengan estrictamente más de 22 años.

```

01 > db.usuarios.find({age:{$gt:22}})
02
03 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"), "nombre" :
04 "Eduardo", "edad" : 23, "altura" : 1.85, "casado" : true,
05 "fechaAlta" : ISODate("2022-04-25T11:44:00.453Z") }
06 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca5"), "nombre" : "María",
07 "apellido1" : "Pérez", "apellido2" : "Rodríguez", "edad" : 32,
08 "altura" : 1.74, "fechaAlta" : ISODate("2022-04-
09 25T11:44:08.062Z") }

```

Tal y como se puede observar, la consulta devuelve todos los documentos que cumplen que el año de los usuarios sea mayor que 22. Sin embargo, si en vez de obtener todo el documento, sólo se quisiera obtener unos determinados campos se pueden indicar como segundo parámetro del método *.find()*. A continuación se muestra un ejemplo donde se especifican los atributos que se quieren obtener de cada uno de los documentos encontrados.

```

01 > db.usuarios.find({edad:{$gt:22}}, {nombre:1, edad:1})
02
03 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"), "nombre" :
04 "Eduardo", "edad" : 23 }
05 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca5"), "nombre" : "María",
06 "edad" : 32

```

El mismo proceso habría que hacer si se quiere obtener los usuarios cuya edad sea menor o igual que 22.

```

01 > db.usuarios.find({edad:{$lte:22}}, {nombre:1, edad:1}).pretty()
02 {
03     "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"),
04     "nombre" : "Concha",
05     "edad" : 22
06 }

```

En este caso, se ha añadido el comando *.pretty()* para demostrar que es compatible con cualquier tipo de búsqueda (independientemente de su complejidad).

Otro operador de comparación es el que permite comparar desigualdades. Para ello MongoDB incluye el operador *\$ne*, que devuelve verdadero cuando el valor de la consulta es falso. En este escenario, si se quisieran obtener los usuarios cuya edad NO sea 32 se haría de la siguiente manera:

```

01 > db.usuarios.find({edad:{$ne:32}}, {edad:1, altura:1}).pretty()
02 {
03     "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"),
04     "edad" : 23,

```

05	"altura" : 1.85
06	}
07	{
08	"_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"),
09	"edad" : 22,
10	"altura" : 1.66
11	}

Finalmente el último operador de comparación incluido en MongoDB es IN que permite encontrar valores que estén dentro de un determinado conjunto de valores. Mientras que en SQL sería lo mismo que usar la combinación WHERE IN en MongoDB se utilizará \$in junto con un array de posibles valores. Un claro ejemplo de uso de este operador sería el de obtener los usuarios que tengan 22 o 23 años de edad.

01	> db.usuarios.find({edad:{\$in:[22,23]}},{nombre:1,edad:1}).pretty()
02	{
03	"_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"),
04	"nombre" : "Eduardo",
05	"edad" : 23
06	}
07	{
08	"_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"),
09	"nombre" : "Concha",
10	"edad" : 22
11	}

Tal y como se puede observar, la consulta devuelve todas las personas cuyas edades son igual a 22 y 23 años.

Operadores de comparación con strings

Los operadores descritos anteriormente son aplicables a los strings. Pero hay que tener en cuenta que MongoDB distingue entre mayúsculas y minúsculas y que utiliza el orden lexicográfico. Esto quiere decir que MongoDB ordena los strings de la misma manera que un diccionario, aunque diferenciando mayúsculas y minúsculas. Este es un ejemplo de orden de strings que hace MongoDB.

01	{ "_id" : "AAab" }
02	{ "_id" : "Abb" }
03	{ "_id" : "Abc" }
04	{ "_id" : "BCb" }
05	{ "_id" : "Bbaab" }
06	{ "_id" : "abb" }
07	{ "_id" : "abc" }
08	{ "_id" : "bcb" }

En orden ascendente las mayúsculas van primero y luego se tiene en cuenta el orden lexicográfico de cada letra. Como se puede observar, el número de caracteres no se tiene en cuenta.

Tal y como se ha comentado anteriormente, MongoDB es una base de datos sin esquema, lo que quiere decir que los documentos, aun siendo de la misma colección, pueden tener distintos campos. Incluso estos campos pueden ser de distintos tipos en cada documento. Por tanto, en ocasiones puede ser útil realizar una consulta que devuelva los documentos en los que exista un determinado campo. La siguiente consulta se obtienen aquellos documentos que contengan el campo casado.

01	db.usuarios.find({casado:{\$exists:true}},
02	{nombre:1,edad:1,casado:1})
03	{ "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"), "nombre" :
04	"Eduardo", "edad" : 23, "casado" : true }

En el caso de que se quisiera buscar los documentos que no tienen un determinado campo, bastará con cambiar el true por un false en el operador \$exists.

Es importante tener en cuenta que MongoDB puede guardar documentos con distintos tipos en el mismo campo. Por ejemplo, aunque “edad” es un número para todos los documentos, podríamos insertar un documento nuevo con un string en ese campo. Por tanto, también podríamos necesitar filtrar por documentos en los cuales un campo será de un determinado tipo. Esto se hace con el operador \$type.

01	db.usuarios.find({edad:{\$type:2}},{nombre:1,edad:1,casado:1})
----	--

En este caso se buscan los documentos cuyo campo edad sea de tipo 2, que es el tipo string. Se puede encontrar el identificador asignado a cada tipo en la ayuda del operador \$type en la página de MongoDB¹.

Operadores lógicos

En bases de datos relacionales es muy típico añadir operadores OR a la cláusula WHERE. Por ejemplo WHERE gender = “female” OR age > 20. Hasta ahora las consultas que hemos visto buscaban por uno o más campos, pero lo hacían con la lógica de un AND, es decir, que todas las condiciones debían cumplirse. Si queremos añadir cláusulas OR usaremos el operador \$or.

01	> db.usuarios.find({\$or:[{edad:{\$eq:22}}, {altura:
02	{\$gt:1.80}}] } ,{nombre:1,edad:1,altura:1})
03	
04	{ "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"), "nombre" :
05	"Eduardo", "edad" : 23, "altura" : 1.85 }
06	{ "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" :
07	"Concha", "edad" : 22, "altura" : 1.66 }

En este caso buscamos los documentos cuyo campo edad sea igual a 20 o cuyo campo “altura” sea mayor que 1.80. Como vemos basta con especificar un array de condiciones para que el operador \$or realice la consulta.

Lo curioso es que también existe un operador \$and. ¿Por qué es curioso? Pensemos en las siguientes consultas.

¹ MongoDB - Tipos: http://docs.mongodb.org/manual/reference/operator/type/#op._S_type

```

01 > db.usuarios.find({edad:{$gt:20},altura:{$gt:1.60}},{nombre:1})
02
03 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"), "nombre" :
04 "Eduardo", "edad" : 23, "altura" : 1.85 }
05 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" :
06 "Concha", "edad" : 22, "altura" : 1.66 }
07
08 > db.usuarios.find($and:[{edad:{$gt:20},altura:{$gt:1.60}},
09 {nombre:1}])
10
11 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"), "nombre" :
12 "Eduardo", "edad" : 23, "altura" : 1.85 }
13 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" :
14 "Concha", "edad" : 22, "altura" : 1.66 }

```

Tal y como se puede observar, ambas consultas son iguales y por tanto MongoDB devuelve los mismos resultados. En la primera, MongoDB hace un and implícito de los parámetros de la consulta, mientras que en la segunda se ha incluido el and explícitamente. Usar el operador explícitamente es útil cuando hay que hacer una consulta que incluya dos veces el mismo campo. Si no utilizamos el and y lo hacemos de esta manera, podemos obtener resultados erróneos. Por ejemplo, las siguientes consultas son iguales, aunque invirtiendo el orden de las condiciones.

```

01 > db.usuarios.find({edad:{$gt:20},edad:{$lt:23}},{nombre:1,
02 edad:1})
03
04 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" :
05 "Concha", "edad" : 22}
06
07 > db.usuarios.find({edad:{$lt:23},edad:{$gt:20}},{nombre:1})
08
09 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"), "nombre" :
10 "Eduardo", "edad" : 23}
11 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" :
12 "Concha", "edad" : 22}

```

Tal y como se puede observar ambas consultas devuelven resultados distintos. Esto es porque MongoDB coge el último valor para realizar la consulta. Por tanto, la consulta correcta sería:

```

01 > db.usuarios.find({$and:[{edad:{$gt:20},{edad:{$lt:23}}}]}],
02 {nombre:1, edad:1})
03
04 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"), "nombre" :
05 "Eduardo", "edad" : 23}
06 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" :
07 "Concha", "edad" : 22}

```

Adicionalmente, se pueden combinar los operadores \$and y \$or indistintamente para conseguir consultas más complejas como la que se muestra a continuación.

```

01 > db.usuarios.find(
02   { $or:
03     [ { nombre: "Eduardo" },
04       { $and: [
05         { edad: { $gt: 30 } },
06         { altura: { $gt: 1.70 } }
07       ] }
08     ]
09   }, { nombre: 1, edad: 1 })
11
12 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"), "nombre" :
13   "Eduardo", "edad" : 23, "altura" : 1.85 }
14 { "_id" : ObjectId("4545a4fea3b331d41dfddeea"), "nombre" : "María",
15   "apellido1": "Pérez", "apellido2": "Rodriguez", "edad" : 32,
16   "altura" : 1.74, "fechaAlta" : ISODate("2022-04-
17   22T13:11:45.128Z") }
```

Tal y como se puede observar, la consulta pretende obtener todos los usuarios que o bien se llamen Eduardo o bien tengan una edad superior a 30 años y una altura superior a 1.70cm.

Además de \$and y \$or, tenemos otros dos operadores lógicos que son \$not y \$nor. El primero (\$not) es bastante sencillo de entender ya que lo que hace es buscar los documentos que no cumplan una determinada condición.

```

01 > db.usuarios.find({ edad: { $not: { $lte: 30 } } })
02
03 { "_id" : ObjectId("4545a4fea3b331d41dfddeea"), "nombre" : "María",
04   "apellido1": "Pérez", "apellido2": "Rodriguez", "edad" : 32,
05   "altura" : 1.74, "fechaAlta" : ISODate("2022-04-
06   22T13:11:45.128Z") }
```

Lo importante en este caso, es saber que \$not solo puede usarse con otros operadores como \$gt o \$lt. No puede usarse con valores directos o documentos. Para eso ya existe el operador \$ne, que se ha explicado antes. También hay que tener en cuenta que la búsqueda pretende buscar usuarios con edades que no sean menores o iguales que 30. Es importante tener en cuenta que si los campos buscados NO existen también serán incluidos en los resultados.

Por otro lado, el comando \$nor acepta dos o más valores. Por ejemplo en la siguiente consulta se buscan los usuarios cuya edad NO sea mayor que 30 y cuyo campo "altura" NO sea menor que 1.65.

```

01 > db.usuarios.find({ $nor:
02   [ { edad: { $gt: 30 } }, { edad: { $exists: false } },
```

```

03      {altura:{$lt:1.65}}, {altura:{$exists:false}}
04    ]
05  }, {nombre:1, edad:1, altura:1})
06
07  { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"), "nombre" :
08    "Eduardo", "edad" : 23, "altura" : 1.85 }
09  { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" :
10    "Concha", "edad" : 22, "altura" : 1.66 }

```

Destacar que al igual que \$not, \$nor devuelve también los documentos si los campos no existen. Para evitar esto, si es un comportamiento no deseable, se puede añadir el operador \$exists tal y como se muestra en el ejemplo anterior.

Operadores de consultas sobre arrays

Como se ha comentado anteriormente, MongoDB permite almacenar un array cualquier tipo de elementos (es decir strings, números, otros arrays o subdocumentos). Para realizar los ejemplos de consultas sobre arrays se van a insertar nuevos datos en la colección de usuarios tal y como se muestra a continuación.

```

01  > usuario4 = {
02    ... .. nombre: "Juan",
03    ... .. apellido1: "Pérez",
04    ... .. apellido2: "Pérez",
05    ... .. edad: 55,
06    ... .. altura: 1.74,
07    ... .. email: ["jpp@empresa.com", "jpp@gmail.com"],
08    ... .. rrss: [{
09      ... .. nombre: "instagram",
10      ... .. usuario: "@juan_pp",
11      ... .. seguidores: 8250 }, {
12      ... .. nombre: "tiktok",
13      ... .. usuario: "@jpp_tiktok",
14      ... .. seguidores: 25}] }
15  {
16    "nombre" : "Juan",
17    "apellido1" : "Pérez",
18    "apellido2" : "Pérez",
19    "edad" : 55,
20    "altura" : 1.74,
21    "email" : [
22      "jpp@empresa.com",
23      "jpp@gmail.com"
24    ],
25    "rrss" : [
26      {
27        "nombre" : "instagram",
28        "usuario" : "@juan_pp",
29        "seguidores" : 8250

```

```
30         },
31         {
32             "nombre" : "tiktok",
33             "usuario" : "@jpp_tiktok",
34             "seguidores" : 25
35         }
36     ]
37 }
38
39 > usuario5 = {
40 ... .. nombre: "Ana",
41 ... .. apellido1: "Vallén",
42 ... .. apellido2: "López",
43 ... .. edad: 29,
44 ... .. altura: 1.69,
45 ... .. email: ["avl@empresa.com", "avl@gmail.com"],
46 ... .. rrss: [{
47 ... .. .. nombre: "instagram",
48 ... .. .. usuario: "@anita_vallen",
49 ... .. .. seguidores: 5000},{
50 ... .. .. nombre: "tiktok",
51 ... .. .. usuario: "@avallen",
52 ... .. .. seguidores: 1200},{
53 ... .. .. nombre: "facebbok",
54 ... .. .. usuario: "ana.vallen.lopez",
55 ... .. .. seguidores: 500} ]}
56 {
57     "nombre" : "Ana",
58     "apellido1" : "Vallén",
59     "apellido2" : "López",
60     "edad" : 29,
61     "altura" : 1.69,
62     "email" : [
63         "avl@empresa.com",
64         "avl@gmail.com"
65     ],
66     "rrss" : [
67         {
68             "nombre" : "instagram",
69             "usuario" : "@anita_vallen",
70             "seguidores" : 5000
71         },
72         {
73             "nombre" : "tiktok",
74             "usuario" : "@avallen",
75             "seguidores" : 1200,
76         },
```

```

77         {
78             "nombre" : "facebook",
79             "usuario" : "ana.vallen.lopez",
80             "seguidores" : 500
81         }
82     ]
83 }
84
85 db.usuarios.insert([usuario4, usuario5])
86 BulkWriteResult({
87     "writeErrors" : [ ],
88     "writeConcernErrors" : [ ],
89     "nInserted" : 2,
90     "nUpserted" : 0,
91     "nMatched" : 0,
92     "nModified" : 0,
93     "nRemoved" : 0,
94     "upserted" : [ ]
95 })

```

Como se puede observar, los últimos documentos insertados en la colección usuarios tiene asociado un campo email, que es un array de strings y un campo rss que es un array de documentos.

En el caso de que por ejemplo, se quisiese buscar un solo elemento dentro del una email bastaría con hacer una consulta similar a la siguiente:

```

01 > db.usuarios.find({email:"jpp@gmail.com"}, {nombre:1,
02 email:1}).pretty()
03
04 {
05     "_id" : ObjectId("62679dc3bd92fbd18b5b6ca7"),
06     "nombre" : "Juan",
07     "email" : [
08         "jpp@empresa.com",
09         "jpp@gmail.com"
10     ]
11 }

```

En este caso MongoDB buscará el elemento "jpp@gmail.com" dentro del array email, devolviendo las personas en cuyo array existe dicho elemento. En este caso la consulta no ha sido diferente de las que se han realizado anteriormente ya que solo se está buscando un solo elemento. En cambio si queremos encontrar todas las personas que contengan varios valores, la consulta sería algo similar a:

```

01 > db.usuarios.find({email:{$all:["jpp@gmail.com",
02 "jpp@empresa.com"]}}, {nombre:1, email:1}).pretty()
03

```

```

04 {
05     "_id" : ObjectId("62679dc3bd92fbd18b5b6ca7"),
06     "nombre" : "Juan",
07     "email" : [
08         "jpp@empresa.com",
09         "jpp@gmail.com"
10     ]
11 }

```

Usando el operador \$all se buscan varios elementos dentro de un array, especificando como entrada un array de elementos a buscar. En el fragmento de código anterior se buscan todas las personas que contengan "jpp@gmail.com" y "jpp@empresa.com" en el campo email. Sólo se devolverán los documentos que contengan ambos valores. En este caso se han especificado dos valores, pero se pueden añadir todos los que se necesiten y solo se devolverán los documentos que los incluyan.

De manera análoga se permite hacer una búsqueda en un array para devolver los documentos que contengan al menos uno de los elementos a buscar.

```

01 > db.usuarios.find({email:{$in:["avl@empresa.com",
02 "jpp@empresa.com"]}}, {nombre:1, email:1}).pretty()
03
04 {
05     "_id" : ObjectId("62679dc3bd92fbd18b5b6ca7"),
06     "nombre" : "Juan",
07     "email" : [
08         "jpp@empresa.com",
09         "jpp@gmail.com"
10     ]
11 }
12 {
13     "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"),
14     "nombre" : "Ana",
15     "email" : [
16         "avl@empresa.com",
17         "avl@gmail.com"
18     ]
19 }

```

En este caso se han utilizado tres valores de búsqueda en conjunción con el operador \$in, que busca todos los documentos que tengan en el campo email al menos uno de los dos elementos. En cuanto se detecte que el documento tiene uno de los valores se devuelve como resultado.

En el caso que se quiera hacer lo mismo pero buscando los documentos que no contengan los elementos especificados en el array de entrada, utilizamos una consulta con el operador \$nin:

```

01 > db.usuarios.find({email:{$nin:["avl@empresa.com",

```

```

02 "jpp@empresa.com"]}}}, {nombre:1, email:1}).pretty()
03
04 {
05     "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"),
06     "nombre" : "Eduardo"
07 }
08 {
09     "_id" : ObjectId("6266bac3bd92fbd18b5b6ca5"),
10     "nombre" : "María"
11 }
12 {
13     "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"),
14     "nombre" : "Concha"
15 }

```

En este caso, la consulta devuelve los documentos que no tienen asignado el campo email ya que los que lo tienen asignado cumplen con los valores especificados en la búsqueda.

Otro operador que puede ser muy útil es \$size. Se utiliza para buscar los documentos que tienen un campo array de un tamaño predeterminado. Es muy sencillo de utilizar. Por ejemplo para devolver todos los documentos cuyo array de tags tiene un tamaño se ejecutaría la consulta:

```

01 > db.usuarios.find({email:{$size:3}}, {nombre:1, email:1}).pretty()
02

```

En este caso la consulta no devuelve ningún valor ya que no existe ningún array email que tenga tres campos.

En cuanto a las proyecciones con arrays, MongoDB dispone de operadores muy útiles. Si se quiere mostrar solo el elemento que está en una determinada posición dentro de un array se utiliza la nomenclatura "<elemento>.\$":<pos>". A continuación se muestra un ejemplo de uso donde se proyecta el email que está en la primera posición del array.

```

01 > db.usuarios.find({email:{$in:["avl@empresa.com",
02 "jpp@empresa.com"]}}, {"email.$":1,nombre:1})
03
04 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca7"), "nombre" : "Juan",
05   "email" : [ "jpp@empresa.com" ] }
06 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana",
07   "email" : [ "avl@empresa.com" ] }

```

La parte que filtra los datos ya se ha utilizado en ejemplo anteriores, sin embargo, se ha modificado un poco la proyección. Es importante recordar que una proyección se utiliza para mostrar los campos concretos que se espera devolver (como los campos de una sentencia SELECT de SQL). En este caso utilizamos entre comillas el operador \$. Concretamente, tal y como se puede observar en el fragmento anterior el elemento "email.\$":1 indica a MongoDB que se quiere devolver el primer elemento del array email.

Otro operador interesante para proyecciones es `$slice`. Este operador permite devolver un número determinado de elementos de un array.

```
01 > db.usuarios.find({email:{$in:["avl@empresa.com",
02 "jpp@empresa.com"]}}, {email:{$slice:2},nombre:1})
03
04 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca7"), "nombre" : "Juan",
05 "email" : [ "jpp@empresa.com", "jpp@gmail.com" ] }
06 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana",
07 "email" : [ "avl@empresa.com", "avl@gmail.com" ] }
```

La consulta anterior permite devolver los documentos filtrados, pero solo se devolverán dos primeros elementos del array email y el nombre de la persona. Si se quisiera devolver elementos por el final bastaría con usar un número negativo.

```
01 > db.usuarios.find({email:{$in:["avl@empresa.com",
02 "jpp@empresa.com"]}}, {email:{$slice:-1},nombre:1})
03
04 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca7"), "nombre" : "Juan",
05 "email" : [ "jpp@gmail.com" ] }
06 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana",
07 "email" : [ "avl@gmail.com" ] }
```

Adicionalmente, `$slice` permite como parámetro un array del tipo `[skip, limit]`. El primer campo del array (*skip*) se usa para definir el número de elementos que se desean ignorar mientras que el segundo (*limit*) sirve para especificar los elementos que se quieren obtener.

```
01 > db.usuarios.find({email:{$in:["avl@empresa.com",
02 "jpp@empresa.com"]}}, {email:{$slice:[1,3]},nombre:1})
03
04 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca7"), "nombre" : "Juan",
05 "email" : [ "jpp@gmail.com" ] }
06 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana",
07 "email" : [ "avl@gmail.com" ] }
```

En este caso, se omite la primera entrada del array y se mostrarían las siguientes 3 entradas. Sin embargo, como los arrays de email solo tienen como máximo elementos, la consulta en MongoDB devuelve únicamente el segundo elemento del array.

Siguiendo el mismo criterio que anteriormente, en el caso de querer empezar a buscar por el final del array el parámetro *skip* pueden tener que ser un número negativo.

```
01 > db.usuarios.find({email:{$in:["avl@empresa.com",
02 "jpp@empresa.com"]}}, {email:{$slice: [-1,1]}})
```

Otra de las posibilidades que ofrece MongoDB es el uso de *Dot Notation*. Dot Notation permite realizar consultas en arrays y en subdocumentos. Se basa en añadir un punto después del identificador del array o subdocumento para realizar consultas sobre un índice

en concreto del array o sobre un campo concreto del subdocumento. A continuación se muestra un ejemplo de una consulta *Dot Notation* con arrays:

```
01 > db.usuarios.find({"email.0":"avl@empresa.com"}, {nombre:1})
02 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana" }
03
04 > db.usuarios.find({"email.1":"jpp@empresa.com"}, {nombre:1})
```

En el fragmento de código anterior se realizan dos consultas (líneas 01 y 04). La primera consulta busca todos los documentos que tengan en la primera posición del array email el valor "avl@empresa.com". Por otro lado, en la segunda consulta se buscan aquellos documentos que tengan en la segunda posición del array email el valor "jpp@empresa.com". Es importante tener en cuenta que en MongoDB el índice de los arrays empieza en 0.

Tal y como se comentó previamente, Dot Notation también puede ser utilizado para realizar consultas sobre subdocumentos. En nuestros datos de ejemplo existe un campo llamado rss, que contiene un array de subdocumentos. Si en dicho campo quisiéramos buscar los elementos que contienen el subdocumento compuesto por el nombre "tiktok" y el usuario "@avallen" se utiliza una consulta como esta:

```
01 > db.usuarios.find({rss:{nombre:"tiktok", usuario:"@avallen"}},
02 {nombre:1})
03
04 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana" }
```

En este caso solo se devuelven los documentos que en el array del campo rss tienen el subdocumento {nombre:"tiktok", usuario:"@avallen"}. Para buscar por un campo del subdocumento en concreto deberemos usar Dot Notation tal y como se muestra a continuación.

```
01 > db.usuarios.find({"rss.nombre":"instagram"}, {nombre:1})
02
03 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca7"), "nombre" : "Juan" }
04 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana" }
```

Se puede ver se ha especificado entre comillas el campo "rss.nombre", lo que quiere decir que se buscará por el campo *name* en el subdocumento *rss*. En este caso se devuelven todos los documentos que cumplen con "rss.nombre":"instagram" independientemente del valor del resto de campos del subdocumento.

Es importante tener en cuenta que **Dot Notation** permite hacer las consultas todo lo complejas que sea necesario.

```
01 > db.usuarios.find({"rss.1.nombre":{"$gte":"T"}}, {rss:{$slice:-
02 1},nombre:1})
03
04 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana",
05 "rss" : [ { "nombre" : "facebook", "usuario" :
06 "ana.vallen.lopez" } ] }
```

Esta consulta permite buscar en el array *rrss* los elementos que estén en la posición 1 y cuyo nombre sea mayor o igual que "i". Además en la proyección se indica el último elemento, que es por el que se está filtrando.

En el ejemplo anterior se ha visto que los operadores \$gt, \$gte, \$lt, ..., se pueden utilizar con *strings*. Sin embargo la función de esos operadores es limitada ya que se basa en realizar las comparaciones algebraicas básicas. Con el fin de exprimir toda la potencia de las búsquedas con strings, MongoDB permite buscar patrones en el texto de los campos mediante el uso de expresiones regulares. Como cada lenguaje de programación utiliza las expresiones regulares de manera diferente, es importante especificar que MongoDB utiliza Perl Compatible Regular Expressions (PCRE)². A continuación se incluye un ejemplo de consulta usando expresiones regulares.

```
01 > db.usuarios.find({nombre:{$regex:".a$"}},{nombre:1})
02
03 {"_id" : ObjectId("6266bac3bd92fbd18b5b6ca5"), "nombre" : "María"}
04 {"_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" : "Concha"}
05 {"_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana"}
```

En este ejemplo se buscan todos los documentos cuyo nombre termine con la letra a minúscula.

```
01 > db.usuarios.find({nombre:{$regex:".a.a$"}},{nombre:1})
02
03 {"_id" : ObjectId("6266bac3bd92fbd18b5b6ca5"), "nombre" : "María"}
04
05 > db.usuarios.find({nombre:{$regex:".A.a$"}},{nombre:1})
06
07 {"_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana"}
```

Las consultas anteriores son muy parecidas. La primera busca nombres que contengan los caracteres a minúscula mientras que la segunda consulta busca nombres que contengan un caracteres a mayúscula y otro minúscula. En MongoDB, por defecto, las expresiones regulares son sensibles a mayúsculas.

```
01 > db.usuarios.find({nombre:{$regex:".a.a$", $options:"i"}},
02 {nombre:1})
03
04 {"_id" : ObjectId("6266bac3bd92fbd18b5b6ca5"), "nombre" : "María"}
05 {"_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana"}
```

Con la opción "i", se indica a MongoDB que las comparaciones no serán sensibles a mayúsculas. Aunque las expresiones regulares pueden ser útiles, no conviene abusar de ellas. No todas pueden hacer uso de los índices y pueden hacer que las consultas sean muy lentas.

Cuando se hace una consulta en la Shell de MongoDB, el servidor devuelve un objeto cursor. Un cursor es un iterador sobre los resultados de una consulta. El cursor está en el servidor, mientras que en el cliente solo tenemos el identificador del mismo. Con este

² PCRE Regex: <http://perldoc.perl.org/perlre.html>

sistema se evitan mover datos innecesarios del servidor al cliente, ya que el cursor por defecto solo devuelve 20 resultados. Por otro lado, si se necesitan muestras más resultados, es necesario escribir "it" en la consola, lo que devolverá los siguientes 20 resultados. Lo bueno de los cursores es que tienen una serie de opciones interesantes que se pueden utilizar para contar el número de resultados u ordenarlos. Los operadores sobre cursores más comunes son: (i) `.count()`, (ii) `.sort()`, (iii) `.limit()`, (iv) `.skip()` y (v) `.toArray()`.

El operador `.count()` permite obtener el número de documentos devueltos en una consulta. A continuación se muestra un fragmento de código donde se ejemplifica el uso del comando:

```
01 > db.usuarios.find().count()
02 5
03
04 > db.usuarios.find({nombre:{$regex:".*$"}, $options:"i"}).count()
05 3
```

Por otro lado, el operador `.sort()`, permite ordenar los resultados por el campo especificado. La siguiente consulta ordena de forma ascendente:

```
01 > db.usuarios.find({nombre:{$regex:".*$"}}, {nombre:1,
02 apellido1:1}).sort({apellido1:1})
03
04 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" :
05 "Concha", "apellido1" : "López" }
06 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca5"), "nombre" : "María",
07 "apellido1" : "Pérez" }
08 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana",
09 "apellido1" : "Vallén" }
```

En el caso de que se quiera hacer la ordenación de forma descendente basta con poner el número del campo en negativo tal y como se muestra a continuación:

```
01 > db.usuarios.find({nombre:{$regex:".*$"}}, {nombre:1,
02 apellido1:1}).sort({apellido1:-1})
03
04 { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana",
05 "apellido1" : "Vallén" }
06 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca5"), "nombre" : "María",
07 "apellido1" : "Pérez" }
08 { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" :
09 "Concha", "apellido1" : "López" }
```

Además se puede especificar más de un criterio de ordenación simplemente separándolo por comas. En el ejemplo siguiente se muestran los resultados ordenados ascendentemente por apellido1 y descendentemente por apellido2

```
01 > db.usuarios.find({nombre:{$regex:".*$"}}, {nombre:1,
```

```

02  apellido1:1, apellido2:1}).sort({apellido1:1,apellido2:-1})
03
04  { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" :
05    "Concha", "apellido1" : "López", "apellido2" : "Arias" }
06  { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca5"), "nombre" : "María",
07    "apellido1" : "Pérez", "apellido2" : "Rodríguez" }
08  { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana",
09    "apellido1" : "Vallén", "apellido2" : "López" }

```

Con respecto al operador *.limit()*, se encarga de acotar el número de resultados devueltos por una consulta. En el siguiente ejemplo se muestra una consulta en la que solo se devuelven los tres primeros resultados.

```

01  > db.usuarios.find({}, {nombre:1}).limit(3)
02
03  { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"), "nombre" : "Eduardo" }
04  { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca5"), "nombre" : "María" }
05  { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" : "Concha" }

```

El operador *.skip()* ignora los N primeros documentos especificados. El siguiente ejemplo salta los 2 primeros documentos y devuelve los siguientes.

```

01  > db.usuarios.find({}, {nombre:1}).skip(2)
02
03  { "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"), "nombre" : "Concha" }
04  { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca7"), "nombre" : "Juan" }
05  { "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"), "nombre" : "Ana" }

```

Finalmente el operador *.toArray()* guarda los resultados en un array que se puede asignar a una variable.

```

01  > usuarios = db.usuarios.find({}, {nombre:1}).toArray()
02  [
03    {
04      "_id" : ObjectId("6266bac3bd92fbd18b5b6ca4"),
05      "nombre" : "Eduardo"
06    },
07    {
08      "_id" : ObjectId("6266bac3bd92fbd18b5b6ca5"),
09      "nombre" : "María"
10    },
11    {
12      "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"),
13      "nombre" : "Concha"
14    },
15    {
16      "_id" : ObjectId("62679dc3bd92fbd18b5b6ca7"),
17      "nombre" : "Juan"

```

```

18     },
19     {
20         "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"),
21         "nombre" : "Ana"
22     }
23 ]

```

Lo bueno de todos estos comandos, es que se pueden concatenar. Por ejemplo, en la consulta siguiente se salta el primer resultado (*skip(1)*), luego se cogen los tres siguientes (*limit(3)*), se ordenan ascendentemente por el primer apellido (*sort({apellido1:1})*) y finalmente se convierten a un array (*.toArray()*) para que sea almacenado en la variable *aux*.

```

01 > aux = db.usuarios.find({},
02 {nombre:1,apellido1:1}).skip(1).limit(3).sort({apellido1:1}).toArray()
03
04
05 [
06     {
07         "_id" : ObjectId("6266bac3bd92fbd18b5b6ca6"),
08         "nombre" : "Concha",
09         "apellido1" : "López"
10     },
11     {
12         "_id" : ObjectId("6266bac3bd92fbd18b5b6ca5"),
13         "nombre" : "María",
14         "apellido1" : "Pérez"
15     },
16     {
17         "_id" : ObjectId("62679dc3bd92fbd18b5b6ca7"),
18         "nombre" : "Juan",
19         "apellido1" : "Pérez"
20     }
21 ]

```

Actualizaciones en MongoDB

Los operadores en MongoDB para la actualización de la base de datos se dividen en dos tipos principales (i) operadores de actualización básicos y (ii) operadores de actualización avanzada.

Operadores de actualización básicos

Los métodos de actualización básica están compuestos por los operadores *.update()* y *.save()* que permiten actualizar el documento en una colección. El método *.update()* actualiza los valores en el documento existente y la sintaxis se define según se indica en la línea 01 del siguiente fragmento de código.

```

01 >db.col_name.update(query, set)

```

Donde:

col_name= nombre de la colección de documentos donde se realizarán las actualizaciones.
 query= permite definir la consulta que se usará para filtrar los documentos a modificar.
 set= permite indicar los elementos que serán modificados.

Si buscamos una analogía con la BD relacionales, query sería el WHERE y set el SET de una sentencia UPDATE relacional.

A continuación se va a ilustrar un ejemplo en el que se usará el comando `.update()` para actualizar la información de un documento. Es importante tener en cuenta que el documento siempre se cambia por el documento JSON que se pasa como parámetro (en `<set>`). Para realizar la operación tendremos que incluir un documento con todos los datos, pero cambiando los que queramos actualizar.

```

01 > db.usuarios.update( { nombre: "Eduardo" }, { nombre: "Eduardo",
02   edad : 23, altura : 1.85, casado : true, fechaAlta : new Date() })
03 WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
04
05 > db.usuarios.find({ nombre : "Eduardo" }).pretty()
06 {
07   "_id" : ObjectId("62668c7ebd92fbd18b5b6ca2"),
08   "nombre" : "Eduardo",
09   "edad" : 23,
10   "altura" : 1.85,
11   "casado" : true,
12   "fechaAlta" : ISODate("2022-04-25T12:01:35.020Z")
13 }
  
```

Tal y como se ilustra en el fragmento anterior, se ha incluido el documento que incluye la información del usuario "Eduardo" incluyendo tanto los campos que no se modifican como los que sí que serán modificados (subrayados en rojo).

Está claro que especificar todos los campos cuándo solo queremos modificar uno de ellos es bastante molesto, por eso entra en escena el operador `$set`. Este operador nos permite modificar (o crear si no existe) uno o varios campos sin tener que introducir el documento JSON completo.

```

01 > db.usuarios.update({ nombre: "María" }, { $set: { edad: 33,
02   casado: true } })
03 WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
04
05 > db.usuarios.find({ nombre : "María" }).pretty()
06 {
07   "_id" : ObjectId("626689e0bd92fbd18b5b6ca0"),
08   "nombre" : "María",
  
```



```

09      "apellido1" : "Pérez",
10      "apellido2" : "Rodríguez",
11      "edad" : 33,
12      "altura" : 1.74,
13      "fechaAlta" : ISODate("2022-04-25T11:44:08.062Z"),
14      "casada" : true
15  }

```

En las actualizaciones anteriores el filtro devolvía un solo documento. Si el filtro devolviese más de uno, la actualización sólo se realizaría sobre el primer documento que coincide. Esto se debe a que por defecto MongoDB solo actualiza un único documento a no se que le indiquemos explícitamente lo contrario. Si se necesita realizar una operación de actualización sobre múltiples documentos hay que usar la opción \$multi como tercer parámetro. En el ejemplo siguiente se muestra un fragmento de código donde se incluirá para todos los usuarios que están casados un campo con el número de hijos (inicialmente a 0).

```

01  > db.usuarios.update({casado:true}, {$set:{numHijos:0}}, {multi:true
02  })
03  WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
04
05  > db.usuarios.find({ nombre : "María" }).pretty()
06  {
07      "_id" : ObjectId("6262a0ccb2a119d14cdadcfc"),
08      "nombre" : "Eduardo",
09      "edad" : 22,
10      "altura" : 1.85,
11      "casado" : true,
12      "fechaAlta" : ISODate("2022-04-22T12:21:10.128Z"),
13      "numHijos" : 0
14  }
15  {
16      "_id" : ObjectId("626689e0bd92fbd18b5b6ca0"),
17      "nombre" : "María",
18      "apellido1" : "Pérez",
19      "apellido2" : "Rodríguez",
20      "edad" : 33,
21      "altura" : 1.74,
22      "fechaAlta" : ISODate("2022-04-25T11:44:08.062Z"),
23      "casado" : true
24      "numHijos" : 0
25  }

```

Además de la opción multi, tenemos disponible la opción upsert, que lo que hace es insertar el documento si este no existe. Por ejemplo la consulta siguiente buscará usuarios con el nombre "Miguel", pero al no existir ninguno insertará un nuevo documento.


```

01 > db.usuarios.update({nombre:"Miguel"},{nombre: "Miguel", edad :
02 42, altura : 1.80, casado : false, fechaAlta : new Date()},
03 {upsert:true})
04 WriteResult({
05     "nMatched" : 0,
06     "nUpserted" : 1,
07     "nModified" : 0,
08     "_id" : ObjectId("62669449a85a751516c735e0")
09 })
10
11 > db.usuarios.find({nombre: "Miguel"})
12 { "_id" : ObjectId("62669449a85a751516c735e0"), "nombre" :
13 "Miguel", "edad" : 42, "altura" : 1.8, "casado" : false,
14 "fechaAlta" : ISODate("2022-04-25T12:30:01.209Z") }
15 > db.usuarios.find({nombre: "Miguel"}).pretty()
16 {
17     "_id" : ObjectId("62669449a85a751516c735e0"),
18     "nombre" : "Miguel",
19     "edad" : 42,
20     "altura" : 1.8,
21     "casado" : false,
22     "fechaAlta" : ISODate("2022-04-25T12:30:01.209Z")
23 }

```

Tal y como se vio anteriormente, para actualizar o añadir un campo se debe usar el modificador \$set para evitar tener que añadir la totalidad del documento.

```

01 > db.usuarios.update({nombre:"Miguel"},{$set:{edad:43}})

```

En la anterior consulta, se usa \$set para modificar el documento con nombre "Miguel". En ese documento se actualiza el campo edad a 43. El documento resultante es el siguiente:

```

01 > db.usuarios.find({nombre:"Miguel"}).pretty()
02 {
03     "_id" : ObjectId("6267ecaaa85a751516c73ae1"),
04     "nombre" : "Miguel",
05     "edad" : 43,
06     "altura" : 1.8,
07     "casado" : false,
08     "fechaAlta" : ISODate("2022-04-26T12:59:22.176Z")
09 }

```

Si por el contrario, se pretende eliminar un campo de un documento, se puede hacer una consulta similar, pero utilizando el parámetro \$unset.

```

01 > db.usuarios.update({nombre:"Miguel"},{$unset:{fechaAlta:1}})
02
03 >db.usuarios.find({nombre:"Miguel"}).pretty()

```

```

04 {
05     "_id" : ObjectId("6267ecaaa85a751516c73ae1"),
06     "nombre" : "Miguel",
07     "edad" : 43,
08     "altura" : 1.8,
09     "casado" : false
10 }

```

Tal y como se puede observar, la consulta es bastante similar a la anterior, pero en este caso \$unset recibe como parámetro el nombre del campo y un 1 indicando que este campo debe eliminarse del documento. Si se ejecuta un *.find()* para ver el resultado (línea 03), se puede observar como el campo a eliminar (fechaAlta) ya no existe.

Operadores de actualización avanzada

Actualizar datos simples es sencillo, pero las operaciones pueden complicarse cuando se quiere actualizar arrays con varios elementos o subdocumentos con varios campos. En primer término se van a describir las operaciones usadas para modificar elementos de un array. Para ello, se van a realizar una consulta que modifique el correo electrónico "avl@gmail.com" por "ana.vallen.lopez@gmail.com" usando *Dot Notation*.

```

01 > db.usuarios.find({nombre:"Ana"},{nombre:1,email:1}).pretty()
02 {
03     "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"),
04     "nombre" : "Ana",
05     "email" : [
06         "avl@empresa.com",
07         "avl@gmail.com"
08     ]
09 }
10
11 > db.usuarios.update({nombre:"Ana"}, {$set:{"email.1":
12 "ana.vallen.lopez@gmail.com"}})
13
14 > db.usuarios.find({nombre:"Ana"},{nombre:1, email:1}).pretty()
15 {
16     "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"),
17     "nombre" : "Ana",
18     "email" : [
19         "avl@empresa.com",
20         "ana.vallen.lopez@gmail.com"
21     ]
22 }

```

En este caso se vuelve a usar \$set para modificar un campo en concreto, solo que esta vez se indica que se quiere modificar el campo "email.1". Tal y como se comentó en la sección anterior (consultas avanzadas), esta terminología indica a MongoDB que se quiere actualizar el elemento 1 del array de características. Como se destacó con anterioridad es importante tener en cuenta que los arrays empiezan con el índice 0, por lo que se está

modificando el segundo valor de dicho array. Si ese valor no existe, será añadido al array. Hay que tener cuidado con la posición del array utilizada, porque si se añade un elemento a un índice cuyos valores anteriores no existen, esos valores serán rellenados con valores nulos. Por ejemplo, en la siguiente consulta se añade un valor en la posición 5 del array, pero como no hay elementos en las posiciones 3 y 4 se rellenan con valores nulos.

```

01 > db.usuarios.update({nombre:"Ana"}, {$set:{"email.5":
02 "info@anavallen.es"}})
03
04 > db.usuarios.find({nombre:"Ana"},{nombre:1, email:1}).pretty()
05 {
06   "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"),
07   "nombre" : "Ana",
08   "email" : [
09     "avl@empresa.com",
10     "ana.vallen.lopez@gmail.com",
11     null,
12     null,
13     null,
14     "info@anavallen.es"
15   ]
16 }

```

Si se desconoce la posición del elemento a modificar, pero se conoce el valor que tiene, se puede utilizar Dot Notation con el operador \$.

```

01 > db.usuarios.update({nombre:"Ana", email:"info@anavallen.es"},
02 {$set:{"email.$": "updated@anavallen.es"}})
03
04 > db.usuarios.find({nombre:"Ana"},{nombre:1, email:1}).pretty()
05 {
06   "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"),
07   "nombre" : "Ana",
08   "email" : [
09     "avl@empresa.com",
10     "ana.vallen.lopez@gmail.com",
11     null,
12     null,
13     null,
14     "updated@anavallen.es"
15   ]
16 }

```

Tal y como se ha podido observar, es posible actualizar arrays con **Dot Notation**, sin embargo, existen varios operadores que pueden facilitarnos la tarea. **Dot Notation** se usa de forma más intuitiva cuándo se quieren actualizar subdocumentos. Las operaciones que podemos realizar sobre subdocumentos son similares a las que se han descrito con los arrays.

Si se quiere añadir un valor a un campo array de un documento existente, se puede utilizar el operador `$addToSet`. Este operador añade el valor que se le pasa como parámetro justo al final del array. Si el campo ya existe, el valor no se añade.

```

01 db.usuarios.update({nombre:"Ana"}, {$addToSet:
02 {"email":"updated@anavallen.es"}})
03 WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 0 })
04
05 > db.usuarios.find({nombre:"Ana"},{nombre:1, email:1}).pretty()
06 {
07   "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"),
08   "nombre" : "Ana",
09   "email" : [
10     "avl@empresa.com",
11     "ana.vallen.lopez@gmail.com",
12     null,
13     null,
14     null,
15     "updated@anavallen.es"
16   ]
17 }

```

Tal y como se puede observar, como el elemento a insertar ya existía la salida del comando `.find()` muestra los elementos originales. Si se quiere añadir varios elementos en un mismo comando, hay que añadir el operador `$each` y un array de elementos justo después del operador `$addToSet`.

```

01 > db.usuarios.update({nombre:"Ana"}, {$addToSet:{email:{$each:
02 ["updated@anavallen.es","latest@example.com"]}}})
03 WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
04
05 > db.usuarios.find({nombre:"Ana"},{nombre:1, email:1}).pretty()
06 {
07   "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"),
08   "nombre" : "Ana",
09   "email" : [
10     "avl@empresa.com",
11     "ana.vallen.lopez@gmail.com",
12     null,
13     null,
14     null,
15     "updated@anavallen.es",
16     "latest@example.com"
17   ]
18 }
19

```

Tal y como se puede observar, aunque se pretendían insertar dos direcciones de correo, en realidad solo se ha insertado una (`latest@example.com`) ya que la otra ya existía. Otra

opción para añadir elementos a un array consiste en usar el operador \$push. Este comando es exactamente igual que \$addToSet, con la diferencia de que, en este caso, el elemento siempre se añade, exista o no.

Si lo que se pretende es eliminar elementos de un array, existen dos opciones: (i) \$pop y (ii) \$pull. El operador \$pop permite eliminar el primer o último elemento de un array. Si al operador \$pop se le pasa un 1, entonces borrará el último elemento del array. Por el contrario, si se le pasa un -1, entonces eliminará el primer elemento del array.

```

01 > db.usuarios.update({nombre:"Ana"}, {$pop:{email:1}})
02 WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
03
04 > db.usuarios.find({nombre:"Ana"},{nombre:1, email:1}).pretty()
05 {
06   "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"),
07   "nombre" : "Ana",
08   "email" : [
09     "avl@empresa.com",
10     "ana.vallen.lopez@gmail.com",
11     null,
12     null,
13     null,
14     "updated@anavallen.es"
15   ]
16 }

```

Tal y como se puede observar en la consulta anterior, se ha eliminado la última entrada del array de correos electrónicos.

El comando \$pop requiere indicar obligatoriamente la posición a borrar, en el caso de que se quiera borrar un elemento en concreto sin saber su posición entonces hay que usar el comando \$pull.

```

01 > db.usuarios.update({nombre:"Ana"}, {$pull:
02   {email:"updated@anavallen.es"}})
03 WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
04
05 > db.usuarios.find({nombre:"Ana"},{nombre:1, email:1}).pretty()
06 {
07   "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"),
08   "nombre" : "Ana",
09   "email" : [
10     "avl@empresa.com",
11     "ana.vallen.lopez@gmail.com",
12     null,
13     null,
14     null
15   ]
16 }

```

Este comando realiza una consulta sobre el array, borrando todos los elementos del mismo que la cumplan. En el ejemplo anterior se han borrado todos los elementos que contenga el texto "updated@anavallen.es".

Si lo que se necesita es borrar varios elementos de un array, entonces se utilizará el comando \$pullAll.

```

01 > db.usuarios.update({nombre:"Ana"}, {$pullAll:{email:
02 [null,"ana.vallen.lopez@gmail.com"]}})
03 WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
04
05 > db.usuarios.find({nombre:"Ana"},{nombre:1, email:1}).pretty()
06 {
07   "_id" : ObjectId("62679dc3bd92fbd18b5b6ca8"),
08   "nombre" : "Ana",
09   "email" : [
10     "avl@empresa.com",
11     "ana.vallen.lopez@gmail.com"
12   ]
13 }

```

Para finalizar, cabe destacar dos operadores que pueden ser útiles a la hora de realizar actualizaciones. Una operación habitual es la de incrementar un valor numérico en una determinada cantidad. A modo de ejemplo, imaginemos que Miguel ha cumplido años y es necesario actualizar su edad en la base de datos. Con lo visto hasta ahora, habría que realizar una consulta a la base de datos, recuperar la edad existente, incrementarla y luego realizar una actualización. Todos estos pasos se pueden evitar usando simplemente una actualización usando el operador \$inc.

```

01 db.usuarios.find({nombre:"Miguel"},{nombre:1, edad:1}).pretty()
02 {
03   "_id" : ObjectId("6267ecaaa85a751516c73ae1"),
04   "nombre" : "Miguel",
05   "edad" : 43
06 }
07
08 > db.usuarios.update({nombre:"Miguel"}, {$inc:{edad:1}})
09 WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
10
11 > db.usuarios.find({nombre:"Miguel"},{nombre:1, edad:1}).pretty()
12 {
13   "_id" : ObjectId("6267ecaaa85a751516c73ae1"),
14   "nombre" : "Miguel",
15   "edad" : 44
16 }

```

Tal y como se puede observar el comando agrega 1 unidad más al campo edad. Por ejemplo, si hay 43 a la hora de actualizar, pasarán a ser 44. Si por el contrario, se usa un valor negativo entonces se restan las unidades.

También es posible equivocarse a la hora de crear un campo, y el nombre sea incorrecto. Para ello, habría que borrarlo y volverlo a crearlo. Para evitar estos pasos, MongoDB dispone del operador \$rename que permite renombrar el nombre de un campo.

Eliminaciones en MongoDB

Para eliminar datos de una colección, MongoDB pone a disposición el comando `.remove()`. Este comando recibe como parámetro la consulta que se utilizará para filtrar los documentos que se borrarán. Si no especificamos ninguna consulta, se eliminarán todos los datos de la colección. Como se puede ver el comportamiento es muy similar al de una operación DELETE de una base de datos relacional ya que si no se especifica un filtro con la sentencia WHERE se borrarán todos los datos de la tabla. En el ejemplo siguiente se ilustra el borrado del documento cuyo nombre de usuario es "Concha".

```

01 > db.usuarios.remove({nombre:"Concha"})
02 WriteResult({ "nRemoved" : 1 })
03
04 > db.usuarios.find()
05 { "_id" : ObjectId("626689e0bd92fbd18b5b6ca0"), "nombre" : "María",
06   "apellido1" : "Pérez", "apellido2" : "Rodríguez", "edad" : 32,
07   "altura" : 1.74, "fechaAlta" : ISODate("2022-04-
08   25T11:44:08.062Z") }
09 { "_id" : ObjectId("62668c7ebd92fbd18b5b6ca2"), "nombre" :
10   "Eduardo", "edad" : 23, "altura" : 1.85, "casado" : true,
11   "fechaAlta" : ISODate("2022-04-25T12:01:35.020Z") }
12 { "_id" : ObjectId("62669449a85a751516c735e0"), "nombre" :
13   "Miguel", "edad" : 42, "altura" : 1.8, "casado" : false,
14   "fechaAlta" : ISODate("2022-04-25T12:30:01.209Z") }

```

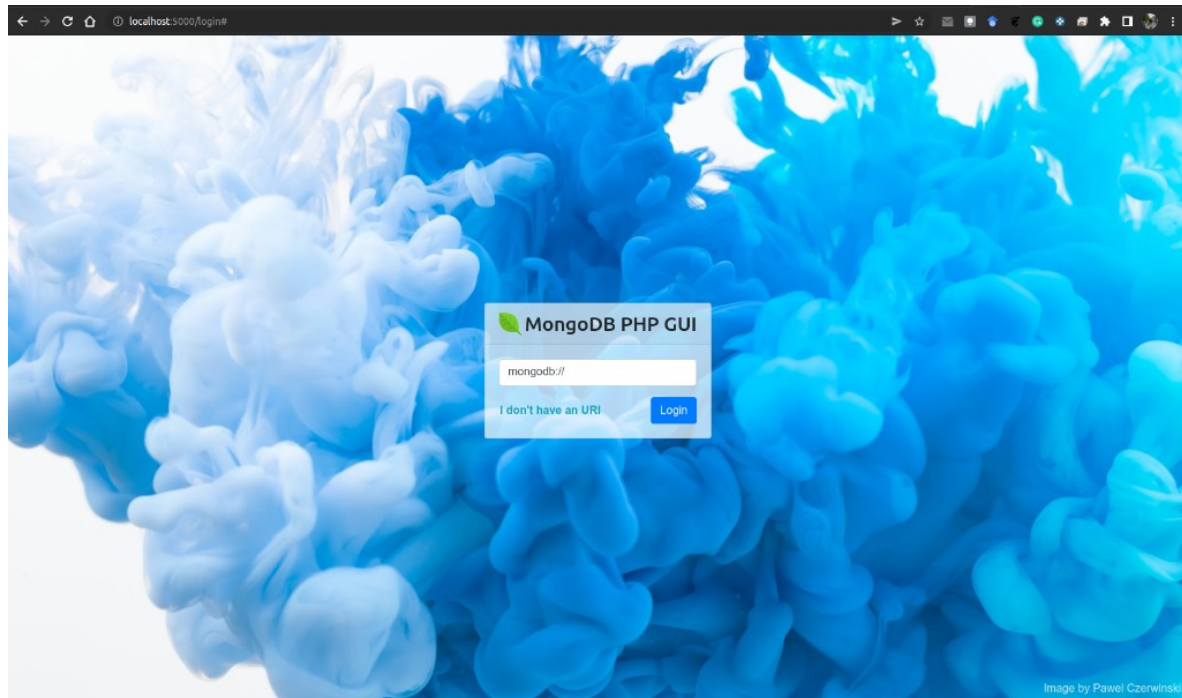
El comando `remove` acepta un segundo parámetro de tipo booleano llamado `justOne`. Por defecto es `false`, pero si lo establecemos como `true`, solo se borrará un documento de la colección, aunque existan varios que cumplan las condiciones especificadas en la consulta.

2.5 El cliente MongoDB-PHP-GUI

Además del uso de la consola por defecto (ver apartado anterior) se ha incluido una interfaz gráfica con el fin de facilitar el manejo y administración de MongoDB. Tal y como se abordó anteriormente, MongoDB-PHP-GUI se trata de un cliente GUI, web (implementado en PHP), por lo que, para acceder a él hay que introducir la URL: `localhost:5000` en el navegador tal y como se muestra en la imagen siguiente. Tal y como se puede observar, el navegador carga automáticamente la página de login por defecto donde hay que ingresar una URI (*Uniform Resource Identification*) que indique la ubicación (local o remota) donde se encuentra el servidor MongoDB.

Según el manual de instalación de MongoDB-PHP-GUI, existen dos formas de conectarse al servidor MongoDB en función de dónde esté localizado.

- Si MongoDB y MongoDB-PHP-GUI están instalados en la misma máquina entonces la conexión es `127.0.0.1:27017`
- Si MongoDB y MongoDB-PHP-GUI están instalados en diferentes máquinas entonces la conexión debe realizarse a la dirección (ip:puerto) `172.17.0.1:27017`

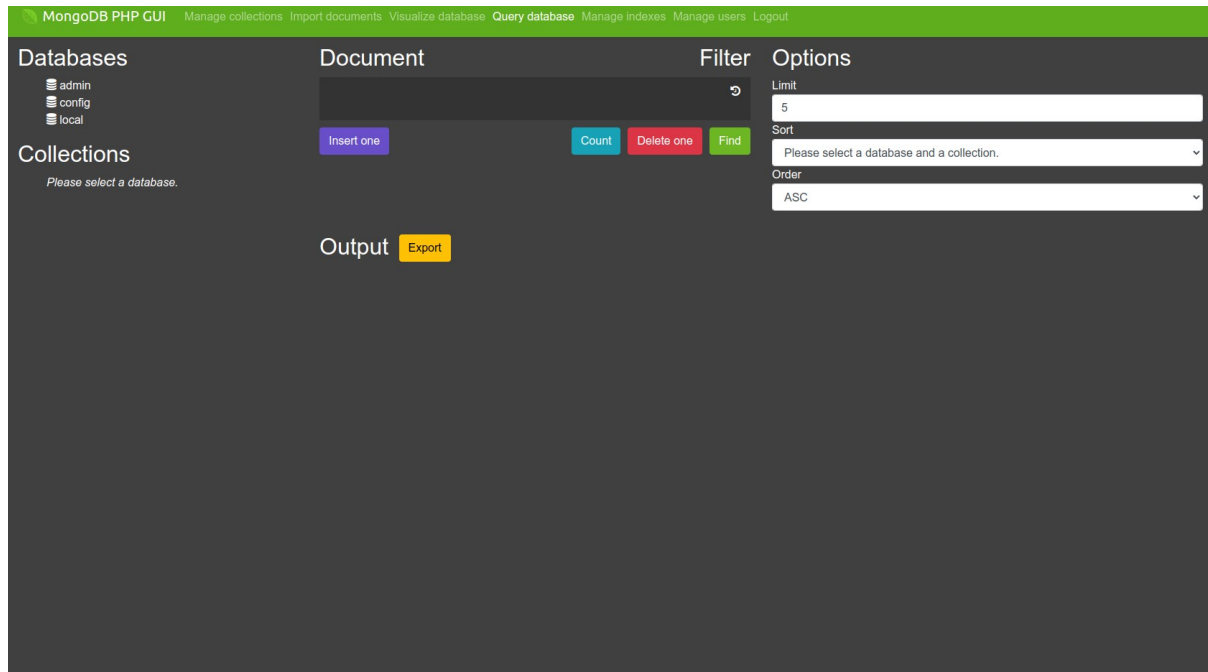


En el caso que nos ocupa, se han instalado los dos servicios (MongoDB y MongoDB-PHP-GUI) en contenedores independientes. Esto quiere decir que están en ubicaciones distintas por lo que se debe conectar usando la segunda opción.

Para la guía que nos ocupa, vamos a utilizar los parámetro de conexión a MongoDB incluidos por defecto por lo que la URI resultante sería: *mongodb://172.17.0.1:27017*.



Finalmente, y tras pinchar sobre el botón Login, se realizará la conexión al servidor MongoDB mostrándose la pantalla que aparece a continuación:



The screenshot displays the MongoDB PHP GUI interface. At the top, a green navigation bar contains the title 'MongoDB PHP GUI' and several menu items: 'Manage collections', 'Import documents', 'Visualize database', 'Query database' (which is highlighted), 'Manage indexes', 'Manage users', and 'Logout'. The main interface is divided into four panels. The 'Databases' panel on the left lists 'admin', 'config', and 'local'. Below it, the 'Collections' panel shows a message 'Please select a database.' The central 'Document' panel features a large text input field for a query, with buttons for 'Insert one' (purple), 'Count' (cyan), 'Delete one' (red), and 'Find' (green). Below the input field is an 'Output' section with an 'Export' button (yellow). The 'Filter' panel on the right is currently empty. The 'Options' panel on the far right includes a 'Limit' input field set to '5', a 'Sort' dropdown menu with the text 'Please select a database and a collection.', and an 'Order' dropdown menu set to 'ASC'.