

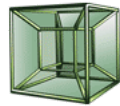
NoSQL
Not Only SQL

APACHE
HBASE

 **Cassandra**


CouchDB
relax

 **riak**



mongoDB

HYPERTABLE INC



Neo4j



redis

En los últimos años están saliendo nuevas Bases de Datos que están huyendo de lo que ellos llaman "rigidez" de las bases de datos relacionales, donde los datos están muy estructuradas: tablas con campos perfectamente definidos, y con las restricciones que hemos visto que se pueden crear.

Y si quiere guardar datos poco estructuradas? Por ejemplo los usuarios de una determinada organización, que no siempre nos guardamos la misma información: unas veces nos guardamos el teléfono, otros no, otros nos guardamos más de un teléfono; otras veces nos queremos guardar su página web, otras veces las aficiones, ... En el modelo relacional, muy estructurado, deberíamos prever todos los campos necesarios, y en caso de que surgiera la necesidad de una nueva información deberíamos cambiar la estructura de la tabla.

U otro ejemplo, queremos guardar sencillamente unas cuantas "variables" de forma permanente. Podría ser por ejemplo guardar las preferencias del usuario en una aplicación para dispositivos móviles. Si lo diseñamos en una mesa, sería una tabla con muchos campos, pero que después sólo habrá una fila, la del usuario.

Otro ejemplo sería guardar información de documentos XML o JSON, los estándares de intercambio de información. En el Modelo Relacional deberíamos guardar el documento XML o JSON prácticamente en un campo, todo junto, y sería costoso buscar dentro de la estructura XML o JSON.

Para salvar estos inconvenientes mencionados han surgido una serie de iniciativas nuevas que se han agrupado en el término **NoSQL** (Not Only SQL). Son en definitiva Bases de Datos que no están basadas en el Modelo Relacional, y que en determinadas ocasiones pueden ser más eficientes que las bases de datos relacionales precisamente para huir de la rigidez que proporciona este modelo, con datos tan bien estructuradas. El término clave es "en determinadas ocasiones", es decir, para guardar un determinado tipo de información. Así nos encontraremos distintos tipos de Bases de Datos NoSQL, dependiendo del tipo de información que quieran guardar:

- **Bases de Datos Orientadas a Objetos** , para poder guardar objetos, como por ejemplo **DB4O**
- **Bases de Datos Orientadas a Documentos** (o simplemente Bases de Datos Documentales), para guardar documentos de determinados tipos: XML, JSON, ... Por ejemplo **existen** que guarda documentos XML, o **MongoDB** que guarda la información en un formato similar a JSON .
- **Bases de Datos Clave-Valor** , para guardar información de forma muy sencilla, guardando únicamente la clave (el nombre de la propiedad) y su valor
- **Bases de Datos Orientadas a Grafos** , para guardar estructuras como los grafos, donde hay una serie de nodos y aristas que comunicarían (relacionarían) los nodos
- Otros tipos como las **Bases de Datos multivalor** , las **Bases de Datos tabulares** , **Bases de Datos Orientadas a Columna** , ...

En este tema sólo vuerem las Bases de Datos **Clave-Valor** , por su sencillez, y otra base de datos, **MongoDB** , para su utilización actual.

2 - Bases de Datos Clave-Valor

Seguramente de todos los tipos de Bases de Datos NoSQL existentes, la de más fácil comprensión es la **Base de Datos Clave-Valor**. En ella se irán guardando parejas clave-valor, es decir, el nombre de una determinada propiedad y su valor. La clave debe ser única, es decir, no se puede repetir ya que de lo contrario no se podría volver a recuperar.

Por lo tanto no hay definición de tablas ni ninguna otra estructura; se guardando parejas clave-valor y suficiente. En el momento de recuperar la información veremos que podremos obtener el contenido de una clave, o el de más de una a la vez.

El ejemplo que veremos de Base de Datos Clave-Valor es **Redis**, muy famosa para su potencia y eficiencia.

La clave siempre es de tipo String, y como ya hemos comentado no pueden haber dos llaves iguales. En cambio en Redis los valores pueden ser de 5 tipos diferentes:

- **Cadenas de caracteres (String)**. Por ejemplo: `nom_1 -> Albert`
- **Registros (hash)** que serán claves que tendrán subcampos (sub-llaves). Por ejemplo: `empleat_1 -> [nombre = "Albert", departamento = "10", sueldo = "1000.0"]`
- **Listas (Lists)** que son conjuntos ordenados de valores. Por ejemplo: `llista_1 -> { "Primero", "Segundo", "Tercer" }`
- **Conjuntos (Sets)** son conjuntos desordenados de valores. No importa su orden, y de hecho será impredecible el orden con el que los devuelve Redis. Por ejemplo: `colores -> { "Azul", "Verde", "Rojo" }`
- **Conjuntos Ordenados (sorted Sets)**. Ya veremos la diferencia entre listas y conjuntos ordenados.

Algunas características de Redis son:

- Es una arquitectura cliente-servidor
- Es extraordinariamente eficiente cuando se puede cargar toda en memoria, aunque si no puede cargarla también funcionará de forma muy rápida. Y además mantiene una sincronización constante a disco para hacer los datos persistentes. Esta tarea la realiza en segundo plano, por lo que no afecta al servicio.
- Para poder soportar ratios de lectura muy altos hace una replicación master / slave, es decir que puede haber más de un servidor, y uno es el que actúa de master y los demás son réplicas del primero. El slave recibe una copia inicial de la Base de Datos entera. A medida que se van realizando escrituras en el master, se van enviando a todos los slaves conectados. Los clientes se pueden conectar a los distintos servidores, bien sea al master o los slaves, sin tener que cargar siempre al master.

Redis está construido para Linux. También funciona, pero, desde Windows como veremos un poco más adelante.

Instalación en Linux

El lugar desde donde bajarlo es la página oficial:

<http://redis.io>

En el momento de hacer estos apuntes, la última versión estable es la **3.0.7**.

Nos bajamos el archivo, lo descomprimos, y luego desde una terminal nos situamos en el directorio donde se ha descomprimido y hacemos **make** para generar los ejecutables. Después de muchos avisos, se debería haber instalado bien, y sin ser necesarios los permisos de administrador. Este sería el resumen de acciones, realizadas todas ellas desde una terminal (pero no es necesario descomprimir desde una terminal) y habiéndonos situado previamente en el lugar donde está el fichero descargado. También supondremos que el archivo está colocado en el lugar donde queremos que esté instalado de forma definitiva. Recuerde que puede descomprimirlo de la manera que le resulte más cómoda:

```
tar xzf redis-3.0.7.tar.gz
```

o también se puede descomprimir desde el navegador de archivos, como ya habíamos comentado.

Una vez descomprimido, desde una terminal nos debemos situar en el directorio recién descomprimir y hacer **make**

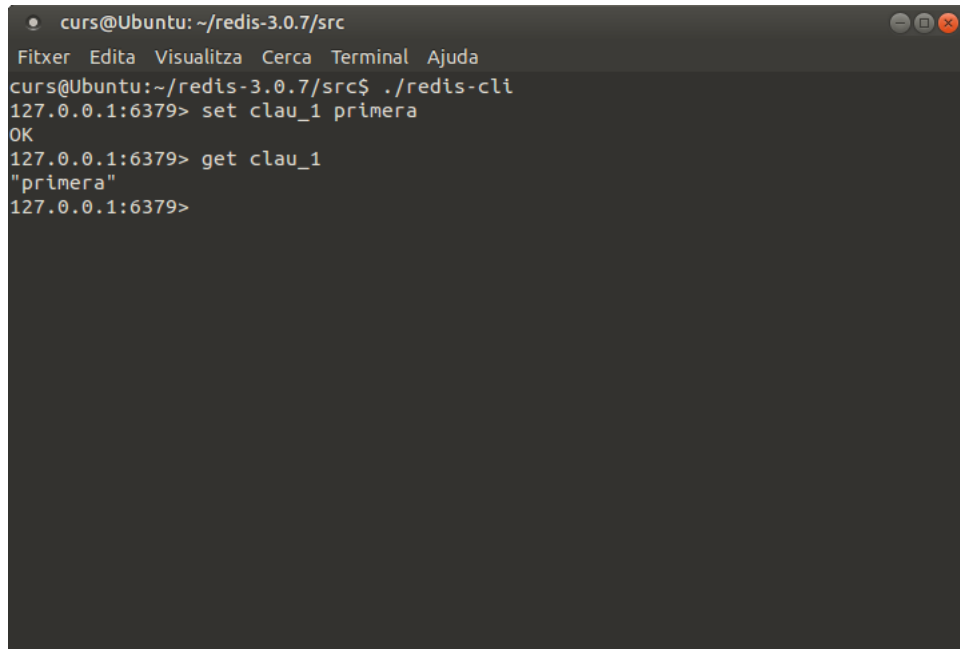
```
cd redis-3.0.7  
make
```

Con esto se deberían haber generado los ejecutables, y ya debería funcionar. Recuerde que no hacen falta permisos de administrador para realizar esto.

Para poner en marcha el servidor, casi que lo más cómodo será abrir un terminal, situarnos en el directorio **redis-3.0.7 / src** y desde ahí ejecutar **redis-server**. Debería salir una ventana similar a la siguiente, con más o menos avisos (obsérvese que al principio de la imagen están las órdenes dadas).

Ya ha hecho la conexión, concretamente en localhost (127.0.0.1) y el puerto 6379, que habíamos quedado que es el puerto por defecto.

Comprobamos que sí funciona. Aún no hay datos, porque la acabamos de instalar. Y recuerde que es una Base de Datos clave-valor. Para crear una entrada pondremos **siete llaves valor** . Para obtenerla pondremos **get clave** . En la imagen se puede comprobar:

A screenshot of a terminal window titled 'curs@Ubuntu: ~/redis-3.0.7/src'. The window contains the following text:

```
Fitxer Edita Visualitza Cerca Terminal Ajuda
curs@Ubuntu:~/redis-3.0.7/src$ ./redis-cli
127.0.0.1:6379> set clau_1 primera
OK
127.0.0.1:6379> get clau_1
"primera"
127.0.0.1:6379>
```

Hemos creado una clave llamada **clau_1** con el valor **primera** , como se puede comprobar en el momento de obtenerla con **get** .

Si el programa **redis-cli** no le ponemos parámetros, intentará hacer una conexión local (localhost). Si queremos conectar a un servidor situado en otra dirección, se la ponemos con el parámetro **-h dirección** , por ejemplo:

```
redis-cli -h 9 9.66.214.106
```

nos haría la conexión al servidor.

Instalación en Windows de 64 bits

Aunque Redis está construido para Linux, hay versiones para Windows, preferiblemente de 64 bits. También podremos encontrar versiones de 32 bits, pero mucho más antiguas.

El lugar donde poder bajar los ficheros de Redis para Windows de 64 bits es:

<https://github.com/MSOpenTech/redis/releases>

The screenshot shows the GitHub repository page for **MSOpenTech / redis**, which is forked from **antirez/redis**. The page displays the latest release, **2.8.2400**, released on January 21 by **enricogior**. This release is based on 2.8.21 from the original repository and includes additional fixes for Windows-specific bugs. The release page provides download links for **Redis-x64-2.8.2400.msi** (6.28 MB) and **Redis-x64-2.8.2400.zip** (5.46 MB), along with links to the **Source code (zip)** and **Source code (tar.gz)**.

Nos bajamos el **zip**, lo descomprimos, y ya lo tendremos disponible (sin hacer **make** ni nada). Obsérvese como en la carpeta resultado de descomprimir ya tenemos los ejecutables **redis-server** y **redis-cli** que son los que nos interesan:

The screenshot shows a Windows File Explorer window displaying the contents of the **Redis-x64-2.8.2400** folder. The folder contains 16 files, including executables, configuration files, and documentation. The files are listed in a table with columns for Name, Date modified, Type, and Size.

Nombre	Fecha de modifica...	Tipo	Tamaño
EventLog.dll	06/05/2016 11:54	Extensión de la apl...	1 KB
Redis on Windows Release Notes	06/05/2016 11:54	Documento XML ...	13 KB
Redis on Windows	06/05/2016 11:54	Documento XML ...	18 KB
redis.windows.conf	06/05/2016 11:54	Archivo CONF	41 KB
redis.windows-service.conf	06/05/2016 11:54	Archivo CONF	41 KB
redis-benchmark	06/05/2016 11:54	Aplicación	401 KB
redis-benchmark.pdb	06/05/2016 11:54	Archivo PDB	4.292 KB
redis-check-aof	06/05/2016 11:54	Aplicación	253 KB
redis-check-aof.pdb	06/05/2016 11:54	Archivo PDB	3.452 KB
redis-check-dump	06/05/2016 11:54	Aplicación	262 KB
redis-check-dump.pdb	06/05/2016 11:54	Archivo PDB	3.420 KB
redis-cli	06/05/2016 11:54	Aplicación	472 KB
redis-cli.pdb	06/05/2016 11:54	Archivo PDB	4.412 KB
redis-server	06/05/2016 11:54	Aplicación	1.395 KB
redis-server.pdb	06/05/2016 11:54	Archivo PDB	6.268 KB
Windows Service Documentation	06/05/2016 11:54	Documento XML ...	15 KB

Ejecutamos **redis-server** directamente y ya lo tendremos en marcha:

```
C:\Users\usuari\Redis 2.8.24>redis-server.exe

[848] 06 May 17:38:57.864 # Warning: no config file specified, using the default
config. In order to specify a config file use C:\Users\usuari\Redis 2.8.24\redi
s-server.exe /path/to/redis.conf

Redis 2.8.2400 (00000000/0) 64 bit
Running in stand alone mode
Port: 6379
PID: 848

http://redis.io

[848] 06 May 17:38:57.883 # Server started, Redis version 2.8.2400
[848] 06 May 17:38:57.883 * The server is now ready to accept connections on por
t 6379
```

Ejecutamos también el **redis-cli** y el resultado será el mismo que en Linux. Hemos incorporar una nueva clave y luego obtenemos su contenido:

```
C:\Users\usuari\Redis 2.8.24>redis-cli.exe

127.0.0.1:6379> set clau_1 primera
OK
127.0.0.1:6379> get clau_1
"primera"
127.0.0.1:6379> _
```

Instalación en Windows de 32 bits

Es un poco más complicada de encontrar. Y sobre todo, es una versión bastante más antigua. Podemos descargarla de la siguiente dirección:

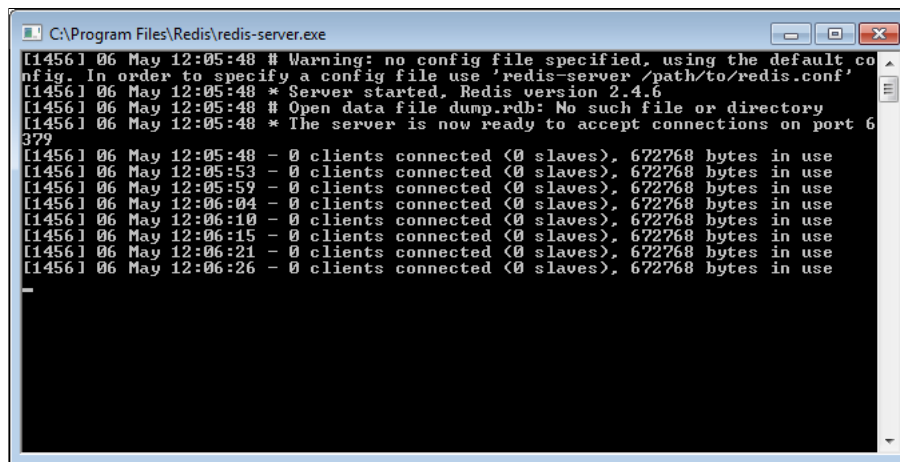
<https://github.com/rgl/redis/downloads>

The screenshot shows the GitHub repository page for **rgl / redis**, which is forked from **dmajic/redis**. The page includes navigation links like Personal, Open source, Business, Explore, Pricing, Blog, and Support. It also shows repository statistics: 54 Watch, 332 Star, and 6,380 Fork. The 'Download Packages' section lists three files:

- 1. **redis-2.4.6-setup-64-bit.exe** — Redis 2.4.6 Windows Setup (64-bit)
796KB · Uploaded on 11 Feb 2012
- 2. **redis-2.4.6-setup-32-bit.exe** — Redis 2.4.6 Windows Setup (32-bit)
768KB · Uploaded on 11 Feb 2012
- 3. **redis-2.2.12-setup-32-bit.exe** — Redis 2.2.12 Windows Setup (32-bit)
748KB · Uploaded on 9 Oct 2011

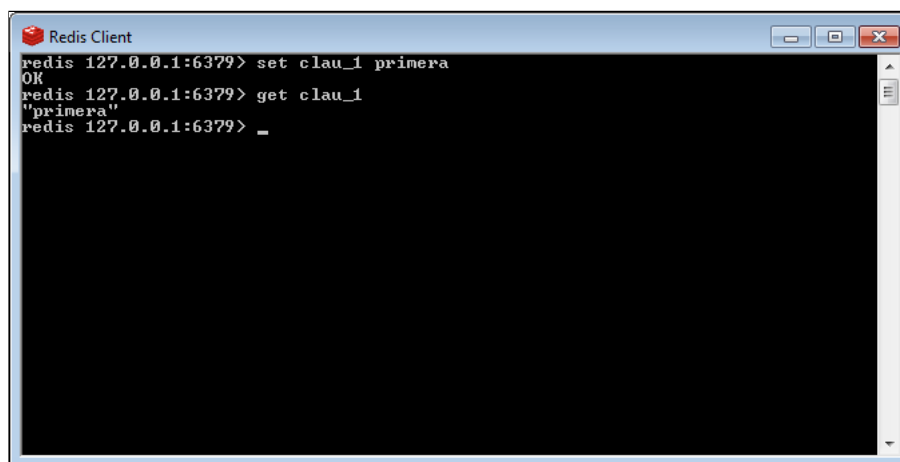
Esta sí que es de instalación (no de descompresión únicamente). La documentación sugiere que se active como un servicio. Por ello, en la lista de programas del Inicio no está el server, únicamente el **cliente**. Pero lo podemos encontrar en el directorio donde se ha instalado: **C:**

Archivos de programa \ Redis . Encontraremos tanto **redis-server.exe** como **redis-cli.exe** . Ponemos en marcha el servidor haciendo doble-clic en **redis-server.exe** :



```
C:\Program Files\Redis\redis-server.exe
[1456] 06 May 12:05:48 # Warning: no config file specified, using the default co
nfig. In order to specify a config file use 'redis-server /path/to/redis.conf'
[1456] 06 May 12:05:48 * Server started, Redis version 2.4.6
[1456] 06 May 12:05:48 # Open data file dump.rdb: No such file or directory
[1456] 06 May 12:05:48 * The server is now ready to accept connections on port 6
379
[1456] 06 May 12:05:48 - 0 clients connected (0 slaves), 672768 bytes in use
[1456] 06 May 12:05:53 - 0 clients connected (0 slaves), 672768 bytes in use
[1456] 06 May 12:05:59 - 0 clients connected (0 slaves), 672768 bytes in use
[1456] 06 May 12:06:04 - 0 clients connected (0 slaves), 672768 bytes in use
[1456] 06 May 12:06:10 - 0 clients connected (0 slaves), 672768 bytes in use
[1456] 06 May 12:06:15 - 0 clients connected (0 slaves), 672768 bytes in use
[1456] 06 May 12:06:21 - 0 clients connected (0 slaves), 672768 bytes in use
[1456] 06 May 12:06:26 - 0 clients connected (0 slaves), 672768 bytes in use
```

Y ya sólo falta poner en marcha el cliente (desde el menú o haciendo doble clic en **redis-cli.exe**). En la imagen se ve como podemos conectar, crear una clave y volver su valor:



```
Redis Client
redis 127.0.0.1:6379> set clau_1 primera
OK
redis 127.0.0.1:6379> get clau_1
"primera"
redis 127.0.0.1:6379> _
```

2.2 - entorno gráfico: Redis Desktop Manager

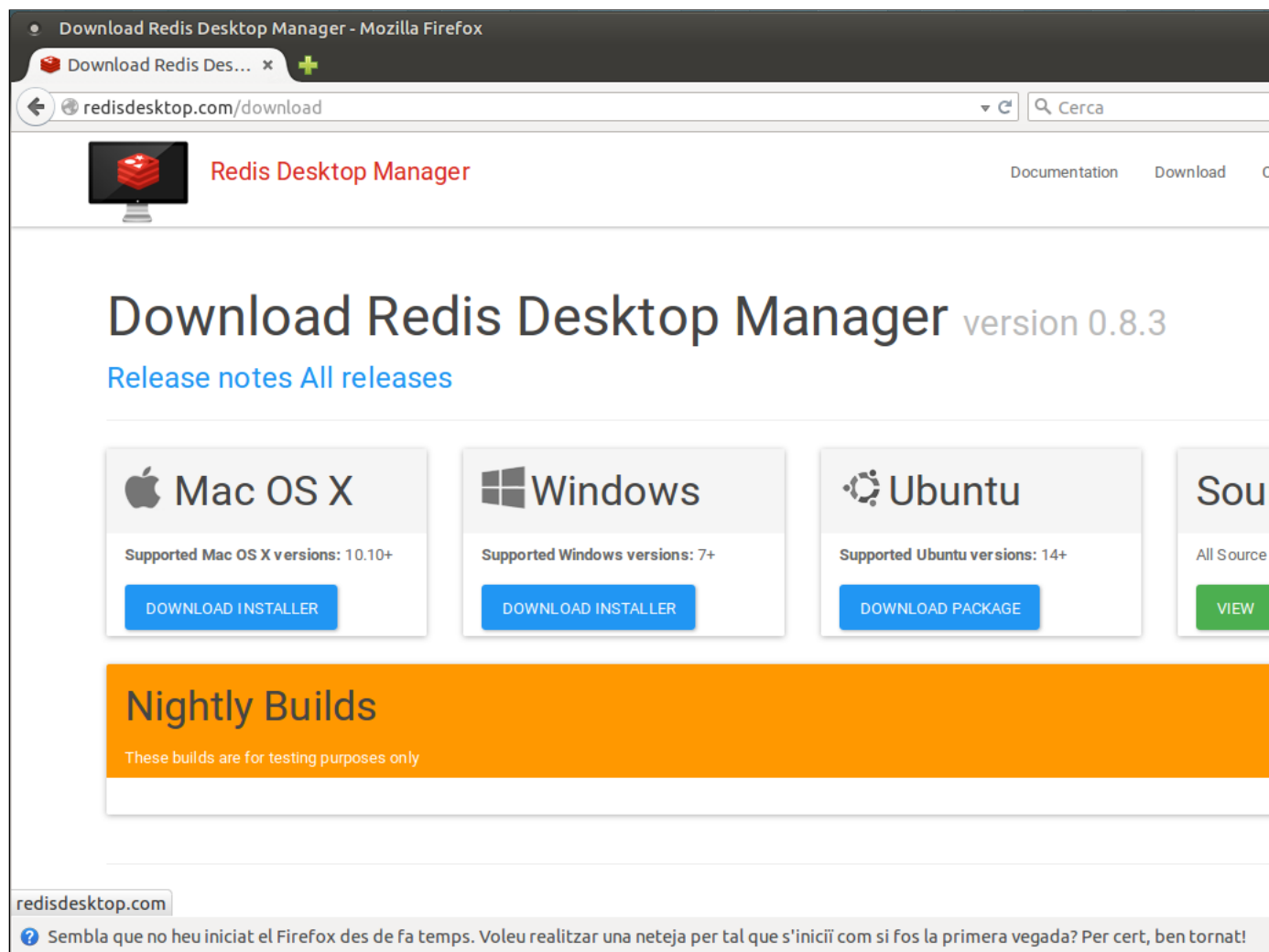
Como hemos comprobado en el punto anterior, la conexión que hacemos desde el cliente es a través de consola. Por lo tanto tendremos que poner comandos y nos contestará su ejecución.

Podemos instalarnos una aplicación gráfica que haga un poco más atractiva la presentación.

La instalación de esta herramienta es **totalmente optativa**, no es necesario que la haga. De hecho, nos los ejemplos que se mostrarán en todo el tema sólo se utilizará el modo consola.

Es completamente independiente del servidor, y podemos instalarla perfectamente sin tener el servidor, utilizándola entonces para conectarse a un servidor remoto.

Lo podemos bajar libremente de la página oficial redisdesktop.com donde podremos comprobar que tenemos para todas las plataformas:



Instalación en Ubuntu

Nos bajaremos el paquete que deberemos instalar bien con el Centro de software de Ubuntu, o bien sencillamente desde una terminal, situados en el directorio donde la hemos bajado haciendo

```
sudo dkpg -y redis-desktop-manager_0.8.3-120_amd64.deb
```

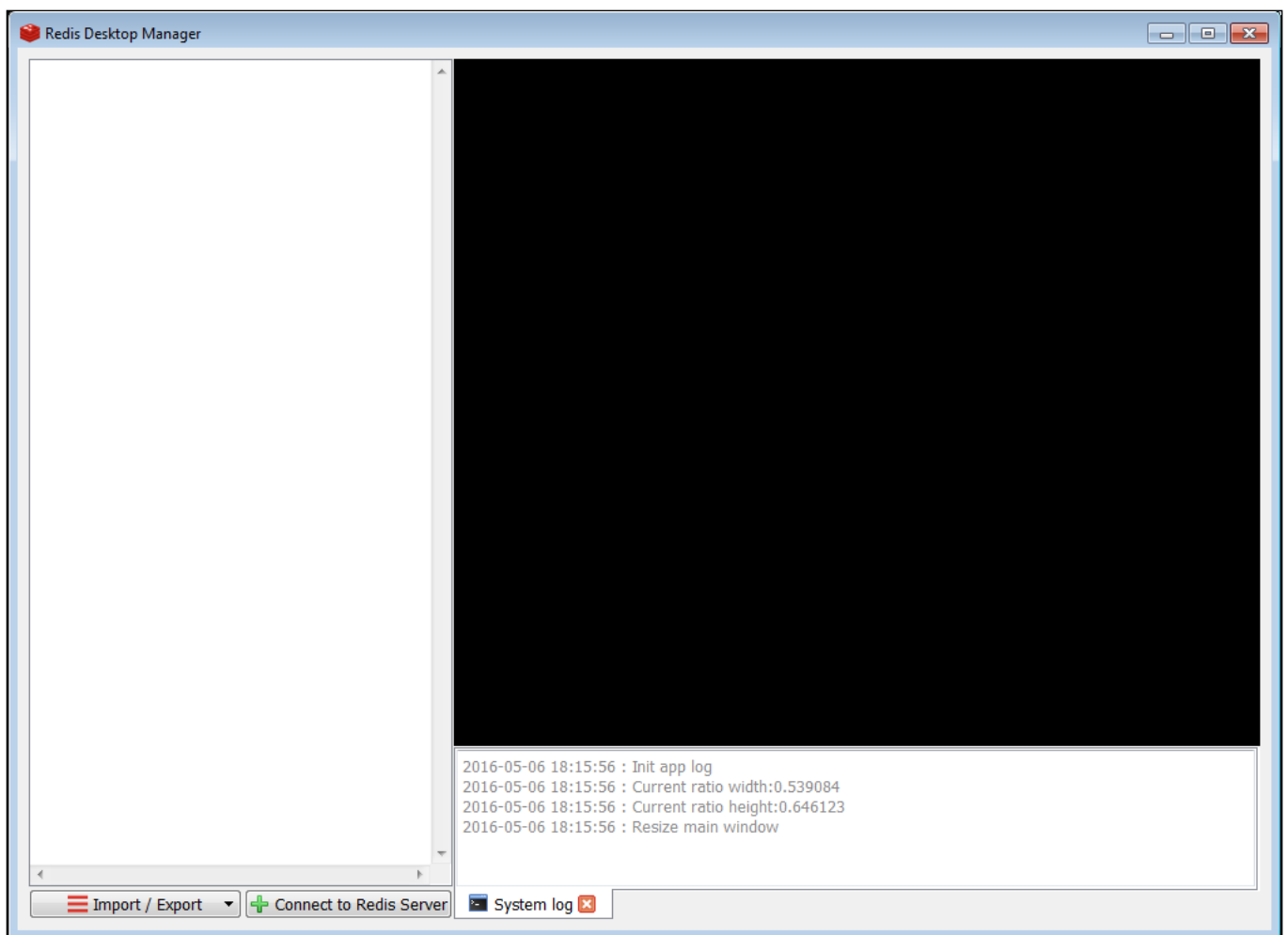
Como veis hacen falta permisos de administrador, y si no hay problemas de dependencias, se instalará sin problemas.

Como es igual que en Windows, si tiene curiosidad vea la imagen posterior de cuando está instalado, para comprobar la conexión al servidor.

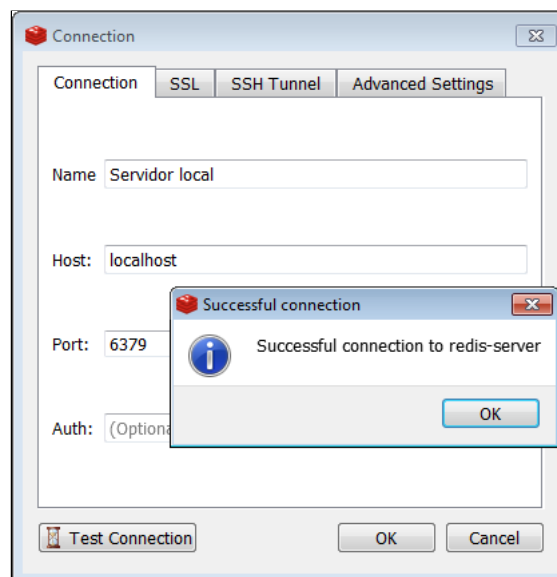
Instalación en Windows

En Windows lo que nos bajaremos es un exe. El ejecutamos (permitiendo la ejecución cuando lo pregunta Windows) y le podemos dar a todas las opciones por defecto.

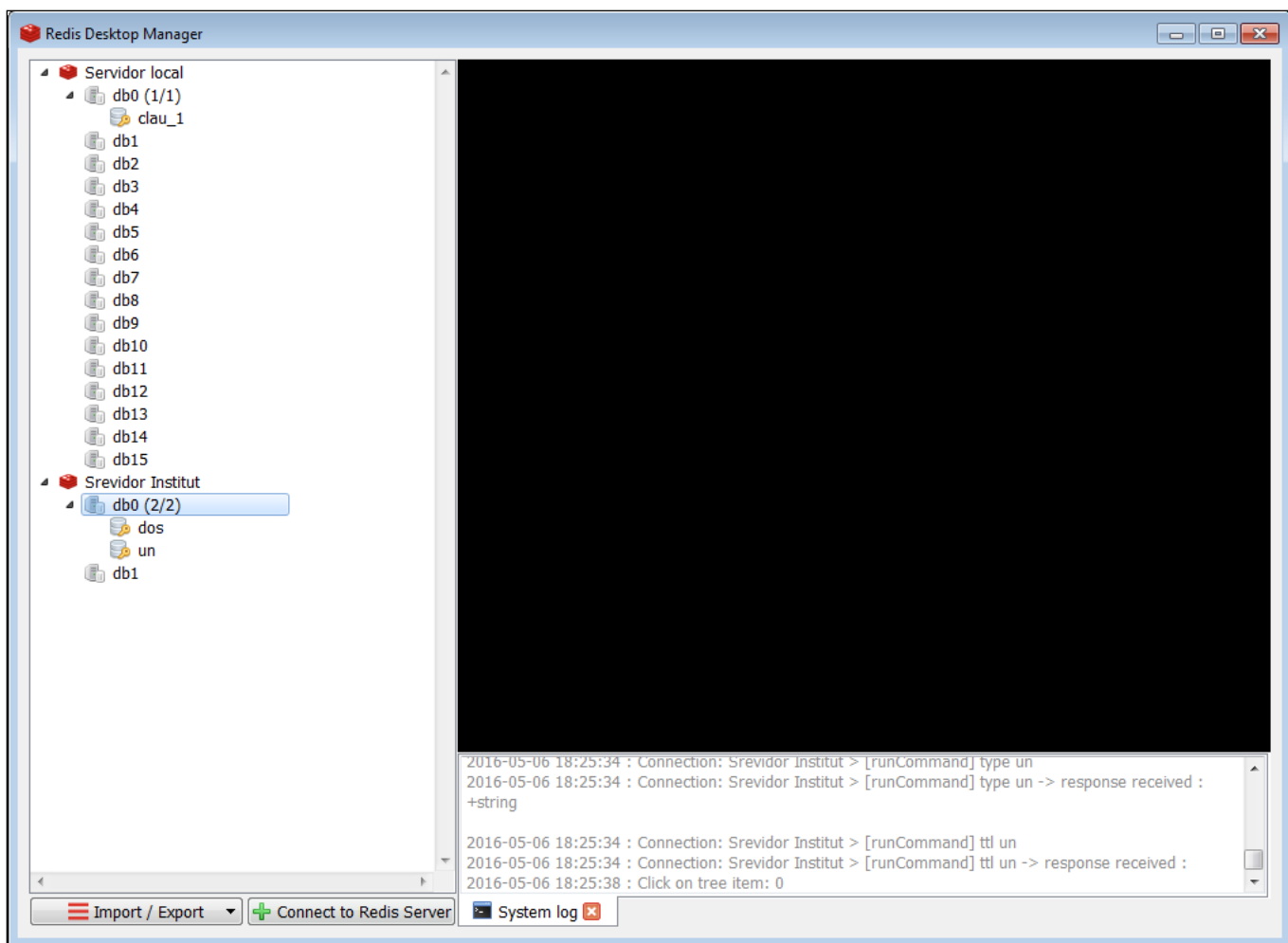
Cuando lo ejecutamos, nos saldrá la siguiente pantalla:



Podemos comprobar que abajo tenemos el botón para conectarse a un servidor. Para conectar al servidor local le ponemos como dirección **localhost** . Para conectar al servidor remoto le ponemos como dirección **la IP del servidor**. En la imagen se ha hecho el test de conexión.



En esta imagen se ve como hemos conectado perfectamente los dos servidores.



Vamos a ver la utilización de Redis. Nos conectaremos como clientes e intentaremos hacer operaciones.

- Las primeras serán las más sencillas, utilizando únicamente el tipo de datos **String** .
- Posteriormente miraremos cómo trabajar con las claves: buscar una, ver si existe, buscar unas cuantas, ...
- Luego ya iremos a por los tipos de datos más complicados:
 - **hash**
 - **List**
 - **siete**
 - **sorted Siete**

Es el tipo de datos más sencillo, más básico. Será una cadena de caracteres de tipo **binary safe** en la que normalmente guardaremos las habituales cadenas de caracteres, pero que también podríamos guardar imágenes u objetos serializados de Java. El tamaño máximo es de 512Mb.

Ahora veremos los comandos más habituales que afectan a este tipo. Como norma general, debemos ser conscientes de que los comandos no son sensibles a mayúsculas o minúsculas, pero las claves y los valores sí lo son. Es decir, el comando **get** también se puede escribir **GET** o **Get**. Pero la clave **Hola** es diferente de la clave **hola**.

Concretamente, los comandos que veremos serán:

`get`, `septiembre`, `Setex`, `psetex`, `mget`, `mset`, `append`, `strlen`, `getrange`, `setrange`, `incr`, `Decr`, `incrby`, `decrby`

GET

sintaxis

```
get clave
```

Vuelve el valor de la clave especificada, siempre que sea de tipo **String**. Si la clave es de otro tipo, dará error. Y si la clave no existe, devolverá el valor especial **nil**.

ejemplos

```
127.0.0.1:6379> get clau_1
"primera"
127.0.0.1:6379> get clau_2
(nil)
```

SET

sintaxis

```
siete llaves valor
```

Asigna a la clave especificada como primer parámetro el valor especificado como segundo parámetro. Si el valor consta de más de una palabra, deberá ir entre comillas dobles.

Redis siempre guardará el valor como string, aunque nosotros pensemos que le pasamos un valor entero o real.

Y otra característica es que si la clave existe ya, matxará su contenido, como era de esperar.

ejemplos

```
127.0.0.1:6379> set clau_2 segunda
OK
127.0.0.1:6379> set texto "Un texto con más de una palabra"
OK

127.0.0.1:6379> set cuatro 4
OK
127.0.0.1:6379> get cuatro
"4"
127.0.0.1:6379> set pi 3.14159265359
OK
127.0.0.1:6379> get pi
"3.14159265359"
```

Nota

Si realiza algún acento, al volver el valor (haciendo get) le parecerá que no se ha guardado bien. Sí que habrá guardado bien, lo que pasa es que posteriormente no se visualiza bien en hacer lo get. Se puede comprobar entrando en el cliente con la opción **raw**, es decir **redis-cli --raw**

El comando **SET** tiene una opción muy interesante, que servirá para dar un tiempo de vida en la llave, transcurrido el cual desaparece la llave (con su valor, claro). Esto se denomina **tiempo de expiración** y se consigue con el parámetro **EX** del comando **SET** seguido del número de segundos que queremos que dure la clave.

ejemplo

```
127.0.0.1:6379> set clau_3 tercera ex 10
OK
```

```
127.0.0.1:6379> get clau_3
"tercera"
127.0.0.1:6379> get clau_3
(nil)
```

Primero sí existe, pero al cabo de 10 segundos ha dejado de existir.

De forma equivalente se puede expresar el tiempo en milisegundos, con el parámetro **PX** en cuenta de **EX**.

Habíamos comentado al principio, que si en el momento de hacer el **SET** la clave ya existía, se reemplazará su contenido. Podemos modificar este comportamiento con el parámetro **NX** (not exist): si no existía la clave, la creará con el valor, pero si ya existía, la dejará como estaba. Nos lo indicará diciendo OK en caso de crearla y NIL en caso de no crearla porque ya existía.

```
127.0.0.1:6379> set clau_4 cuarta nx
OK
127.0.0.1:6379> set clau_1 cuarta nx
(nil)
127.0.0.1:6379> get clau_4
"cuarta"
127.0.0.1:6379> get clau_1
"primera"
```

Y de forma inversa, si ponemos el parámetro **XX**, si ya existe la clave, reemplazará el valor, pero si no existía, no hará nada.

SETEX

sintaxis

```
Setex clave según valor
```

Funciona igual que el **SET** con el parámetro **EX**: creará la clave con el valor, pero tendrá una existencia de los segundos indicados.

PSETEX

sintaxis

```
psetex clave milisegundos valor
```

Funciona igual que el anterior, pero el que especificamos son los milisegundos de existencia.

MGET

sintaxis

```
mget clave1 clau2 clauN
```

Vuelve una lista de valores, los de las claves indicadas.

ejemplo

```
127.0.0.1:6379> set más1 enero
OK
127.0.0.1:6379> set mes2 febrero
OK
127.0.0.1:6379> set mes3 marzo
OK
127.0.0.1:6379> mget más1 mes2 mes3
1) "Enero"
2) "febrero"
3) "mar \ xc3 \ xa7"
```

Nota

Recuerde que los caracteres como vocales acentuadas, ç, ñ, ... se han introducido bien, pero quizás no se visualizan bien. Se puede evitar entrando en el cliente de esta manera **redis-cli --raw**

Si alguna de las claves no éxito, volverá **nil** en su lugar

MSET

sintaxis

```
mset clave1 valor1 clau2 valor2 clauN valorN
```

Asigna los valores correspondientes a las claves. Es una operación atómica: se ponen (o cambian) todos los valores a la hora.

```
127.0.0.1:6379> mset mes4 abril mes5 mayo mes6 junio mes7 julio
OK
```

```
127.0.0.1:6379> mget más1 mes2 mes3 mes4 mes5 mes6 mes7
1) "Enero"
2) "febrero"
3) "mar \ xc3 \ xa7"
4 ) "abril"
5) "mayo"
6) "junio"
7) "julio"
```

Si no quiera reemplazar valores, podríamos utilizar el comando **MSETNX** , totalmente equivalente, pero deberíamos tener en cuenta que si alguna ya existe y por tanto no puede cambiar el valor, no haría la operación, es decir, tampoco crearía las otras.

APPEND

sintaxis

```
append clave1 valor1
```

Si la clave no existe la crea asignándole el valor (como el **SET**), pero si ya existe, concatena el valor al final de la cadena que ya había.

```
127.0.0.1:6379> append saludo Hola
(integer) 4
127.0.0.1:6379> get saludo
"Hola"
127.0.0.1:6379> append saludo ", como?"
(integer) 13
127.0.0.1:6379> get saludo
"Hola, cómo va?"
```

STRL

sintaxis

```
strlen clave1
```

Vuelve el número de caracteres que hay en el valor de la clave. Si la clave no existe, volverá 0. Si la clave es de otro tipo, volverá error.

```
127.0.0.1:6379> strlen saludo
(integer) 13
127.0.0.1:6379> strlen sal
(integer) 0
```

GETRANGE

sintaxis

```
getrange clave1 inicio final
```

Extrae una subcadena del valor de la clave (debe ser de tipo **String**) desde el número de carácter de inicio hasta el número del final (ambos incluidos). El primer carácter es el 0. Si ponemos de final un número mayor que el último, igual lo sacará hasta el final.

Se pueden poner también valor negativos que nos ayudan a coger la cadena desde el final. El -1 es el último carácter, el -2 el penúltimo, ... Y se pueden mezclar números positivos y negativos. Así, **rango 0 -1** es toda la cadena.

```
127.0.0.1:6379> getrange saludo 1 3
"ola"
127.0.0.1:6379> getrange saludo 6 50
"como va?"
127.0.0.1:6379> getrange saludo 6 -1
"como va?"
127.0.0.1:6379> getrange saludo -7 -5
"como"
127.0.0.1:6379> getrange saludo 0 -1
"Hola, cómo va?"
```

SETRANGE

sintaxis

```
setrange clave1 desplazamiento valor
```

Sustituye parte del valor de la cadena, a partir del desplazamiento, con el vaolr proporcionado. No se admiten en desplazamiento valores negativos.

```
127.0.0.1:6379> get saludo
"Hola, cómo va?"
127.0.0.1:6379> setrange saludo 4 ". C"
(integer) 13
127.0.0.1:6379> get saludo
"Hola. ¿Cómo?"
```


INCR

sintaxis

```
incr clavel
```

A pesar de que Redis guarda los strings como tales, como cadenas de caracteres, en algunas ocasiones es capaz de transformar la cadena a un número. Es el caso del comando **INCR**, que convierte la cadena en un entero (si puede) e incrementa este valor en una unidad.

Si la clave no existe la crea asumiendo que valía 0, y por tanto después valdrá 1.

Si el valor de la clave no era un número entero, dará un error.

```
127.0.0.1:6379> set compt1 20
OK
127.0.0.1:6379> get compt1
"20"
127.0.0.1:6379> incr compt1
(integer) 21
127.0.0.1:6379> get compt1
"21"
127.0.0.1:6379> incr compt2
(integer) 1
127.0.0.1:6379> get compt2
"1"
127.0.0.1:6379> incr clau_1
(error) ERR value is not an integer oro out of range
127.0.0.1:6379> set compt3 04:25
OK
127.0.0.1:6379> incr compt3
(error) ERR value is not an integer oro out of range
```

decrece

sintaxis

```
Decr clavel
```

Reducir en una unidad el valor de la clave (SENP que sea un entero).

Puede tomar valores negativos.

Si la clave no existe la crea asumiendo que valía 0, y por tanto después valdrá -1.

```
127.0.0.1:6379> Decr compt2
(integer) 0
127.0.0.1:6379> Decr compt2
(integer) -1
127.0.0.1:6379> get compt2
"-1"
```

INCRBY

sintaxis

```
incrby clavel incremento
```

Incrementa el valor de la clave en el número de unidades indicado en **incremento** (el valor debe ser entero). El incremento puede ser negativo.

```
127.0.0.1:6379> incrby compt1 10
(integer) 31
127.0.0.1:6379> incrby compt1 -20
(integer) 11
```

DECRBY

sintaxis

```
decrby clavel decremento
```

Reducir el valor de la clave el número de unidad indicado en **decremento**.

```
127.0.0.1:6379> decrby compt1 5
(integer) 6
```

Ahora vamos a ver comandos que nos permiten trabajar con las claves, para buscarlas, ver si existen, etc. No importará el tipo de las claves (de momento sólo hemos trabajado con claves de tipo **String**, pero si ya tuviéramos los otros tipos también se verían afectadas). En ningún caso de estos comandos accederemos al valor de las claves.

La lista de comandos de claves que veremos es:

keys, **exists**, **del**, **type**, **rename**, **renamenx**, **expire**, **pexpire**, **ttl**, **pttl**, **persist**

KEYS

sintaxis

keys patrón

Vuelve todas las claves que coinciden con el patrón. En el patrón podemos poner caracteres comodín:

- ***** : Equivale a 0 o más caracteres. Por ejemplo "Mar * a" podría volver "Mara", "María", "Marta", "Margarita", ...
- **?** : Equivale exactamente a un carácter. Por ejemplo "Mar? A" podría volver "María" o "Marta", pero no "Mara", "Margarita", ...
- **[ab]** : será cierto si en el lugar correspondiente hay uno de los caracteres especificados entre los corchetes. Por ejemplo "Mar [it] en" podría volver "María" o "Marta", pero no "Marga"

Para devolver todas las claves utilizaremos **keys ***

```
127.0.0.1:6379> keys *
1) "mes3"
2) "saludo"
3) "pi"
4) "clau_4"
5) "compt2"
6) "mes6"
7) "más1"
8) "mes4"
9) "clau_1"
10) "compt3"
11) "cuatro"
12) "mes7"
13) "mes2"
14) "mes5"
15) "compt1"
16) "clau_2"
17) "texto"
```

```
127.0.0.1:6379> keys mes?
1) "mes5"
2) "mes3"
3) "mes7"
4) "mes2"
5) "mes6"
6) "mes4"
7) "más1"
```

```
127.0.0.1:6379> keys c *
1) "clau_4"
2) "compt2"
3) "clau_1"
4) "compt3"
5) "compt1"
6) "clau_2"
```

```
127.0.0.1:6379> keys mes [125]
1) "mes5"
2) "mes2"
3) "más1"
```

Nota

El orden en el que se mostrarán las claves no tiene por qué ser igual para todos nosotros. El más normal es que cada uno de nosotros tenga un orden diferente. Esto tiene que ver con los conjuntos (**septiembre**), un tipo de datos que veremos más adelante.

EXISTS

sintaxis

exists clave

Vuelve 1 si la clave existe, y 0 si no existe. No importa de qué tipo sea la clave.

```
127.0.0.1:6379> exists clau_1
(integer) 1
127.0.0.1:6379> exists clau_25
(integer) 0
```

DEL

sintaxis

```
del clave1 clau2 clauN
```

Elimina la clave o claves especificadas. Si ponemos más de una clave y alguna no existe, la ignorará y sí que borrará las otras.

```
127.0.0.1:6379> del compt2
(integer) 1
127.0.0.1:6379> del mes6 mes7 mes8 mes9
(integer) 2
```

Obsérvese que nos indica cuántas claves ha borrado. En el primer ejemplo ha borrado la clave especificada, y en el segundo dice que ha borrado 2, que serán **mes6** y **mes7**, ya que **mes8** y **mes9** no existían.

TYPE

sintaxis

```
type clave
```

Vuelve el tipo de la clave especificada. Los valores posibles son:

- string
- hash
- list
- septiembre
- zset (conjunto ordenado)

ejemplos

```
127.0.0.1:6379> type clau_1
string
```

RENAME

sintaxis

```
rename clave novaclau
```

Cambia el nombre de la clave a la clave nueva, conservando el valor. Da error si la clave antigua no existe. Si la clave nueva ya existía reemplazará su valor.

ejemplos

```
127.0.0.1:6379> get saludo
"Hola. ¿Cómo?"
127.0.0.1:6379> rename saludo saludar
OK
127.0.0.1:6379> get saludo
(nil)
127.0.0.1:6379> get saludar
"Hola. ¿Cómo?"
127.0.0.1:6379> rename clau_22 clau_23
(error) ERR no such key
```

RENAMENX

sintaxis

```
renamenx clave novaclau
```

Igual que el anterior pero únicamente si la clave nueva no existía. Si ya existía no hace nada (volviendo 0 para indicarle h0).

ejemplos

```
127.0.0.1:6379> renamenx compt1 compt3
(integer) 0
127.0.0.1:6379> get compt1
"9"
```

Estamos suponiendo que la clave **compt3** ya existe

EXPIRE

sintaxis

```
expire clave según
```

Asigna como tiempo de expiración de la clave los segundos especificados. Si ya tenía tiempo de expiración, lo modifica poniéndole ese valor especificado.

En caso de que a una llave con tiempo de expiración le cambiamos el nombre con **RENAME**, continuará con tiempo de expiración que le quedaba.

PEXPIRE

sintaxis

```
pexpire clave milisegundos
```

Lo mismo pero en milisegundos

TTL

sintaxis

```
ttl clave
```

Vuelve el tiempo de vida (hasta la expiración) de una clave. Si la clave no tiene tiempo de expiración, devuelve -1.

ejemplos

```
127.0.0.1:6379> expire compt3 10
(integer) 1
127.0.0.1:6379> ttl compt3
(integer) 6
127.0.0.1:6379> ttl compt3
(integer) 3
127.0.0.1:6379> ttl compt3
(integer) 0
127.0.0.1:6379> get compt3
(nil)
```

PTTL

sintaxis

```
pttl clave
```

Al igual que el anterior, pero nos devuelve el tiempo en milisegundos.

persisten

sintaxis

```
persist clave
```

Elimina el tiempo de expiración de una clave, si es que tenía. Ahora la clave no expirará nunca.

ejemplos

```
127.0.0.1:6379> expire compt1 20
(integer) 1
127.0.0.1:6379> ttl compt1
(integer) 12
127.0.0.1:6379> ttl compt1
(integer) 7
127.0.0.1:6379> persist compt1
(integer) 1
127.0.0.1:6379> ttl compt1
(integer) -1
127.0.0.1:6379> get compt1
"9"
```

Ya habíamos comentado que el tipo **Hash** es una especie de registro, con subcampos (en realidad deberíamos decir sub-llaves). Puede tener cualquier número de subcampos que serán de tipo String.

Redis es muy eficiente en cuanto al espacio que ocupan los **Hash**, y sobre todo en el tiempo de recuperación de los datos.

Los comandos que vimos para el **String** no se pueden aplicar al **Hash**. Sin embargo los comandos del **Hash** son muy similares a aquellos, empezando siempre por **H**.

Veremos los siguientes comandos:

hset, **hget**, **hgetall**, **hdel**, **hkeys**, **Hvala**, **hmget**, **hmset**, **hexists**, **hsetnx**, **hincrby**

HSET

sintaxis

```
hset clave campo valor
```

Asigna al campo especificado de la clave especificada el valor especificado. Si el valor consta de más de una palabra, deberá ir entre comillas dobles.

Si la clave no existía, la creará, y si ya existía, sencillamente añadirá el campo. Y si de esta clave ya existía el campo, modificará su valor.

Evidentemente, en claves diferentes pueden haber campos con los mismos nombres.

ejemplos

```
127.0.0.1:6379> hset empleat_1 nombre Andreu
(integer) 1
127.0.0.1:6379> hset empleat_1 departamento 10
(integer) 1
127.0.0.1:6379> hset empleat_1 sueldo 1000.0
(integer) 1
```

```
127.0.0.1:6379> hset empleat_2 nombre Berta
(integer) 1
127.0.0.1:6379> hset empleat_2 sueldo 1500.0
(integer) 1
```

HGET

sintaxis

```
hget clave campo
```

Vuelve el valor del campo de la clave. Si no existía (el campo o la clave) vuelve **nil**. Sólo podemos especificar un campo.

ejemplos

```
127.0.0.1:6379> hget empleat_1 nombre
"Andrés"
127.0.0.1:6379> hget empleat_1 departamento
"10"
127.0.0.1:6379> hget empleat_2 nombre
"Berta"
127.0.0.1:6379> hget empleat_2 departamento
(nil)
```

HGETALL

sintaxis

```
hgetall clave
```

Devuelve una lista con todos los campos y sus valores de la clave. La secuencia es: campo1 valor1 campo2 valor2 ... Pero no nos podemos fiar de que el orden sea el mismo orden que cuando el definimos.

ejemplos

```
127.0.0.1:6379> hgetall empleat_1
1) "nombre"
2) "Andreu"
3) "departamento"
```

```
4) "10"  
5) "sueldo"  
6) "1000.0"
```

HDEL

sintaxis

```
hdel clave campo1 campo2 campN
```

Elimina el o los campos especificados. Si no existen alguno de ellos, sencillamente lo ignora y si que elimina los otros.

ejemplos

```
127.0.0.1:6379> hdel empleat_1 departamento  
(integer) 1  
127.0.0.1:6379> hgetall empleat_1  
1) "nombre"  
2) "Andreu"  
3) "sueldo"  
4) "1000.0"
```

HKEYS

sintaxis

```
hkeys clave
```

Devuelve una lista con los campos de la clave. Si la clave no existía, devuelve una lista vacía

ejemplos

```
127.0.0.1:6379> hkeys empleat_1  
1) "nombre"  
2) "sueldo"
```

HVALA

sintaxis

```
Hvala clave
```

Devuelve una lista con los valores (únicamente los valores) de todos los campos de la clave. Si la clave no existía, devuelve una lista vacía

ejemplos

```
127.0.0.1:6379> hvala empleat_1  
1) "Andreu"  
2) "1000.0"
```

otros Comandos

También existen otros comandos, de funcionamiento como cabría esperar (los hemos visto todos en el caso de **String**):

- **hmget** : Vuelve más de un campo de la clave
- **hmset** : asigna más de un campo a una clave
- **hexists** : indica si existe el subcampo de la clave
- **hsetnx** : asigna únicamente en caso de que no exista el campo.
- **hincrby** : incrementa el campo de la clave

Las **Listas** en **Redis** son listas de Strings ordenadas, donde cada elemento está asociado a un índice de la lista. Se pueden recuperar los elementos tanto de forma ordenada (por el índice) como accediendo directamente a una posición.

Los elementos se pueden añadir al principio, al final o también en una posición determinada.

La lista se crea en el momento en que se inserta el primer elemento, y desaparece cuando quitamos el último elemento que quede.

Están muy bien optimizadas para la inserción y para la consulta.

Los comandos que afectan a las listas empiezan casi todos por **L**, excepto algunos que empiezan por **R** indicando que hacen la operación por la derecha.

Por cierto, los valores de los elementos se pueden repetir.

La lista de comandos que afectan a listas que veremos es:

lpush, **rpush**, **lpop**, **rpop**, **lset**, **índice**, **linsert**, **lrange**, **len**, **LRM**, **LTrim**

LPUSH

sintaxis

```
lpush clave valor1 valor2 valorN
```

Introduce los valores en la lista (creando la clave si es necesario). Los inserta en la primera posición, o también podríamos decir que para la izquierda (**Left PUSH**), imaginando que los elementos están ordenados de izquierda a derecha. Si ponemos más de un valor, se irán introduciendo siempre en la primera posición. El comando devolverá el número de elementos (strings) de la lista después de la inserción.

ejemplos

```
127.0.0.1:6379> lpush lista1 primera segunda tercera
(integer) 3
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "tercera"
2) "segunda"
3) "primera"
```

```
127.0.0.1:6379> lpush lista1 cuarta quinta
(integer) 5
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "quinta"
2) "cuarta"
3) "tercera"
4) "segunda"
5) "primera"
```

Nota

Para ver el contenido de la lista utilizaremos el comando **lrange lista 0 -1**, que devuelve la lista entera. Veremos de forma más completa este comando con posterioridad.

RPUSH

sintaxis

```
rpush clave valor1 valor2 valorN
```

Introduce los valores en la lista (creando la clave si es necesario). Los inserta en la última posición, o también podríamos decir que por la derecha (**Right PUSH**), imaginando que los elementos están ordenados de izquierda a derecha. El comando devolverá el número de elementos (strings) de la lista después de la inserción.

ejemplos

```
127.0.0.1:6379> rpush lista1 sexta séptima
(integer) 7
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "quinta"
2) "cuarta"
3) "tercera"
4) "segunda"
```

```
5) "primera"
6) "sexta"
7) "séptima"
```

LPOP

sintaxis

```
lpop clave
```

Vuelve y elimina el primer elemento (el de más a la izquierda).

ejemplos

```
127.0.0.1:6379> lpop lista1
"quinta"
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "tercera"
3) "segunda"
4) "primera"
5) "sexta"
6) "séptima"
```

RPOP

sintaxis

```
rpop clave
```

Vuelve y elimina el último elemento (el de más a la derecha).

ejemplos

```
127.0.0.1:6379> rpop lista1
"séptima"
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "tercera"
3) "segunda"
4) "primera"
5) "sexta"
```

LSET

sintaxis

```
lset clave índice valor
```

Sustituye el valor de la posición indicada por el índice. Tanto la clave como el elemento de la posición indicada deben existir, sino dará error. Ahora la **L** no significa **Left** sino **List**.

La primera posición es la 0. Y también se pueden poner números negativos: -1 es el último, -2 el penúltimo, ...

ejemplos

```
127.0.0.1:6379> lset lista1 2 cuarta
OK
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "tercera"
3) "cuarta"
4) "primera"
5) "sexta"
```

```
127.0.0.1:6379> lset lista1 -1 quinta
OK
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "tercera"
3) "cuarta"
4) "primera"
5) "quinta"
```

Observe cómo se pueden repetir los valores

índice

sintaxis

```
El índice clave índice
```

Vuelve el elemento situado en la posición indicada por el índice, **pero sin eliminarlo de la lista** .

ejemplos

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "tercera"
3) "cuarta"
4) "primera"
5) "quinta"
```

```
127.0.0.1:6379> índice lista1 0
"cuarta"
```

```
127.0.0.1:6379> índice lista1 3
"primera"
```

```
127.0.0.1:6379> índice lista1 -1
"quinta"
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "tercera"
3) "cuarta"
4) "primera"
5) "quinta"
```

LINSERT

sintaxis

```
linsert clave BEFORE | AFTER valor1 valor2
```

Insertar el valor2 antes o después (según lo que elegimos) de la primera vez que encuentra el valor1. No sustituye, sino que inserta en una determinada posición. Los elementos que van después del elemento introducido verán actualizado su índice.

ejemplos

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "tercera"
3) "cuarta"
4) "primera"
5) "quinta"
```

```
127.0.0.1:6379> linsert lista1 AFTER cuarta segunda
(integer) 6
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "segunda"
3) "tercera"
4) "cuarta"
5) "primera"
6) "quinta"
```

```
127.0.0.1:6379> linsert lista1 BEFORE quinta sexta
(integer) 7
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "segunda"
3) "tercera"
4) "cuarta"
5) "primera"
6) "sexta"
7) "quinta"
```

Si intentamos insertar antes o después un elemento que no existe, volverá -1 indicando que no lo ha encontrado y no hará la inserción.

```
127.0.0.1:6379> linsert lista1 BEFORE décima séptima
(integer) -1
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "segunda"
3) "tercera"
4) "cuarta"
5) "primera"
6) "sexta"
7) "quinta"
```

LRange

sintaxis

```
lrange clave inicio final
```

Vuelve los elementos de la lista que hay entre los índices inicio y final, ambos inclusive. El primer elemento es el 0. Se pueden poner valores negativos, siendo -1 el último, -2 el penúltimo, ...

ejemplos

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "segunda"
3) "tercera"
4) "cuarta"
5) "primera"
6) "sexta"
7) "quinta"
```

```
127.0.0.1:6379> lrange lista1 2 4
1) "tercera"
2) "cuarta"
3) "primera"
```

```
127.0.0.1:6379> lrange lista1 1 -2
1) "segunda"
2) "tercera"
3) "cuarta"
4) "primera"
5) "sexta"
```

```
127.0.0.1:6379> lrange lista1 4 4
1) "primera"
```

LEN

sintaxis

```
len clave
```

Vuelve el número de elementos de la lista

ejemplos

```
127.0.0.1:6379> len lista1
(integer) 7
```

LREM

sintaxis

```
LREM clave número valor
```

Elimina elementos de la lista que coincidan con el valor proporcionado. Ya sabemos que los valores se pueden repetir. Con el número indicamos cuántos elementos queremos que se borran: si ponemos 1 borrará el primer elemento con este valor, si ponemos 2 borrarán los dos primeros elementos (los de más a la izquierda) que tenga este valor. Si ponemos 0 borrarán todos los elementos con este valor

ejemplos

```
127.0.0.1:6379> rpush lista1 segunda
(integer) 8
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "segunda"
3) "tercera"
4) "cuarta"
5) "primera"
6) "sexta"
7) "quinta"
8) "segunda "
```

```
127.0.0.1:6379> LREM lista1 1 segunda
(integer) 1
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "cuarta"
2) "tercera"
3) "cuarta"
4) "primera"
5) "sexta"
6) "quinta"
7) "segunda"
```

```
127.0.0.1:6379> LREM lista1 0 cuarta
(integer) 2
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "tercera"
2) "primera"
3) "sexta"
4) "quinta"
5) "segunda"
```

LTrim

sintaxis

```
LTrim clave inicio final
```

Eliminar elementos que quedan fuera de los índices inicio y final, es decir elimina los que estén a la izquierda de inicio, y los que estén a la derecha de final.

ejemplos

```
127.0.0.1:6379> LTrim lista1 1 -2
OK
```

```
127.0.0.1:6379> lrange lista1 0 -1
1) "primera"
2) "sexta"
3) "quinta"
```

Los **Sets** de **Redis** son conjuntos de valores de tipo String no ordenados. Podremos añadir, actualizar y borrar estos elementos de forma cómoda y eficiente. No se permitirán los valores duplicados.

Además **Redis** nos ofrece operaciones interesantes como la unión, intersección y diferencia de conjuntos.

Como siempre, los comandos son específicos, es decir no nos valen los de Strings, Hash o List. Todos los comandos empiezan por **S**.

Esta es la lista de los que veremos:

Sadd , **smembers** , **sismember** , **Scardem** , **Srem** , **SPOp** , **srandmember** , **Sunion** , **sunionstore** , **sdiff** , **sdiffstore** , **Sinter** , **sinterstore** , **smove**

SADD

sintaxis

```
Sadd clave valor1 valor2 valorN
```

Agregar los valores al conjunto (creando la clave si es necesario). Recordemos que el orden no es importante, y que no se pueden repetir los valores; si intentamos introducir un repetido, no dará error, pero no lo introducirá. El comando devolverá el número de elementos que realmente se han añadido.

ejemplos

```
127.0.0.1:6379> Sadd colores rojo verde azul
(integer) 3
```

```
127.0.0.1:6379> Sadd colores verde amarillo
(integer) 1
```

SMEMBERS

sintaxis

```
smembers clave
```

Vuelve todos los valores del conjunto. Si la clave no existe volverá un conjunto vacío. Recuerde que el orden de los elementos no es predecible

ejemplos

```
127.0.0.1:6379> smembers colores
1) "amarillo"
2) "verde"
3) "rojo"
4) "azul"
```

SISMEMBER

sintaxis

```
sismember clave valor
```

Comprueba si el valor está en el conjunto, volviendo 1 en caso afirmativo y 0 en caso negativo.

ejemplos

```
127.0.0.1:6379> sismember colores verde
(integer) 1
```

```
127.0.0.1:6379> sismember colores negro
(integer) 0
```

Scardem

sintaxis

```
sardo clave
```

Vuelve la cardinalidad, es decir, el número de elementos del conjunto en la actualidad.

ejemplos

```
127.0.0.1:6379> Scardem colores  
(integer) 4
```

Srem

sintaxis

```
Srem clave valor1 valor2 valorN
```

Elimina los valores del conjunto. Si el conjunto se queda vacío, eliminará la clave también. Si alguno de los valor no es ningún elemento del conjunto, sencillamente ignorará. El comando devuelve el número de elementos realmente eliminado.

ejemplos

```
127.0.0.1:6379> Srem colores verde negro  
(integer) 1
```

```
127.0.0.1:6379> smembers colores  
1) "amarillo"  
2) "rojo"  
3) "azul"
```

SPOp

sintaxis

```
SPOp clave
```

Vuelve y elimina un valor **aleatorio** del conjunto. Recuerde que además de devolverlo, lo elimina del conjunto.

ejemplos

```
127.0.0.1:6379> smembers colores  
1) "amarillo"  
2) "rojo"  
3) "azul"
```

```
127.0.0.1:6379> SPOp colores  
"amarillo"
```

```
127.0.0.1:6379> smembers colores  
1) "rojo"  
2) "azul"
```

SRANDMEMBER

sintaxis

```
randmember clave
```

Muy parecido al anterior. Vuelve un valor aleatorio del conjunto, pero en esta ocasión no lo elimina del conjunto.

ejemplos

```
127.0.0.1:6379> srndmember colores  
"azul"
```

```
127.0.0.1:6379> smembers colores  
1) "rojo"  
2) "azul"
```

Súion

sintaxis

```
Sunion clave1 clau2 clauN
```

Vuelve la unión de los elementos de los conjuntos especificados. Es una unión correcta, es decir, no se repetirá ningún valor.

No modifica ningún conjunto, y el resultado únicamente se vuelve, no se guarda en ningún lugar de forma permanente.

ejemplos

```
127.0.0.1:6379> smembers colores
1) "rojo"
2) "azul"
```

```
127.0.0.1:6379> Sadd colors1 verde rojo amarillo
(integer) 3
```

```
127.0.0.1:6379> smembers colors1
1) "amarillo"
2) "rojo"
3) "verde"
```

```
127.0.0.1:6379> Sunion colores colors1
1) "verde"
2) "amarillo"
3) "rojo"
4) "azul"
```

SUNIONSTORE

sintaxis

```
sunionstore clau_destí clavel clau2 clauN
```

Al igual que el anterior, pero ahora sí que se guarda el resultado de la unión en un conjunto, **clau_destí** (el primer especificado). Si la clau_destí ya existía, sustituirá el contenido.

ejemplos

```
127.0.0.1:6379> smembers colores
1) "rojo"
2) "azul"
```

```
127.0.0.1:6379> smembers colors1
1) "amarillo"
2) "rojo"
3) "verde"
```

```
127.0.0.1:6379> sunionstore colors2 colores colors1
(integer) 4
```

```
127.0.0.1:6379> smembers colors2
1) "verde"
2) "amarillo"
3) "rojo"
4) "azul"
```

SDIFF

sintaxis

```
sdiff clavel clau2 clauN
```

Vuelve la diferencia de los elementos del primer conjunto respecto de la unión de todos los demás. Es decir, vuelve los elementos del primer conjunto que no pertenecen a ninguno de los otros.

No modifica ningún conjunto, y el resultado únicamente se vuelve, no se guarda en ningún lugar de forma permanente.

ejemplos

```
127.0.0.1:6379> smembers colores
1) "rojo"
2) "azul"
```

```
127.0.0.1:6379> smembers colors1
1) "amarillo"
2) "rojo"
3) "verde"
```

```
127.0.0.1:6379> sdiff colors1 colores
1) "verde"
2) "amarillo"
```

SDIFFSTORE

sintaxis

```
sdiffstore clau_destí clavel clau2 clauN
```

Al igual que el anterior, pero ahora sí que se guarda el resultado de la diferencia en un conjunto, **clau_destí** (el primer especificado). Si la clau_destí ya existía, sustituirá el contenido.

ejemplos

```
127.0.0.1:6379> smembers colores
1) "rojo"
2) "azul"
```

```
127.0.0.1:6379> smembers colors1
1) "amarillo"
2) "rojo"
3) "verde"
```

```
127.0.0.1:6379> sdiffstore colors3 colors1 colores
(integer) 2
```

```
127.0.0.1:6379> smembers colors3
1) "verde"
2) "amarillo"
```

SINTER

sintaxis

```
Sinter clau1 clau2 clauN
```

Vuelve la intersección de los elementos de los conjuntos. Es decir, vuelve los elementos que pertenecen a todos los conjuntos especificados.

No modifica ningún conjunto, y el resultado únicamente se vuelve, no se guarda en ningún lugar de forma permanente.

ejemplos

```
127.0.0.1:6379> smembers colores
1) "rojo"
2) "azul"
```

```
127.0.0.1:6379> smembers colors1
1) "amarillo"
2) "rojo"
3) "verde"
```

```
127.0.0.1:6379> Sinter colores colors1
1) "rojo"
```

SINTERSTORE

sintaxis

```
sinterstore clau_destí clau1 clau2 clauN
```

Al igual que el anterior, pero ahora sí que se guarda el resultado de la intersección en un conjunto, **clau_destí** (el primer especificado). Si la clau_destí ya existía, sustituirá el contenido.

ejemplos

```
127.0.0.1:6379> smembers colores
1) "rojo"
2) "azul"
```

```
127.0.0.1:6379> smembers colors1
1) "amarillo"
2) "rojo"
3) "verde"
```

```
127.0.0.1:6379> sinterstore colors4 colores colors1
(integer) 1
```

```
127.0.0.1:6379> smembers colors4
1) "rojo"
```

SMOVE

sintaxis

```
smove clau_font clau_destí valor
```

Menea el valor del conjunto de origen (el primer conjunto) al conjunto destino (el segundo). Esto supondrá eliminarlo del primero y añadirlo al segundo. Volverá 1 si la ha removido. También, y 0 si no lo ha removido. También.

ejemplos

```
127.0.0.1:6379> smembers colores
1) "rojo"
```

```
2) "azul"
```

```
127.0.0.1:6379> smembers colors1
```

```
1) "amarillo"
```

```
2) "rojo"
```

```
3) "verde"
```

```
127.0.0.1:6379> smove colors1 colores verde  
(integer) 1
```

```
127.0.0.1:6379> smembers colores
```

```
1) "verde"
```

```
2) "rojo"
```

```
3) "azul"
```

```
127.0.0.1:6379> smembers colors1
```

```
1) "amarillo"
```

```
2) "rojo"
```


2.3.6 - Siete ordenado

Los **Sets Ordenados** (**sorted Set**) de **Redis** son Sets que además de guardar los valores, guardan también una **puntuación** (**score**) para cada valor, y **Redis** mantendrá el conjunto **ordenado** por esta puntuación.

Los valores no se podrán repetir, pero sí las puntuaciones.

Algunos de los comandos serán iguales que los del **Siete** , ya que un conjunto ordenado no deja de ser un conjunto, pero con la información de la puntuación (y que no se pueden repetir los valores). En esta ocasión comenzarán por **Z** .

Y esta es la lista de los que veremos:

zadd , **zcard** , **zscore** , **zcount** , **zrange** , **zrangebyscore** , **zrank** , **zrem** , **zremrangebyscore** , **zincrby**

ZADD

sintaxis

```
zadd clave puntuación1 valor1 puntuación2 valor2 puntuaciones valorN
```

Agregar los valores al conjunto (creando la clave si es necesario) con las puntuaciones correspondientes. Las puntuaciones serán Strings de valores reales (float). No se pueden repetir los valores, pero sí las puntuaciones. Si intentamos introducir un valor repetido, lo que hará será actualizar la puntuación. El comando devolverá el número de elementos que realmente se han añadido.

ejemplos

```
127.0.0.1:6379> zadd puntuaciones 1 Nombre1 2 nombre2 5 Nom3 4 Nom4
(integer) 4
```

```
127.0.0.1:6379> zrange puntuaciones 0 -1
1) "Nombre1"
2) "nombre2"
3) "Nom4"
4) "Nom3"
```

ZCARD

sintaxis

```
zcard clave
```

Vuelve la cardinalidad, es decir, el número de elementos del conjunto ordenado en la actualidad.

ejemplos

```
127.0.0.1:6379> zcard puntuaciones
(integer) 4
```

ZSCORE

sintaxis

```
zscore clave valor
```

Vuelve la puntuación (score) del valor especificado del conjunto ordenado. Si no existe el valor o no existiera la llave, vuelve nil.

ejemplos

```
127.0.0.1:6379> zscore puntuaciones Nom3
"5"
```

```
127.0.0.1:6379> zscore puntuaciones Nom7
(nil)
```

ZCOUNT

sintaxis

```
zcount clave min max
```

Vuelve el número de valores que están entre las puntuaciones especificadas (ambas incluidas).

ejemplos

```
127.0.0.1:6379> zcount puntuaciones 2 5
(integer) 3
```

ZRANGE

sintaxis

```
zrange clave inicio final [withscores]
```

Vuelve los elementos del conjunto ordenado que existe entre los índices inicio y final, ambos inclusive. Y se sacan por orden Ascent de puntuación. El primer elemento es el 0. Se pueden poner valores negativos, siendo -1 el último, -2 el penúltimo, ... Opcionalmente podemos poner **WITHSCORES** para que nos devuelva también la puntuación de cada elemento

ejemplos

```
127.0.0.1:6379> zrange puntuaciones 0 -1
1) "Nombre1"
2) "nombre2"
3) "Nom4"
4) "Nom3"
```

```
127.0.0.1:6379> zrange puntuaciones 0 -1 withscores
1) "Nombre1"
2) "1"
3) "nombre2"
4) "2"
5) "Nom4"
6) "4"
7) "Nom3"
8) " 5 "
```

Si quiera sacar el conjunto en orden inverso de puntuación, utilizaríamos el comando **ZREVRANGE** (**reverse range**).

```
127.0.0.1:6379> zrevrange puntuaciones 0 -1 withscores
1) "Nom3"
2) "5"
3) "Nom4"
4) "4"
5) "nombre2"
6) "2"
7) "Nombre1"
8) " 1 "
```

ZRANGEBYSCORE

sintaxis

```
zrangebyscore clave min max [withscores]
```

Vuelve los elementos del conjunto ordenado que tienen una puntuación comprendida entre **min** y **max** (ambas incluidas). Y se sacan por orden Ascent de puntuación. Opcionalmente podemos poner **WITHSCORES** para que nos devuelva también la puntuación de cada elemento.

ejemplos

```
127.0.0.1:6379> zrangebyscore puntuaciones 2 5
1) "nombre2"
2) "Nom4"
3) "Nom3"
```

```
127.0.0.1:6379> zrangebyscore puntuaciones 2 5 withscores
1) "nombre2"
2) "2"
3) "Nom4"
4) "4"
5) "Nom3"
6) "5"
```

Si quiere que las puntuaciones fueron estrictamente mayores que la puntuación mínima y / o estrictamente menor que la puntuación máxima, pondríamos un **paréntesis** ante **min** y / o **max** :

```
127.0.0.1:6379> zrangebyscore puntuaciones 2 (5 withscores
1) "nombre2"
2) "2"
3) "Nom4"
4) "4"
```

Y si quiere sacar el conjunto en orden inverso de puntuación, utilizaríamos el comando **ZREVRANGEBYSCORE** (**reverse range**). Cuidad que como en orden inverso, por ejemplo el valor máximo debe ser el primero, y el mínimo el segundo.

```
127.0.0.1:6379> zrevrangebyscore puntuaciones 5 2 withscores
1) "Nom3"
```

```
2) "5"
3) "Nom4"
4) "4"
5) "nombre2"
6) "2"
```

ZRANK

sintaxis

```
zrank clave valor
```

Vuelve el número de orden del elemento con el valor especificado. El primer valor es el 0. Si no existe, vuelve **nil**.

ejemplos

```
127.0.0.1:6379> zrank puntuaciones Nombre1
(integer) 0
```

```
127.0.0.1:6379> zrank puntuaciones Nom4
(integer) 2
```

```
127.0.0.1:6379> zrank puntuaciones Nom7
(nil)
```

Si queremos saber el número de orden pero desde el final de la lista (en orden inverso), debemos utilizar **ZREVRANK**:

```
127.0.0.1:6379> zrevrank puntuaciones Nombre1
(integer) 3
```

```
127.0.0.1:6379> zrevrank puntuaciones Nom4
(integer) 1
```

ZREM

sintaxis

```
zrem clave valor1 valor2 valorN
```

Eliminar elementos con los valores especificados. Si algún valor no existe, sencillamente lo ignora. Vuelve el número de elementos realmente eliminados.

ejemplos

```
127.0.0.1:6379> zrem puntuaciones Nombre1
(integer) 1
```

```
127.0.0.1:6379> zrange puntuaciones 0 -1 withscores
1) "nombre2"
2) "2"
3) "Nom4"
4) "4"
5) "Nom3"
6) "5"
```

ZREMRANGEBYSCORE

sintaxis

```
zremrangebyscore clave min max
```

Eliminar elementos con puntuación comprendida entre el mínimo y el máximo de forma inclusiva. Si queremos hacerlo de forma exclusiva (sin incluir las puntuaciones de los extremos) pondremos un paréntesis frente al mínimo y / o el máximo. Vuelve el número de elementos realmente eliminados.

ejemplos

```
127.0.0.1:6379> zremrangebyscore puntuaciones (2 4
(integer) 1
```

```
127.0.0.1:6379> zrange puntuaciones 0 -1 withscores
1) "nombre2"
2) "2"
3) "Nom3"
4) "5"
```

ZINCRBY

sintaxis

zincrby clave incremento valor

Incrementa la puntuación del elemento especificado. El valor de la puntuación a incrementar es un número real. Vuelve el valor la puntuación final del elemento. Si el elemento no existía, la insertará, asumiendo una puntuación inicial de 0.

ejemplos

```
127.0.0.1:6379> zincrby puntuaciones 1.5 nombre2  
"3.5"
```

```
127.0.0.1:6379> zincrby puntuaciones 2.75 Nom5  
"2.75"
```

```
127.0.0.1:6379> zrange puntuaciones 0 -1 withscores  
1) "Nom5"  
2) "2.75"  
3) "nombre2"  
4) "3.5"  
5) "Nom3"  
6) "5"
```

La utilización desde Java es muy sencilla, y podremos utilizar todos los comandos que hemos visto.

Tendremos que incorporar en el proyecto una librería. Podemos utilizar por ejemplo la de **Jedis** (acrónimo de **J**ava **R**edis). En la siguiente dirección se puede encontrar (no es la última versión, pero nos vale):

<http://search.maven.org/remotecontent?filepath=redis/clients/jedis/2.4.2/jedis-2.4.2.jar>

Para hacer pruebas, creamos un nuevo proyecto llamado **Tema8** , e incorporamos la librería de **Jedis** . Para tenerlo un poco ordenado, creamos un paquete llamado **pruebas** .

conexión

La conexión es tan sencilla como el siguiente:

```
Jedis cono = new Jedis ( "localhost");
con.connect ();
```

Es decir, obtenemos un objeto **Jedis** pasándole al constructor la dirección del servidor, y luego conectamos con el método **connect** .

Si el servidor no lo tenemos en la misma máquina, sólo tendremos que sustituir **localhost** por la dirección donde esté el servidor.

El objeto Jedis representará una conexión con el servidor **Redis** . Todos los comandos que hemos visto para utilizar desde el cliente de Redis, serán métodos de este objeto. Sólo tendremos que tener cuidado con lo que nos devuelve el servidor cuando hacemos una petición. Muchas veces será un **String** , pero en muchas otras ocasiones serán colecciones: sets, lists, ...

Para cerrar la conexión:

```
con.close ();
```

Comandos que vuelven Strings

Todos los comandos que vimos serán ahora métodos del objeto **Jedis** . En concreto obtener el valor de una clave (comando **get clave**) será el método **get** al que le pasaremos la clave como parámetro:

Aquí tenemos ya la primera prueba:

```
importe redis.clients.jedis.Jedis;

public class Prova1 {

    public static void main (String [] args) {
        Jedis cono = new Jedis ( "localhost");
        con.connect ();

        System.out.println (con.get ( "saludo"));

        con.close ();
    }
}
```

Si queremos guardar una llave con un valor, utilizaremos el método **set (clave, valor)** , al que como vemos le debemos pasar los dos parámetros:

```
importe redis.clients.jedis.Jedis;

public class prueba2 {

    public static void main (String [] args) {
        Jedis cono = new Jedis ( "localhost");
        con.connect ();

        String valor = "Esta clave es una clave creada desde Java";
        con.set ( "clau_Java", valor);

        System.out.println (con.get ( "clau_Java"));
        con.close ();
    }
}
```

Comandos que vuelven conjuntos

En muchas ocasiones, lo que nos volverá **Redis** no es un único String, sino un conjunto de Strings. Esto será cierto cuando trabajamos con **Sets** , pero también en muchas otras ocasiones. Por ejemplo cuando pedimos unas cuantas claves con **MGET** , o cuando utilizamos **KEYS**

para que nos devuelva las llaves que coinciden con el patrón.

En algunas ocasiones tendremos que recoger con un objeto **Set** , cuando no importe el orden. En otras ocasiones con un objeto **List** cuando este orden sí que importe.

Por ejemplo, si queremos obtener con **MGET** los valores de varias llaves, sí importa el orden (el primer valor es la primera clave, el segundo de la segunda). Entonces lo obtendremos en un **List** :

```
importe java.util.List;
importe redis.clients.jedis.Jedis;

public class Prova3 {

    public static void main (String [] args) {
        Jedis cono = new Jedis ( "localhost");
        cono.connect ();

        List <String> c = cono.mget ( "más1", "mes2", "mes3");
        for (String s: c)
            System.out.println (s);

        cono.close ();
    }
}
```

Pero en cambio si queremos obtener todas las claves utilizaremos el método **keys** pasándole el patrón como parámetro. El orden no importa, y además no lo podemos predecir. Por lo tanto tenemos que recoger el resultado con un **Set** :

```
importe java.util.Set;
importe redis.clients.jedis.Jedis;

public class Prova4 {

    public static void main (String [] args) {
        Jedis cono = new Jedis ( "localhost");
        cono.connect ();

        Siete <String> c = cono.keys ( "*");
        for (String s: c)
            System.out.println (s);

        cono.close ();
    }
}
```

Y evidentemente también será el caso cuando accedemos a los tipos de datos **List** , **Set** y seguramente también **Hash** .

Para acceder a todo el contenido de un **List** utilizamos por ejemplo **lrange 0 -1** . Si utilizamos este método de **Jedis** nos volverá un **List** (de Java):

```
importe java.util.List;
importe redis.clients.jedis.Jedis;

public class Prova5 {

    public static void main (String [] args) {
        Jedis cono = new Jedis ( "localhost");
        cono.connect ();

        List <String> ll = cono.lrange ( "lista1", 0, -1);
        for (String e: ll)
            System.out.println (e);

        cono.close ();
    }
}
```

Si es un **Set** accederemos con el método **smembers** que volverá un **Set** (de Java):

```
importe java.util.Set;
importe redis.clients.jedis.Jedis;

public class Prova6 {

    public static void main (String [] args) {
        Jedis cono = new Jedis ( "localhost");
        cono.connect ();

        Siete <String> s = cono.smembers ( "colores");
        for (String e: s)
            System.out.println (e);

        cono.close ();
    }
}
```

Y en el caso de **Hash** , con el método **hkeys** podemos obtener todos los campos (subclaves), ya partir de ellos sus valores. Lo que vuelve **hkeys** es un **Set** (de Java).

```

importe java.util.Set;
importe redis.clients.jedis.Jedis;

public class Prova7 {

    public static void main (String [] args) {
        Jedis cono = new Jedis ( "localhost");
        cono.connect ();

        Siete <String> subcampos = cono.hkeys ( "empleat_1");
        for (String subcampo: subcampos)
            System.out.println (subcampo + ":" + cono.hget ( "empleat_1", subcampo));

        cono.close ();
    }
}

```

Tratamiento de los conjuntos ordenados

Los conjuntos ordenados (**sorted Sets**) tienen métodos específicos, al igual que todos los tipos. Algunos de estos métodos vuelven Strings, y un otros conjuntos (**Set** de **Java**). No hay problema con estos tipos, que ya los hemos tratado.

Pero hay unos métodos que vuelven listas **con más de un valor para cada elemento** . Es el caso de los métodos **ZRANGE** (con las variantes **ZREVRANGE** , **ZRANGEBYSCORE** y **ZREVRANGEBYSCORE**), que tienen la posibilidad de llevar el parámetro **WITHSCORES** . En este caso cada elemento constará del valor y de la puntuación. Lo que volverán los métodos es un **Set** de **Tuplas** , objeto proporcionado por **Jedis** , que dispondrá de los métodos **getElement ()** para el valor y **getScore ()** para la puntuación.

```

importe java.util.Set;
importe redis.clients.jedis.Jedis;
importe redis.clients.jedis.Tuple;

public class Prova8 {

    public static void main (String [] args) {
        Jedis cono = new Jedis ( "localhost");
        cono.connect ();

        Siete <tuple> conjOrd = cono.zrangeWithScores ( "puntuaciones", 0, -1);
        for (tuple t: conjOrd)
            System.out.println (t.getElement () + "--->" + t.getScore ());

        cono.close ();
    }
}

```

Seguramente **MongoDB** es el más famoso de los Sistemas Gestores de Bases de Datos **NoSQL** .

El nombre de **MongoDB** proviene de la palabra inglesa *hu **mongo** os* , que significa enorme, que es el propósito de esta Base de Datos: guardar grandes cantidades de información. Es de código abierto y está programada en C ++. El creó la empresa **10gen** (actualmente **MongoDB Inc.**)

Es una SGBD **Documental** , es decir, que servirá para guardar documentos. La manera interna de guardarlos es en formato **BSON** (Binary JSON) que en esencia es una variante del JSON para poder guardar físicamente los datos de una manera más eficiente.

En un servidor Mongo pueden haber más de una Base de Datos (aunque nosotros sólo gasta una: **test**).

- En cada Base de Datos la información se guardará en **colecciones** .
- Cada colección constará de unos cuantos **documentos** .
- Y cada documento serán una serie de datos guardados en forma de **clave-valor** , de los tipos soportados por MongoDB, y con el formato JSON (en realidad BSON)

Por tanto, en Mongo no hay mesas. Miremos unos ejemplos de documentos JSON para guardar la información de libros y autores.

Dependen de cómo se vaya a acceder a la información nos podemos plantear guardar los libros con sus autores, o guardar los autores, con sus libros. Incluso nos podríamos guardar los dos, para poder acceder de todas las maneras, aunque es a costa de doblar la información.

De la primera manera, guardando los libros con su autor, podríamos tener documentos con esta estructura, que se podrían guardar en una colección llamada **Libros** :

```
{
  _id: 101,
  titulo: "El secreto de Khadrell",
  autor: {
    nombre: "Pep",
    apellidos: "Castellano Puchol",
    any_naixement: 1960
  },
  ISBN: "84-95620-72-3"
},
{
  _id : 102,
  titulo: "La sombra del viento",
  autor: {
    nombre: "Carlos",
    apellidos: "Ruiz Zafon",
    país: "España"
  },
  páginas 490,
  editorial: "Planeta"
}
```

Observe cómo los objetos no tienen por qué tener la misma estructura. La manera de acceder al nombre de un autor sería esta:

objecte.autor.nom

Una manera alternativa de guardar la información, como habíamos comentado antes sería organizar por autores, con sus libros. De esta manera podríamos ir llenando la colección **Autores** con uno o más documentos de este estilo:

```
{
  _id: 201,
  nombre: "Pep",
  apellidos: "Castellano Puchol",
  any_naixement: 1960,
  libros: [
    {
      titulo: "El secreto de Khadrell",
      ISBN: "84-95620-72-3"
    },
    {
      titulo : "Habitación 502",
      editorial: "Tabarca"
    }
  ]
},
{
  _id: 202,
  nombre: "Carlos",
  apellidos: "Ruiz Zafon",
  país: "España",
  libros: [
    {
      titulo: "La Sombra del viento ",
      páginas: 490,
      editorial: "Planeta"
    }
  ]
}
```


Observe como para un autor, ahora tenemos un array (los corchetes: `[]`) con sus libros.

Cuál de las dos maneras es mejor para guardar la información? Pues depende del acceso que se tenga que hacer a los datos. La mejor será seguramente aquella que dependiendo de los accesos que se tengan que hacer, devuelva la información de forma más rápida.

3.1 - Instalación de MongoDB

Podremos instalar MongoDB en cualquier plataforma. E incluso sin tener permisos de administrador, como veremos en el caso de Ubuntu.

Instalación en Linux

Para poder hacer la instalación más básica, podremos hacerlo sin permisos de administrador. Si los tenemos todo es más cómodo, pero si no tenemos también lo podemos hacer, como veremos y remarcaremos a continuación.

De la página de **MongoDB** (<https://mongodb.org>) nos bajamos la versión apropiada para nuestro Sistema Operativo. Obsérvese como en el caso de Linux hay muchas versiones. En el caso de **Ubuntu 14.04 de 64 bits** , este archivo es: **MongoDB-linux-x86_64-ubuntu1404-3.2.1.tgz** . Pero recuerde que debe asegurarse de la versión

Sencillamente descomprimir este archivo donde queramos, y ya estará hecha la instalación básica.

Por defecto el directorio de la Base de Datos es **/ data / db**

El único problema que podríamos tener si no somos administradores es que no tengamos permiso para crear este directorio. Entonces crearemos otro directorio y en el momento de arrancar el servidor, le especificaremos este sitio.

La manera de arrancar el servidor será:

```
<Directorio raíz MongoDB> / bin / mongodes
```

Opcionalmente le podemos decir donde está la Base de Datos (si no lo especificamos asumirá que está en **/ data / db**):

```
<Directorio raíz MongoDB> / bin / mongodes --dbpath <directorio de la BD>
```

Resumiendo, y estando situados en el directorio donde hemos descomprimido MongoDB:

- Si somos administradores:
 - Creamos el directorio de datos:

```
mkdir / data  
mkdir / data / db
```

- Arrancamos el servidor:

```
./bin/mongod
```

- Si no somos administradores:
 - Creamos el directorio de datos:

```
mkdir fecha  
mkdir fecha / db
```

- Arrancamos el servidor:

```
./bin/mongod --dbpath ./data/db
```

La siguiente imagen ilustra esta segunda opción:

```

● curs@Ubuntu: ~/mongodb-linux-x86_64-ubuntu1404-3.2.1
Fitxer Edita Visualitza Cerca Terminal Ajuda
curs@Ubuntu:~$ cd mongodb-linux-x86_64-ubuntu1404-3.2.1
curs@Ubuntu:~/mongodb-linux-x86_64-ubuntu1404-3.2.1$ mkdir data
curs@Ubuntu:~/mongodb-linux-x86_64-ubuntu1404-3.2.1$ mkdir data/db
curs@Ubuntu:~/mongodb-linux-x86_64-ubuntu1404-3.2.1$ ./bin/mongod --dbpath ./data/db
2016-02-17T13:58:34.474+0100 I CONTROL [initandlisten] MongoDB starting : pid=7445 port=27017 dbpath=./data/db 64-
2016-02-17T13:58:34.476+0100 I CONTROL [initandlisten] db version v3.2.1
2016-02-17T13:58:34.476+0100 I CONTROL [initandlisten] git version: a14d55980c2cdc565d4704a7e3ad37e4e535c1b2
2016-02-17T13:58:34.476+0100 I CONTROL [initandlisten] OpenSSL version: OpenSSL 1.0.1f 6 Jan 2014
2016-02-17T13:58:34.476+0100 I CONTROL [initandlisten] allocator: tcmalloc
2016-02-17T13:58:34.476+0100 I CONTROL [initandlisten] modules: none
2016-02-17T13:58:34.476+0100 I CONTROL [initandlisten] build environment:
2016-02-17T13:58:34.476+0100 I CONTROL [initandlisten]     distmod: ubuntu1404
2016-02-17T13:58:34.476+0100 I CONTROL [initandlisten]     distarch: x86_64
2016-02-17T13:58:34.476+0100 I CONTROL [initandlisten]     target_arch: x86_64
2016-02-17T13:58:34.476+0100 I CONTROL [initandlisten] options: { storage: { dbPath: "./data/db" } }
2016-02-17T13:58:34.500+0100 I STORAGE [initandlisten] wiredtiger_open config: create,cache_size=1G,session_max=20
ax=4),config_base=false,statistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snappy),file_manag
000),checkpoint=(wait=60,log_size=2GB),statistics_log=(wait=0),
2016-02-17T13:58:34.580+0100 I CONTROL [initandlisten]
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/enabled is
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten] **          We suggest setting it to 'never'
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten]
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/defrag is '
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten] **          We suggest setting it to 'never'
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten]
2016-02-17T13:58:34.582+0100 I FTDC [initandlisten] Initializing full-time diagnostic data capture with directo
c.data'
2016-02-17T13:58:34.583+0100 I NETWORK [HostnameCanonicalizationWorker] Starting hostname canonicalization worker
2016-02-17T13:58:34.663+0100 I NETWORK [initandlisten] waiting for connections on port 27017

```

Una vez en marcha el servidor, no debemos cerrar esta terminal, ya que pararíamos el servidor. Para conectar un cliente, abrimos una segunda terminal y ejecutamos el cliente **mongo** :

```
./bin/mongo
```

```

● curs@Ubuntu: ~/mongodb-linux-x86_64-ubuntu1404-3.2.1
Fitxer Edita Visualitza Cerca Terminal Ajuda
curs@Ubuntu:~/mongodb-linux-x86_64-ubuntu1404-3.2.1$ ./bin/mongo
MongoDB shell version: 3.2.1
connecting to: test
Server has startup warnings:
2016-02-17T13:58:34.580+0100 I CONTROL [initandlisten]
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/enabled is
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten] **          We suggest setting it to 'never'
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten]
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/defrag is '
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten] **          We suggest setting it to 'never'
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten]
>

```

Para probar su funcionamiento, vamos a hacer un par de comandos: uno para guardar un documento y otro para recuperarlo.

Para cualquier operación debe ponerse **db** seguido del nombre de la colección, y después la operación que queremos hacer. Con sgüent:

```
> Db.example.save ({msg: "Hola, ¿qué tal?"})
```

Nos contestará:

```
WriteResult ({ "nInserted": 1})
```

Indicando que ha insertado un documento en la colección **ejemplo** (si no estaba creada, la creará).

Y con el siguiente comando recuperamos la información:

```
> Db.example.findOne ()
```

Que nos volverá:

```
{ "_id": ObjectId ( "56cc130590d651d45ef3d3be"), "msg": "Hola, ¿qué tal?" }
```

Todo lo hace en la misma terminal, ya cada uno de nosotros nos dará un número diferente en **ObjectId** . En la siguiente imagen se ven las dos operaciones:

```
● curs@Ubuntu: ~/mongodb-linux-x86_64-ubuntu1404-3.2.1
Fitxer Edita Visualitza Cerca Terminal Ajuda
curs@Ubuntu:~/mongodb-linux-x86_64-ubuntu1404-3.2.1$ ./bin/mongo
MongoDB shell version: 3.2.1
connecting to: test
Server has startup warnings:
2016-02-17T13:58:34.580+0100 I CONTROL [initandlisten]
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/enabled is
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten] **          We suggest setting it to 'never'
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten]
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/defrag is '
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten] **          We suggest setting it to 'never'
2016-02-17T13:58:34.581+0100 I CONTROL [initandlisten]
> db.exemple.save( {msg:"Hola, què tal?"} )
WriteResult({ "nInserted" : 1 })
> db.exemple.findOne()
{ "_id" : ObjectId("56cc130590d651d45ef3d3be"), "msg" : "Hola, què tal?" }
>
```

En realidad estamos conectados a una Base de Datos llamada **test** . Podemos crear y utilizar más de una Base de Datos, pero en este curso tendrá más que suficiente con esta Base de Datos. Para comprobarlo podemos ejecutar la siguiente sentencia, que nos devuelve el nombre de la Base de Datos:

```
> Db.getName ()
test
```

Instalación en Windows

No ofrece ninguna dificultad. Nos bajamos la versión apropiada de MongoDB para Windows, dependiendo de si es de 32 o 64 bits nuestra versión, que resultará ser un .msi directamente ejecutable. En el momento de hacer estos apuntes, el de la versión de 64 bits era el archivo:

[MongoDB-win32-x86_64-2008plus-ssl-3.2.3-signed.msi](#)

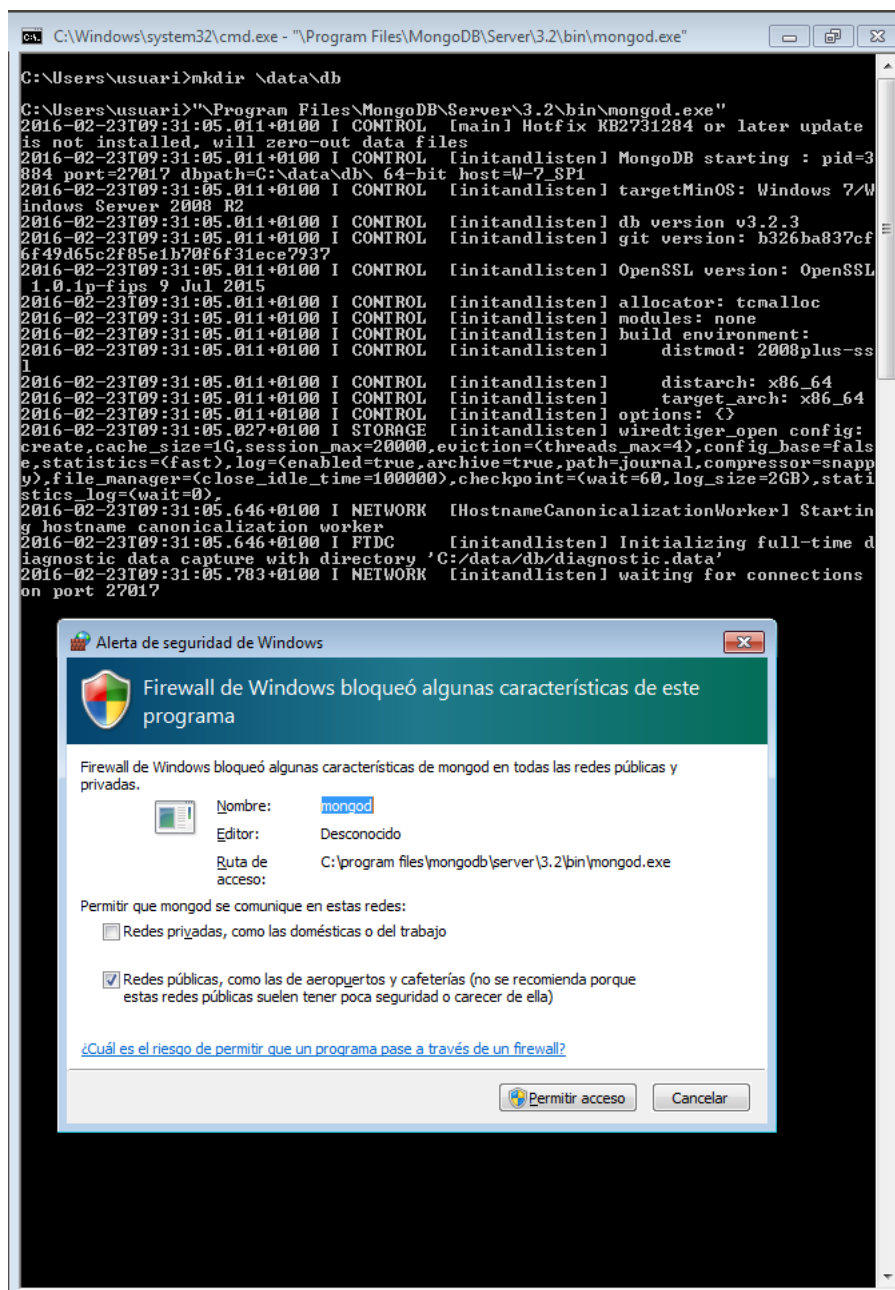
Una vez descargado, ejecutamos el archivo. Tendremos que aceptar la licencia, instalar la versión completa, y aceptar cuando Windows nos avise de que un programa quiere instalar software. El de siempre.

Como en el caso de Linux, antes de ejecutar el servidor tendremos que tener el directorio creado. Por defecto el directorio será **\ data \ db**

Este serían las órdenes para crear el directorio y luego arrancar el servidor:

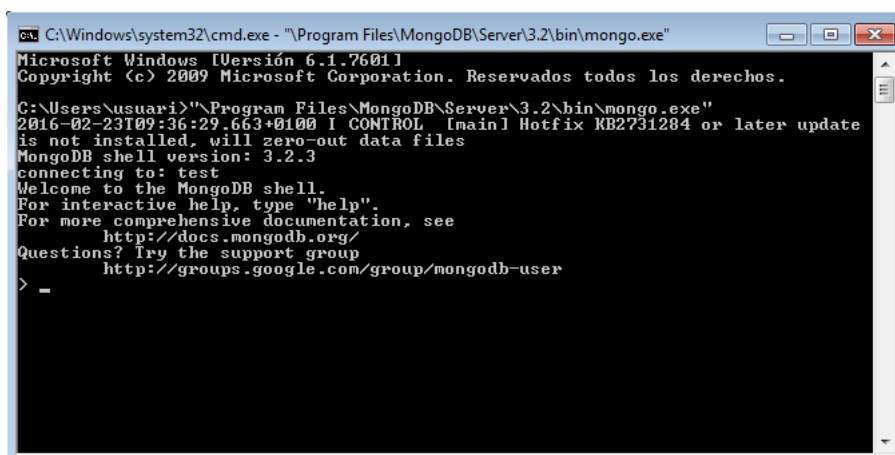
```
mkdir \ data \ db
C: \ Archivos de programa \ MongoDB \ Server \ 3.2 \ bin \ mongod.exe
```

En esta imagen se observa que al intentar poner en marcha el servidor, el Firewall de Windows lo detecta, y solicita permiso para ponerlo en marcha. Aceptamos y suficiente:



Para conectarnos como clientes, lo tendremos que hacer desde otra terminal, ya que si cerramos esta pararemos el servidor. El programa es **mongo.exe** :

C: \ Archivos de programa \ MongoDB \ Server \ 3.2 \ bin \ mongo.exe



Para probar su funcionamiento, vamos a hacer un par de comandos: uno para guardar un documento y otro para recuperarlo.

Para cualquier operación debe ponerse **db** seguido del nombre de la colección, y después la operación que queremos hacer. Con sgüent:

> Db.example.save ({msg: "Hola, ¿qué tal?"})

Nos contestará:

```
WriteResult ({ "nInserted": 1 })
```

Indicando que ha insertado un documento en la colección **ejemplo** (si no estaba creada, la creará).

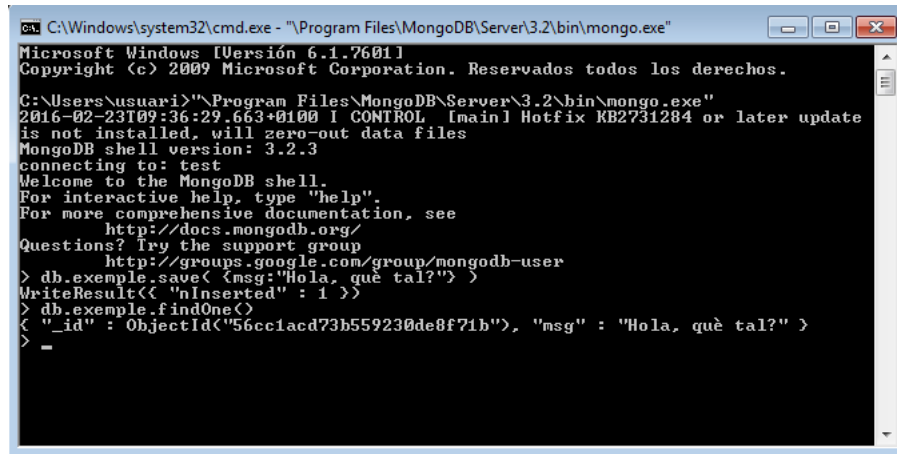
Y con el siguiente comando recuperamos la información:

```
> Db.example.findOne ()
```

Que nos volverá:

```
{ "_id": ObjectId ( "56cc1acd73b559230de8f71b"), "msg": "Hola, ¿qué tal?" }
```

Todo lo hace en la misma terminal, ya cada uno de nosotros nos dará un número diferente en **ObjectId** . En la siguiente imagen se ven las dos operaciones:



```
C:\Windows\system32\cmd.exe - "Program Files\MongoDB\Server\3.2\bin\mongo.exe"
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\usuari>"Program Files\MongoDB\Server\3.2\bin\mongo.exe"
2016-02-23T09:36:29.663+0100 I CONTROL [main] Hotfix KB2731284 or later update
is not installed, will zero-out data files
MongoDB shell version: 3.2.3
connecting to: test
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
http://docs.mongodb.org/
Questions? Try the support group
http://groups.google.com/group/mongodb-user
> db.example.save( {msg:"Hola, qué tal?"} )
WriteResult({ "nInserted" : 1 })
> db.example.findOne()
{ "_id" : ObjectId("56cc1acd73b559230de8f71b"), "msg" : "Hola, qué tal?" }
>
```

En realidad estamos conectados a una Base de Datos llamada **test** . Podemos crear y utilizar más de una Base de Datos, pero en este curso tendrá más que suficiente con esta Base de Datos. Para comprobarlo podemos ejecutar la siguiente sentencia, que nos devuelve el nombre de la Base de Datos:

```
> Db.getName ()
test
```

3.2 - Utilización de MongoDB

Comenzaremos la utilización de MongoDB desde la consola que habíamos arrancado al final de la instalación.

Recuerde que tendremos dos terminal:

- Una con el servidor en marcha (y que no debemos cerrar): **mongod**
- Otra con el cliente que se conecta al servidor: **mongo**

En esta última consola del cliente podemos utilizar sentencias del lenguaje **Javascript**, pero lo que más nos interesará, evidentemente, son las sentencias de acceso a datos. Del lenguaje Javascript prácticamente el único que utilizaremos son variables y algunas funciones.

Utilización de variables

Como comentábamos lo que más utilizaremos del lenguaje **Javascript** es el empleo de variables, que nos puede ser muy útil en algunas ocasiones. Podremos utilizarlas durante la sesión, pero evidentemente no perdurarán de una sesión a otra.

Para definir una variable podemos poner opcionalmente ante la palabra reservada **var**, pero no es necesario. Pondremos el nombre de la variable, el signo igual, ya continuación el valor de la variable, que puede ser una constante, o una expresión utilizando constantes, operadores, otras variables, funciones de Javascript, ...

Especialmente interesante son las variables que pueden contener un documento JSON.

Por ejemplo:

```
> A = 30
30
> b = a / 4
7.5
> Math.sqrt (b)
2.7386127875258306
> doc = {camp1: "Hola", camp2: 45, camp3: new Date ()}
{
  "camp1": "Hola",
  "camp2": 45,
  "camp3": ISODate ( "2016-02-23T19: 10: 29.560Z")
}
>
```

Una variable de tipo JSON se podrá modificar muy fácilmente, toda ella, o alguno de los elementos. Para llegar a los elementos pondremos **nom_variable.nom_camp**:

```
> Doc.camp4 = 3.141592
3.141592
```

```
> Doc.camp5 = [2, 4, 6, 8]
[2, 4, 6, 8]
```

Y si ahora intentamos sacar el contenido de la variable:

```
> Doc
{
  "camp1": "Hola",
  "camp2": 45,
  "camp3": ISODate ( "2016-02-24T22: 31: 12.724Z"),
  "Camp4": 3.141592,
  "camp5": [
    2,
    4,
    6,
    8
  ]
}
>
```

También debemos hacer constar que en un documento, que será de tipo JSON (prácticamente), será un conjunto de pares clave-valor, con algunas restricciones:

- El documento (que muchas veces el asociaremos objeto de JSON) va entre llaves (**{ }**)
- Los elementos de un objeto van separados por comas, y son parejas clave-valor.
- La clave no puede ser nula, ni repetirse en el mismo objeto (sí en diferentes objetos, claro)
- Los valores son los tipos que veremos en la pregunta 3.2.2
- Un documento guardado debe contener obligatoriamente un campo llamado **_id**, y que contendrá un valor único en la colección y servirá para identificarlo. Si en guardar un documento no le hemos puesto campo **_id**, el generará automáticamente MongoDB.

3.2.1 - Tipos de datos

Los valores de los elementos, es decir de las parejas clave valor, pueden ser de una amplia gama de tipos. Hacemos un rápido repaso.

En los ejemplos que van a continuación definimos sencillamente parejas clave-valor de los diferentes tipos, o en todo caso nos lo guardamos en variables, pero no guardaremos aún en la Base de Datos (lo haremos en la siguiente pregunta).

Cuando guardamos en una variable se mostrará el prompt, la definición de la variable y luego el resultado de haber guardado la variable. Utilizaremos recuadros blancos. Los recuadros amarillos son únicamente de la definición de una clave-valor de un determinado tipo

NULL

Más que un tipo de datos es un valor, mejor dicho, la ausencia de valor

```
{ "X": null }
```

boolean

El tipo booleano, que puede coger los valores true o false.

```
{ "X": true }
```

```
{ "Y": false }
```

NUMBER

Por defecto, el tipo de datos numéricos será el de coma flotante (**float**), simple precisión. Si queremos otro tipo (entero, doble precisión, ...) tendremos que indicar expresamente. Así los dos siguientes valores son float:

```
{ "X": 14.3 }
```

```
{ "Y": 3 }
```

Si queremos que sea estrictamente entero, por ejemplo, tendremos que utilizar una función de conversión:

```
{ "X": NumberDouble ( "03/14" ) }
```

```
{ "Y": Numbers ( "3" ) }
```

STRING

Se puede guardar cualquier cadena con caracteres de la codificación UTF-8

```
{X: "Hola, ¿qué tal?" }
```

DATE

Se guarda fecha y hora, e internamente se guardan en milisegundos desde el año inicial. No se guarda el *Time zone* , es decir, la desviación respecto a la hora internacional.

```
{X: ISODate ( "2016-02-24T11: 15: 27.471Z" ) }
```

Normalmente utilizaremos funciones de tratamiento de la fecha-hora. El anterior era para convertir el string en fecha-hora. La siguiente es para obtener la fecha-hora actual:

```
{X: new Date ( ) }
```

Es decir, que si no ponemos parámetro, nos da la fecha-hora actual. Pero le podemos poner como parámetro la fecha-hora que queremos que genere. En este ejemplo, sólo ponemos fecha, por lo tanto se lo será las 00:00:

```
> Z = new Date ( "2016-08-01")  
ISODate ( "2016-08-01T00: 00: 00Z")
```

En este sí ponemos una determinada hora, y observe como hemos deponer T (Time) entre el día y la hora:

```
> Z = new Date ( "2016-08-01T10: 00")  
ISODate ( "2016-08-01T08: 00: 00Z")
```

Es muy importante que pongamos siempre **New Date ()** para generar una fecha-hora. Si ponemos únicamente **Date ()** , lo que estamos generando es un string (con la fecha y hora, pero un string):


```
> z = Date ( "2016-08-01")
Fri Feb 26 2016 08:30:13 GMT + 0100 (CET)
```

ARRAY

Es un conjunto de elementos, cada uno de cualquier tipo, aunque lo más habitual es que sean del mismo tipo. Van entre corchetes (`[]`) y los elementos separados por comas.

```
{x: [2, 4, 6, 8]}
```

Como comentábamos, cada elemento del array puede ser de cualquier tipo:

```
{y: [2, 3.14, "Hola", new Date ()]}
```

En MongoDB podremos trabajar muy bien con arrays, y tendremos operaciones para poder buscar dentro del array, modificar un elemento, crear índice, ...

DOCUMENTOS (OBJETOS)

Los documentos pueden contener como elementos otros documentos (**objetos** en la terminología JSON, pero **documentos** en la terminología de MongoDB).

Como siempre van entre llaves (`{}`), y los elementos que contendrán van separados por comas y serán parejas clave-valor de cualquier tipo (incluso otros documentos).

```
{x: {a: 1, b: 2}}
```

Poner documentos dentro de otros documentos (lo que se llama *embedded document*) nos permite guardar la información de una manera más real, no tan plana. Así por ejemplo, los datos de una persona los podríamos definir de la siguiente manera. Los pondremos en una variable, para ver después cómo podemos acceder a los diferentes elementos, aunque lo más normal es guardarlo en la Base de Datos (con **insert ()** o **save ()**). Si copiamos lo que va a continuación al terminal de Mongo, nos aparecerá con un formato extraño. Es para que la sentencia de asignación a la variable ocupa más de una línea, y aparecerán 3 puntos al principio para indicar que continúa la sentencia. Pero funcionará perfectamente:

```
doc = {
  nombre: "Joan Martí",
  dirección: {
    calle: "Mayor",
    número: 1,
    población: "Castellón"
  },
  teléfonos: [964.223.344, 678345123]
}
```

Observe cómo esta estructura que ha quedado tan clara, seguramente en una Base de Datos Relacional nos habría tocado guardar en 3 tablas: la de personas, la de direcciones y la de teléfonos.

Para acceder a los elementos de un documento poníamos el punto. Pues lo mismo para los elementos de un documento dentro de un documento. Y también podemos acceder a los elementos de un array, poniendo el índice entre corchetes.

```
> Doc.nom
Joan Martí

> doc.adreça
{ "calle": "Mayor", "número": 1, "población": "Castellón" }

> doc.adreça.carrer
Mayor

> doc.telèfons
[964223344, 678345123]

> doc.telèfons [0]
964 223 344
```

OBJECT ID

Es un tipo que define MongoDB para poder obtener valores únicos. Es el valor por defecto del elemento **_id** , necesario en todo documento (atención: en un documento, no en un elemento de tipo documento que hemos dicho equivalente al objeto de JSON). Es un número largo, es decir que utiliza 24 bytes.

Haremos pruebas de su utilización en la siguiente pregunta, en el momento de insertar diferentes documentos.

En este punto vamos a ver las operaciones más básicas, para poder trabajar sobre ejemplos prácticos, y así disponer ya de unos datos iniciales para practicar.

inserción elemental

La función **insert** añadirá documentos a una colección. En el parámetro ponemos el documento directamente, o una variable que contenga el documento. Si la colección no existía, la creará y luego añadirá el documento. En la siguiente sentencia estamos trabajando sobre la colección **ejemplo**, que seguramente ya existirá de cuando la pregunta 3.1 de instalación de MongoDB, que para probar insertamos un documento. Pero si no existía, la creará sin problemas.

```
> Db.example.insert ({msg2: "¿Cómo va la cosa?"})
WriteResult ({ "nInserted": 1})
```

Acabamos de insertar un nuevo documento, y así nos lo avisa ({ "nInserted": 1} , se ha insertado un documento). Automáticamente habrá creado un elemento **_id** de tipo **ObjectId**, ya que le hace falta para identificar el documento entre todos los demás de la colección.

Y en este ejemplo nos guardamos el documento en la variable **doc**, y luego el insertamos

```
> Doc = {msg3: "Por aquí no nos podemos quejar ..."}
{ "msg3": "Por aquí no nos podemos quejar ..."}
> db.example.insert (doc)
WriteResult ({ "nInserted" : 1})
```

También nos indica que ha insertado un documento. Y habrá creado también el campo **_id** como veremos en el siguiente punto

lectura

Tenemos dos funciones para recuperar información: **find** y **findOne**.

- **find ()** : recuperará todos los documentos de la colección, aunque podremos poner criterios para que nos devuelva todos los documentos que cumplan estos criterios (lo veremos más adelante).
- **findOne ()** : sólo volverá un documento, en principio el primero. Puede ser sobre todos los documentos (y por tanto sería el primer documento), o poner una condición, y volvería el primero que cumplirá la condición.

Ejemplo de **find ()** :

```
> Db.example.find ()
{ "_id": ObjectId ( "56ce310bc61e04ba81def50b"), "msg": "Hola, ¿qué tal?" }
{ "_id": ObjectId ( "56ce31f6c61e04ba81def50c"), "msg2": "¿Cómo va la cosa?" }
{ "_id": ObjectId ( "56ce3237c61e04ba81def50d"), "msg3": "Por aquí no nos podemos quejar ..."}
>
```

Ejemplo de **findOne ()** :

```
> Db.example.findOne ()
{ "_id": ObjectId ( "56ce310bc61e04ba81def50b"), "msg": "Hola, ¿qué tal?" }
>
```

En todos los casos podemos comprobar que es cierto lo que veníamos afirmando, que ha creado automáticamente el elemento **_id** para cada documento guardado. Evidentemente, cada uno de nosotros tendrá una valores diferentes.

Inserción especificando el id

Ahora que ya sabemos consultar los documento de la colección con **find ()** vamos a continuar las inserciones de documentos, para ver las posibilidades que tenemos.

En los documento que hemos insertado hasta el momento, no hemos especificado el campo **_id**, y Mongo lo ha generado automáticamente de tipo **ObjectId**.

Pero nosotros podremos poner este campo **_id** con el valor que queramos. Eso sí, tendremos que estar seguros de que este valor no lo coge otro dcument de la colección, o nos dará un error.

Así por ejemplo vamos a insertar la información de unos alumnos. Los pondremos en una colección nueva llamada **alumnos**, y los intentaremos poner un **_id** personal. Por ejemplo pondremos los números 51, 52, 53, ...

```
> Db.alumnes.insert ({_id: 51, nombre: "Rebeca", apellidos: "Martí Peral"})
WriteResult ({ "nInserted": 1})
```

Ha ido bien, y si miramos los documentos que tenemos en la colección, comprobaremos que nos ha respetado el `_id` :

```
> Db.alumnos.find ()
{ "_id": 51, "nombre": "Rebeca", "apellidos": "Martí Peral" }
>
```

Pero si intentamos insertar otro documento con el mismo `_id` (51), nos dará error:

```
> Db.alumnos.insert ({_id: 51, nombre: "Raquel", apellidos: "Gomis Arnau"})
WriteResult ({
  "nInserted": 0,
  "writeError": {
    "code": 11000,
    "errmsg": "E11000 duplicate key error collection: test.alumnos index: _id_ dup key:
{: 51.0}"
  }
})
>
```

Nos avisa que estamos duplicando la *clave principal* , es decir el identificador.

inserción múltiple

Cuando los documentos que queremos insertar son sencillos, podemos insertar más de uno a la vez, poniendo dis del `insert ()` un **array** con todos los elementos. En el siguiente ejemplo creamos unos cuantos números primos en la colección del mismo nombre:

```
> Db.nombresprimers.insert ([{_id: 2}, {_id: 3}, {_id: 5}, {_id: 7}, {_id: 11}, {_id: 13},
{_id: 17}, {_id: 19}])
BulkWriteResult ({
  "writeErrors": [],
  "writeConcernErrors": [],
  "nInserted": 8,
  "nUpserted": 0,
  "nMatched": 0,
  "nModified": 0,
  "nRemoved": 0,
  "upserted": []
})
>
```

Nos avisa que ha hecho 8 inserciones, y aquí los tenemos:

```
> Db.nombresprimers.find ()
{ "_id": 2 }
{ "_id": 3 }
{ "_id": 5 }
{ "_id": 7 }
{ "_id": 11 }
{ "_id": 13 }
{ "_id": 17 }
{ "_id": 19 }
>
```

borrado

Para borrar un documento de una colección utilizaremos la función **remove** , pasándole como parámetro la condición del documento o documentos a borrar.

```
> Db.nombresprimers.remove ({ "_id": 19 })
WriteResult ({ "nRemoved": 1 })
>
```

Nos avisa que ha borrado un documento.

La condición no es necesario que sea sobre el campo `_id` . Puede ser sobre cualquier campo, y se eliminarán todos los que coinciden.

```
> Db.exemple.remove ({ "msg3": "Por aquí no nos podemos quejar ..." })
WriteResult ({ "nRemoved": 1 })
>
```

También tenemos la posibilidad de borrar toda una colección con la función **drop ()** . Preste atención porque es muy sencilla de eliminar, y por tanto, potencialmente muy peligrosa.

```
> Db.nombresprimers.drop ()
true
>
```

actualización

La función **update** servirá para actualizar un documento ya guardado. Tendrá dos parámetros:

- El primer parámetro será la condición para encontrar el documento que se va a actualizar.

- El segundo parámetro será el nuevo documento que sustituirá al anterior

Por ejemplo, si miramos los datos actuales:

```
> Db.example.find ()
{ "_id": ObjectId ( "56ce310bc61e04ba81def50b"), "msg": "Hola, ¿qué tal?" }
{ "_id": ObjectId ( "56ce31f6c61e04ba81def50c"), "msg2": "¿Cómo va la cosa?" }
```

Podemos comprobar el contenido del segundo documento, el que tiene **msg2** . Vamos a modificarlo: en el primer parámetro ponemos condición de búsqueda (sólo habrá uno) y en el segundo ponemos el nuevo documento que sustituirá al anterior

```
> Db.example.update ({msg2: "¿Cómo va la cosa?"}, {Msg2: "¿Qué? ¿Cómo va la cosa?"})
WriteResult ( { "nMatched": 1, "nUpserted": 0, "nModified ": 1})
```

Obsérvese que la contestación del **update ()** es que ha hecho **match** (ha habido coincidencia) con un documento, y que ha modificado una. Si no encuentra ninguna, no dará error, sencillamente dirá que ha hecho match con 0 documentos, y que ha modificado 0 documentos. Miremos como efectivamente ha cambiado el segundo documento

```
> Db.example.find ()
{ "_id": ObjectId ( "56ce310bc61e04ba81def50b"), "msg": "Hola, ¿qué tal?" }
{ "_id": ObjectId ( "56ce31f6c61e04ba81def50c"), "msg2": "¿Qué? ¿Cómo va la cosa?" }
```

Nos vendrán muy bien las variables para las actualizaciones, ya que en muchas ocasiones será modificar ligeramente el documento, cambiando o añadiendo algún elemento. Lo podremos hacer cómodamente con la variable: primero guardamos el documento a modificar en una variable; después modificamos la variable; y por último hacemos la operación de actualización. Evidentemente si tenemos alguna variable con el contenido del documento nos podríamos ahorrar el primer paso.

```
> Doc1 = db.example.findOne ()
{ "_id": ObjectId ( "56ce310bc61e04ba81def50b"), "msg": "Hola, ¿qué tal?" }
```

```
> Doc1.titol = "Mensaje 1"
Mensaje 1
```

```
> Db.example.update ({msg: "Hola, ¿qué tal?"}, Doc1)
WriteResult ( { "nMatched": 1, "nUpserted": 0, "nModified": 1})
> db.example.findOne ()
{
  "_id": ObjectId ( "56ce310bc61e04ba81def50b"),
  "msg": "Hola, ¿qué tal?",
  "titulo": "Mensaje 1"
}
```

función Save

Hemos visto la manera de insertar nuevos documentos con **insert ()** y de actualizar documentos existentes con **update ()** . Pero si intentamos insertar un documento ya existente (la manera de saberlo es por el campo **_id**) nos dará error. Y si intentamos actualizar un documento no existente, no dará error, pero no actualizará nada.

La función **save ()** es una mezcla de los dos: si el documento que vamos a salvar ya existe, pues el modificará, y si no existe lo creará.

Probemos con un ejemplo, y aprovechamos de que en la variable **doc1** tenemos el contenido de un documento. Hacemos una modificación, por ejemplo añadiendo el campo destinatario:

```
> Doc1.destinatari = "Ferran"
Ferran
```

```
> Doc1
{
  "_id": ObjectId ( "56ce310bc61e04ba81def50b"),
  "msg": "Hola, ¿qué tal?",
  "Titulo": "Mensaje 1",
  "destinatario": "Ferran"
}
```

Vamos a guardarlo con **save ()** :

```
> Db.example.save (doc1)
WriteResult ( { "nMatched": 1, "nUpserted": 0, "nModified": 1})
```

Ya nos avisa que como ha encontrado el documento ("**nMatched: 1**") ha modificado uno. Efectivamente, miramos como ha modificado el documento existente:

```
> Db.example.find ()
{ "_id": ObjectId ( "56ce310bc61e04ba81def50b"), "msg": "Hola, ¿qué tal?", "Titulo": "Mensaje 1", "destinatario": "Ferran" }
{ "_id": ObjectId ( "56ce31f6c61e04ba81def50c"), "msg2": "¿Qué? ¿Cómo va la cosa?" }
```

Vamos a hacer ahora una modificación del campo **_id** . A todos los efectos será un documento nuevo, ya que la manera de identificar un documento es para este campo:

```
> Doc1._id = 100
100
```

```
> Doc1
{
  "_id": 100,
  "msg": "Hola, ¿qué tal?",
  "Titulo": "Mensaje 1",
  "destinatario": "Ferran"
}
```

Y ahora vamos a hacer lo **save ()** de este documento:

```
> Db.example.save (doc1)
WriteResult ({ "nMatched": 0, "nUpserted": 1, "nModified": 0, "_id": 100})
```

Observe como avisa que no encontró ninguno igual, y lo que hace es insertarlo (nos dice **nUpserted** , que es una mezcla de **Up** dated y **In**serted ; más adelante volveremos a esta palabra)

```
> Db.example.find ()
{ "_id": ObjectId ( "56ce310bc61e04ba81def50b"), "msg": "Hola, ¿qué tal?", "Titulo": "Mensaje 1", "destinatario": "Ferran"}
{ "_id": ObjectId ( "56ce31f6c61e04ba81def50c"), "msg2": "¿Qué? ¿Cómo va la cosa?" }
{ "_id": 100, "msg": "Hola, ¿qué tal?", "Titulo": "Mensaje 1", "destinatario": "Ferran"}
```

Efectivamente se ha guardado como un nuevo documento

3.2.3 - Operaciones de actualización avanzada

Al final de la pregunta anterior vimos la actualización de documentos ya existentes en la Base de Datos. Esta actualización la hacíamos modificando todo el documento, aunque tenemos la variante de guardar el documento en una variable, modificar esta variable y luego hacer la actualización con esta variable. Pero obsérvese que sigue siendo una modificación de todo el documento, una sustitución del documento antiguo por un documento nuevo.

En esta pregunta veremos la utilización de unos modificadores (*modifiers*) de la operación **update ()** , que nos permitirán modificar documentos de forma potente: creando y eliminando claves (elementos) de un documento, o cambiándolos, y hasta todo añadir o eliminar elementos de un array.

El modificador **\$ Siete** asigna un valor a un campo del documento seleccionado de la Base de Datos. Si el campo ya existía, modificará el valor, y si no existía el creará.

La sintaxis del modificador **\$ Siete** es la siguiente:

```
{ $ Set: {clave: valor}}
```

Pero recuerde que es un modificador, y la tenemos que utilizar dentro de una operación de actualización. Irá en el segundo parámetro del **update ()**, y por tanto con estos modificadores ya no ponemos todo el documento en el segundo parámetro, sino únicamente el operador de modificación.

Mirémoslo mejor en un ejemplo:

```
> Db.alumnes.insert ({nombre: "Abel", apellidos: "Bernat Carrera"})
WriteResult ({ "nInserted": 1 })
>
> db.alumnes.findOne ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8 " ),
  "nombre ": " Abel ",
  "apellidos ": " Bernat Carrera "
}
>
```

Supongamos ahora que queremos añadir la edad. Antes lo haríamos guardando el documento en una variable, y añadiendo el campo, para guardar después. Ahora lo tenemos más fácil:

```
> Db.alumnes.update ({nombre: "Abel"}, {$ set: {edad: 21}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
```

Ha encontrado uno, y lo ha modificado. Evidentemente, si hubiera más de un alumno a on el nombre Abel, los modificaría todos.

```
> Db.alumnes.findOne ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8" ),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 21
}
```

Se puede especificar más de un campo con los valor correspondientes. Si no existían se crearán, y si ya existían se modificarán:

```
> Db.alumnes.update ({nombre: "Abel"}, {$ set: {nota: 8.5, edad: 22}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.alumnes.findOne ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8" ),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "nota": 8.5
}
```

E incluso se puede cambiar el tipo de un campo determinado, y utilizar arrays, y objetos, ...

```
> Db.alumnes.update ({nombre: "Abel"}, {$ set: {nota: [8.5,7.5,9], dirección: {calle:
"Mayor", numero: 7, cp: "12001"}}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.alumnes.findOne ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8" ),
  "nombre": "Abel ",
  "apellidos ": " Bernat Carrera ",
  "edad ": 22,
  "nota ": [
    8.5,
    7.5,
    9
  ],
  "dirección ": {
    "calle ": " Mayor ",
    "numero ": 7,
    "cp ": " 12001 "
  }
}
```

Podemos incluso modificar ahora sólo el valor de un campo de un objeto del documento. Por ejemplo, vamos a modificar el código postal de la anterior alumno. La manera de llegar al código postal será **adreça.cp**, pero tendremos que tener cuidado de que vaya entre comillas para que el encuentro:

```
> Db.alumnes.update ({nombre: "Abel"}, {$ set: {adreça.cp: "12502"}})
2016-03-08T13: 19: 59.744 + 0100 E QUERY [thread1] SyntaxError: missing: after property id @
(shell): 1: 49

> db.alumnes.update ({nombre: "Abel"}, {$ set: { "adreça.cp": "12502"}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
> db.alumnes.findOne ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8"),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera ",
  "edad ": 22,
  " nota ": [
    8.5,
    7.5,
    9
  ],
  "dirección ": {
    " calle ":" Mayor ",
    " numero ": 7,
    " cp ":" 12502 "
  }
}
```


El modificador **\$ unset** servirá para **eliminar** elementos (campos) de uno o unos documentos. Si el campo existía, lo eliminará, y si no existía, no dará error (avisará que se han modificado 0 documentos).

La sintaxis es:

```
{ $ Unset: { campo: 1 } }
```

Tendremos que poner un valor en el campo que vamos a borrar para mantener la sintaxis correcta, y ponemos 1 que equivale a true. También podríamos poner -1, que equivale a false, pero entonces no lo borraría, y por tanto no haríamos nada. Siempre pondremos 1.

Miremos el siguiente ejemplo. Añadimos un campo, que será el número de orden, y después el quitaremos.

```
> Db.alumnos.update ({nombre: "Abel"}, {$ set: {num_orden: 10}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db .alumnos.findOne ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8" ),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "nota": [
    8.5,
    7.5,
    9
  ],
  "dirección": {
    "calle": "Mayor",
    "numero": 7,
    "cp": "12502"
  },
  "num_orden": 10
}

> db.alumnos.update ({nombre: "Abel"}, {$ unset: {num_orden: 1}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> Db.alumnos.update ({nombre: "Abel"}, {$ unset: {puntuacion: 1}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 0 })
```

Hemos añadido primero el campo **num_orden** , y hemos mostrado el documento para comprobar que existe. Después borramos el campo **num_orden** (y nos confirma que ha modificado un documento). Después intentamos borrar un campo que no existe, **puntuacion** . No da error, pero nos avisa que ha modificado 0 documentos. Podemos comprobar al final como el documento ha quedado como esperábamos.

```
> Db.alumnos.findOne ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8" ),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "nota": [
    8.5,
    7.5,
    9
  ],
  "dirección": {
    "calle": "Mayor",
    "numero": 7,
    "cp": "12502"
  }
}
```

El modificador **\$ rename** cambiará el nombre de un campo. Si no existía, no dará error y sencillamente no lo modificará. Debemos cuidar de poner el nuevo nombre del campo entre comillas, para que no de error.

La sintaxis es:

```
{ $ Rename: { campo1: "nou_nom1", campo2: "nou_nom2", ... }}
```

Por ejemplo, cambiamos el nombre del campo **nota** a **notas** :

```
> Db.alumnes.update ({nombre: "Abel"}, {$ rename: {nota: "notas"}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
> db.alumnes.findOne ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8" ),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "dirección": {
    "calle ": " Mayor ",
    " numero ": 7,
    " cp ": " 12502 "
  },
  " notas ": [
    8.5,
    7.5,
    9
  ]
}
```

Obsérvese que la ha cambiado de lugar, lo que nos hace pensar que al cambiar de nombre un campo, lo que hace es volver a crearlo con el nuevo nombre, y borrar el campo antiguo.

En este ejemplo volvemos a cambiar el nombre a **nota** , e intentamos cambiar el nombre a un campo inexistente, **campo1** . No dará error.

```
> Db.alumnes.update ({nombre: "Abel"}, {$ rename: {campo1: "camp2", notas: "nota"}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, " nModified ": 1})
> db.alumnes.findOne ()
{
  " _id ": ObjectId ( " 56debe3017bf4ed437dc77c8 " ),
  " nombre ": " Abel ",
  " apellidos ": " Bernat Carrera ",
  " edad ": 22,
  " dirección ": {
    " calle ": " Mayor ",
    " numero ": 7,
    " cp ": " 12502 "
  },
  " nota ": [
    8.5,
    7.5,
    9
  ]
}
```

Como cabría esperar, el modificador **\$ inc** servirá para incrementar un campo numérico. Si el campo existía, lo incrementará en la cantidad indicada. Si no existía, creará el campo con un valor inicial de 0, e incrementará el valor con la cantidad indicada. La cantidad puede ser positiva, negativa o incluso con parte fraccionaria. Siempre funcionará bien, excepto cuando el campo a incrementar no sea numérico, que dará error.

La sintaxis es la siguiente:

```
{ $ Inc: {campo: cantidad}}
```

En los siguientes ejemplos, incrementamos un campo nuevo (por lo tanto el creará con el valor especificado), y después la incrementamos en cantidades positivas, negativas y fraccionarias, concretamente la inicializamos con **2**, y después la incrementamos en **5**, en **-4** y en **2.25**, por lo tanto el resultado final será **25.5**:

```
> Db.alumnos.update ({nombre: "Abel"}, {$ inc: {puntuacion: 2}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
> db .alumnos.findOne ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8" ),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "nota": [
    8.5,
    7.5,
    9
  ],
  "dirección": {
    "calle": "Mayor",
    "numero": 7,
    "cp": "12502"
  },
  "puntuacion": 2
}
> db.alumnos.update ({nombre: "Abel"}, {$ inc: {puntuacion: 5}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
> Db.alumnos.update ({nombre: "Abel"}, {$ inc: {puntuacion: -4}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
> db.alumnos.update ({nombre: "Abel"}, {$ inc: {puntuacion: 2.25}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
> db. alumnos.findOne ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8" ),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "nota": [
    8.5,
    7.5,
    9
  ],
  "dirección": {
    "calle": "Mayor",
    "numero": 7,
    "cp": "12502"
  },
  "puntuacion": 25.5
}
```

3.2.3.5 - Elementos de un array

Para acceder directamente a un elemento de un array de un determinado documento se puede utilizar la siguiente sintaxis:

```
"Array.index"
```

Debemos tener presente que el primer elemento del array es el de subíndice 0. Y no se olvide de cerrar todo entre comillas para que lo pueda encontrar.

Si no existe el elemento con el subíndice indicado, dará error.

Por ejemplo, vamos a subir un punto la primera nota del alumno que estamos utilizando en todos los ejemplos:

```
> Db.alumnes.update ({nombre: "Abel"}, {$ inc: { "nota.0": 1}})
```

```
> Db.alumnes.findOne ()
{
  "_id": ObjectId ( "56df11d778549bdfbf2125e3"),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "dirección": {
    "calle ": " Mayor ",
    " numero ": 7,
    " cp ": " 12502 "
  },
  " nota ": [
    9.5,
    7.5,
    9
  ]
}
```

3.2.3.6 - Inserción en Arrays: \$ push

La manera más sencilla de introducir un elemento en un array es utilizar **\$ push** sin más. Si existía el array, introducirá el o los nuevos elementos al final. Si no existía el array, el creará con este o estos elementos.

La sintaxis es:

```
{ $ Push: { clave: elemento } }
```

Por ejemplo vamos a añadir una nota al alumno de siempre, y pongamos la diferente para ver que se introduce al final:

```
> Db.alumnos.update ({nombre: "Abel"}, { $ push: {nota: 7}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
```

```
> Db.alumnos.findOne ()
{
  "_id": ObjectId ( "56df11d778549bdfbf2125e3"),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "dirección": {
    "calle ":" Mayor ",
    " numero ": 7,
    " cp ":" 12502 "
  },
  " nota ": [
    9.5,
    7.5,
    9,
    7
  ]
}
```

También hay manera de introducir un elemento en una determinada posición que no sea el final, pero se complica bastante la cosa, ya que tenemos que utilizar por un lado el modificador **\$ position** para decir donde se insertará, y por otro lado el modificador **\$ each** para poder especificar el o los valores que se quieren insertar. Se pone a continuación únicamente de forma ilustrativa.

Para insertar en una determinada posición tenemos que utilizar obligatoriamente 2 modificadores más:

- **\$ position** indicará a partir de qué posición se hará la acción (normalmente de insertar en el array, es decir, **\$ push**)
- **\$ each** nos permite especificar una serie de valores como un array, y significa que se hará la operación para cada valor del array

Los dos modificadores seguirán la sintaxis de siempre, de clave valor, por lo tanto el conjunto de la sintaxis es:

```
{ $ Push:
  { clau_del_array:
    { $ position: posición ,
      $ each: [ valores ]
    }
  }
}
```

Aquí tenemos un ejemplo donde introducimos una nota en la primera posición:

```
> Db.alumnos.update ({nombre: "Abel"}, { $ push: {nota: { $ position: 0, $ each:
[5]}}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
```

```
> Db.alumnos.findOne ()
{
  "_id": ObjectId ( "56df11d778549bdfbf2125e3"),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "dirección": {
    "calle ":" Mayor ",
    " numero ": 7,
    " cp ":" 12502 "
  },
  " nota ": [
    5,
    9.5,
    7.5,
    9,
    7
  ]
}
```

3.2.3.7 - Eliminación en arrays: \$ pop y \$ pull

Hay más de una manera de eliminar elementos de un array.

\$_pop

Si queremos eliminar el primer elemento o el último, el modificador adecuado es **\$ pop** . La sintaxis es

```
{ $ Pop: { clave: posicion } }
```

Donde en posición podremos poner:

- -1, y borrará el primer elemento
- 1, y borrará el último

En los siguientes ejemplos se borran primero el último elemento y luego el primero.

```
> Db.alumnes.findOne ()
{
  "_id": ObjectId ( "56df11d778549bdfbf2125e3" ),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "dirección": {
    "calle ": " Mayor ",
    " numero ": 7,
    " cp ": " 12502 "
  },
  " nota ": [
    5,
    9.5,
    7.5,
    9,
    7
  ]
}
```

```
> Db.alumnes.update ({nombre: "Abel"}, { $ pop: {nota: 1}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
> db .alumnes.findOne ()
{
  "_id": ObjectId ( "56df11d778549bdfbf2125e3" ),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "dirección": {
    "calle": "Mayor",
    "numero": 7,
    "cp": "12502"
  },
  "nota": [
    5,
    9.5,
    7.5,
    9
  ]
}
```

```
> Db.alumnes.update ({nombre: "Abel"}, { $ pop: {nota: -1}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
> db.alumnes.findOne ()
{
  "_id": ObjectId ( "56df11d778549bdfbf2125e3" ),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "dirección": {
    "calle" : "Mayor",
    "numero": 7,
    "cp": "12502"
  },
  "nota": [
    9.5,
    7.5,
    9
  ]
}
```

\$_pull

Con este modificador borraremos los elementos del array que coincidan con una condición, estén en la posición que estemos. Observe cómo se puede eliminar más de un elemento.

Para poder comprobar bien, primero insertamos otro elemento al final del array, con el valor **7.5** (si ha seguido los mismos ejemplos que en estos apuntes, este valor ya se encuentra en la segunda posición).

```
> Db.alumnes.update ({nombre: "Abel"}, {$ push: {nota: 7.5}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
```

```
> Db.alumnes.findOne ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8" ),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "dirección": {
    "calle ":" Mayor ",
    " numero ": 7,
    " cp ":" 12502 "
  },
  " nota ": [
    9.5,
    7.5,
    9,
    7.5
  ]
}
```

Ahora vamos a borrar con **\$ pull** el elemento de valor **7.5**

```
> Db.alumnes.update ({nombre: "Abel"}, {$ pull: {nota: 7.5}})
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
```

```
> Db.alumnes.findOne ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8" ),
  "nombre": "Abel",
  "apellidos": "Bernat Carrera",
  "edad": 22,
  "dirección": {
    "calle ":" Mayor ",
    " numero ": 7,
    " cp ":" 12502 "
  },
  " nota ": [
    9.5,
    9
  ]
}
```

Esta palabra ya la habíamos comentada en un punto anterior.

En el **update ()** normal, si la condición de búsqueda no daba ningún resultado (hablando rápido, si no presenta *matching* con ningún documento), pues no actualizaba ningún documento y punto.

El **Upsert** es una variante del update, que cuando no coincida ningún documento con la condición, creará un documento nuevo que será el resultado de combinar el criterio que se ha utilizado en la condición con las operaciones de actualización realizadas en el segundo parámetro

Para que un **Update** actúe de esta manera, le tenemos que poner un tercer parámetro con el valor **true** :

```
update ({...}, {...}, true)
```

Recuerde que el primer parámetro era la condición, y el segundo la actualización.

Mirémoslo en el ejemplo de los alumnos. Si vamos a actualizar los apellidos, y se encuentra el documento, se actualizará:

```
> Db.alumnes.update ({nombre: "Abel"}, {$ set: {apellidos: "Bernat Cantera"}}, true)
WriteResult ({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
```

Efectivamente, nos dice que ha modificado un documento.

Pero si no se encuentra el documento (por ejemplo porque le hemos puesto el nombre **Berta**):

```
> Db.alumnes.update ({nombre: "Berta"}, {$ set: {apellidos: "Bernat Cantero"}}, true)
WriteResult ({
  "nMatched": 0,
  "nUpserted": 1,
  "nModified": 0,
  "_id": ObjectId ( "56dfdbd136d8b095cb6bd57a")
})
```

Ya nos avisa que no ha hecho ningún *matching* , y ha hecho un **Upsert** . Lo podemos comprobar mirando todos los documento de la colección:

```
> Db.alumnes.find ()
{ "_id": ObjectId ( "56debe3017bf4ed437dc77c8"), "nombre": "Abel", "apellidos": "Bernat Cantera", "edad": 22, "dirección": { "calle ": " Mayor ", " numero ": 7, " cp ": " 12502 " }, " nota ": [9.5, 9]}
{"_id ": ObjectId ( " 56dfdbd136d8b095cb6bd57a " ), " nombre ": " Berta " , "apellidos": "Bernat Cantero"}
>
```

El nuevo documenttindrà los campos:

- **_id** , con lo que nos había avisado que generaría
- Los campos de la condición, que en nuestro ejemplo es **{nombre: "Berta"}**
- Los campos de la actualización, que en nuestro ejemplo eran los apellidos

3.3 - Consulta de documentos

En la pregunta anterior hemos visto cómo introducir, eliminar y modificar documentos. Las consultas de documentos han sido muy sencillas, para comprobar únicamente los resultados.

En esta pregunta veremos en profundidad la consulta de documentos.

- Funciones **find ()** y **findOne ()** , que son las que hemos utilizado hasta ahora. Veremos en profundidad su sintaxis y potencia.
- Limitaremos y ordenaremos también los resultados
- Incluso podremos elaborar más los resultados, agrupando los resultados, utilizando funciones de agregación (o mejor decir operadores de agregación) y dándoles un aspecto diferente

3.3.1 - Parámetros de las funciones find () y findOne ()

Las funciones **find ()** y **findOne ()** son absolutamente equivalentes, con la única diferencia que la primera devuelve todos los documentos encontrados, mientras que la segunda sólo devuelve el primer documento encontrado.

Para una mejor comprensión, utilizaremos únicamente **find ()**, para ver todos los resultados obtenidos.

La función **find ()** se ha comparado tradicionalmente con la sentencia SELECT de SQL. Siempre devolverá un conjunto de documentos, que pueden variar desde no volver ningún documento, a devolverlos todos los de la colección.

La función **find ()** puede tener varios parámetros.

- El primero indica una condición o criterio, y devolverá aquellos documentos de la colección que cumplan la condición o criterio. Esta condición viene dada en forma de documento (u objeto) JSON, y es como la habíamos visto en la función **update ()**:

```
db.col_leccio1.find ({clave1: valor1})
```

Volverá todos los documentos de la colección **col_leccio1** que tengan el campo **clave1** y que en ella tengan el valor **valor1**. Este criterio puede ser el complicado que haga falta, formándolo en JSON. En definitiva, devolverá aquellos documentos que hagan *matching* con el documento del criterio, es decir, funcionaría como un **and**

```
db.col_leccio1.find ({clave1: valor1, clau2: valor2})
```

que devolvería aquellos documentos de la **col_leccio1** que tienen el campo **clave1** con el valor **valor1** y que tienen el campo **clau2** con el valor **valor2**

Si no queremos poner ningún criterio, para que los devuelva todos, no ponemos nada como parámetro, o aún mejor, le pasamos un documento (objeto) vacío, por lo que todos los documentos de la colección harán *matching* con él.

```
db.col_leccio1.find ({})
```

Tendremos esto presente, sobre todo cuando nos toque utilizar el segundo parámetro de **find**. Si no queremos ningún criterio, pondremos el documento vacío como el ejemplo anterior

- El segundo parámetro nos servirá para delimitar los campos de los documentos que se devolverán. También tendrá el formato JSON de un objeto al que le pondremos como claves los diferentes campos que queremos que aparezcan o no, y como valor 1 para que sí aparezcan y 0 para que no aparezcan.

Si ponemos algún campo a que sí aparezca (es decir, con el valor 1), los únicos que aparecerán serán los mismos, además del **_id** que por defecto siempre aparece.

```
> Db.alumnes.find ({}, {nombre: 1})
{ "_id": ObjectId ( "56debe3017bf4ed437dc77c8"), "nombre": "Abel" }
{ "_id": ObjectId ( "56dfdbd136d8b095cb6bd57a"), "nombre": "Berta" }
```

Por lo tanto si no queremos que aparezca **_id** pondremos:

```
> Db.alumnes.find ({}, { _id: 0})
{ "nombre": "Abel", "apellidos": "Bernat Cantera", "edad": 22, "dirección": { "calle": "Mayor", "numero": 7, "cp": "12502"}, "nota": [9.5, 9]}
{ "nombre": "Berta", "apellidos": "Bernat Cantero" }
```

Y si queremos sacar únicamente el nombre:

```
> Db.alumnes.find ({}, {nombre: 1, _id: 0})
{ "nombre": "Abel" }
{ "nombre": "Berta" }
```

Por último, como que a partir de ahora utilizaremos documentos más complicados, si queremos que nos aparezcan los campos que devolvemos de una forma un poco más elegante o bonita (*pretty*), pondremos esta función al final: **find (). Pretty ()**

```
> Db.alumnes.find (). Pretty ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8"),
  "nombre": "Abel",
  "apellidos": "Bernat Cantera",
  "edad": 22,
  "dirección": {
    "calle": "Mayor",
    "numero": 7,
    "cp": "12502"
  },
  "nota": [
    9.5,
    9
  ]
}
{
  "_id": ObjectId ( "56dfdbd136d8b095cb6bd57a"),
```

```
"nombre" : "Berta",  
"apellidos": "Bernat Cantero"  
}
```

3.3.2 - Operadores de las condiciones

Antes de empezar esta pregunta, vamos a coger unos datos de prueba, que están en el fichero `libros_ejemplo.json`

Sólo tiene que copiar el contenido del archivo en la terminal del cliente de Mongo.

Ponemos aquí el contenido para que pueda pegarle una miraditas sin necesidad de abrirlo. Irá bien para los ejemplos posteriores.

```
db.libro.save ({
  "_id": "9788408117117",
  "titulo": "Circo Máximo",
  "autor": "Santiago Posteguillo",
  "editorial": "Planeta",
  "enstock": true,
  "páginas": 1100,
  "precio": 21.75,
  "fecha": new ISODate ( "2013-08-29T00: 00: 00Z"),
  "resumen": "asedios sin fin, dos aurigas rivales, el Anfiteatro, los gladiadores y tres carreras de cuadrigas
})

Db.libro.save ({
  "_id": "9788401342158",
  "titulo": "El juego de Ripper",
  "autor": "Isabel Allende",
  "editorial": "Plaza & Janes",
  "enstock": true,
  "páginas": 480,
  "precio": 21.75,
  "fecha": new ISODate ( "2014-03-01T00: 00: 00Z"),
  "resumen": "Tal como predijo la astróloga más reputada de San Francisco, una oleada de crímenes C
})

db.libro.save ({
  "_id": "9788496208919 ",
  "titulo": "Juego de tronos: Canción de hielo y fuego 1",
  "autor": "George RR Martin",
  "editorial": "Gigamesh",
  "enstock": true,
  "páginas": 793,
  "precio": 9.5,
  "fecha": new ISODate ( "2011-11-24T00: 00: 00Z"),
  "Resumen": "Tras el largo verano, el invierno se Acerca a los Siete Reinos. Lord Eddar Stark, señor
})

Db.libro.save ({
  "_id": "9788499088075",
  "titulo": "La ladrona de libros",
  "autor": "Markus Zusak",
  "editorial": "Debolsillo",
  "enstock": false ,
  "páginas": 544,
  "precio": 09.45,
  "fecha": new ISODate ( "2009-01-09T00: 00: 00Z"),
  "resumen": "En plena II Guerra Mundial, la pequeña Liesel hallará sume salvación en la lectura. Una r
})

db.libro.save ({
  "_id": " 9788415140054 ",
  "titulo": " la princesa de hielo ",
  "autor": " Camilla Läckberg ",
  "editorial": "Embolsillo",
  "enstock": true,
  "precio": 11,
  "fecha": new ISODate ( "2012-10-30T00: 00: 00Z"),
  "resumen": "Misterio y secretos familiares en una emocionante novela de suspense Erica vuelve a super
})

Db.libro.save ({
  "_id": "9788408113331",
  "titulo": "

  "editorial": "Planeta",
  "enstock": false,
  "páginas": 290,
  "precio": 17.23,
  "fecha": new ISODate ( "2013-06-04T00: 00: 00Z"),
  "resumen": "En la pequeña isla de Thisby, cada noviembre los caballos de agua de la mitología celta e
})

db.libro.save ( {
  "_id": "9788468738895",
  "titulo": "Las reglas del juego",
  "autor": "Anna Casanovas",
  "enstock": true,
  "páginas": null,
  "precio": 15.90,
  "fecha": new ISODate ( "2014-02-06T00: 00: 00Z"),
  "resumen": "¿Por qué no le Había sucedido antes? Se supone que el Y Susana no se soportaban ¿Desde cuándo Siente
})
```

Como puede comprobar están los comandos de inserción (`save()`) y también se ve bastante bien los campos de cada documento.

Puede comprobar que hay 7 documentos en la nueva colección **libro** :

```
> Db.libro.count ()
7
```

Y también podemos consultar los títulos de forma cómoda:

```
> Db.libro.find ({}, {titulo: 1})
{ "_id": "9788408117117", "titulo": "Circo Máximo" }
{ "_id": "9788401342158", "titulo": "El juego de Ripper " }
{ "_id": "9788496208919", "titulo": "Juego de tronos: Canción de hielo y fuego 1 " }
{ "_id": "9788499088075", "titulo": "La ladrona de libros " }
{ "_id": "9788415140054", "titulo": "La princesa de hielo" }
{ "_id": "9788408113331", "titulo": "Las carreras de Escorpio" }
{ "_id": "9788468738895", "titulo": "Las reglas del juego " }
```

Y otro ejemplo, donde consultamos los libros que están en stock (hay un campo booleano que lo dice: **enstock**), mostrando título, editorial y precio

```
> Db.libro.find ({enstock: true}, {titulo: 1, editorial: 1, precio: 1})
{ "_id": "9788408117117", "titulo": "Circo Máximo", "editorial": " planeta ", " precio ": 21.75 }
{ "_id": "9788401342158", "titulo": "El juego de Ripper ", " editorial ": " Plaza & Janes ", " precio ": 21.75 }
{ "_id": "9788496208919", "titulo": "Juego de tronos: Canción de hielo y fuego 1", "editorial": "Gigamesh", "precio": 9.5 }
{ "_id": "9788415140054", "titulo": "La princesa de hielo ", " editorial ": " Embolsillo ", " precio ": 11 }
```

Y un último ejemplo, donde consultamos los libros que están en stock y tienen un precio de 21.75 €, mostrando todo excepto el **_id** y el resumen

```
> Db.libro.find ({enstock: true, precio: 21.75}, {titulo: 1, editorial: 1, precio: 1})
{ "_id": "9788408117117", "titulo": "Circo Máximo", " editorial ": " Planeta ", " precio ": 21.75 }
{ "_id": "9788401342158", "titulo": "El juego de Ripper ", " editorial ": " Plaza & Janes ", " precio ": 21.75 }
```

Vamos a mirar ahora operadores que nos servirán para hacer mejor las consultas.

Operadores de comparación

Hasta ahora en todas las condiciones hemos utilizado la igualdad, si un determinado campo era igual a un determinado valor. Pero hay infinidad de consultas en las que querremos otras operaciones de comparación: mayor, mayor o igual, menor, ...

Estos son los operadores de comparación:

- **\$lt** (*less than*) **menor**
- **\$lte** (*less than or equal*) **menor o igual**
- **\$gt** (*greater than*) **mayor**
- **\$gte** (*greater than or equal*) **mayor o igual**
- **\$ne** (*not equal*) **distinto**
- **\$eq** (*equal*) **igual** (pero este casi no haría falta, porque en no poner nada se refiere a la igualdad como hasta ahora)

La sintaxis para su utilización es, como siempre, acoplarse a la sintaxis JSON:

```
clave: { $ operador: valor [...]
```

Así por ejemplo, para buscar los libros de más de 10 €:

```
> Db.libro.find ({precio: { $ gt: 10 }}, {titulo: 1, precio: 1})
{ "_id": "9788408117117", "titulo": "Circo Máximo", "precio": 21.75 }
{ "_id": "9788401342158", "titulo": "El juego de Ripper", "precio": 21.75 }
{ "_id": "9788415140054", "titulo": "La princesa de hielo", "precio": 11 }
{ "_id": "9788408113331", "titulo": "Las carreras de Escorpio", "precio": 17.23 }
{ "_id": "9788468738895", "titulo": "Las reglas del juego", "precio": 15.9 }
```

Y para buscar los libros entre 10 y 20 €:

```
> Db.libro.find ({precio: { $ gt: 10, $ lt: 20 }}, {titulo: 1, precio: 1})
{ "_id": "9788415140054", "titulo": "La princesa de hielo", "precio": 11 }
{ "_id": "9788408113331", "titulo": "Las carreras de Escorpio", "precio": 17.23 }
{ "_id": "9788468738895", "titulo": "Las reglas del juego", "precio": 15.9 }
```

Es especialmente útil para las fechas, ya que difícilmente encontraremos una fecha (y hora) exacta, y querremos casi siempre los documentos anteriores a una fecha, o posteriores, o entre dos fechas. Tendremos que tener cuidado por el tratamiento especial de las fechas: debemos comparar cosas del mismo tipo, y por lo tanto la fecha con la que queremos comparar la tendremos que tener en forma de fecha:

```
> Var d = new ISODate ( "2013-01-01T00: 00: 00Z" )
> db.libro.find ({fecha: { $ GTE: d }}, {fecha: 1})
{ "_id": "9788408117117", "fecha": ISODate ( " 2013-08-29T00: 00: 00Z " ) }
{ "_id": "9788401342158", "fecha": ISODate ( " 2014-03-01T00: 00: 00Z " ) }
```

```
{ "_id": "9788408113331", "fecha": ISODate ( "2013-06-04T00: 00: 00Z" ) }
{ "_id": "9788468738895", "fecha": ISODate ( "2014-02-06T00: 00 : 00Z " ) }
```

\$ in

Servirá para comprobar si el valor de un campo está entre los de una lista, proporcionada como un array. La sintaxis es:

```
clave: { $ in: [valor1, valor2, ..., valorN] }
```

Y aquí tenemos un ejemplo, los libros de las editoriales Planeta y Debolsillo:

```
> Db.libro.find ({editorial: { $ in: [ "Planeta", "Debolsillo" ] }}, {titulo: 1, editorial: 1})
{ "_id": "9788408117117", "titulo": " Circo Máximo ", " editorial ":" Planeta " }
{ "_id ":" 9788499088075 ", " titulo ":" La ladrona de libros ", " editorial ":" Debolsillo " }
{ "_id ":" 9788408113331 ", " titulo ":" Las carreras de Escorpio ", " editorial ":" Planeta " }
```

\$ niño

Es lo contrario, sacará los que no están en la lista.

```
> Db.libro.find ({editorial: { $ niño: [ "Planeta", "Debolsillo" ] }}, {titulo: 1, editorial: 1})
{ "_id": "9788401342158", "titulo": " el juego de Ripper ", " editorial ":" Plaza & Janes " }
{ "_id ":" 9788496208919 ", " titulo ":" juego de tronos: Canción de hielo y fuego 1 ", " editorial ":" Gigamesh " }
{ "_id": "9788415140054", "titulo": "La princesa de hielo", "editorial": "Embolsillo" }
{ "_id": "9788468738895", "titulo": "Las reglas del juego" }
```

Observe como saca los libros que no tienen editorial, como es el caso del último libro, Las reglas del juego

\$ oro

El operador anterior, \$ in , ya hacía una especie de OR, pero siempre sobre el mismo campo. Si la operación OR la queremos hacer sobre campos distintos, tendremos que utilizar el operador \$ oro . Su sintaxis debe jugar con la posibilidad de poner muchos elementos, y por tanto conviene el array:

\$ Oro: [{clave1: valor1}, {clave2: valor2}, ..., {claveN: valorN}]

Será cierto si se cumple alguna de las condiciones. Por ejemplo, sacar los libros que no están en stock o que no tienen editorial:

```
> Db.libro.find ({ $ oro: [{enstock: false}, {editorial: null}] }, {titulo: 1, enstock: 1, editorial: 1})
{ "_id": "9788499088075", " titulo ":" La ladrona de libros ", " editorial ":" Debolsillo ", " enstock ":" false" }
{ "_id ":" 9788408113331 ", " titulo ":" Las carreras de Escorpio ", " editorial ":" Planeta ", " enstock ":" false" }
{ "_id ":" 9788468738895 ", " titulo ":" Las reglas del juego ", " enstock ":" true" }
```

\$ not

Sirve para negar otra condición.

```
$ Not: {condición}
```

Por ejemplo los libros que no son de la editorial Planeta (obsérvese que sería más sencillo utilizar el operador \$ ne , pero es para mostrar su funcionamiento:

```
> Db.libro.find ({editorial: { $ not: { $ eq: "Planeta" } }}, {titulo: 1, editorial: 1})
{ "_id": "9788401342158", "titulo": "El juego de Ripper ", " editorial ":" Plaza & Janes " }
{ "_id ":" 9788496208919 ", " titulo ":" juego de tronos: Canción de hielo y fuego 1 ", " editorial ":" Gigamesh " }
{ "_id": "9788499088075", "titulo": "La ladrona de libros", "editorial": "Debolsillo" }
{ "_id": "9788415140054", "titulo": "La princesa de hielo", "editorial": "Embolsillo" }
{ "_id": "9788468738895", "titulo": "Las reglas del juego" }
```

\$ exists

Servirá para saber los documentos que tienen un determinado campo

```
clave: { $ exists: boolean }
```

Depenet del valor *boolean* , el funcionamiento será:

- **true** : devuelve los documentos en los que existe el campo, aunque su valor sea nulo
- **false** : devuelve los documentos que no tienen el campo.

Vamos a sacar los libros que tienen el campo **páginas** :

```
> Db.libro.find ({páginas: {$ exists: true}}, {titulo: 1, páginas: 1})
{ "_id": "9788408117117", "titulo": "Circo Máximo", "páginas": 1100}
{ "_id": "9788401342158", "titulo": "El juego de Ripper", "páginas": 480}
{ "_id": "9788496208919", "titulo": "juego de tronos: Canción de hielo y fuego 1 ", "páginas": 793}
{ "_id": "9788499088075 ", "titulo": "La ladrona de libros ", "páginas": 544}
{ "_id": "9788408113331 ", "titulo": "las carreras de Escorpio ", "páginas": 290}
{ "_id": "9788468738895 ", "titulo": "Las reglas del juego", "páginas": null}
```

Observe cómo nos aparece también el último libro, que tiene el campo **páginas** con el valor **nulo** . En cambio si hubiéramos hecho la consulta preguntando por los que son diferente de nulo, no aparecería este último libro:

```
> Db.libro.find ({páginas: {$ ne: null}}, {titulo: 1, páginas: 1})
{ "_id": "9788408117117", "titulo": "Circo Máximo", "páginas": 1100}
{ "_id": "9788401342158", "titulo": "El juego de Ripper", "páginas": 480}
{ "_id": "9788496208919", "titulo": "juego de tronos: Canción de hielo y fuego 1 ", "páginas": 793}
{ "_id": "9788499088075 ", "titulo": "La ladrona de libros ", "páginas": 544}
{ "_id": "9788408113331 ", "titulo": "las carreras de Escorpio ", "páginas": 290}
```

Y si ponemos **false** al valor en el **\$ exists** , únicamente nos aparecerá el libro que no tiene el campo:

```
> Db.libro.find ({páginas: {$ exists: false}}, {titulo: 1, páginas: 1})
{ "_id": "9788415140054", "titulo": "La princesa de hielo"}
```

Y por la misma razón que antes, si sacamos los que tienen páginas en null, nos saldrá tanto quien no tiene el campo, como el que lo tiene pero con valor nulo:

```
> Db.libro.find ({páginas: null}, {titulo: 1, páginas: 1})
{ "_id": "9788415140054", "titulo": "La princesa de hielo"}
{ "_id": "9788468738895 ", "titulo": "Las reglas del juego ", "páginas": null}
```

Por lo tanto, para según qué cosas, nos interesa el operador **\$ exists** , en lugar de jugar con el nulo.

expresiones regulares

Mongo acepta las expresiones regulares de forma nativa, lo que da mucha potencia para poder buscar información diversa.

Las expresiones regulares en Mongo tienen la misma sintaxis que en Perl, y que es muy muy parecida a la mayor parte de lenguajes de programación.

Miremos algunos ejemplos. Los libros dentro de los cuales está la palabra **juego** :

```
> Db.libro.find ({titulo: / juego /}, {titulo: 1})
{ "_id": "9788401342158", "titulo": "El juego de Ripper"}
{ "_id": "9788468738895", "titulo": "Las reglas del juego"}
```

Ahora que tienen la palabra **juego** sin importar mayúsculas o minúsculas:

```
> Db.libro.find ({titulo: / juego / y}, {titulo: 1})
{ "_id": "9788401342158", "titulo": "El juego de Ripper"}
{ "_id": "9788496208919", "titulo": "juego de tronos: Canción de hielo y fuego 1"}
{ "_id": "9788468738895", "titulo": "Las reglas del juego"}
```

Y ahora que tienen la palabra **juego** sólo al principio.

```
> Db.libro.find ({titulo: / ^ juego / y}, {titulo: 1})
{ "_id": "9788496208919", "titulo": "Juego de tronos: Canción de hielo y fuego 1"}
```

Y ahora los libros que en el resumen (**resumen**) que tienen la palabra **amiga** o **amigo** , es decir **amig** seguido de una **a** o una **o** :

```
> Db.libro.find ({resumen: / a medio [ao] / y}, {titulo: 1})
{ "_id": "9788415140054", "titulo": "La princesa de hielo"}
{ "_id": "9788468738895", "titulo": "Las reglas del juego"}
```

arrays

Las consultas dentro de arrays de Mongo son muy sencillas.

La más sencilla es como cuando buscamos un valor de un tipo sencillo, y en este caso lo que hará Mongo es buscar en todo el array por si está este valor. Es decir, exactamente igual que lo que hemos hecho hasta ahora.

```
db.col_leccion1.find ({clau_array: valor})
```

Miremos en un ejemplo. Vamos a crear dos documentos que tengan un array cada uno, por ejemplo de colores. El creamos en una colección nueva, llamada **colorines** , en dos documentos con el mismo campo de tipo array, **color** , pero con datos diferentes:

```
> Db.colorins.insert ({color: [ "rojo", "azul", "amarillo"]})
WriteResult ({ "nInserted": 1})
```

```
> Db.colorins.insert ({color: [ "negro", "blanco", "rojo"]})
WriteResult ({ "nInserted": 1})
```

```
> Db.colorins.find ();
{ "_id": ObjectId ( "56e1438ff6663c8169030e09"), "color": [ "rojo", "azul", "amarillo"]}
{ "_id": ObjectId ( "56e14398f6663c8169030e0a"), "color": [ "negro", "blanco", "rojo"]}
```

Como se ve en la sintaxis, elegir los documentos que tienen un campo (en este caso de array) que contenga un valor, es igual de sencillo que cuando se trata de un campo de tipo string, por ejemplo:

```
> Db.colorins.find ({color: "rojo"})
{ "_id": ObjectId ( "56e1438ff6663c8169030e09"), "color": [ "rojo", "azul", "amarillo"]}
{ "_id": ObjectId ( "56e14398f6663c8169030e0a"), "color": [ "negro", "blanco", "rojo"]}
```

También podemos utilizar cualquiera de los operadores vistos hasta el momento, como por ejemplo el operador `$ in`, que tratará los documentos que tienen alguno de los colores que se especifica a continuación:

```
> Db.colorins.find ({color: {$ in: [ "amarillo", "lila"]}})
{ "_id": ObjectId ( "56e1438ff6663c8169030e09"), "color": [ "rojo", "azul", "amarillo"]}
```

O por ejemplo también utilizar **expresiones regulares**:

```
> Db.colorins.find ({color: / bl /})
{ "_id": ObjectId ( "56e1438ff6663c8169030e09"), "color": [ "rojo", "azul", "amarillo"]}
{ "_id": ObjectId ( "56e14398f6663c8169030e0a"), "color": [ "negro", "blanco", "rojo"]}
```

\$ ajo

El operador `$ ajo` lo podemos utilizar cuando queramos seleccionar los documentos que en el array esté **todos** los elementos especificados.

Por ejemplo, vamos a buscar los documento que tienen el color rojo y azul.

```
> Db.colorins.find ({color: {$ all: [ "rojo", "azul"]}})
{ "_id": ObjectId ( "56e1438ff6663c8169030e09"), "color": [ "rojo", "azul", "amarillo"]}
```

subíndice

Si queremos mirar exactamente una determinada posición del array, podemos especificar la posición inmediatamente después de la clave, **separada por un punto**. Recuerde que la primera posición es la `0`. Debemos poner entre comillas la clave y la posición, sino no sabrá encontrarla.

Por ejemplo, buscamos los documentos que tienen el rojo en la primera posición.

```
> Db.colorins.find ({ "color.0": "rojo"})
{ "_id": ObjectId ( "56e1438ff6663c8169030e09"), "color": [ "rojo", "azul", "amarillo"]}
```

Nota

Acceder a una determinada posición es fácil, pero no es tan fácil acceder a una posición calculada, por ejemplo en la última posición. Hace falta conocimientos un poco más avanzados de JavaScript, para poner dentro del `find ()` una función en JavaScript, y actuar dentro de esta.

Únicamente de manera ilustrativa, ponemos aquí la manera de sacar los documentos, el último color de los cuales es el rojo. En ella nos creamos una variable con el último elemento del array (con `pop ()`), y lo comparamos con el color rojo, volviendo true en caso de que sí sean iguales:

```
> Db.colorins.find (function () {var a = this.color.pop (); return (a == "rojo")})
{ "_id": ObjectId ( "56e14398f6663c8169030e0a"), "color": [ "negro", "blanco", "rojo"]}
```

También hay una forma alternativa de hacerlo, que es utilizando el operador `$ where`, que nos permite crear condiciones con sintaxis JavaScript:

```
> Db.colorins.find ({ $ where: "this.color [this.color.length - 1] == 'rojo'" })
{ "_id": ObjectId ( "56e14398f6663c8169030e0a"), "color": [ " negro ", " blanco ", " rojo " ]}
```

\$ size

El operador `$ size` nos servirá para hacer condiciones sobre el número de elementos de un array.

Incorporamos 2 documentos nuevos, con 2 y 4 elementos respectivamente, para poder comprobarlo:

```
> Db.colorins.insert ({color: [ "negro", "blanco"]})
WriteResult ({ "nInserted": 1})
```

```
> Db.colorins.insert ({color: [ "naranja", "gris", "lila", "verde"]})
WriteResult ({ "nInserted": 1})
```



```
> Db.colorins.find ( )
{ "_id": ObjectId ( "56e1438ff6663c8169030e09"), "color": [ "rojo", "azul", "amarillo"] }
{ "_id": ObjectId ( "56e14398f6663c8169030e0a"), "color": [ "negro", "blanco", "rojo"] }
{ "_id": ObjectId ( "56e16972aa3c92aaed389da6"), "color": [ "negro", "blanco"] }
{ "_id": ObjectId ( "56e16990aa3c92aaed389da7"), "color": [ "naranja", "gris", "lila", "verde"] }
```

Ahora vamos a seleccionar los documentos que tienen 4 colores

```
> Db.colorins.find ({color: {$ size: 4}})
{ "_id": ObjectId ( "56e16990aa3c92aaed389da7"), "color": [ "naranja", "gris", "lila", "verde" ] }
```

Nota

El operador **\$ size** sólo admite un valor numérico, y no se pueden concatenar expresiones con otros operadores, como por ejemplo intentar la condición de que el tamaño del array sea menor o igual a un determinado valor. Se puede volver a esquivar la cuestión con el operador **\$ where**, y poner la condición en JavaScript. Así la consulta de los documentos que tienen 3 o menos colores la podríamos sacar de esta manera:

```
> Db.colorins.find ({ $ where: "this.color.length <= 3" })
{ "_id": ObjectId ( "56e1438ff6663c8169030e09"), "color": [ "rojo", "azul", "amarillo" ] }
{ "_id": ObjectId ( " 56e14398f6663c8169030e0a "), " color ": [ " negro ", " blanco ", " rojo " ] }
{ "_id": ObjectId ( " 56e16972aa3c92aaed389da6 "), " color ": [ " negro ", " blanco " ] }
```

\$ slice

El operador **\$ slice** no es un operador que se pueda poner en las condiciones (criterios), sino que servirá para extraer determinados elementos del array, por número de orden de estos elementos en y el array. Sólo lo podremos poner, por tanto, en el segundo parámetro del **find()**.

La sintaxis es:

```
clave: { $ slice: x }
```

Los valores que puede tomar **x** son:

- Números positivos: será el número de elementos del principio (por la izquierda)
- Números negativos: será el número de elementos del final (por la derecha)
- Un array de 2 elementos (**[x, y]**): sacará a partir de la posición **x** (0 es el primero), tantos elementos como indique **y**

Por ejemplo, vamos a sacar los dos primeros colores de cada documento:

```
> Db.colorins.find ({}, {color: { $ slice: 2}})
{ "_id": ObjectId ( "56e1438ff6663c8169030e09"), "color": [ "rojo", "azul"] }
{ "_id": ObjectId ( " 56e14398f6663c8169030e0a "), " color ": [ " negro ", " blanco " ] }
{ "_id": ObjectId ( " 56e16972aa3c92aaed389da6 "), " color ": [ " negro ", " blanco " ] }
{ "_id": ObjectId ( "56e16990aa3c92aaed389da7"), "color": [ "naranja", "gris"] }
```

O sacar el último color:

```
> Db.colorins.find ({}, {color: { $ slice: -1}})
{ "_id": ObjectId ( "56e1438ff6663c8169030e09"), "color": [ "amarillo"] }
{ "_id": ObjectId ( "56e14398f6663c8169030e0a"), "color": [ "rojo"] }
{ "_id": ObjectId ( "56e16972aa3c92aaed389da6"), "color": [ "blanco"] }
{ "_id": ObjectId ( "56e16990aa3c92aaed389da7"), "color": [ "verde"] }
```

O sacar el tercer elemento, tengan los que tengan. Recuerde que el segundo elemento, es el de la posición 2, y queremos sacar 1.

```
> Db.colorins.find ({}, {color: { $ slice: [2,1]}})
{ "_id": ObjectId ( "56e1438ff6663c8169030e09"), "color": [ "amarillo"] }
{ "_id": ObjectId ( " 56e14398f6663c8169030e0a "), " color ": [ " rojo " ] }
{ "_id": ObjectId ( " 56e16972aa3c92aaed389da6 "), " color ": [ ] }
{ "_id": ObjectId ( " 56e16990aa3c92aaed389da7 "), "color": [ "lila"] }
```

Búsquedas en objetos

Para hacer búsquedas en campos que a su vez son objetos (o documentos dentro de documentos, en la terminología de Mongo), sólo tenemos que poner la ruta de las claves separando por medio de puntos, y cuidar de ponerlas entre comillas.

Así, por ejemplo, vamos a hacer una consulta sobre la colección de alumnos, que eran unos documentos en los que había algún campo de tipo objeto.

```
> Db.alumnos.find (). Pretty ( )
{
  "id": ObjectId ( "56debe3017bf4ed437dc77c8"),
  "nombre": "Abel",
  "apellidos": "Bernat Cantera",
```

```

    "edad": 22,
    "dirección": {
      "calle": "Mayor",
      "numero": 7,
      "cp": "12502"
    },
    "nota": [
      9.5,
      9
    ]
  }
}
{
  "_id": ObjectId ( "56dfdbd136d8b095cb6bd57a"),
  "nombre": "Berta",
  "apellidos": "Bernat Cantero"
}

```

Se podrían sacar los documentos (los alumnos) que viven en el código postal 12502. Nos debe salir el único alumno del que tenemos la dirección, que justamente tiene este código postal. Recuerde que en la llave (realmente clau.subclau), debe ir entre comillas. Hemos puesto al final **pretty ()** para una mejor lectura, pero evidentemente no es necesario.

```

> Db.alumnes.find ({ "adreça.cp": "12502"}). Pretty ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8"),
  "nombre": "Abel",
  "apellidos": "Bernat Cantera ",
  "edad": 22,
  "dirección": {
    "calle": " Mayor ",
    "numero": 7,
    "cp": " 12502 "
  },
  "nota": [
    9.5,
    9
  ]
}

```

Y funcionaría igual con cualquier número de subniveles, es decir, documentos que tienen objetos, los cuales tienen objetos, ... Y también con otros tipos de operadores, o expresiones regulares, ...

Por ejemplo, todos los alumnos de Castellón (el código postal debe comenzar por 12 y contener 3 cifras más, es decir, carácter del 0 al 9, y 3 veces.

```

> Db.alumnes.find ({ "adreça.cp": / ^ 12 [0-9] {3} /}). Pretty ()
{
  "_id": ObjectId ( "56debe3017bf4ed437dc77c8"),
  "nombre": "Abel ",
  "apellidos": " Bernat Cantera ",
  "edad": 22,
  "dirección": {
    "calle": " Mayor ",
    "numero": 7,
    "cp": " 12502 "
  },
  "nota": [
    9.5,
    9
  ]
}

```

Limit, Skip y Suerte

Una vez tenemos hecha una consulta, podemos limitar el número de documentos que nos ha de volver, o ordenarlos.

Para ello hay unos métodos que apliquemos al final del **find ()**, es decir, después del **find ()**, separados por un punto.

Lo aplicaremos los libros, que es donde tenemos más documentos. Y no mostramos todos los campos, para una mejor lectura:

```

> Db.libro.find ({}, {titulo: 1, precio: 1, editorial: 1})
{ "_id": "9788408117117", "titulo": "Circo Máximo", "editorial": "Planeta", "precio": 21.75}
{ "_id": "9788401342158", "titulo": "El juego de Ripper", "editorial": "Plaza & Janes",
"precio": 21.75}
{ "_id": "9788496208919", "titulo": "Juego de tronos: Canción de hielo y fuego 1",
"editorial": "Gigamesh", "precio": 9.5}
{ "_id": "9788499088075", "titulo": "La ladrona de libros", "editorial": " Debolsillo ",
"precio": 9.45}
{ "_id": "9788415140054", "titulo": " La princesa de hielo ", "editorial": " Embolsillo ",
"precio": 11}
{ "_id": "9788408113331", "titulo": " Las carreras de Escorpio ", "editorial": " Planeta
", "precio": 17.23}
{ "_id": "9788468738895", "titulo": "Las reglas del juego", "precio": 15.9}

```

límite (n)

Limita el número de documentos devueltos a *n* documentos.

```

> Db.libro.find ({}, {titulo: 1, precio: 1, editorial: 1}). Límite (3)
{ "_id": "9788408117117", "titulo": "Circo Máximo", "editorial": "Planeta", "precio": 21.75}

```

```
{ "id": "9788401342158", "titulo": "El juego de Ripper", "editorial": "Plaza & Janes",
"precio": 21.75}
{ "id": "9788496208919", "titulo": "Juego de tronos: Canción de hielo y fuego 1",
editorial": "Gigamesh", "precio": 9.5}
```

Si el número de documentos que hace la consulta es menor que n , donc se volverán menos. Así por ejemplo, de la editorial Planeta sólo hay dos libros. Aunque ponemos límite (3), se volverán 2.

```
> Db.libro.find ({editorial: "Planeta"}, {titulo: 1, precio: 1, editorial: 1}). Límite (3)
{ "id": "9788408117117", "titulo": "Circo Máximo", "editorial": "Planeta", "precio":
21.75}
{ "id": "9788408113331", "titulo": "Las carreras de Escorpio", "editorial": "Planeta",
"precio": 17.23}
```

skip(n).

Se saltarán los primeros n documentos. Si hubiera menos documentos de los que se saltan, pues no se mostraría ninguna.

```
> Db.libro.find ({}, {titulo: 1, precio: 1, editorial: 1}). Skip (2)
{ "id": "9788496208919", "titulo": "Juego de tronos: Canción de hielo y fuego 1",
editorial": "Gigamesh", "precio": 9.5}
{ "id": "9788499088075", "titulo": "La ladrona de libros", "editorial": "Debolsillo",
"precio": 9.45}
{ "id": "9788415140054", "titulo": "La princesa de hielo", "editorial": "Embolsillo",
"precio": 11}
{ "id": "9788408113331", "titulo": "Las carreras de Escorpio", "editorial": "Planeta",
"precio": 17.23}
{ "id": "9788468738895", "titulo": "Las reglas del juego", "precio": 15.9}
```

suerte().

Sirve para ordenar. Como parámetro se le pasará un objeto JSON con las claves para ordenar, y los valores serán:

- 1: orden ascendente
- -1: orden descendente

Si ponemos más de una clave, se ordenará por el primero, en caso de empate por el segundo, ...

En este ejemplo ordenamos por el precio

```
> Db.libro.find ({}, {titulo: 1, precio: 1, editorial: 1}). Suerte ({precio: 1})
{ "id": "9788499088075", "titulo": "La ladrona de libros", "editorial": "Debolsillo",
"precio": 9.45}
{ "id": "9788496208919", "titulo": "Juego de tronos: Canción de hielo y fuego 1",
editorial": "Gigamesh", "precio": 9.5}
{ "id": "9788415140054", "titulo": "La princesa de hielo", "editorial": "Embolsillo",
"precio": 11}
{ "id": "9788468738895", "titulo": "Las reglas del juego", "precio": 15.9}
{ "id": "9788408113331", "titulo": "Las carreras de Escorpio", "editorial": "Planeta",
"precio": 17.23}
{ "id": "9788408117117", "titulo": "Circo Máximo", "editorial": "Planeta", "precio":
21.75}
{ "id": "9788401342158", "titulo": "El juego de Ripper", "editorial": "Plaza & Janes",
"precio": 21.75}
```

Y como decíamos, se puede poner más de un campo de ordenación. Por ejemplo, para editorial en orden ascendente, y por precio en orden descendente

```
> Db.libro.find ({}, {titulo: 1, precio: 1, editorial: 1}). Suerte ({editorial: 1, precio:
-1})
{ "id": "9788468738895", "titulo": "Las reglas del juego", "precio": 15.9}
{ "id": "9788499088075", "titulo": "La ladrona de libros", "editorial": "Debolsillo",
"precio": 9.45}
{ "id": "9788415140054", "titulo": "La princesa de hielo", "editorial": "Embolsillo",
"precio": 11}
{ "id": "9788496208919", "titulo": "Juego de tronos: Canción de hielo y fuego 1",
editorial": "Gigamesh", "precio": 9.5}
{ "id": "9788408117117", "titulo": "Circo Máximo", "editorial": "Planeta", "precio":
21.75}
{ "id": "9788408113331", "titulo": "Las carreras de Escorpio", "editorial": "Planeta",
"precio": 17.23}
{ "id": "9788401342158", "titulo": "El juego de Ripper", "editorial": "Plaza & Janes",
"precio": 21.75}
```

Observe como el primero es el que no tiene editorial (equivalente a null). Y como hay dos de la editorial Planeta, aparece primero el más caro, y después el más barato.

Y evidentemente, se pueden combinar los métodos límite, skip y suerte.

En este ejemplo vamos a sacar el segundo y tercer libro más caro. Para ello ordenamos por precio de forma descendente, saltamos uno y limitamos a 2. No importa el orden como colocar skip, límite y suerte.

```
> Db.libro.find ({}, {titulo: 1, precio: 1, editorial: 1}). Suerte ({precio: -1}). Skip (1)
.limit (2)
{ "id": "9788401342158", "titulo": "El juego de Ripper", "editorial": "Plaza & Janes",
"precio": 21.75}
{ "id": "9788408113331", "titulo": "Las carreras de Escorpio", "editorial": "Planeta",
"precio": 17.23}
```

3.3.3 - Agregación

La agregación nos permitirá hacer consultas muy avanzadas. Es un proceso un poco complicado pero muy potente. Nos dará una potencia casi como la del SQL cuando empezamos a utilizar el GROUP BY y HAVING.

La técnica que se utiliza es la del *pipeline*, es decir hacer una serie de comandos, cada uno coge los datos que proporciona el anterior ya su vez proporciona los datos al siguiente comando. De esta manera se tratará un conjunto de documentos y se harán "operaciones" sobre ellos secuencialmente en bloques: filtrado, proyección, agrupaciones, ordenación, limitación y *skipping* (saltar algunos).

La sintaxis será:

```
db.colleccion.aggregate ( operador $ match, operador $ project, operador $ group, operador $ suerte, operador $ limit, operador $ skip )
```

El orden de los operadores puede cambiar, pero debemos tener en cuenta que los comandos se ejecutan en el orden en que los ponemos (de izquierda a derecha). Así, por ejemplo, puede ser muy conveniente poner el primer operador el \$ match, que es el de seleccionar documentos, así las demás operaciones se harán sobre menos documentos e irán más rápidas.

Cada parámetro del Aggregate, es decir, cada operador tendrá formato JSON, y por lo tanto siempre será del estilo:

```
{ $ Operador: { clave: valor, ... } }
```

Antes de ver los comandos de agrupación, miramos un ejemplo para poder darnos cuenta de la potencia

Operador \$ match

Servirá para filtrar los documentos. Entonces, la agregación sólo afectará a los documentos seleccionados. Se pueden utilizar todos los operadores que hemos ido estudiando.

El siguiente ejemplo selecciona (para Aggregate, pero como no hacemos más sencillamente selecciona los documentos) los documentos de la editorial Planeta.

```
> Db.libro.aggregate ({ $ match: { editorial: "Planeta" } })
```

En el siguiente ejemplo, además de seleccionar los de la editorial Planeta después aplicamos una proyección sobre los campos título y editorial, para poder visualizar mejor el resultado.

```
> Db.libro.aggregate ({ $ match: { editorial: "Planeta" } }, { $ project: { titulo: 1, editorial: 1 } })
{ "_id": "9788408117117", "titulo": "Circo Máximo", "editorial": "Planeta" }
{ "_id": "9788408113331", "titulo": "Las carreras de Escorpio", "editorial": "Planeta" }
```

Operador \$ project

Nos permite proyectar sobre determinados campos del documento, pero es mucho más completo que en la proyección "normal" que habíamos hecho hasta ahora, ya que permite también renombrar campos, hacer cálculos, etc.

proyección

La manera más sencilla, evidentemente es proyectar sobre algunos campos de los existentes, y el funcionamiento es idéntico al de la otra vez (valores 1 para que aparezcan, 0 para que no aparezcan, por defecto `_id` siempre aparece):

```
> Db.libro.aggregate ({ $ project: { titulo: 1, editorial: 1 } })
{ "_id": "9788408117117", "titulo": "Circo Máximo", "editorial": "Planeta" }
{ "_id": "9788401342158", "titulo": "El juego de Ripper", "editorial": "Plaza & Janes" }
{ "_id": "9788496208919", "titulo": "juego de tronos: Canción de hielo y fuego 1", "editorial": "Gigamesh" }
{ "_id": "9788499088075", "titulo": "La ladrona de libros", "editorial": "Debolsillo" }
{ "_id": "9788415140054", "titulo": "La princesa de hielo", "editorial": "Embolsillo" }
{ "_id": "9788408113331", "titulo": "Las carreras de Escorpio", "editorial": "Planeta" }
{ "_id": "9788468738895", "titulo": "Las reglas del juego" }
```

renombrar

\$ Project también nos permite renombrar campos existentes (después veremos que también cálculos). La manera será poner de este modo:

```
{ $ Project: { "nombre_nuevo": "$ campo_viejo" } }
```

El secreto está en el dólar que va delante del campo viejo, ya que de esta manera nos referimos al valor de este campo. Así por ejemplo renombrar el campo `enstock` a `disponible`, además de sacar el título:

```
> Db.libro.aggregate ({ $ project: { titulo: 1, disponible: "$ enstock" } })
{ "_id": "9788408117117", "titulo": "Circo Máximo", "disponible": true }
```

campos calculados

- **Expresiones matemáticas** : Podremos aplicar fórmulas para sumar (**\$ add**), restar (**\$ sustractivo**), multiplicar (**\$ multiply**), dividir (**\$ divide**) y más cosas (potencia, raíz cuadrada, valor absoluto, módulo, ...). Cada operación tiene su operador que será una palabra precedida por dólar, y con la sintaxis de JSON, donde pondremos los operandos en un array.

- **Expresiones de fechas** : Ya veremos y ya podemos ir intuyendo que muchas agregaciones estarán basadas en el tiempo, para poder hacer consultas de documentos de la semana pasada, o el mes pasado, ... Para poder hacer estas agregaciones, hay un conjunto de expresiones que permiten extraer fácilmente de una fecha su día, mes, año, ... en forma de número

- **\$ substr: [exp, inicio, longitud]** : extrae una subcadena del string del primer parámetro, desde la posición que indica el segundo parámetro (o primer carácter) y tantos caracteres como el tercer parámetro
- **\$ concat: [exp1, exp2, ...]** : concatena las expresiones que hay en el array
- **\$ toLower: exp** y **\$ ToUpper: exp** : convierten la expresión a mayúsculas y minúsculas respectivamente

```
> Db.libro.aggregate ({ $ project: { "Libro": { $ concat: [ "$ titulo", "(", "$ autor",
")"] } } })
{ "_id": " 9788408117117 ", " Libro": " Circo Máximo (Santiago Posteguillo) " }
{ "_id": " 9788401342158 ", " Libro": " El juego de Ripper (Isabel Allende) " }
{ "_id": " 9788496208919 ", " Libro": " Juego de tronos: Canción de hielo y fuego 1
(George RR Martin)" }
{ "_id": "9788499088075", "Libro": "La ladrona de libros (Markus Zusak)" }
{ "_id": "9788415140054", "Libro": "La princesa de hielo (Camilla Läckberg)" }
{ "_id": "9788408113331 ", " Libro": " Las carreras de Escorpio (Maggie Stiefvater) " }
{ "_id": " 9788468738895 ", " Libro": " Las reglas del juego (Anna Casanovas) " }
```

```
> Db.libro.aggregate ({ $ project: { "Libro": { $ concat: [ { $ ToUpper: "$ titulo" }, "(",  
"$ autor", ")"] } } } )  
{ " id": "9788408117117", "Libro": "Circo máximo (Santiago Posteguillo)" }
```

```
{ "_id": "9788401342158", "Libro": "EL JUEGO DE RIPPER (Isabel Allende)" }
{ "_id": "9788496208919", "Libro": "Juego de tronos: Canción de HIELO Y FUEGO 1 (George RR Martin)" }
{ "_id": "9788499088075", "Libro": "LA Ladrón de LIBROS (Markus Zusak)" }
{ "_id": "9788415140054", "Libro": "LA PRINCESA DE HIELO (Camilla Läckberg)" }
{ "_id": "9788408113331", "Libro": "LAS CARRERAS DE ESCORPIO (Maggie Stiefvater)" }
{ "_id": "9788468738895", "Libro": "LAS REGLAS DEL JUEGO (Anna Casanovas)" }
```

Operador \$group

Realiza grupos sobre los documentos seleccionados previamente, para valores iguales del campo o expresiones que determinamos. Posteriormente, con los grupos, podremos realizar operaciones, como sumar o sacar la media de alguna cantidad de los documentos del grupo, o el máximo o mínimo, ...

Para poder agrupar, deberemos definir como `_id` del grupo el campo o campos por los valores de los que queremos agrupar. Por ejemplo, si queremos agrupar los libros para la editorial, tendremos que definir el `_id` del grupo el campo editorial

```
$ group: { "_id": campo o campos }
```

Si agrupamos por un único campo, sencillamente lo ponemos con un dólar frente y entre comillas. Si es más de un campo, los ponemos como un objeto. Por ejemplo, agrupamos por editorial:

```
> Db.libro.aggregate ({ $ group: { "_id": "$ editorial" } })
{ "_id": "Debolsillo" }
{ "_id": null }
{ "_id": "Gigamesh" }
{ "_id": "Embolsillo" }
{ "_id": "Plaza & Janes" }
{ "_id": "Planeta" }
```

Podemos observar como hay algún libro que no tiene editorial.

Ahora agrupamos por año de publicación (el extraeremos del campo `fecha`):

```
> Db.libro.aggregate ({ $ group: { "_id": { "año": { $ year: "$ fecha" } } } })
{ "_id": { "año": 2012 } }
{ "_id": { "año": 2009 } }
{ "_id": { "año": 2011 } }
{ "_id": { "año": 2014 } }
{ "_id": { "año": 2013 } }
```

Y ahora agrupamos por editorial y año de publicación (los dos libros de Planeta son de 2013)

```
> Db.libro.aggregate ({ $ group: { "_id": { "Editorial": "$ editorial", "año": { $ year: "$ fecha" } } } })
{ "_id": { "Editorial": "Embolsillo", "año": 2012 } }
{ "_id": { "Editorial": "Plaza & Janes", "año": 2014 } }
{ "_id": { "Editorial": "Debolsillo", "año": 2009 } }
{ "_id": { "Editorial": "Gigamesh", "año": 2011 } }
{ "_id": { "Editorial": "Plaza & Janes", "año": 2014 } }
{ "_id": { "Editorial": "Planeta", "año": 2013 } }
```

Operadores de agrupación

Nos permitirán hacer alguna operación sobre los documentos del grupo. Se ponen como segundo parámetro del grupo (después de la definición del `_id`).

- **\$ sum: valor** : sumará el valor de todos los documentos del grupo. El valor puede ser un campo numérico, o alguna otra cosa más complicada.
- **\$ avg: valor** : calculará la media de los valores para los documentos del grupo
- **\$ max: valor** : máximo
- **\$ min: valor** : mínimo
- **\$ first: exp** : cogerá el primer valor de la expresión del grupo, ignorando las otras del grupo
- **\$ last: exp** : cogerá el último

Por ejemplo, la suma de los precios de los libros de cada editorial:

```
> Db.libro.aggregate ({ $ group: { "_id": "$ editorial", "suma_preus": { $ sum: "$ precio" } } })
{ "_id": "Debolsillo", "suma_preus": 9.45 }
{ "_id": null, "suma_preus": 15.9 }
{ "_id": "Gigamesh", "suma_preus": 9.5 }
{ "_id": "Embolsillo", "suma_preus": 11 }
{ "_id": "Plaza & Janes", "suma_preus": 21.75 }
{ "_id": "Planeta", "suma_preus": 38.980000000000004 }
```

O la media de los precios de cada año:

```
> Db.libro.aggregate ({ $ group: { "_id": { "año": { $ year: "$ fecha" } }, "media_precios": { $ avg: "$ precio" } } })
{ "_id": { "año": 2012 }, "media_precios": 11 }
{ "_id": { "año": 2009 }, "media_precios": 9.45 }
{ "_id": { "año": 2011 }, "media_precios": 9.5 }
```

```
{ "_id": { "año": 2014}, "media precios": 18.825}
{ "_id": { "año": 2013}, "media precios": 19.490000000000002}
```

Operador \$ suerte

Sirve para ordenar y sigue la misma sintaxis que en las consultas normal (1: orden ascendente; -1: orden descendente). Podremos ordenar los campos normales o por campos calculats.

Por ejemplo ordenamos por la suma de precios de cada editorial:

```
> Db.libro.aggregate ({ $ group: { "_id": "$ editorial", "suma_preus": { $ sum: "$ precio" } } },
{ $ suerte: { suma_preus: 1 } })
{ "_id": " DeboIsillo ", " suma_preus ": 9:45}
{ "_id": " Gigamesh ", " suma_preus ": 9.5}
{ "_id": " Embolsillo ", " suma_preus ": 11}
{ "_id": null, "suma_preus": 15.9}
{ "_id": "Plaza & Janes", "suma_preus": 21.75}
{ "_id": "Planeta", "suma_preus": 38.980000000000004}
```

Y ahora ordenamos de forma descendente por la media de precios de cada año:

```
> Db.libro.aggregate ({ $ group: { "_id": { "año": { $ year: "$ fecha" } } }, "media precios": { $
avg: "$ precio" } } }, { $ suerte: { "media precios": - 1 } })
{ "_id": { "año": 2013}, "media precios": 19.490000000000002}
{ "_id": { "año": 2014}, "media precios ": 18.825}
{ "_id": { "año ": 2012}, " media precios ": 11}
{ "_id": { "año ": 2011}, " media precios ": 9.5}
{ "_id": { "año": 2.009}, "media precios": 9:45}
```

Operador \$ límite

Limita el resultado del Aggregate al número indicado.

Por ejemplo, los tres años de media de precios más cara. Es como el último ejemplo, añadiendo el límite:

```
> Db.libro.aggregate ({ $ group: { "_id": { "año": { $ year: "$ fecha" } } }, "media precios": { $
avg: "$ precio" } } }, { $ suerte: { "media precios": - 1 } }, { $ limite: 3 })
{ "_id": { "año": 2013}, "media precios": 19.490000000000002}
{ "_id": { "año": 2014}, "media precios": 18.825}
{ "_id": { "año": 2012}, "media precios": 11}
```

Operador \$ skip

Salta el número indicado

En el ejemplo anterior, ahora saltamos los 3 primeros:

```
> Db.libro.aggregate ({ $ group: { "_id": { "año": { $ year: "$ fecha" } } }, "media precios": { $
avg: "$ precio" } } }, { $ suerte: { "media precios": - 1 } }, { $ skip: 3 })
{ "_id": { "año": 2011}, "media precios": 9.5}
{ "_id": { "año": 2009}, "media precios": 9:45}
```


3.4 - Conexión desde Java

Para poder conectar desde Java nos será suficiente con un driver, que tendremos que incorporar al proyecto. En el momento de hacer estos apuntes, el último driver disponible es el siguiente:

<https://oss.sonatype.org/content/repositories/releases/org/mongodb/mongo-java-driver/3.2.2/mongo-java-driver-3.2.2.jar>

Para separar las pruebas y ejercicios de la parte de **Redis**, creamos en el proyecto **Tema8** un paquete llamado **provesMongo**.

conexión

La conexión es tan sencilla como el siguiente:

```
MongoClient cono = new MongoClient ( "localhost", 27017);
MongoDatabase bd = cono.getDatabase ( "test");
```

Es decir, obtenemos un objeto **MongoClient** pasándole al constructor la dirección del servidor y el puerto de conexión (que por defecto es 27017).

Posteriormente debemos conectar con la Base de Datos. Ya habíamos comentado en la instalación de Mongo que nosotros sólo utilizaríamos una Base de Datos ya creada llamada **test**. Obtenemos un objeto **MongoDatabase** que hará referencia a la Base de Datos, y es el objeto que utilizaremos a partir de ahora. Evidentemente lo podríamos haber hecho en una única línea.

Si el servidor no lo tenemos en la misma máquina, sólo tendremos que sustituir **localhost** por la dirección donde esté el servidor.

Para cerrar la conexión:

```
con.close ();
```

Inserción de documentos

Desde Java podremos insertar documentos con la misma facilidad que desde la consola. Sólo tendremos que crear un objeto **Documento** de **BSON** (recuerde que es el formato interno de Mongo, absolutamente similar a **JSON**). La manera de ir poniendo parejas clave-valor en este documento es por medio del método **put**. Hacemos un ejemplo muy sencillo donde sencillamente guardamos un documento con una pareja clave-valor de un mensaje:

```
importe org.bson.Document;

importe com.mongodb.MongoClient;
importe com.mongodb.client.MongoDatabase;

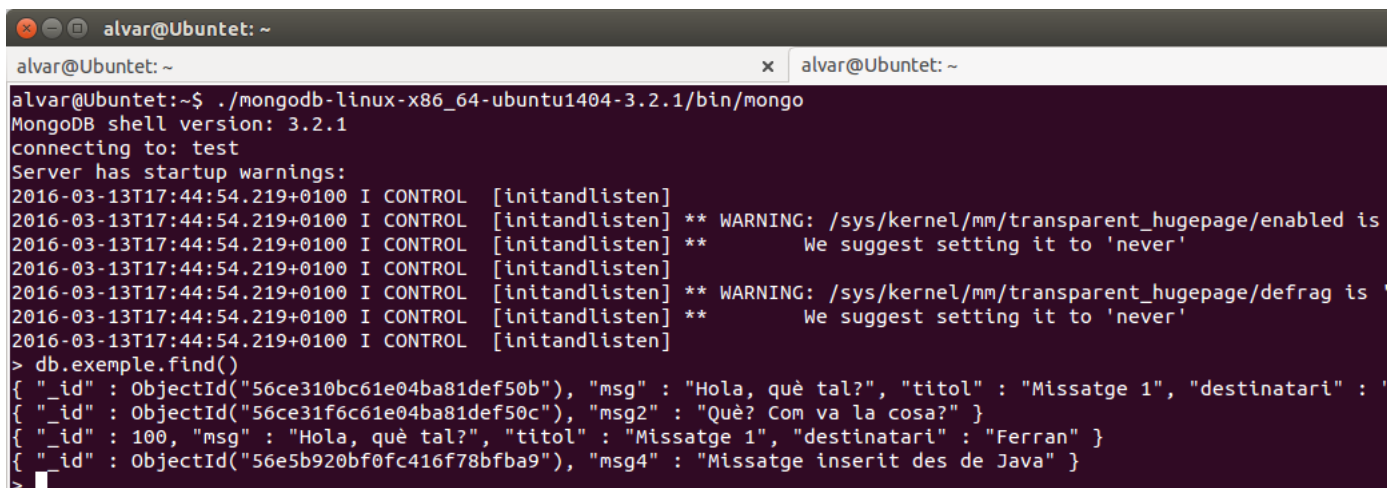
public class Prova1 {

    public static void main (String [] args) {
        MongoClient cono = new MongoClient ( "localhost", 27017);
        MongoDatabase bd = cono.getDatabase ( "test");

        Documento doc = new Documento ();
        doc.put ( "msg4", "Mensaje insertado desde Java");
        bd.getCollection ( "ejemplo").insertOne (doc);

        con.close ();
    }
}
```

Seguramente sacará avisos en la consola, pero sólo son avisos. Podemos comprobar en la terminal como se ha insertado el documento:



```
alvar@Ubuntet:~$ ./mongodb-linux-x86_64-ubuntu1404-3.2.1/bin/mongo
MongoDB shell version: 3.2.1
connecting to: test
Server has startup warnings:
2016-03-13T17:44:54.219+0100 I CONTROL [initandlisten]
2016-03-13T17:44:54.219+0100 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/enabled is
2016-03-13T17:44:54.219+0100 I CONTROL [initandlisten] ** We suggest setting it to 'never'
2016-03-13T17:44:54.219+0100 I CONTROL [initandlisten]
2016-03-13T17:44:54.219+0100 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/defrag is
2016-03-13T17:44:54.219+0100 I CONTROL [initandlisten] ** We suggest setting it to 'never'
2016-03-13T17:44:54.219+0100 I CONTROL [initandlisten]
> db.example.find()
{ "_id" : ObjectId("56ce310bc61e04ba81def50b"), "msg" : "Hola, què tal?", "titol" : "Missatge 1", "destinatari" : "Ferran" }
{ "_id" : ObjectId("56ce31f6c61e04ba81def50c"), "msg2" : "Què? Com va la cosa?" }
{ "_id" : 100, "msg" : "Hola, què tal?", "titol" : "Missatge 1", "destinatari" : "Ferran" }
{ "_id" : ObjectId("56e5b920bf0fc416f78bfa9"), "msg4" : "Missatge inserit des de Java" }
>
```


consultas

Tenemos el método `find ()` para hacer consultas, y le podemos poner un documento como parámetro para seleccionar determinados documentos o sacar determinada información.

```
importe org.bson.Document;

importe com.mongodb.MongoClient;
importe com.mongodb.client.FindIterable;
importe com.mongodb.client.MongoDatabase;

public class prueba2 {

    public static void main (String [] args) {
        MongoClient cono = new MongoClient ( "localhost", 27017);
        MongoDatabase bd = cono.getDatabase ( "test");

        FindIterable <Documento> libros = bd.getCollection ( "libro"). Find ();

        for (Documento libro: libros)
            System.out.println (llibre.get ( "titulo"));

        cono.close ();
    }
}
```

Y como comentábamos podemos poner como parámetros en el `find ()` para seleccionar determinados documentos, ordenar, etc. Sólo tenemos que cuidar que lo tenemos que poner en **JSON** (mejor dicho **BSON**), y por lo tanto tendremos que crear un documento para ello.

```
importe org.bson.Document;

importe com.mongodb.MongoClient;
importe com.mongodb.client.FindIterable;
importe com.mongodb.client.MongoDatabase;

public class Prova3 {

    public static void main (String [] args) {
        MongoClient cono = new MongoClient ( "localhost", 27017);
        MongoDatabase bd = cono.getDatabase ( "test");

        Documento ordenar = new Documento ();
        ordenar.put ( "precio", -1);

        FindIterable <Documento> libros = bd.getCollection ( "libro"). Find (). Suerte
(ordonar);

        for (Documento libro: libros)
            System.out.println ( "Título:" + llibre.get ( "titulo") + ". Precio:" +
llibre.get ( "precio"));

        cono.close ();
    }
}
```



ejercicio 8.1

Sobre la Base de Datos del Servidor local hacer las siguientes operaciones, tanto para guardar una serie de datos, como para recuperarlas. Siempre pondremos a las llaves el **prefijo 999x_**, donde deberá sustituir 999 por las 3 últimas cifras de su DNI, y la x por la letra del NIF.

Copia en un único archivo de texto, de forma numerada. Es este archivo el que deberás subir.

- a. Crea la clave **999x_ Nombre** con tu nombre
- b. Crea la clave **999x_ Apellidos** con tus apellidos. Una de las dos menos, nombre o apellidos, debe constar de más de una palabra.
- c. Mostrar todas las claves tus, y únicamente tus.
- d. Crea la clave **999x_ dirección**, de tipo Hash, con los subcampos **calle**, **numero** y **cp**. No importa que los datos sean falsas. Puedes hacerlo en una sentencia o en más de una.
- e. Añadir a lo anterior el subcampo **población**
- f. Ver toda la información de su dirección (sólo la información, no las subclaves)
- g. Crea la clave **999x_ Moduls_DAW**, de tipo Set, con todos los módulos del DAW, que son: SI, BD, PR, LM, ED, FOL, DWEC, DWES, DAW, DIW, EIE, PROY y FCT. Puedes hacerlo en una o en más de una sentencia.
- h. Crea la clave **999x_ Moduls_meus**, de tipo Set, con todos los módulos a los que estás matriculado. Puedes hacerlo en una o en más de una sentencia.
- i. Guarda en la clave **999x_ Moduls_altres** los módulos en los que no estás matriculado actualmente. Debe ser por medio de operaciones de conjuntos. Puedes comprobar que el resultado es correcto con **smembers**
- j. Crea una lista con el nombre **999x_ Notes_BD** con la nota de 4 ejercicios de BD. Las notas serán: 7, 9, 6, 10. Deben quedar en este orden (no en orden inverso)