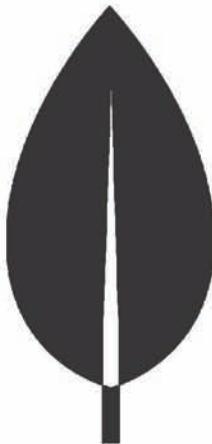


TECNOLOGÍA

# ANTONIO SARASA

## INTRODUCCIÓN A LAS BASES DE DATOS NoSQL USANDO **MongoDB**



EDITORIAL UOC



# **Introducción a las bases de datos NoSQL usando MongoDB**

Antonio Sarasa

Diseño de la colección: Editorial UOC  
Diseño de la cubierta: Natàlia Serrano

Primera edición en lengua castellana: abril 2016  
Primera edición digital: mayo 2016

© Antonio Sarasa, del texto

© Editorial UOC (Oberta UOC Publishing, SL), de esta edición, 2016  
Rambla del Poblenou, 156  
08018 Barcelona  
<http://www.editorialuoc.com>

Realización editorial: Oberta UOC Publishing, SL  
Maquetación: Natàlia Serrano

ISBN: 978-84-9116-250-6

Ninguna parte de esta publicación, incluyendo el diseño general y de la cubierta, no puede ser copiada, reproducida, almacenada o transmitida de ninguna forma ni por ningún medio, ya sea eléctrico, químico, mecánico, óptico, de grabación, de fotocopia o por otros métodos, sin la autorización previa por escrito de los titulares del *copyright*.

**Antonio Sarasa**

Estudió la licenciatura en C.C. Matemáticas en la especialidad de C.C. Computación en la Facultad de Matemáticas de la Universidad Complutense de Madrid. Posteriormente estudió ingenierías técnicas en Informática de Gestión y de Sistemas en la UNED, e Ingeniería en Informática y graduado en Ingeniería Informática en la UOC. Es doctor en Informática por la Universidad Complutense de Madrid.

Actualmente, es profesor contratado doctor en el Departamento de Sistemas Informáticos y Computación de la Facultad de Informática de la Universidad Complutense de Madrid. Asimismo, ha sido profesor tutor de asignaturas de las carreras de Informática en varios centros asociados de la UNED, y es consultor de la asignatura de «Bases de datos NoSQL» del máster de Business Intelligent de la UOC.

Sus áreas de investigación se centran en la aplicación de las técnicas de procesadores de lenguajes y compiladores al desarrollo de aplicaciones. Asimismo ha trabajado en temas de patrimonio digital y eLearning, en los que se destaca su participación en el proyecto estatal de la red de repositorios de material educativo digital Agrega. Ha realizado más de cincuenta publicaciones en congresos y revistas nacionales e internacionales.

Es miembro de varios comités de estandarización de AENOR.



*Este libro está dedicado a mi mujer, a mis hijos, a mis padres,  
a mis hermanas y al resto de mi familia.*

*También me gustaría agradecer a los responsables del máster Business  
Intelligent de la UOC, y en particular a la profesora responsable  
de la asignatura de «Bases de datos NoSQL» de la UOC por la  
oportunidad que me dieron de impartir esta asignatura.*





# Índice

<b>Prólogo .....</b>	15
<b>Capítulo I. Introducción a las bases de datos NoSQL ....</b>	17
1. Introducción.....	17
2. El papel de las base de datos en las aplicaciones software ...	18
3. Las limitaciones de las bases de datos relacionales .....	20
4. Bases de datos NoSQL .....	24
5. Modelos de bases de datos NoSQL orientados hacia agregados .....	28
6. El modelos de distribución de datos de las bases de datos NoSQL .....	34
7. La consistencia de los datos en las bases de datos NoSQL.....	35
8. El teorema CAP .....	41
Bibliografía .....	44
<b>Capítulo II. Conceptos básicos .....</b>	45
1. Introducción.....	45
2. Documentos.....	46
3. Tipos de datos .....	47
4. Colecciones .....	51
5. Bases de datos.....	54
6. La Shell de comandos de MongoDB.....	56
7. Operaciones básicas en la Shell .....	58
8. Observaciones .....	66
Bibliografía .....	68

<b>Capítulo III. Operaciones CRUD .....</b>	69
1. Introducción.....	69
2. Inserción de documentos.....	69
3. Borrado de documentos .....	71
4. Actualización de documentos .....	72
5. Modificadores de los arrays .....	80
6. Upsert.....	90
7. Actualización de múltiples documentos.....	94
8. Eficiencia de las modificaciones .....	95
9. Ejercicios propuestos .....	100
Bibliografía .....	103
 <b>Capítulo IV. Queryng en MongoDB.....</b>	105
1. Introducción.....	105
2. El método <code>find()</code> .....	105
3. Operadores condicionales: <code>\$lt</code> , <code>\$lte</code> , <code>\$gt</code> , <code>\$gte</code> .....	109
4. Los operadores <code>\$not</code> , <code>\$ and</code> y <code>\$or</code> .....	111
5. Expresiones regulares .....	114
6. Consultas sobre arrays.....	115
7. Consultas sobre documentos embebidos.....	122
8. Consultas <code>\$where</code> .....	124
9. Configuración de las consultas .....	126
10. Otras características .....	131
11. Ejercicios propuestos .....	134
Bibliografía .....	139
 <b>Capítulo V. La framework de agregación.....</b>	141
1. Introducción.....	141
2. Agregación mediante tuberías .....	141
3. Map Reduce.....	149
4. Operaciones de propósito único .....	154

5. Ejercicios propuestos .....	157
Bibliografía .....	160
<b>Capítulo VI. Indexación.....</b>	<b>161</b>
1. Introducción.....	161
2. Los índices en MongoDB.....	162
3. Creación de un índice simple .....	163
4. Creación de un índice compuesto .....	165
5. Índices de subdocumentos .....	166
6. Índices creados manualmente .....	167
7. Especificar opciones sobre los índices .....	169
8. Eliminación de índices .....	172
9. Reindexación de una colección.....	173
10. Selección de índices .....	173
11. El comando hint ().....	174
12. Optimización del almacenamiento de pequeños objetos.....	177
13. Índices geoespaciales .....	179
Bibliografía .....	183
<b>Capítulo VII. Replicación.....</b>	<b>185</b>
1. Introducción.....	185
2. Replica sets .....	186
3. El proceso de replicación .....	186
4. Configuración de nodos secundarios.....	187
5. Recuperación ante un fallo del primario .....	189
Bibliografía .....	204
<b>Capítulo VIII. Sharding .....</b>	<b>205</b>
1. Introducción.....	205
2. Sharding .....	206

3. Clave de sharding .....	208
4. Configuración de la distribución de datos .....	211
5. Creación de un clúster.....	213
6. Adición de shards a un clúster.....	219
7. Eliminación de un shard de un clúster.....	223
8. Estado de un clúster .....	226
9. Conversión de un replica set en un sharded clúster replicado .....	229
Bibliografía .....	237
 <b>Capítulo IX. Optimización.....</b>	 239
1. Introducción.....	239
2. MongoDB Profiler.....	240
3. Explain() .....	246
4. El valor queryPlanner.....	248
5. El valor executionStats .....	250
6. Server Info .....	251
7. El optimizador de consultas.....	252
8. Index Filter.....	254
9. Observaciones .....	255
10. Índice que cubre una consulta .....	257
11. Selectividad de las consultas.....	259
12. Ejercicios propuestos .....	260
Bibliografía .....	262
 <b>Capítulo X. GridFS .....</b>	 263
1. GridFS.....	263
2. Mongofile .....	264
3. Colecciones GridFS .....	266
4. Índices GridFS.....	268
5. Carga de un documento en GridFS.....	269

6. Búsqueda y recuperación de archivos .....	271
7. Eliminación de archivos .....	271
Bibliografía .....	272
 <b>Capítulo XI. Operaciones de administración.....</b>	 273
1. Introducción.....	273
2. Realizar backups del servidor.....	273
3. Restauración de una copia de seguridad.....	277
4. Exportación de datos .....	280
5. Importación de datos .....	283
6. Creación de usuarios.....	287
7. Otras operaciones de administración.....	291
8. Ejercicios propuestos .....	294
Bibliografía .....	297
 <b>Anexo I. Instalación y puesta en marcha de MongoDB....</b>	 299



## Prólogo

En las últimas décadas, el paradigma imperante en el mundo de las bases de datos ha sido el modelo relacional. En este sentido, en muchos de los sistemas de información que utilizamos día a día, la información se almacena en bases de datos relacionales. Un elemento clave del éxito de este paradigma ha sido disponer del lenguaje estándar de consultas SQL. Sin embargo, el surgimiento del denominado «Big Data» ha traído consigo la aparición de nuevas necesidades y formas de procesamiento y almacenamiento de la información. En este nuevo contexto, lo más frecuente es tratar con datos no estructurados o semiestructurados, en cantidades gigantescas, en las que se necesita realizar procesamientos rápidos y donde características como la consistencia de los datos deja de ser un requisito rígido y puede ser flexibilizado o donde la existencia de un esquema fijo para el modelo de datos deja de ser necesario. Para dar respuesta a estas necesidades nuevas, el procesamiento distribuido se convierte en una característica de primer orden, y surgen conceptos tales como los clústeres, la replicación de datos y el escalado horizontal. Aunque es cierto que el modelo relacional puede adaptarse a estas nuevas necesidades, sin embargo, no es la mejor solución. Es en este nuevo contexto donde surge una nueva clase de bases de datos que no siguen el modelo relacional y que tratan de cubrir las nuevas necesidades surgidas, que se han denominado bases de datos NoSQL. Este tipo de bases de datos no es homogéneo y es posible encontrar distintas aproximaciones a los mismos problemas. En particular se encuentra la aproximación

documental, en la que la información se almacena en un tipo de entidades denominadas «documentos» cuya estructura es accesible desde diferentes, y donde no existe un esquema fijo para la estructuración de la información. Dentro de esta aproximación documental, destaca MongoDB.

Este manual pretende realizar una introducción a los conceptos principales en los que se basan las bases de datos NoSQL e ilustrarlos en el caso concreto de MongoDB. Para ello, el libro se estructura en un primer capítulo introductorio a los conceptos genéricos de las bases de datos NoSQL, y el resto de los capítulos están dedicados íntegramente a exponer las diferentes funcionalidades y características de MongoDB. Algunos de los capítulos sobre MongoDB van acompañados de una colección de ejercicios propuestos que permiten al lector asentar los conceptos vistos. Asimismo las soluciones a todos los ejercicios propuestos se pueden encontrar en la dirección <http://www.editorialuoc.cat/introduccion-a-las-bases-de-datos-nosql-usando-mongodb>.

Espero que el libro sea ameno para el lector, y que cumpla su objetivo de servir de medio de acercamiento a este nuevo ámbito de las bases de datos NoSQL y en particular a MongoDB.

*Madrid, 4 de diciembre de 2015*

**Antonio Sarasa Cabezuelo**

## Capítulo I

# Introducción a las bases de datos NoSQL

### 1. Introducción

En este capítulo se hace una revisión de los principales conceptos relativos a las bases de datos NoSQL. Se comienza justificando el papel que desempeñan las bases de datos en el ámbito de los sistemas de información actuales. A continuación se reflexiona sobre las limitaciones que presentan las bases de datos relacionales. Estas han sido la solución preferida para implementar la persistencia de los sistemas de información en las últimas décadas, y han conseguido unos resultados con gran éxito. Las limitaciones discutidas sirven de punto de partida para introducir las bases de datos NoSQL y las razones que provocaron su aparición, y comentar algunas de sus características comunes. A continuación se enumeran las principales familias de modelos de bases de datos NoSQL, centrándose en los modelos orientados a agregados. Por último se revisan brevemente los conceptos de distribución y consistencia en el contexto de las bases de datos NoSQL, y se presenta el teorema CAP.

## 2. El papel de las base de datos en las aplicaciones software

Se pueden diferenciar dos usos muy comunes en las bases de datos, como elemento de integración o bien como una aplicación única.

En el primer caso, las bases de datos permiten integrar múltiples aplicaciones, generalmente realizadas por diferentes equipos, que almacenan sus datos en una base de datos común. Esto mejora la comunicación, puesto que todas las aplicaciones operan sobre un conjunto consistente de datos persistentes. También existen desventajas para esta aproximación, dado que una estructura que está diseñada para integrar muchas aplicaciones acaba siendo muy compleja. Por ejemplo, en este contexto, cuando una aplicación quiere hacer cambios sobre los datos almacenados, entonces necesita coordinarse con el resto de las aplicaciones que usan la base de datos. Además, las diferentes aplicaciones tienen diferentes estructuras y necesidades, así, por ejemplo, una aplicación puede requerir un índice, pero este índice puede que cause un problema para realizar inserciones de datos a otra aplicación. Por otra parte, el hecho de que cada aplicación normalmente es implementada por un equipo separado también tiene como consecuencia que la base de datos no pueda confiar a las aplicaciones la actualización de los datos de forma que se preserve la integridad de la base de datos, de forma que es la propia base de datos la responsable de asegurarla.

Otro caso de uso es como una única aplicación, de manera que hay una única aplicación que accede a la base de datos. En este caso, solo hay un equipo de desarrollo que debe conocer la estructura de la base de datos, lo que hace más fácil mantener y realizar cambios sobre la misma. Así mismo, dado que solo

existe una aplicación que interactúe con la base de datos, la integración de ambas puede realizarse en el propio código de la aplicación. Así, mismo, cualquier tema de interoperabilidad puede ser gestionado desde las interfaces de la aplicación, lo que permite una mejor interacción con los protocolos y llevar a cabo cambios en los mismos. Por ejemplo, en los últimos tiempos se ha utilizado una arquitectura orientada a servicios que se basa en servicios web. En estos casos, la comunicación se realiza vía HTTP, lo que permite un mecanismo de comunicación ampliamente utilizado. Un aspecto interesante en el uso de los servicios web como mecanismo de integración resulta en ser más flexible para la estructura de datos que es intercambiada. Si se comunica usando SQL, los datos deben estar estructurados como relaciones. Sin embargo, con un servicio web, se pueden usar estructuras de datos más ricas con registros anidados y listas, que normalmente están representados como documentos XML o JSON. En general, en las comunicaciones remotas, interesa reducir el número de comunicaciones envueltas en cada interacción, de manera que es útil poder usar una estructura de información rica en cada petición o respuesta. Hay que observar que en este contexto existe libertad para elegir la base de datos, dado que existe un desacoplamiento entre la base de datos interna y los servicios que se usan para interaccionar con el mundo exterior (no es necesario que el resto de las aplicaciones conozcan cómo se almacenan los datos).

### 3. Las limitaciones de las bases de datos relacionales

Las bases de datos relacionales han sido durante décadas el modelo de persistencia más utilizado en la mayoría de las aplicaciones software. En este sentido se van a revisar las ventajas y desventajas que presentan. Desde el punto de vista de los beneficios que aportan las bases de datos relacionales, se puede destacar:

- *Persistencia de datos.* Una de las misiones más importantes de las bases de datos es mantener cantidades enormes de datos persistentes. En la mayoría de las aplicaciones es necesario disponer de un almacén de datos que actúe de backup. Estos almacenes se pueden estructurar de muchas formas, como, por ejemplo, un archivo del sistema de archivos del sistema operativo. Sin embargo, la forma preferida es una base de datos, pues ofrece más flexibilidad que el sistema de archivos cuando almacena grandes cantidades de datos a la hora de permitir obtener conjuntos de información de una forma rápida y fácil.
- *Concurrencia.* En general, en las aplicaciones normales suele haber muchas personas que están trabajando sobre los mismos datos, y puede que estén modificando los mismos datos. Esta situación obliga a coordinar las diferentes interacciones sobre los mismos datos para evitar inconsistencias en estos. La gestión directa de la concurrencia es complicada y existen muchas posibilidades de cometer errores. Las bases de datos ayudan a controlar todos los accesos a los datos usando transacciones. Aunque no es un sistema que no pueda dar lugar a errores (pueden ocurrir errores en las transacciones y enton-

ces hay que retroceder la transacción), sin embargo, sí permite gestionar la complejidad que supone la concurrencia.

- *Integración.* Con frecuencia, las aplicaciones requieren interactuar con otras aplicaciones de forma que comparten los mismos datos, y unas y otras utilizan los datos que han sido modificados por el resto de las aplicaciones. Una forma de implementar esta interacción consiste en usar una integración basada en compartir una base de datos, es decir, las diferentes aplicaciones que interactúan almacenan sus datos en una única base de datos. Usar una única base de datos permite que todas las aplicaciones usen los datos de las restantes de una forma fácil. Además, el sistema de control de la concurrencia de las bases de datos permite que existan múltiples aplicaciones usando la misma base de datos.
- *Modelo estándar.* Las bases de datos relacionales han tenido éxito debido a que proporcionan los beneficios clave de constituir un sistema estándar. Una persona puede aprender unos conocimientos generales sobre el modelo relacional que son comunes a las diferentes bases de datos relacionales y podrá aplicarlos en cualquier caso particular. Es decir, los mecanismos nucleares son los mismos y las diferencias serán, por ejemplo, la existencia de dialectos de SQL o pequeñas diferencias de funcionamiento de las transacciones, pero esencialmente el funcionamiento es el mismo.

Sin embargo, las bases de datos relacionales no son perfectas. Uno de los principales problemas hace referencia a la diferencia entre el modelo relacional y las estructuras de datos en memoria que algunos autores denominan «**impedancia**». El modelo relacional organiza los datos en una estructura de tablas y filas (también denominado «relaciones y tuplas»). En el modelo relacional, una

tupla es un conjunto de pares nombre-valor y una relación es un conjunto de tuplas. Todas las operaciones en SQL consumen y retornan relaciones de acuerdo con el álgebra relacional. Este uso de las relaciones proporciona cierta elegancia y simplicidad, pero introduce un conjunto de limitaciones. En particular, los valores en una tupla relacional tienen que ser simples (no pueden tener estructura tal como un registro anidado o una lista). Esta limitación no se da en las estructuras de datos en memoria, donde existe un conjunto de tipos de estructuras mucho más ricas que las relaciones. Como consecuencia, si se quiere usar una estructura de datos más compleja que la relación, hay que traducirla a una representación relacional para poderla almacenar en disco.

Este problema llevó a pensar que las bases de datos relacionales serían sustituidas por bases de datos que replicaran las estructuras de datos en memoria. Y en este sentido la aparición de los lenguajes orientados a objetos, y junto con ellos de las bases de datos orientadas a objetos, hizo pensar que estas últimas acabarían ocupando el lugar de las bases de datos relacionales. Así, los lenguajes de programación orientados a objetos triunfaron, sin embargo, las bases de datos orientadas a objetos quedaron en el olvido. La razón del éxito de las bases de datos relacionales se ha debido a que constituyen un mecanismo de integración muy sólido basado en la existencia de un lenguaje estándar de manipulación de datos, el lenguaje SQL, y en la existencia de dos roles en los equipos de desarrollo: desarrolladores y los administrados de las bases de datos, que permiten llevar a cabo esta integración y coordinación. Hay que observar por otro lado que en los últimos tiempos ha aparecido un conjunto de frameworks que mapean información del mundo relacional al mundo de la orientación de objetos aliviando el problema de la impedancia. Sin embargo, el mapeo tiene también sus propios problemas cuando al hacer uso

del mismo se obvia la existencia de una base de datos y se llevan a cabo consultas que hacen descender de una forma importante el rendimiento de la base de datos.

Otra limitación presente en las bases de datos relacionales es su integración en un **clúster**. Un clúster es un conjunto de máquinas que permite implementar soluciones para escalar una aplicación (escalado horizontal). Es una solución más resistente que disponer de una única máquina más potente (escalado vertical), dado que, cuando se produce un fallo en un clúster, la aplicación podrá continuar funcionando, proporcionando, así, una alta fiabilidad. Desde el punto de vista de las bases de datos relacionales, existe un problema, dado que estas no están diseñadas para ejecutarse sobre un clúster. Existen algunas bases de datos relacionales que funcionan sobre el concepto de subsistema de disco compartido, usando un sistema de archivos compatible con un clúster que escribe sobre un subsistema de disco de alta disponibilidad, pero eso significa que el clúster tiene el subsistema de disco como un único punto de fallo. Por otra parte, las bases de datos relacionales podrían ejecutarse en servidores separados para diferentes conjuntos de datos, implementando lo que se denomina «una solución de **sharding** de la base de datos». Sin embargo, esta solución plantea el problema de que todo el sharding tiene que ser controlado por la aplicación, por lo que deberá mantener el control sobre qué servidor debe ser preguntado para cada tipo de datos. Además, se pierden determinadas características acerca de las consultas que se pueden realizar, integridad referencial, transacciones o control de la consistencia. Otro problema adicional es el coste: normalmente las bases de datos relacionales están pensadas para ejecutarse en un único servidor, de manera que si se tienen que ejecutarse en un clúster requieren de varias instancias de la base de datos, lo que aumenta, así, el coste económico.

## 4. Bases de datos NoSQL

Debido a las limitaciones mencionadas en la sección anterior, algunas empresas decidieron usar alternativas a las bases de datos relacionales. Este fue el caso de Google y Amazon, que desarrollaron sistemas de almacenamiento basado en el uso de clústeres: las Big Tables de Google y Dynamo de Amazon. Estos sistemas permitían gestionar grandes cantidades de datos en ambientes distribuidos y llevar a cabo su procesamiento, por lo que se consideran el origen de las denominadas «**bases de datos NoSQL**». Aunque muchas empresas no gestionaban las escalas de datos de las empresas antes mencionadas, sin embargo, muchas de ellas empezaron a diseñar sus aplicaciones para ejecutarse en estos ambientes distribuidos y usar este tipo de bases de datos al descubrir las ventajas que ofrecían:

- *Productividad del desarrollo de aplicaciones.* En los desarrollos de muchas aplicaciones se invierte un gran esfuerzo en realizar mapeos entre las estructuras de datos que se usan en memoria y el modelo relacional. Las bases de datos NoSQL proporcionan un modelo de datos que encaja mejor con las necesidades de las aplicaciones simplificando así la interacción, lo que resulta en tener que codificar, depurar y evolucionar menos las aplicaciones.
- *Datos a gran escala.* Las organizaciones han encontrado muy valiosa la posibilidad de tener muchos datos y procesarlos rápidamente. En general, es caro (en el caso de ser posible) hacerlo con una base de datos relacional. La primera razón es que las bases de datos relacionales están diseñadas para ejecutarse en una única máquina, y por otro lado es mucho más barato ejecutar muchos datos y procesarlos si se encuen-

tran cargados sobre clústeres de muchas maquinas pero más pequeñas y baratas. La mayoría de las bases de datos NoSQL están diseñadas para ejecutarse sobre clústeres, por lo que encajan mejor para estas situaciones.

Algunas de las características compartidas por las bases de datos NoSQL son:

- No usan SQL como lenguaje de consultas, sin embargo, algunas de ellas utilizan lenguajes de consultas similares a SQL, tales como CQL en Cassandra.
- En general se trata de proyectos de código abierto.
- Muchas de las bases de datos NoSQL nacieron por la necesidad de ejecutarse sobre clúster, lo que ha influido sobre su modelo de datos como su aproximación acerca de la consistencia. Las bases de datos relacionales usan transacciones ACID para manejar la consistencia de la base de datos, sin embargo, esto choca con un entorno de clúster, de manera que ofrecen diversas opciones para implementar la consistencia y la distribución. Sin embargo, no todas las bases de datos NoSQL están orientadas a correr sobre clúster.
- Las bases de datos NoSQL no tienen un esquema fijo.

De todas las características sobre las bases de datos NoSQL, destaca la no existencia de un esquema. Cuando se van almacenar datos en una base de datos relacional, lo primero que hay que hacer es definir un esquema que indique que tablas existen, qué columnas tiene cada tabla y qué tipo de datos tiene cada columna. Sin embargo, las bases de datos NoSQL operan sin esquema, lo que permite añadir libremente campos a los registros de la base de datos sin tener que redefinir la estructura. Esto es particular-

mente útil cuando se trata con campos de datos personalizados y no uniformes (datos donde cada registro tiene un conjunto de campos diferentes) que fuerzan a las base de datos relacionales a usar campos que son difíciles de procesar y entender, dado que un esquema relacional pondrá todas las filas de una tabla en un esquema rígido,<sup>1</sup> lo que se complica cuando se tienen diferentes tipos de datos en diferentes filas, de manera que al final se tienen muchas columnas que son nulas (una tabla dispersa) o bien columnas que no tienen significado. Cuando no se usa esquema, se evita esto, dado que se permite que cada registro pueda contener solo aquello que necesita. Los defensores de no usar esquemas apuntan a este grado de libertad y flexibilidad. Con un esquema es necesario saber antes lo que se necesita almacenar, lo cual en ocasiones puede no conocerse. Sin embargo, cuando no se tiene que cumplir un esquema, se puede almacenar fácilmente lo que se necesite, permitiendo cambiar<sup>2</sup> los datos almacenados según se va conociendo más acerca del proyecto. También, si hay información que no es necesaria seguir almacenando, se puede dejar de almacenar sin preocuparse por perder datos antiguos, como si se borrasen columnas en un esquema relacional. En este

---

1 Aunque se critica a los esquemas relacionales por tener que definir a priori los esquemas y ser inflexibles, esto no es del todo cierto, puesto que los esquemas pueden ser cambiados en cualquier momento con determinados comandos de SQL. Así, por ejemplo, se pueden crear nuevas columnas para almacenar datos no uniformes. Sin embargo, en muchas ocasiones la no uniformidad en los datos es una buena razón para usar una base de datos sin esquema. En este sentido, la no existencia de esquema tiene un gran impacto sobre cambios en la estructura a lo largo del tiempo, especialmente en datos más uniformes.

2 Aunque no se haga habitualmente, los cambios en un esquema de bases de datos relacionales pueden ser hechos de forma controlada, y de forma similar se debe controlar cuándo se cambia la forma en que se almacenan los datos en una base de datos sin esquema de forma que sea fácil acceder tanto a los datos nuevos como a los antiguos.

sentido es muy interesante usar soluciones sin esquemas, pero también tienen desventajas. Muchas veces con los datos se hacen procesamientos en los que es necesario conocer el nombre de los campos, el tipo de los datos que tiene cada campo, etc., de manera que la mayoría de las veces los programas que acceden a los datos descansan sobre alguna forma implícita de esquema que asume que ciertos nombres de campos llevan determinados datos de un tipo de datos con un cierto significado. Así, aunque la base de datos no tenga esquemas, existe un esquema implícito, que es el conjunto de suposiciones acerca de la estructura de los datos en el código que manipula los mismos. Tener el esquema implícito en el código de la aplicación produce algunos problemas, como, por ejemplo, para saber qué datos están presentes, hay que consultar el código de la aplicación, de forma que si está bien estructurado será fácil de deducir el esquema, de lo contrario, puede convertirse en una tarea tediosa. Por otro lado, la base de datos permanece ignorante del esquema y no puede usarlo para decidir cómo almacenar y recuperar datos de manera eficiente, no puede aplicar sus propias validaciones sobre los datos de manera que se asegure que diferentes aplicaciones no manipulan datos de una forma inconsistente. Estas son algunas de las razones por las que las bases relationales han fijado sus esquemas, y el valor que tienen los mismos. Esencialmente, las bases de datos sin esquemas cambian el esquema dentro del código de la aplicación que lo accede. Esto se convierte en problemático si existen múltiples aplicaciones desarrolladas por diferentes personas que acceden a la misma base de datos. Estos problemas pueden ser mitigados bien encapsulando toda la interacción de la base de datos en una única aplicación e integrarlo con otras aplicaciones usando servicios web o bien estableciendo claramente diferentes puntos de acceso a los datos almacenados por parte de las aplicaciones.

## 5. Modelos de bases de datos NoSQL orientados hacia agregados

Aunque resulta complicado realizar una categorización de las bases de datos NoSQL puesto que en muchos casos algunas bases de datos comparten características de varias familias, sin embargo, de acuerdo con el modelo de datos se pueden distinguir:<sup>3</sup>

- Bases de datos clave-valor: Riak, Redis, Dynamo, Voldemort.
- Bases de datos orientadas a documento: MongoDB, CouchDB.
- Bases de datos basadas en columnas: Cassandra, Hypertable, HBase, SimpleDB
- Bases de datos de grafos: Neo4J, Infinite Graph.

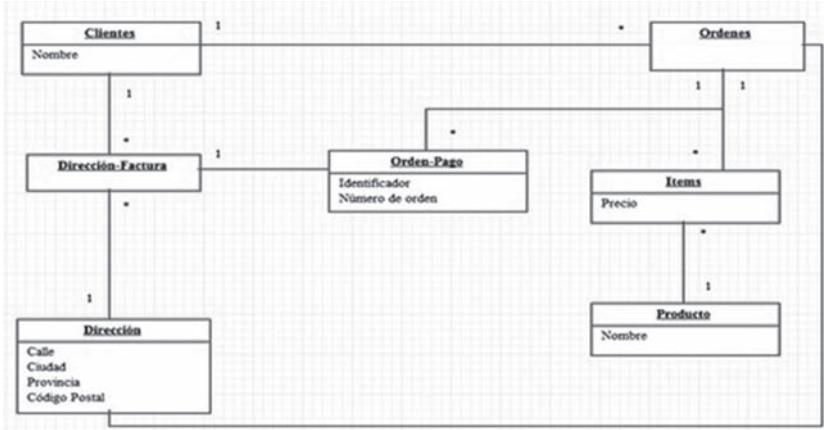
Las tres primeras familias de bases de datos NoSQL mencionadas comparten una característica común en sus modelos de datos y se trata de modelos orientados hacia agregados. En el modelo relacional se toma la información que se quiere almacenar y se divide en tuplas (filas). Una tupla es una estructura de datos limitada, dado que captura un conjunto de valores de manera que no se puede anidar una tupla dentro de otra para conseguir registros anidados, ni se puede poner una lista de valores o tuplas dentro de otra. Esta simplicidad permite pensar cuando se opera que se toman tuplas y se retornan tuplas. Sin embargo, en la orientación agregada hay un cambio de aproximación, y se parte del hecho de que a veces interesa operar con estructuras más complejas que un conjunto de tuplas tales

---

<sup>3</sup> Hay que observar que esta clasificación no es exhaustiva, y se puede encontrar una clasificación más detallada en la siguiente dirección: <http://nosql-database.org>.

como un registro complejo que puede contener listas y otras estructuras de registro que están anidadas dentro del mismo. Estos registros complejos se van a denominar **«agregados»**, y van a representar un conjunto de datos relacionados que son tratados como una unidad. En particular son una unidad para la manipulación de datos y para la gestión de la consistencia, es decir, se actualizarán agregados con operaciones atómicas y se interaccionarán con la base de datos en términos de agregados. También el uso de agregados facilita que las bases de datos puedan operar sobre clústeres, pues el concepto de agregado se convierte en una unidad natural para la replicación y la distribución. Asimismo desde el punto de vista del programador, es más fácil operar con agregados, pues habitualmente operan con datos a través de estructuras agregadas.

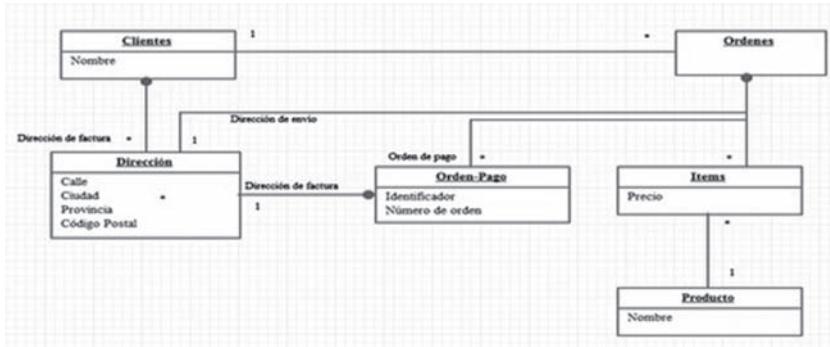
Para fijar las ideas, se va a comparar el modelo relacional con la agregación de datos en un escenario en el que se quiere gestionar información de una tienda virtual en la que se venden productos de algún tipo y hay que almacenar información acerca de los usuarios, el catálogo de productos, órdenes de compra, direcciones, etc. Esta información podría modelarse tanto usando el modelo relacional como usando agregación. Así, si se usa un modelo relacional, se podría modelar usando el diagrama UML de la figura 1. En este modelo supuestamente normalizado, no hay repeticiones de datos, hay integridad referencial, etc., y se requieren siete tablas.

**Figura 1.** Solución relacional al supuesto planteado.

Sin embargo, si se usa un modelo agregado, se podría modelar de acuerdo con la figura 2. En este modelo se tienen dos agregados: «clientes» y «órdenes». El cliente contiene una lista de direcciones de facturación y las órdenes contienen una lista de productos vendidos, direcciones de envío y órdenes de pago. Las órdenes de pago en sí mismas contienen una dirección de facturación para ese pago. Asimismo el registro de la dirección aparece tres veces, pero, en vez de usar ID, se trata como un valor y se copia cada vez (esto es coherente con el dominio que se modeliza en el que no se quiere que se cambie ni la dirección de envío ni la dirección de pago). En el modelo relacional, se aseguraría que las filas de la dirección no son actualizadas para ese caso haciendo una nueva fila en su lugar. Con los agregados se puede copiar la estructura entera de la dirección en el agregado tal como se necesita. Por otro lado, el enlace entre el cliente y la orden es una relación entre agregados. De la misma forma, se podría haber definido un enlace entre los ítems y los productos, siendo los productos un agregado separado, aunque en este caso no se ha hecho y el nombre del producto se muestra como parte

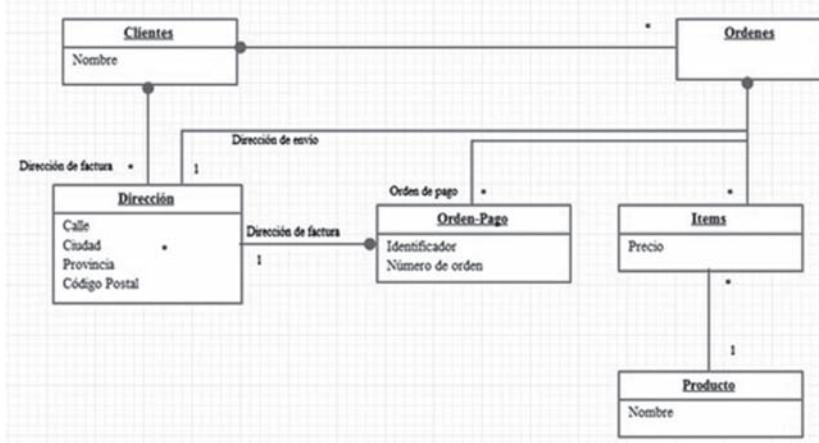
del ítem con el objetivo de minimizar el número de agregados a los que hay que acceder.

**Figura 2.** Solución agregada al supuesto planteado.



Hay que observar que en el caso agregado, cuando se modeliza hay que pensar en cómo se va a acceder a los datos, lo cual afectará al desarrollo de la aplicación que use ese modelo de datos. En este sentido igual de válido habría sido pensar un modelo de datos en el que todas las órdenes son parte de un agregado denominado «clientes» (figura 3). En este sentido no existe un modelo mejor que otro, todo depende de la forma en la que se accede a los datos. Si, por ejemplo, se prefiere que al acceder a un cliente se acceda a todas las órdenes asociadas al cliente, se optaría por este último modelo. Sin embargo, si se prefiere gestionar una orden cada vez, es preferible el primer modelo, y tener varios agregados.

**Figura 3.** Otra solución agregada al supuesto planteado.



Según ejemplo descrito se puede observar que el modelo relacional captura los diferentes elementos de datos y las relaciones que existen entre ellos sin usar ninguna noción de agregación entre ellos (en el ejemplo se podría deducir que una orden está constituida por un conjunto de ítems, una dirección de envío y una orden de pago), sino que lo expresa en términos de claves ajenas en las relaciones (pero no existe nada para distinguir relaciones que representen agregaciones de aquellas que no lo son, perdiendo el conocimiento acerca de la estructura de agregación para almacenar y distribuir los datos). Por esta razón se dice que las bases de datos relacionales ignoran las agregaciones. Un modelo que ignora las agregaciones permite fácilmente ver los datos desde distintas perspectivas, de manera que será una buena elección cuando no existe una estructura clara de manipulación de los datos.

Cuando se trabaja con bases de datos orientadas a la agregación, se tiene una semántica más clara respecto a la agregación al tratarse de la unidad de interacción con el sistema de almacenamiento. No es una propiedad lógica de los datos, sino que tiene

que ver con cómo los datos están siendo usados por las aplicaciones. A veces es complicado definir agregados si los mismos datos van a ser usados en diferentes contextos, pues una estructura de agregación puede ser útil con algunas interacciones de datos pero ser un obstáculo para otros casos. Sin embargo, la razón básica para usar una orientación agregada es cuando se quiere utilizar un clúster. En este caso, hay que minimizar el número de nodos<sup>4</sup> que se necesitan consultar cuando se están recopilando datos. Así, cuando se definen agregaciones, se ofrece información acerca de qué conjuntos de datos deben ser manipulados juntos, y, por tanto, deberían encontrarse en el mismo nodo.

Otro aspecto clave sobre los modelos orientados a agregaciones se refiere al tema de las transacciones. Las bases de datos relacionales permiten manipular cualquier combinación de filas de cualquier tabla en una única transacción ACID (atómica, consistente, aislada y durable).<sup>5</sup> En general, es cierto que las bases de datos orientadas a agregados no soportan las transacciones ACID para múltiples agregados, sin embargo, soportan manipulaciones atómicas de un único agregado cada vez. Eso significa que, si se necesita manipular múltiples agregados de una manera atómica, habrá que hacerlo desde el código de la aplicación. En la práctica, se puede ver que muchas veces se puede mantener la atomicidad dentro de una única agregación, de hecho, esta situación es parte de la consideración que se debe hacer a la hora de dividir los datos en agregados.

---

4 En un clúster un nodo es cualquier instancia de una base de datos que se está ejecutando de forma distribuida con otras instancias.

5 La atomicidad hace referencia a que muchas filas que abarcan muchas tablas son actualizadas en una sola operación que tiene éxito o bien falla de manera completa. Además, las operaciones concurrentes son aisladas una de cada otra de manera que no puede realizarse una actualización parcial.

## 6. El modelos de distribución de datos de las bases de datos NoSQL

Cuando la cantidad de datos que manejar es enorme, se hace complicado y costoso escalar las bases de datos, y una posible solución consiste en la compra de un gran servidor que ejecute la base de datos (escalado vertical). Sin embargo, una opción mejor consiste ejecutar la base de datos en un clúster de servidores. Por esta razón, las bases de datos NoSQL son interesantes por su capacidad<sup>6</sup> para ejecutarse sobre clústeres de grandes dimensiones.

Dependiendo del modelo de distribución que se elija, se pueden obtener ventajas del tipo: capacidad de manejar grandes cantidades de datos, capacidad de procesar un mayor tráfico de lectura/escritura, mayor disponibilidad cuando se producen ralentizaciones en la red o un fallo de la red, etc. Sin embargo, la distribución conlleva un coste en términos de complejidad.

Esencialmente, hay dos modelos de distribución de datos: replicación y sharding. La replicación toma los mismos datos y los copia en múltiples servidores, mientras que el sharding distribuye diferentes datos a través de múltiples servidores de manera que cada servidor actúa como una única fuente para un subconjunto de datos. En este sentido se trata de dos modelos ortogonales, de manera que puede usarse en solitario uno de ellos o bien ambos.

La replicación puede ser de dos formas: maestro-esclavo o peer-to-peer. La replicación maestro-esclavo hace de un nodo la copia autorizada que soporta las escrituras mientras que

---

<sup>6</sup> Capacidad basada en el concepto usado para organizar la información en forma de agregados de datos.

los nodos secundarios se sincronizan con el nodo primario y soportan las lecturas. En el caso de la replicación peer-to-peer, se permite la escritura sobre cualquier nodo, de manera que los nodos se coordinan para sincronizar las copias de los datos. Por una parte, la replicación maestro-esclavo reduce las posibilidades de conflictos de actualización, y la replicación peer-to-peer evita cargar las escrituras sobre un único punto de fallo.

## 7. La consistencia de los datos en las bases de datos NoSQL

El modelo de ejecución distribuido propio de las bases de datos NoSQL plantea varios tipos de problemas de consistencia en los datos:

- *Consistencia en las escrituras.* Cuando dos usuarios tratan de actualizar el mismo dato al mismo tiempo se produce un conflicto de escritura-escritura. Cuando las escrituras llegan al servidor, las serializa y entonces decide aplicar una y a continuación la otra. De esta forma, uno de los usuarios sufre una pérdida de actualización, puesto que la última escritura que se aplica sobrescribe la primera escritura. Existen varias aproximaciones para mantener la consistencia. Una aproximación pesimista<sup>7</sup> consiste en tener bloqueos de escritura, de manera que, si se quiere cambiar un valor, es necesario

---

<sup>7</sup> Las aproximaciones pesimistas a menudo degradan severamente la capacidad de respuesta de un sistema hasta el grado de no ser útiles para su propósito. Este problema empeora por el peligro de errores, debidos a interbloqueos que son difíciles de prevenir y depurar.

adquirir un bloqueo. El sistema asegura que solo un cliente puede conseguir un bloqueo a la vez. Otra aproximación más positiva consiste en la actualización condicional donde algún cliente que hace una actualización testea el valor justo antes de actualizarlo para ver si ha sido cambiado desde su última lectura. Ambas aproximaciones descansan sobre una serialización consistente de las actualizaciones. Si existe un único servidor, tiene que elegir una y a continuación la otra. Sin embargo, si existe más de un servidor, varios nodos podrían aplicar las actualizaciones en orden diferente, por lo que resultará un valor diferente actualizado. Por esta razón se habla de consistencia secuencial, que consiste en asegurar que todos los nodos apliquen las operaciones en el mismo orden. Existen otras aproximaciones,<sup>8</sup> pero en cualquiera de ellas el objetivo es mantener el equilibrio entre evitar errores tales como actualizaciones conflictivas y la rapidez de respuesta al usuario. Por otra parte, la replicación hace mucho más probable encontrarse con conflictos de escritura-escritura. Si diferentes nodos tienen diferentes copias de algunos datos que pueden ser actualizados de manera independiente, se llegarán a conflictos a menos que se tomen medidas específicas para evitarlos. Usando un único nodo para todas las escrituras de algunos datos hace que sea más fácil mantener la consistencia de las actualizaciones.

- *Consistencia en las lecturas.* Una inconsistencia de lectura o conflicto de lectura-escritura se produce cuando un usuario reali-

---

<sup>8</sup> Existe otro camino optimista para tratar con los conflictos escritura-escritura, guardar todas las actualizaciones y registrar que están en conflicto. El siguiente paso consiste en mezclar las dos actualizaciones. El problema es que cualquier mezcla automatizada de conflictos escritura-escritura es específica del dominio y necesita ser programada para cada caso particular.

za una lectura en la mitad de la escritura de otro sobre los mismos datos que se están leyendo. Para evitar una inconsistencia lógica<sup>9</sup> debido a un conflicto lectura-escritura, en las bases de datos relacionales se usa el concepto de transacción, que asegura que en estas situaciones el usuario que lee o bien lee los datos antes de la escritura o bien los datos después de la escritura. En general se afirma que las bases de datos NoSQL no soportan transacciones, sin embargo, no es cierto que todas las bases de datos NoSQL no soporten transacciones ACID (por ejemplo, las orientadas hacia grafos suelen soportarlas) y además las bases de datos orientadas hacia agregados soportan actualizaciones atómicas sobre un único agregado (por lo que consiguen consistencia lógica dentro de un agregado pero no entre agregados). En la replicación aparece un nuevo tipo de consistencia denominada «consistencia de replicación», que consiste en asegurar que los mismos datos tienen el mismo valor cuando son leídos desde diferentes réplicas. En este sentido se dice que los nodos son «eventualmente consistentes», pues cuando se produce una actualización en un nodo puede que exista algún período en el que algunos nodos tengan inconsistencias, pero, si no se realizan más actualizaciones con el tiempo, al final, todos tomarán el mismo valor. Aunque la consistencia de replicación es independiente de la consistencia lógica, la replicación puede dar lugar a una inconsistencia lógica alargando su ventana de inconsistencia.<sup>10</sup>

---

9 Se denomina «consistencia lógica» aquella que asegura que los diferentes datos tienen sentido juntos.

10 La longitud de tiempo en la que una inconsistencia está presente se denomina «ventana de inconsistencia». La presencia de una ventana de inconsistencia significa que diferentes usuarios verán diferentes cosas al mismo tiempo. En los sistemas NoSQL, la ventana de inconsistencia generalmente es bastante pequeña.

Así, por ejemplo, dos actualizaciones diferentes sobre el nodo maestro pueden ser realizadas en una sucesión rápida, dejando una ventana de inconsistencia de milisegundos, de manera que un retardo en la red podría hacer que la misma ventana de inconsistencia se hiciera mucho más larga sobre los nodos esclavos.

- *Consistencia de sesión.* Las ventanas de inconsistencia pueden ser problemáticas cuando se producen inconsistencias con uno mismo. A menudo, los sistemas mantienen la carga ejecutándose sobre un clúster y balanceando la carga enviando las peticiones a diferentes nodos, de manera que, por ejemplo, si se publica un mensaje en un sitio web y se refresca el navegador, puede que no se vea la actualización, dado que el refresco se ha redirigido a otro nodo diferente que aún no ha recibido el mensaje para publicar, lo que produce confusión en el usuario. En estas situaciones se necesita una consistencia del tipo «leer lo que tú escribes», es decir, que, una vez que se ha realizado una actualización, se debe garantizar que se continúa viendo esa actualización. Un camino para conseguir esto en un sistema eventualmente consistente es proporcionar consistencia de sesión (dentro de una sesión de usuario existe una consistencia de «leer lo que tú escribes»). Esto significa que el usuario puede perder la consistencia si su sesión finaliza por alguna razón o si el usuario accede al mismo sistema de forma simultánea desde diferentes ordenadores. Existen un par de técnicas para proporcionar consistencia de sesión. Una aproximación consiste en tener una «sticky sesión», es decir, una sesión que está ligada a un nodo (también llamada «sesión de afinidad»). Este tipo de sesiones permite asegurar que el tiempo que se mantenga la consistencia de «leer lo que tú escribes» sobre el nodo también se mantendrá para las

sesiones. La principal desventaja es que este tipo de sesiones reduce la posibilidad de balancear la carga. Otra aproximación es usar marcas de versión y asegurar que cada interacción con la base de datos incluye la última marca de versión vista por una sesión. El nodo servidor debe asegurar que tiene las actualizaciones que incluyen la marca de versión antes de responder a una petición. Mantener la consistencia de sesión con «sticky sessions» y con una replicación maestro-esclavo puede ser complicado si se quiere leer de los esclavos para mejorar el rendimiento de las lecturas pero se necesita escribir en el nodo maestro. Una forma de conseguirlo es que las escrituras sean enviadas al esclavo, que tiene la responsabilidad de reenviárselas al nodo maestro mientras mantiene la consistencia de sesión para su cliente. Otra aproximación es cambiar la sesión al nodo maestro temporalmente cuando se está haciendo una escritura, justo el tiempo necesario para que las lecturas sean hechas al nodo maestro hasta que los nodos esclavos hayan conseguido actualizarse.

Aunque es posible diseñar sistemas que eviten las inconsistencias, a menudo no es posible hacerlo sin sacrificar otras características del sistema, de manera que existe una compensación de la consistencia del sistema con otras características. En este sentido, diferentes dominios pueden tener diferente tolerancia a la inconsistencia, y esta tolerancia se debe tener en cuenta cuando se diseña un sistema. Operar sin consistencia también ocurre en los sistemas relacionales. Las transacciones garantizan la consistencia, pero es posible relajar los niveles de aislamiento, permitiendo que las consultas lean datos que aún no han sido comprometidos. Así, hay sistemas que relajan la consistencia por debajo del más alto nivel de aislamiento (serialización) con el objetivo de conse-

uir un rendimiento efectivo, e incluso es posible encontrarse el uso de un nivel de transacción de lectura comprometida, lo cual elimina algunos conflictos de lectura-escritura aunque permite otros. Algunas aplicaciones directamente renuncian a las transacciones, puesto que el impacto sobre el rendimiento es demasiado alto. Incluso sin estas restricciones, muchas aplicaciones empresariales necesitan interactuar con sistemas remotos que no pueden ser incluidos adecuadamente dentro del contexto de una transacción, de forma que las actualizaciones se realizan fuera del contexto de las transacciones. Por último, hay que observar que la garantía de consistencia no es algo que sea global a toda la aplicación. En este sentido se puede especificar el nivel de consistencia que se quiera para peticiones individuales, lo que permite usar un nivel de consistencia débil la mayoría de las veces y solicitar un nivel de consistencia mayor cuando sea necesario.

Un concepto relacionado con la consistencia es el problema de la durabilidad de la replicación. Un fallo de la durabilidad de la replicación ocurre cuando un nodo procesa una actualización pero falla antes de que la actualización es replicada al resto de los nodos. Un caso simple de esta situación ocurre si se tiene un modelo de distribución maestro-esclavo donde los esclavos apuntan a un nuevo maestro automáticamente si el nodo maestro falla. Si un nodo maestro falla, algunas escrituras no pasadas sobre las réplicas se perderán. Si el nodo maestro vuelve a levantarse, esas actualizaciones entrarán en conflicto con las actualizaciones que han ocurrido desde entonces. Es un problema de durabilidad, puesto que se piensa que la actualización ha sido exitosa, ya que el nodo maestro lo ha reconocido, pero un fallo en el nodo maestro causó su perdida. Si se tiene suficiente confianza para pensar que el nodo maestro volverá a estar online rápidamente, es una razón para no realizar la commutación a un

esclavo. Se puede mejorar la durabilidad de la replicación, asegurando que el maestro espera a que algunas replicas reconozcan la actualización antes de que el nodo maestro lo reconozca al cliente. Obviamente, esto ralentiza las actualizaciones y hace que el clúster no esté disponible si los esclavos fallan. Así, se tiene un compromiso, dependiendo de cuán vital es la durabilidad.

## 8. El teorema CAP

El teorema establece que, si se consideran las propiedades consistencia, disponibilidad y tolerancia a la partición en un sistema, solo es posible conseguir dos de ellas a la vez. La consistencia ya ha sido comentada en la sección anterior. En cuanto a la disponibilidad, significa que si se tiene acceso a un nodo en un clúster, se puede leer y escribir datos. Y la toleración al particionamiento significa que el clúster puede sobrevivir a roturas de la comunicación en el clúster que lo separan en múltiples particiones que no puedan comunicarse entre sí (conocido como «split brain»).

Un sistema de un único servidor es un ejemplo de sistema que tiene consistencia y disponibilidad pero no tiene tolerancia al particionamiento dado que una única máquina no puede particionarse y no tiene que preocuparse de la tolerancia al particionamiento. Existe un único nodo, de manera que si está en funcionamiento, está disponible. Estar en funcionamiento y mantener la consistencia es razonable. Este escenario es el típico de las bases de datos relacionales.

Teóricamente, es posible tener un clúster que tenga consistencia y disponibilidad. Pero esto significa que, si se produce un particionamiento en el clúster, todos los nodos en el clúster se

caerán y el cliente no podrá comunicarse con los nodos. Desde el punto de vista de la definición usual de «disponibilidad», esta situación significaría una falta de disponibilidad; sin embargo, en el teorema CAP el uso del concepto de «disponibilidad» es algo confuso. En CAP se define «disponibilidad» como que cada petición recibida por un nodo no caído en el sistema debe resultar en una respuesta. Es por ello que un fallo o no respuesta de un nodo no infiere una falta de disponibilidad en CAP. Esto implica que se puede construir un clúster consistente y con disponibilidad pero hay que asegurar que solo se particionará raramente y completamente.

Aunque el teorema establece que solo se pueden conseguir dos de las tres propiedades, en la práctica lo que está estableciendo es que un sistema puede sufrir particiones y hay que compensar consistencia para conseguir disponibilidad. El sistema no será completamente consistente ni disponible, pero tendrá una combinación que será razonable para las necesidades particulares. Hay que observar que existen casos donde se puede tratar adecuadamente con respuestas inconsistentes a las peticiones. Estas situaciones están estrechamente vinculadas al dominio y se requiere un conocimiento del dominio para saber cómo resolverlas. La posibilidad de mantener inconsistencias ofrece la posibilidad de incrementar la disponibilidad y rendimiento del sistema. Un razonamiento similar se aplica a la lectura consistente. En estos casos, es necesario saber cuán tolerante es el dominio a las lecturas de datos «caducados» y qué longitud puede tener la ventana de inconsistencia (a menudo en términos de la longitud media, el peor caso, o alguna medida de la distribución de las longitudes). Diferentes datos pueden tener diferentes tolerancias a la «caducidad», lo que puede llevar a necesitar diferentes configuraciones en el diseño de la replicación.

En general, el compromiso entre consistencia y disponibilidad es un compromiso entre consistencia y latencia, puesto que la consistencia se puede mejorar disponiendo de más nodos envueltos en la interacción, pero cada nodo que se añade incrementa el tiempo de respuesta de esa interacción. Así, la disponibilidad es el límite de la latencia que es tolerable, de manera que cuando la latencia se hace demasiado grande entonces los datos se tratan como no disponibles.

## Bibliografía

- Mohamed, M. A.; Altrafi, O. G.; Ismail, M. O.** (2014). «Relational vs. NoSQL Databases: A Survey». *International Journal of Computer and Information Technology* (ISSN: 2279–0764) Volume.
- Moniruzzaman, A. B. M.; Hossain, S. A.** (2013). «Nosql database: New era of databases for big data analytics-classification, characteristics and comparison». ArXiv preprint arXiv: 1307.0191.
- Pramod, J. S.; Fowler, M.** (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional.
- Redmond, E.; Wilson, J. R.** (2012). *Seven databases in seven weeks: a guide to modern databases and the NoSQL movement*. Pragmatic Bookshelf.
- Tiwari, S.** (2011). *Professional NoSQL*. John Wiley & Sons.
- Vaish, G.** (2013). *Getting started with NoSQL*. Packt Publishing Ltd.

## Capítulo II

# Conceptos básicos

### 1. Introducción

El modelo de datos documental está fuertemente orientado a agregados, dado que una base de datos consiste en un conjunto de agregados denominados «documentos». Las bases de datos documentales se caracterizan por que definen un conjunto de estructuras y tipos permitidos que pueden ser almacenados, y además es posible acceder a la estructura del agregado, teniendo como ventaja que se consigue más flexibilidad en el acceso. En este sentido se pueden realizar consultas a la base de datos según los campos del agregado, se pueden recuperar partes del agregado en vez del agregado completo, y además se pueden crear índices basados en el contenido del agregado. Dado que cada agregado tiene asociado un identificador, es posible realizar búsquedas del estilo clave-valor.

MongoDB es una base de datos NoSQL orientada a documentos creada por la compañía 10gen en el año 2007. Se caracteriza por que almacena los datos en documentos de tipo JSON con un esquema dinámico denominado «BSON».

En este capítulo se exponen los conceptos básicos y la terminología fundamental necesaria para poder entender qué es una base de datos en MongoDB y qué elementos la forman. En este sentido se definen los conceptos de documento, colección y base de datos, así como los tipos de datos soportados por MongoDB. Asimismo se realiza una introducción a la Shell de

comandos que constituye la principal forma de interactuar con MongoDB.

## 2. Documentos

Los documentos son la unidad básica de organización de la información en MongoDB, y desempeñan un papel equivalente a una fila en las bases de datos relacionales. Un documento es un conjunto ordenado de claves que tienen asociados valores, y que se corresponden con algunas estructuras de datos típicas de los lenguajes de programación tales como tablas hash o diccionarios. En general, los documentos contendrán múltiples pares clave-valor, como, por ejemplo, `{"Nombre":"Juan","País":"España"}`. Sus principales características son:

- Las claves son cadenas, por lo que se permite cualquier carácter con un par de excepciones:
  - La clave no pueden contener el carácter nulo «\0».
  - El punto «.» y el «\$» deben evitarse, pues tienen propiedades especiales.
- MongoDB es sensitivo tanto a las mayúsculas/minúsculas como a los tipos de datos. Así, por ejemplo, los siguientes documentos se consideran distintos: `{"Edad":3}`, `{"Edad":"3"}`, `{"edad":3}`, `{"edad":"3"}`.
- Los documentos no pueden tener claves duplicadas. Así, por ejemplo, el siguiente documento es incorrecto: `{"edad":3,"edad":56}`.
- Los pares clave-valor están ordenados en los documentos. Por ejemplo, el documento `{"x":3,"y":5}` no es igual que

{“y”:5,”x”:3}. Es importante no definir las aplicaciones pensando en el orden de los campos, pues MongoDB puede reordenarlos automáticamente en determinadas situaciones.

- Los valores de un documento pueden ser de diferentes tipos.

### 3. Tipos de datos

Los principales tipos de datos soportados por los documentos en MongoDB son:

- Nulo: representa el valor nulo o bien un campo que no existe. Por ejemplo, {“x”:null}.
- Booleanos: representa el tipo booleano, que puede tomar los valores de true o false. Por ejemplo, {“x”:true}.
- Números: distingue entre números reales, como, por ejemplo, {“x”:3.14}, y números enteros, como, por ejemplo, {“x”:45}.
- Cadenas: cualquier cadena de caracteres, como, por ejemplo, {“x”：“Ejemplo”}.
- Fechas: almacena la fecha en milisegundos, pero no la zona horario. Por ejemplo, {“x”:new Date()}.
- Expresiones regulares: se pueden usar expresiones regulares para realizar consultas.
- Arrays: se representa como un conjunto o lista de valores. Por ejemplo, {“x”:[“a”,“b”,“c”]}.
- Documentos embebidos: los documentos pueden contener documentos embebidos como valores de un documento padre. Por ejemplo, {“x”:{“y”:45}}.
- Identificadores de objetos: es un identificador de 12 bytes para un documento. Por ejemplo, {“x”: ObjectId()}.

- Datos binarios: es una cadena de bytes arbitraria que no puede ser manipulada directamente desde el Shell y que sirve para representar cadenas de caracteres no UTF8.
- Código Javascript: los documentos y las consultas pueden contener código JavaScript. Por ejemplo, {“x: function () { ...}}.

Hay que observar:

**a) Fechas.** Para crear un objeto de tipo fecha se usa el comando new Date (). Sin embargo, si se llama sin new (solo Date ()) , se retorna una cadena que representa la fecha. Y por tanto se trata de diferentes tipos de datos. Las fechas en el Shell son mostradas usando la configuración local de la zona horaria, sin embargo, la base de datos las almacena como un valor en milisegundos sin referencia a la zona horaria (aunque podría almacenarse este valor definiendo una clave para el mismo).

**b) Arrays.** Pueden ser usados tanto en operaciones en las que el orden es importante, tales como listas, pilas o colas, como en operaciones en las que el orden no es importante, tales como conjuntos. Los arrays pueden contener diferentes tipos de valores, como, por ejemplo, {“Cosas”: [“edad”,45]} (de hecho, soporta cualquiera de los tipos de valores soportados para los documentos, por lo que se pueden crear arrays anidados). Una propiedad importante en MongoDB es que reconoce la estructura de los arrays y permite navegar por el interior de los arrays para realizar operaciones sobre sus contenidos, como consultas, o crear índices sobre sus contenidos. En el ejemplo anterior se podría crear una consulta para recuperar todos aquellos documentos donde 3.14 es un elemento del array «Cosas», y si, por

ejemplo, esta fuera una consulta habitual entonces incluso se podría crear un índice sobre la clave «Cosas» y mejorar el rendimiento de la consulta. Asimismo MongoDB permite realizar actualizaciones que modifican los contenidos de los arrays, tales como cambiar un valor del array por otro.

c) *Documentos embebidos.* Los documentos pueden ser usados como valores de una clave, y en este caso se denominan «documentos embebidos». Se suelen usar para organizar los datos de una manera lo más natural posible. Por ejemplo, si se tiene un documento que representa a una persona y se quiere almacenar su dirección, podría crearse anidando un documento «dirección» al documento asociado a una persona, como, por ejemplo:

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

MongoDB es capaz de navegar por la estructura de los documentos embebidos y realizar operaciones con sus valores, como, por ejemplo, crear índices, consultas o actualizaciones. La principal desventaja de los documentos embebidos se debe a la repetición de datos. Por ejemplo, supóngase que las direcciones se encuentran en una tabla independiente y que hay que rectificar un error tipográfico, entonces, al rectificar la dirección en la tabla direcciones, se rectifica para todos los usuarios que comparten

la misma dirección. Sin embargo, en MongoDB habría que rectificar las direcciones una a una en cada documento, aunque se compartan direcciones.

#### d) Identificador de objetos

Cada documento tiene que tener un clave denominada «`_id`»:

- El valor de esta clave puede ser de cualquier tipo pero por defecto será de tipo `ObjectId`.
- En una colección, cada documento debe tener un valor único y no repetido para la clave «`_id`», lo que asegura que cada documento en la colección pueda ser identificado de manera única. Así, por ejemplo, dos colecciones podrían tener un documento con «`_id`» con el valor 123, pero en una misma colección no podría haber dos documentos con valor de «`_id`» de 123.
- El tipo `ObjectId` es el tipo por defecto para los valores asociados a la clave «`_id`». Es un tipo de datos diseñado para ser usado en ambientes distribuidos de manera que permita disponer de valores que sean únicos globalmente. Cada valor usa 12 bytes, lo que permite representar una cadena de 24 dígitos hexadecimales (2 dígitos por cada byte). Si se crean múltiples valores del tipo `ObjectId` sucesivamente, solo cambian unos pocos dígitos del final y una pareja de dígitos de la mitad. Esto se debe a la forma en la que se crean los valores del tipo `ObjectIDs`. Los 12 bytes se generan así:

Timestamp				Machine			PID		Increment		
0	1	2	3	4	5	6	7	8	9	10	11

Hay que observar que:

- Los primeros 4 bytes son una marca de tiempo en segundos, que, combinados con los siguientes 4 bytes, proporciona unicidad a nivel de segundo y que identifican de manera implícita cuando el documento fue creado. Por otro lado, a causa de que la marca de tiempo aparece en primer lugar, los ObjectIDs se ordenan obligatoriamente en orden de inserción, lo que hace que la indexación sobre ObjectIDs sea eficiente.
- Los siguientes 3 bytes son un identificador único de la máquina que lo genera, lo que garantiza que diferentes máquinas no generan colisiones.
- Para conseguir unicidad entre diferentes procesos que generan ObjectIDs concurrentemente en una misma máquina, se usan los siguientes 2 bytes, que son tomados del identificador del proceso que genera un ObjectId.
- Los primeros 9 bytes aseguran la unicidad a través de máquinas y procesos para un segundo, y los últimos 3 bytes son un contador que se incrementa y que es el responsable de la unicidad dentro de un segundo en un proceso.
- Este sistema permite generar 2563 únicos ObjectIDs por proceso en un segundo.

Cuando un documento se va a insertar, si no tiene un valor para la clave «`_id`», es generado automáticamente por MongoDB.

## 4. Colecciones

Una colección es un grupo de documentos, y desempeña el papel análogo a las tablas en las bases de datos relacionales.

Las colecciones tienen esquemas dinámicos, lo que significa que dentro de una colección puede haber cualquier número de documentos con diferentes estructuras. Por ejemplo, en una misma colección podrían estar los siguientes documentos diferentes: `{"edad":34}`, `{"x":"casa"}` que tienen diferentes claves y diferentes tipos de valores.

Dado que cualquier documento, se puede poner en cualquier colección, y dado que no es necesario disponer de esquemas distintos para los diferentes tipos de documentos, entonces surge la pregunta de por qué se necesita usar más de una colección y tener que separar los documentos mediante colecciones separadas:

- Si se mantuvieran diferentes tipos de documentos en la misma colección, produciría problemas para asegurar que cada consulta solo recupera documentos de un cierto tipo o que en el código de la aplicación implementa consultas para cada tipo de documento.
- Es más rápido obtener una lista de colecciones que extraer una lista de los tipos de documentos en una colección.
- La agrupación de documentos del mismo tipo juntos en la misma colección permite la localidad de los datos.
- Cuando se crean índices se impone cierta estructura a los documentos (especialmente en los índices únicos). Estos índices están definidos por colección de forma que poniendo documentos de un solo tipo en la misma colección entonces se podrán indexar las colecciones de una forma más eficiente.

Por tanto, estas razones hacen razonable crear un esquema y agrupar los tipos relacionados de documentos juntos aunque MongoDB no lo imponga como obligatorio.

Una colección se identifica por su nombre, que es una cadena con las siguientes restricciones:

- La cadena vacía no es un nombre válido para una colección.
- Los nombres de las colecciones no pueden contener el carácter nulo «\0», pues este símbolo se usa para indicar el fin del nombre de una colección.
- No se debe crear ninguna colección que empiece con «system», dado que es un prefijo reservado para las colecciones internas. Por ejemplo, la colección system.users contiene los usuarios de la base de datos, la colección system.namespaces contiene información acerca de todas las colecciones de la base de datos.
- Las colecciones creadas por los usuarios no deben contener el carácter reservado «\$» en su nombre.

Una convención para organizar las colecciones consiste en definir subcolecciones usando espacios de nombres separados por el carácter «.». Por ejemplo, una aplicación que contuviese un blog podría tener una colección denominada «blog.posts» y otra colección denominada «blog.autores» con un propósito organizativo y que, sin embargo, ni exista la colección blog y en caso de existir no exista una relación entre la colección padre blog y las subcolecciones.

Aunque las subcolecciones no tienen propiedades especiales, son útiles por algunas razones:

- Existe un protocolo en MongoDB para almacenar archivos muy extensos denominado «GridFS» que usa las subcolecciones para almacenar los archivos de metadatos separados de los datos.

- Algunas librerías proporcionan funciones para acceder de una forma simple a las subcolecciones de una colección. Por ejemplo, db.blog proporciona acceso a la colección blog y db.blog.posts permite acceder la colección blog.posts

## 5. Bases de datos

Las colecciones se agrupan en bases de datos, de manera que una única instancia de MongoDB puede gestionar varias bases de datos cada una agrupando cero o más colecciones.

Hay que observar que:

- Cada base de datos tiene sus propios permisos y se almacena en ficheros del disco separados.
- Una buena regla general consiste en almacenar todos los datos de una aplicación en la misma base de datos.
- Las bases de datos separadas son útiles cuando se almacenan datos para aplicaciones o usuarios diferentes que usan el mismo servidor de MongoDB.

Las bases de datos se identifican mediante nombres que son cadenas con las siguientes restricciones:

- La cadena vacía no es un nombre válido para una base de datos.
- El nombre de una base de datos no puede contener ninguno de los siguientes caracteres: \, /, ., ”, \*, <, >, :, |, ?, \$, espacio o \0(valor nulo).
- Los nombres de las bases de datos son sensativos a mayúsculas y minúsculas incluso sobre sistemas de archivos que

no lo sean. Una regla práctica es usar siempre nombres en minúscula.

- Los nombres están limitados a un máximo de 64 bytes.
- Existen nombres que no pueden usarse para las bases de datos por estar reservados:

**1) admin.** Es el nombre de la base de datos «root» en términos de autenticación. Si un usuario es añadido a esta base de datos, entonces el usuario hereda los permisos para todas las bases de datos. Existen determinados comandos que solo pueden ser ejecutados desde esta base de datos, tales como listar todas las bases de datos o apagar el servidor.

**2) local.** Esta base de datos nunca será replicada y sirve para almacenar cualquier colección que debería ser local a un servidor.

**3) configura.** Cuando en MongoDB se usa una configuración con sharding, se usa esta base de datos para almacenar información acerca de los fragmentos o shards que se crean.

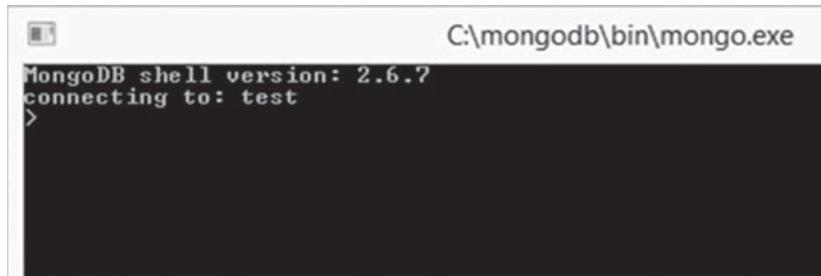
Mediante la concatenación del nombre de una base de datos con una colección de la base de datos se consigue una cualificación entera del nombre de la colección denominado «espacio de nombres». Por ejemplo, si se usa la colección blog.posts en la base de datos cms, el espacio de nombres de esa colección sería cms.blog.posts. Los espacios de nombres están limitados a 121 bytes de longitud, aunque en la práctica es mejor que sean menores de 100 bytes.

## 6. La Shell de comandos de MongoDB

La Shell es un cliente de MongoDB que permite interaccionar con una instancia de MongoDB desde una línea de comandos con el objetivo de realizar funciones administrativas, inspeccionar la instancia que se está ejecutando o llevar a cabo determinadas operaciones de gestión sobre la base de datos. Se trata de la principal interfaz para comunicarse con una instancia de MongoDB.

Para empezar a utilizarla se ejecuta el archivo mongo.exe (figura 4), que se encuentra en el subdirectorio bin del directorio elegido para realizar la instalación de MongoDB. Hay que observar que antes de ejecutar la Shell, se ha debido ejecutar el archivo mongod.exe, que arranca el servidor de MongoDB.

**Figura 4.** Shell de MongoDB.



```
C:\mongodb\bin\mongo.exe
MongoDB shell version: 2.6.7
connecting to: test
>
```

La Shell intenta conectarse al servidor de MongoDB que se esté ejecutando, y en caso contrario se producirá un fallo. Si la conexión se ha realizado con éxito, lo primero que hace es conectarse a una base de datos por defecto denominada «test» y asignar esta conexión a la variable global db. Esta variable es el punto de acceso primario al servidor MongoDB a través de la Shell. Para ver la base de datos que tiene asignada en un momento dado, basta con teclear en la Shell «db» (figura 5).

**Figura 5.** Obtención de la base de datos actual.

```
> db  
test  
>
```

En particular, la Shell es un intérprete de JavaScript que es capaz de ejecutar cualquier programa en JavaScript, como, por ejemplo, ejecutar operaciones matemáticas (figura 6).

**Figura 6.** Ejemplo de operaciones aritméticas en la Shell.

```
C:\mongod  
MongoDB shell version: 2.6.7  
connecting to: test  
200  
5
```

También es capaz de utilizar librerías de JavaScript (figura 7).

**Figura 7.** Ejemplo de uso de librerías de JavaScript en la Shell.

```
> Math.sin(Math.PI / 2);  
1  
> new Date("2014/6/1");  
ISODate("2014-05-31T22:00:00Z")  
> "¡Hola Mundo!".replace("Mundo", "Juan");  
¡Hola Juan!
```

Asimismo es posible definir y llamar a funciones de JavaScript (figura 8).

**Figura 8.** Ejemplo de definición y ejecución de funciones de JavaScript en la Shell.

```
> function factorial (n) {  
...   if (n <= 1) return 1;  
...   return n * factorial(n - 1);  
... }  
> factorial(5);  
120  
>
```

En la Shell se pueden crear comandos que se encuentren en varias líneas, de manera que la Shell detectará si el comando está completo cuando se presiona Enter. En caso de no estar completo, la Shell permite continuar escribiendo en la siguiente línea. Si se presiona tres veces Enter, se cancela y se vuelve a una nueva línea.

La Shell contiene algunos comandos adicionales que no son válidos en JavaScript pero que simplifican las operaciones con las bases de datos, como, por ejemplo, el comando «use», que permite seleccionar o cambiar de base de datos (figura 9).

**Figura 9.** Ejemplo de uso del comando use.

```
> use prueba
switched to db prueba
> db
prueba
>
```

Las colecciones pueden ser accedidas desde la variable db, como, por ejemplo, db.blog retornaría la colección blog en la base de datos actual.

## 7. Operaciones básicas en la Shell

En la Shell se pueden realizar cuatro operaciones básicas para manipular y ver datos denominadas CRUD (crear, leer, actualizar o borrar):

a) *Crear*: El comando **insert** añade un documento a una colección. Por ejemplo, si se quiere almacenar un post de un blog (figura 10), primero se crea una variable post que representa al documento y que tendrá como claves título, contenido y fecha de publicación:

**Figura 10.** Creación de la variable post.

```
> post= { "titulo": "Mi primer post de mi blog",
... "contenido" : "Este es mi primer post de mi blog",
... "date": new Date<>
{
    "titulo" : "Mi primer post de mi blog",
    "contenido" : "Este es mi primer post de mi blog",
    "date" : ISODate("2015-11-08T00:09:21.135Z")
}
> -
```

A continuación se usa insert para añadirlo a la colección blog (figura 11).

**Figura 11.** Inserción del documento post en blog.

```
> db.blog.insert(post)
WriteResult({ "nInserted" : 1 })
```

Ahora se puede recuperar con una llamada a la función find sobre la colección (figura 12).

**Figura 12.** Ejecución de la función find para recuperar el contenido de blog.

```
> db.blog.find()
{ "_id" : ObjectId("563e932c22cf0c8f162ed7bc"), "titulo" : "Mi primer post de mi blog", "contenido" : "Este es mi primer post de mi blog", "date" : ISODate("2015-11-08T00:09:21.135Z") }
```

En el ejemplo se puede observar que se ha añadido una clave «\_id» y que el resto de las claves se han mantenido intactas.

**b) Leer.** Los comandos **find** y **findOne** pueden ser usados para consultar una colección. Si solo se quiere recuperar un único documento de una colección, se usará findOne (figura 13):

**Figura 13.** Figura 13. Invocación del comando findOne.

```
> db.blog.findOne()
{
    "_id" : ObjectId("5649b05329a5a1b615dd2e97"),
    "titulo" : "Mi primer post de mi blog",
    "contenido" : "Este es mi primer post de mi blog",
    "date" : ISODate("2015-11-16T10:30:25.092Z")
}
```

Tanto a `find()` como a `findOne()` se les puede pasar como parámetro una condición de recuperación que permite restringir los documentos recuperados. La Shell mostrará automáticamente 20 documentos pero se pueden recuperar más.

**c) Actualizar.** Para actualizar se dispone del comando `update`, que toma al menos dos parámetros: un criterio para encontrar el documento que se desea actualizar y el nuevo documento. Por ejemplo, supóngase que en el ejemplo anterior se quiere añadir una nueva clave que represente los comentarios a un post del blog, y que tomará como valor un array de comentarios. Entonces, lo primero que hay que hacer es modificar la variable `post` y añadir una clave «comentarios» (figura 14):

**Figura 14.** Adición del campo «comentarios» a la variable `post`.

```
> post.comentarios=[ ]
```

A continuación, se actualiza el documento sustituyendo el anterior documento por el nuevo (figura 15):

**Figura 15.** Sustitución del documento anterior por otro nuevo.

```
> db.blog.update({ "titulo": "Mi primer post de mi blog"}, post)
> writeResult{ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }
```

Ahora se puede consultar y ver los cambios mediante `findOne` (figura 16):

**Figura 16.** Consulta del nuevo documento que ha sido cambiado.

```
> db.blog.findOne()
{
  "_id" : ObjectId("5649b05329a5a1b615dd2e97"),
  "titulo" : "Mi primer post de mi blog",
  "contenido" : "Este es mi primer post de mi blog",
  "date" : ISODate("2015-11-16T10:30:25.092Z"),
  "comentarios" : [ ]
```

**d) Borrar.** El comando **remove** elimina de forma permanente los documentos de una base de datos. Si se llama sin parámetros, elimina todos los documentos de una colección. También puede tomar un documento especificando criterios para su eliminación, como el siguiente ejemplo, que elimina los documentos que tenga en la clave «título» el valor «Mi primer post de mi blog» (figura 17):

**Figura 17.** Eliminación de un documento con remove.

```
> db.blog.remove({ "titulo": "Mi primer post de mi blog" })
writeResult: { "nRemoved" : 1 }
```

Además de las operaciones básicas de manipulación de la base de datos, la Shell dispone de otros comandos útiles más orientados a manipular la propia Shell:

**a) Configuración de la Shell.** Por defecto, la Shell se conecta a una instancia local de MongoDB, sin embargo, es posible conectarse a una instancia diferente o puerto especificando el hostname, puerto y base de datos: mongo hostname: puerto/nombre\_base\_datos.

A veces puede interesar ejecutar la Shell sin conectarse a una instancia de MongoDB, para lo que se utiliza el comando mongo --nodb (figura 18).

**Figura 18.** Conexión a la Shell sin conectarse a una instancia de MongoDB.

```
C:\mongodb\bin>mongo --nodb
MongoDB shell version: 2.6.7
> -
```

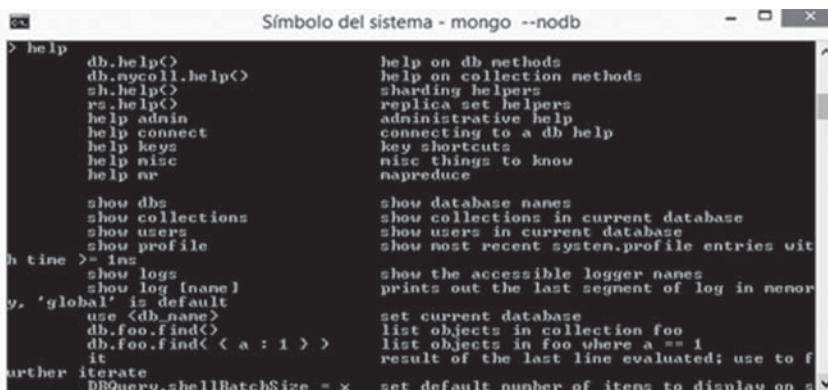
Una vez conectado se puede conectar a una instancia de MongoDB mediante el comando new Mongo (hostname) y a continuación conectarse a la base de datos (figura 19):

**Figura 19.** Conexión a una instancia de MongoDB mediante el comando Mongo.

```
> conn=new Mongo("localhost")
connection to localhost
> db=conn.getDB("Prueba")
Prueba
> _
```

A partir de este momento se puede usar de forma normal la base de datos. Hay que observar que con estos comandos es posible conectarse a diferentes bases de datos o servidores en cualquier momento.

**b) Ayuda.** Dado que MongoDB es una Shell basado en JavaScript, se puede encontrar información sobre su funcionamiento consultando la documentación de JavaScript. Asimismo se puede encontrar información sobre la funcionalidad de la Shell específica de MongoDB utilizando el comando help (figura 20).

**Figura 20.** Invocación del comando help ()�


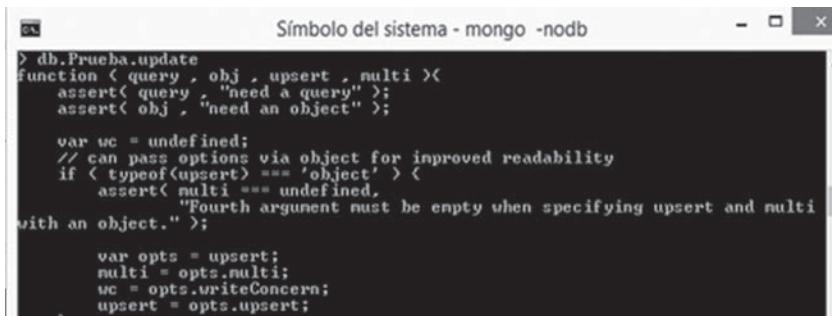
```
Símbolo del sistema - mongo --nodb
> help
db.help()                      help on db methods
db.mycoll.help()                help on collection methods
sh.help()                       sharding helpers
rs.help()                        replica set helpers
help admin                      administrative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

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

En cuanto a la base de datos también se puede encontrar ayuda mediante el comando db.help () y en cuanto a colección mediante el comando db.Nombre\_Colección.help (). También

se puede encontrar información sobre una función determinada introduciendo el nombre de la misma sin paréntesis. En este caso se imprimirá el código fuente en JavaScript de la función. Por ejemplo, si se quiere ver cómo funciona la función update, se utilizaría el comando mostrado en figura 21.

**Figura 21.** Muestra de la ayuda del comando update.



```
Símbolo del sistema - mongo -nodb
db.Pruna.update
function (query, obj, upsert, multi) {
    assert(query, "need a query");
    assert(obj, "need an object");

    var uc = undefined;
    // can pass options via object for improved readability
    if (typeof(upsert) === 'object') {
        assert(multi === undefined,
            "Fourth argument must be empty when specifying upsert and multi
with an object.");
        var opts = upsert;
        multi = opts.multi;
        uc = opts.writeConcern;
        upsert = opts.upsert;
    }
}
```

c) *Ejecución de JavaScript.* Otra característica de la Shell es la posibilidad de ejecutar archivos JavaScript. Para ello se ejecutan en la Shell los scripts de la siguiente forma:

```
mongo Nombre_Script1.js Nombre_Script2.js.
```

La Shell ejecuta cada script listado y a continuación abandona la Shell. También es posible ejecutar los scripts en una conexión que no sea la que aparece por defecto especificando primero la dirección: host, puerto y una base de datos concreta, y a continuación la lista de scripts:

```
mongo --quiet hostname:puerto/base_datos Nombre_
Script1.js Nombre_Script2.js.
```

Hay que observar en el ejemplo anterior que las opciones de la línea de comandos para ejecutar la Shell van siempre antes de la dirección. En este caso se utiliza la opción `--quiet`, que evita que se imprima la información de cabecera «MongoDB Shell version....». En este sentido, esta opción se puede utilizar para canalizar la salida de un script de la Shell a otro comando. También es posible imprimir a la salida estándar los scripts usando la función `print()`.

Otra forma de ejecutar scripts desde dentro de la Shell es usar la función `load` (“Nombre\_Script.js”).

Los scripts tienen acceso a la variable `db` como a otras variables globales, sin embargo, no funcionan algunos de los comandos propios de MongoDB, aunque existen otras alternativas a estos en JavaScript (tabla 1):

<code>use prueba</code>	<code>db.getSiblingDB("prueba")</code>
<code>show dbs</code>	<code>db.getMongo().getDBs()</code>
<code>show collections</code>	<code>db.getCollectionNames()</code>

Tabla 1. Comandos alternativos en JavaScript.

También es posible usar scripts para añadir variables en la Shell. Por ejemplo, se podría tener un script `conectarseA()` para realizar una inicialización que conecte a una instancia local de MongoDB que se ejecuta sobre un puerto dado y configura la variable `db` a esa conexión. Si se carga este script en la Shell, el script está ya definido en MongoDB (figura 22):

**Figura 22.** Definición de un script en MongoDB.

```
> var conectarseA= function (puerto, dbnombre){  
... if (!puerto) {  
... puerto=27017;  
}  
... if (!dbnombre) {  
... dbnombre="test";  
}  
... db= connect("localhost:"+puerto+"/"+dbnombre);  
... return db;  
};  
> typeof conectarTo  
function
```

Los scripts también pueden usarse para automatizar tareas comunes y actividades administrativas. Por defecto, la Shell mirará en el directorio en que se inició, de manera que, si el script no está en el directorio actual, se puede especificar a la Shell una dirección relativa o absoluta. Por ejemplo, si se quiere poner los scripts de la Shell en el directorio `~/mis-scripts`, para cargar el script `conectararseA.js` se usaría el comando: `load ("~/mis-scripts/conectararseA.js")`. Hay que observar que `load ()` no es capaz de resolver el símbolo `~`.

Asimismo si se quiere usar el comando `run ()` para ejecutar programas desde la línea de comandos de la Shell, se pasan los argumentos a la función como parámetros:

- `run ("ls", "-l", "/home/usuario/mis-scripts/")`
- `sh70352| -rw-r--r-- 1 usuario usuario 2012-12-13 13:15 defineconectararseA.js`
- `sh70532| -rw-r--r-- 1 myUser myUser 2013-02-22 15:10 script1.js`
- `sh70532| -rw-r--r-- 1 myUser myUser 2013-02-22 15:12 script2.js`
- `sh70532| -rw-r--r-- 1 myUser myUser 2013-02-22 15:13 script3.js`

Hay que observar que su uso es limitado, dado que la salida se formatea de forma extraña y no soporta la canalización entre diferentes funciones.

## 8. Observaciones

Cuando se consulta una colección con el comando db.Nombre\_colección, siempre funciona salvo que el nombre de la colección sea un nombre reservado o sea un nombre inválido de propiedad de JavaScript. Por ejemplo, si se intenta acceder a una colección denominada «version», no puede usar «db.version», dado que se trata de un método sobre db que retorna la versión del servidor de MongoDB que se está usando (figura 23).

**Figura 23.** Intento de acceder a la colección version mediante db.

```
> db.version
function () {
    return this.serverBuildInfo().version;
}
>
```

Así, para acceder a la colección de nombre «version», se usa la función getCollection (figura 24).

**Figura 24.** Acceso a la colección version mediante getCollection.

```
> db.getCollection("version")
test.version
> _
```

Este procedimiento también puede ser usado para acceder a nombres de colecciones que usan caracteres que no son válidos para nombres de propiedades JavaScript (los nombres deben

contener letras, números, «\$» y «\_» y no pueden empezar con un número), tal como 123abc. Otra forma de resolver este problema es usar el acceso via array; así, en JavaScript x.y es idéntico a x ['y'] (es decir, que las subcolecciones pueden ser accedidas usando variables no solo como nombres literales). Por ejemplo, si se quiere realizar alguna operación sobre cada subcolección del blog, se podría iterar a través de ellas usando el siguiente código:

```
var colecciones = ["posts", "comentarios", "autores"];
for (var i in colecciones) {
    print(db.blog [colecciones[i]]);
}
```

Hay que observar que, si se escribe db.blog.i, sería interpretado como test.blog.i y no como test.blog.posts. Es por ello que hay que usar la sintaxis db.blog[i] para que sea interpretado como una variable. Esta técnica también es útil para acceder a colecciones que tengan nombres complicados, como, por ejemplo:

```
> var nombre = "@#&!";
> db [nombre].find ()
```

Hay que observar que la consulta db. @#&! sería ilegal; sin embargo, db [nombre] sí funcionaría.

## Bibliografía

- Banker, K.** (2011). *MongoDB in action*. Manning Publications Co.
- Chodorow, K.** (2013). *MongoDB: the definitive guide*. O'Reilly Media, Inc.
- Hows, D.; Membrey, P.; Plugge, E.** (2014). *Installing MongoDB*. In *MongoDB Basics* (págs. 19-36). Apress.
- Membrey, P.; Plugge, E.; Hawkins, D.** (2010). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress.
- Sitio oficial MongoDB, «Getting Started with MongoDB».  
<<https://docs.mongodb.org/v2.6/tutorial/getting-started/>>

## Capítulo III Operaciones CRUD

### 1. Introducción

Este capítulo se centra en las principales operaciones de manipulación de datos, más conocidas como CRUD (crear, leer, actualizar o borrar). Mediante estas operaciones, el usuario puede gestionar los datos de una forma directa a través de la Shell de comandos de MongoDB. Asimismo se hace una especial referencia a las operaciones de modificación sobre los arrays, al tratarse de un objeto que es considerado de primer orden en MongoDB.

### 2. Inserción de documentos

Para insertar un documento en una colección se usa el método `insert` (figura 25):

**Figura 25.** Ejemplo de inserción de un documento.

```
> db.prueba.insert({ "titulo": "El Quijote" })
> writeResult({ "nInserted" : 1 })
> -
```

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

Cuando es necesario insertar un conjunto de documentos, se puede pasar como parámetro un array con el conjunto de documentos que deben ser insertados (figura 26):

**Figura 26.** Ejemplo de inserción de múltiples documentos.

```
> db.prueba.insert([{"_id" : 0}, {"_id" : 1}, {"_id" : 2}])
BulkWriteResult{{
    "writeErrors" : [ ],
    "writeConcernErrors" : [ ],
    "nInserted" : 3,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
}}
>
```

Se pueden insertar mediante un array múltiples documentos siempre que se vayan almacenar en una única colección; en caso de varias colecciones no es posible y habría que usar otras herramientas tales como mongoimport.

Hay que observar:

- Cuando se inserta usando un array, si se produce algún fallo en algún documento, se insertan todos los documentos anteriores al que tuvo el fallo, y los que hay a continuación no se insertan. Este comportamiento se puede cambiar usando la opción «`continueOnError`», que, en caso de encontrarse un error en un documento, lo salta, y continúa insertando el resto de los documentos. Esta opción no está disponible directamente en la Shell, pero sí en los drivers de los lenguajes de programación.
- Actualmente existe un límite de longitud de 48 MB para las inserciones realizadas usando un array de documentos.

Cuando se inserta un documento MongoDB, se realizan una serie de operaciones con el objetivo de evitar inconsistencias tales como:

- Se añade el campo «`_id`» en caso de no tenerlo.
- Se comprueba la estructura básica. En particular se comprueba el tamaño del documento (debe ser más pequeño de 16 MB). Para saber el tamaño de un documento se puede usar el comando `Object.bsonsize(doc)`.
- Existencia de caracteres no válidos.

### 3. Borrado de documentos

El método `remove` elimina todos los documentos de una colección, pero no elimina la colección ni la metainformación acerca de la colección (figura 27):

**Figura 27.** Eliminación de los documentos de una colección.

```
> db.prueba.find()
{ "_id" : 0 }
{ "_id" : 1 }
{ "_id" : 2 }
> db.prueba.remove({})
writeResult({ "nRemoved" : 3 })
>
```

El método permite opcionalmente tomar una condición de búsqueda, de forma que eliminará solo aquellos documentos que encajen con la condición dada. Por ejemplo, si se quisiera eliminar todos los documentos de la colección `correo.lista` donde el valor para el campo «`salida`» es cierto, entonces se usaría el siguiente comando:

```
db.correo.lista.remove ({ "salida" : true })
```

Una vez que se ha realizado el borrado no se puede dar revertir y se pierden todos los documentos borrados. A veces, si se van a borrar todos los documentos, es más rápido eliminar toda la colección en vez de los documentos. Para ello se usa el método drop:

```
db.prueba.drop()
```

## 4. Actualización de documentos

Para modificar un documento almacenado se usa el método update, que toma dos parámetros: una condición de búsqueda que localiza el documento que actualizar y un conjunto de cambios que realizar sobre el documento.

Las actualizaciones son atómicas, de manera que, si se quieren realizar dos a la vez, la primera que llegue es la primera en realizarse y a continuación se hará la siguiente.

- Reemplazamiento de documentos

El tipo de actualización más simple consiste en reemplazar un documento por otro. Por ejemplo, que se quiere cambiar el siguiente documento:

```
{  
  "nombre": "Juan",  
  "amigos": 32,  
  "enemigos": 2  
}
```

Y se quiere crear un campo «relaciones» que englobe a los campos «amigos» y «enemigos» como subdocumentos, esta operación se puede llevar a cabo con un update (figura 28):

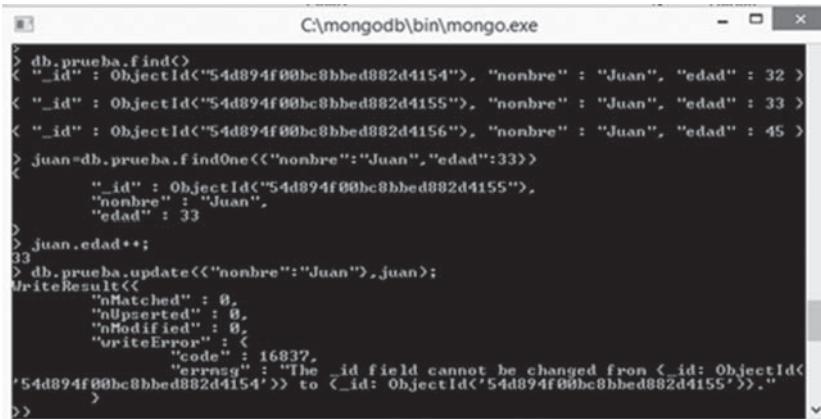
**Figura 28.** Ejemplo de actualización con update.



The screenshot shows a terminal window titled 'C:\mongodb\bin\mongo.exe'. The command entered is:

```
> var juan = db.prueba.findOne({"nombre" : "Juan"});
> juan.relaciones = {"amigos":juan.amigos, "enemigos":juan.enemigos};
> juan.amigos = 32, "enemigos" : 2
> juan.PrimerNombre=Juan.nombre
> Juan
> delete juan.amigos
true
> delete juan.enemigos
true
> delete juan.nombre
true
> db.prueba.update({"nombre":"Juan"}, juan);
WriteResult({"nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.prueba.findOne()
{
    "_id" : ObjectId("54d892c60bc8bbcd882d4153"),
    "relaciones" : {
        "amigos" : 32,
        "enemigos" : 2
    },
    "PrimerNombre" : "Juan"
}
```

Un error común es cuando encaja más de un documento con el criterio de búsqueda y se crea un campo duplicado «`_id`» con el segundo parámetro. En este caso, la base de dato genera un error y ningún documento es actualizado. Por ejemplo, supóngase que se crean varios documentos con el mismo valor para el campo «`nombre`», sea «`Juan`», y se quiere actualizar el valor del campo «`edad`» de uno de ellos (se quiere aumentar el valor de la edad del segundo «`Juan`») (figura 29):

**Figura 29.** Ejemplo de error en la actualización.


```
C:\mongodb\bin\mongo.exe
> db.prueba.find()
< "_id" : ObjectId("54d894f800bc8bbcd882d4154"), "nombre" : "Juan", "edad" : 32 >
< "_id" : ObjectId("54d894f800bc8bbcd882d4155"), "nombre" : "Juan", "edad" : 33 >
< "_id" : ObjectId("54d894f800bc8bbcd882d4156"), "nombre" : "Juan", "edad" : 45 >
> juan=db.prueba.findOne({ "nombre": "Juan", "edad": 33 })
< "_id" : ObjectId("54d894f800bc8bbcd882d4155"),
  "nombre" : "Juan",
  "edad" : 33
> juan.edad++;
33
> db.prueba.update({ "nombre": "Juan" }, juan);
WriteResult({
  "nMatched" : 0,
  "nUpserted" : 0,
  "nModified" : 0,
  "writeError" : {
    "code" : 16837,
    "errmsg" : "The _id field cannot be changed from < id: ObjectId('54d894f800bc8bbcd882d4154')> to < id: ObjectId('54d894f800bc8bbcd882d4155')>"}
})
>
```

Se produce un error, dado que el método update busca un documento que encaje con la condición de búsqueda y el primero que encuentra es el referido al «Juan», que tiene 32 años. Intenta cambiar ese documento por el actualizado, y se encuentra que, si hace el cambio, habría dos documentos con el mismo «\_id», y eso no es posible (el «\_id» debe ser único). Para evitar estas situaciones, lo mejor es usar el método update con el campo «\_id», que es único. En el ejemplo anterior se podría hacer la actualización si se hiciera de la siguiente forma (figura 30):

**Figura 30.** Ejemplo de actualización correcta.

```
> db.prueba.update({ "_id": ObjectId("54d894f800bc8bbcd882d4155") }, juan);
WriteResult({
  "nMatched" : 1,
  "nUpserted" : 0,
  "nModified" : 1
})
```

Otra ventaja de usar el campo «\_id» es que el documento está indexado por este campo.

- Modificadores

En muchas ocasiones, el tipo de actualización que se quiere realizar consiste en añadir, modificar o eliminar claves, manipular

arrays y documentos embebidos, etc. Para estos casos se va a usar un conjunto de operadores de modificación.

- `$inc`. Este operador permite cambiar el valor numérico de una clave que ya existe incrementando su valor por el especificado junto al operador, o bien puede crear una clave que no existía inicializándola al valor dado.

Por ejemplo, supóngase que se mantienen los datos estadísticos de un sitio web en una colección de manera que se incrementa un contador cada vez que alguien visita una página. Para ello se tiene un documento que almacena la URL y el número de visitas de la página (figura 31).

**Figura 31.** Documento que almacena información de una página web.

```
> db.prueba.findOne()
{
  "_id" : ObjectId("54d89c5f0bc8bbed882d4158"),
  "URL" : "www.ejemplo.es",
  "visitas" : 34
}
```

Cada vez que alguien visita una página se busca la página a partir de su URL y se incrementa el campo de «visitas» con el modificador `«$inc»`, que incrementa el campo dado en el valor descrito (figura 32):

**Figura 32.** Ejemplo de uso del modificador `$inc`.

```
> db.prueba.update({ "URL": "www.ejemplo.es" }, { "$inc": { "visitas": 1 } })
> db.prueba.findOne()
{
  "_id" : ObjectId("54d89c5f0bc8bbed882d4158"),
  "URL" : "www.ejemplo.es",
  "visitas" : 35
}
```

También sería posible incrementar el valor por un valor mayor que 1 (figura 33).

**Figura 33.** Ejemplo de incremento de un valor mayor que 1.

```
> db.prueba.findOne()
{
    "_id" : ObjectId("54d8a9e00bc8bbbed882d415d"),
    "URL" : "www.ejemplo.es",
    "visitas" : 34
}
> db.prueba.update({ "URL": "www.ejemplo.es"}, { "$inc": { "visitas": 30 } })
writeResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.prueba.findOne()
{
    "_id" : ObjectId("54d8a9e00bc8bbbed882d415d"),
    "URL" : "www.ejemplo.es",
    "visitas" : 64
}
>
```

Y de la misma forma se podría decrementar usando números negativos (figura 34).

**Figura 34.** Ejemplo de decremento de un valor.

```
> db.prueba.update({ "URL": "www.ejemplo.es"}, { "$inc": { "visitas": -37 } })
writeResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.prueba.findOne()
{
    "_id" : ObjectId("54d8a9e00bc8bbbed882d415d"),
    "URL" : "www.ejemplo.es",
    "visitas" : 27
}
>
```

Y por ejemplo, se podría añadir un nuevo campo para indicar el número de enlaces de la página (figura 35):

**Figura 35.** Inserción de un nuevo campo en el documento.

```
> db.prueba.update({ "URL": "www.ejemplo.es"}, { "$inc": { "enlaces": 20 } })
writeResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.prueba.findOne()
{
    "_id" : ObjectId("54d8a9e00bc8bbbed882d415d"),
    "URL" : "www.ejemplo.es",
    "visitas" : 27,
    "enlaces" : 20
}
>
```

Hay que observar:

- Cuando se usan operadores de modificación el valor del campo «\_id», no puede ser cambiado (en cambio, cuando se reemplaza un documento entero, sí es posible cambiar el campo «\_id»). Sin

embargo, los valores para cualquier otra clave incluyendo claves indexadas únicas sí pueden ser modificadas.

- Este operador solo puede ser usado con números enteros, enteros largos o double, de manera que, si se usa con otro tipo de valores (incluido los tipos que algunos lenguajes tratan como números tales como booleanos, cadenas de números, nulos, etc.), producirá un fallo. En el ejemplo se intenta incrementar un campo con un valor no numérico (figura 36).

**Figura 36.** Error producido al intentar incrementar con un valor no numérico

```
> db.prueba.update({ "URL": "www.ejemplo.es"}, { "$inc": { "enlaces": "20" } })
{
  "nMatched": 0,
  "nUpserted": 0,
  "nModified": 0,
  "writeError": {
    "code": 14,
    "errmsg": "Cannot increment with non-numeric argument: enlaces
: \"20\""
  }
}
```

- «\$set» y «\$unset». Este operador establece un valor para un campo dado, y, si el campo dado no existe, lo crea. En este sentido, es útil para modificar el esquema de un documento o añadir claves definidas por el usuario. Por ejemplo, supóngase que se tiene el perfil de usuario almacenado en un documento (figura 37):

**Figura 37.** Ejemplo de documento con el perfil de un usuario.

```
> db.prueba.findOne()
{
  "_id": ObjectId("54d8a03e0bc8bbed882d415a"),
  "nombre": "Juan",
  "edad": 34,
  "sexo": "Varon",
  "localizacion": "Madrid"
}
```

Si el usuario desea añadir un campo sobre su libro favorito, se podría hacer usando el modificador «\$set» (figura 38):

**Figura 38.** Ejemplo de uso del modificador \$set para añadir un campo.

```
> db.prueba.update({ "_id": ObjectId("54d8a03e0bc8bbcd882d415a") },
... { "$set": { "libroFavorito": "Guerra y Paz" } })
> db.prueba.writeResult()
> db.prueba.findOne()
{
    "_id" : ObjectId("54d8a03e0bc8bbcd882d415a"),
    "nombre" : "Juan",
    "edad" : 34,
    "sexo" : "Varon",
    "localizacion" : "Madrid",
    "libroFavorito" : "Guerra y Paz"
}
```

Nuevamente, si el usuario desea cambiar el valor del campo sobre su libro favorito por otro valor, se podría hacer usando también el modificador «\$set» (figura 39):

**Figura 39.** Ejemplo de uso del modificador \$set para cambiar un valor.

```
> db.prueba.update({ "_id": ObjectId("54d8a03e0bc8bbcd882d415a") }, { "$set": { "libroFavorito": "El Quijote" } })
> db.prueba.writeResult()
> db.prueba.findOne()
{
    "_id" : ObjectId("54d8a03e0bc8bbcd882d415a"),
    "nombre" : "Juan",
    "edad" : 34,
    "sexo" : "Varon",
    "localizacion" : "Madrid",
    "libroFavorito" : "El Quijote"
}
```

También con el modificador «\$set» es posible cambiar el tipo de un campo que se modifica. Por ejemplo, si se quiere que el campo «libroFavorito» sea un array en vez de un valor único, también puede usarse el modificador «\$set» (figura 40):

**Figura 40.** Ejemplo de uso del modificador \$set para insertar un array.

```
> db.prueba.update({ "_id": ObjectId("54d8a03e0bc8bbcd882d415a") }, { "$set": { "libroFavorito": [ "Guerra y Paz", "El Quijote" ] } })
> db.prueba.writeResult()
> db.prueba.findOne()
{
    "_id" : ObjectId("54d8a03e0bc8bbcd882d415a"),
    "nombre" : "Juan",
    "edad" : 34,
    "sexo" : "Varon",
    "localizacion" : "Madrid",
    "libroFavorito" : [
        "Guerra y Paz",
        "El Quijote"
    ]
}
```

También mediante el operador «\$set» es posible realizar cambios en documentos embebidos, para lo cual solo es necesario indicar el campo en el que se encuentran. Por ejemplo, considerar el siguiente documento (figura 41):

**Figura 41.** Ejemplo de documento con elementos embebidos.

```
> db.prueba.findOne()
<
  "_id" : ObjectId("54d8a48f0bc8bbed882d415b"),
  "titulo" : "Post de un blog",
  "contenido" : "el contenido",
  "autor" : {
    "nombre" : "Isabel",
    "email" : "isa@prueba.com"
  }
>
```

Se quiere cambiar el valor del campo del nombre del autor del post, entonces se haría lo siguiente (figura 42):

**Figura 42.** Modificación de un elemento embebido con \$set.

```
> db.prueba.update({ "autor.nombre": "Isabel"}, { "$set": { "autor.nombre": "Isabel Sanz" } })
> db.prueba.findOne()
<
  "_id" : ObjectId("54d8a48f0bc8bbed882d415b"),
  "titulo" : "Post de un blog",
  "contenido" : "el contenido",
  "autor" : {
    "nombre" : "Isabel Sanz",
    "email" : "isa@prueba.com"
  }
>
```

Hay que observar:

- Existe un operador denominado «\$unset» que permite eliminar campos de un documento. En el ejemplo anterior, si se quiere eliminar el campo «libroFavorito» (figura 43):

**Figura 43.** Ejemplo de eliminación con \$unset.

```
> db.prueba.findOne()
{
  "_id" : ObjectId("54d8a6230bc8bbcd882d415c"),
  "nombre" : "Juan",
  "edad" : 34,
  "sexo" : "Varon",
  "localizacion" : "Madrid",
  "libroFavorito" : [
    "Guerra y Paz",
    "El Quijote"
  ]
}
> db.prueba.update({"_id" : ObjectId("54d8a6230bc8bbcd882d415c")}, {"$unset": {"libroFavorito": 1}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.prueba.findOne()
{
  "_id" : ObjectId("54d8a6230bc8bbcd882d415c"),
  "nombre" : "Juan",
  "edad" : 34,
  "sexo" : "Varon",
  "localizacion" : "Madrid"
}
>
```

- Para añadir, modificar o eliminar claves se debe usar siempre los modificadores «\$». En este sentido, hay que observar que, si intentara hacer un cambio en las claves con un comando como el siguiente: db.prueba.update (criterio, {"edad": "país"}), tendría como efecto reemplazar el documento que encaje con el criterio de búsqueda por el documento {"edad": "país"}.

## 5. Modificadores de los arrays

En MongoDB, los arrays son estructuras de datos de primer nivel que disponen de un conjunto de modificadores propios:

- Adición de elementos.* El modificador «\$push» añade elementos al final del array si existe o bien crea uno nuevo si no existe. Por ejemplo, supóngase que se almacenan posts de un blog y se quiere añadir una clave «comentarios» que contenga un array de comentarios. Esto puede hacerse usando el modificador «\$push», que en el ejemplo crea una nueva clave denominada «comentarios» (figura 44):

**Figura 44.** Ejemplo de creación de una nueva clave con \$push.

```
> db.prueba.findOne()
<
  "_id" : ObjectId("54d8c0a60bc8bbed882d415f"),
  "titulo" : "Posts de un blog",
  "contenido" : "el contenido"
>
> db.prueba.update({ "titulo": "Posts de un blog"}, { "$push": { "comentarios": {
    "nombre": "Juan",
    "email": "juan@ejemplo.com",
    "contenido": "buen post." } } })
> WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.prueba.findOne()
<
  "_id" : ObjectId("54d8c0a60bc8bbed882d415f"),
  "titulo" : "Posts de un blog",
  "contenido" : "el contenido",
  "comentarios" : [
    {
      "nombre" : "Juan",
      "email" : "juan@ejemplo.com",
      "contenido" : "buen post."
    }
  ]
>
```

A continuación, si se quiere añadir otro comentario, se usa nuevamente el modificador «\$push» (figura 45):

**Figura 45.** Ejemplo de inserción de datos usando \$push.

```
> db.prueba.update({ "titulo": "Posts de un blog"}, { "$push": { "comentarios": {
    "nombre": "Isabel",
    "email": "isabel@ejemplo.com",
    "contenido": "buen post." } } })
> WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.prueba.findOne()
<
  "_id" : ObjectId("54d8c2d40bc8bbed882d4160"),
  "titulo" : "Posts de un blog",
  "contenido" : "el contenido",
  "comentarios" : [
    {
      "nombre" : "Juan",
      "email" : "juan@ejemplo.com",
      "contenido" : "buen post."
    },
    {
      "nombre" : "Isabel",
      "email" : "isabel@ejemplo.com",
      "contenido" : "buen post."
    }
  ]
>
```

Este modificador también puede usarse junto al modificador «\$each» para añadir múltiples valores en una sola operación. Por ejemplo, si se quisiera añadir tres valores a un array denominado «horas» (figura 46):

**Figura 46.** Ejemplo de inserción utilizando \$push y \$each.

```
> db.prueba.update({"_id" : ObjectId("54d8c58e0bc8bbed882d4162")}, {"$push": {"horas": {"$each": [34, 56, 33]}}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.prueba.findOne()
{
    "_id" : ObjectId("54d8c58e0bc8bbed882d4162"),
    "horas" : [
        34,
        56,
        33
    ]
}
>
```

Hay que observar que si se especifica un array con un único elemento, su comportamiento es similar a un «\$push» sin «\$each».

También es posible limitar la longitud hasta la que puede crecer un array usando el operador «\$slice» junto al operador «\$push». Por ejemplo, en el siguiente array se limita el crecimiento hasta 10 elementos, de manera que si se introducen 10 elementos se guardan todos, pero si se introducen más de 10 elementos entonces solo se guardan los 10 últimos elementos (figura 47):

**Figura 47.** Ejemplo de uso de \$slice.

```
> db.peliculas.update({"genero": "historicas"}, {"$push": {"top10": {"$each": ["Los cañones del Navarone", "El Cid Campeador"]}}, {"$slice": -10}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.peliculas.findOne()
{
    "_id" : ObjectId("54d8d4690bc8bbed882d4163"),
    "genero" : "historicas",
    "top10" : [
        "Los cañones del Navarone",
        "El Cid Campeador"
    ]
}
```

Hay que observar que el operador «\$slice» puede tomar valores negativos (empieza a contar desde el final) o bien valores positivos (empieza a contar desde el principio) y esencialmente actúa creando una cola o una pila en el documento.

Por último, el operador «\$sort» permite ordenar los elementos indicando el campo de ordenación y el criterio en forma

de 1(ascendente) 0-1(descendente). En el siguiente ejemplo se guardan los primeros 10 elementos ordenados de acuerdo con el campo «valoración» en orden ascendente (figura 48):

**Figura 48.** Ejemplo de uso de \$sort

```
> db.peliculas.update({ "genero" : "historica"}, { "$push" : { "top10" : { "$each" : [ { "nombre" : "Los cañones del Navarone", "valoracion" : 6.6}, { "nombre" : "El Cid canpeador", "valoracion" : 4.3} ]}, "$slice" : -10, "$sort" : { "valoracion" : -1 } })
> writeResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.peliculas.findOne()
{
  "_id" : ObjectId("54d8d8030bc8bbed882d4165"),
  "genero" : "historica",
  "top10" : [
    {
      "nombre" : "Los cañones del Navarone",
      "valoracion" : 6.6
    },
    {
      "nombre" : "El Cid canpeador",
      "valoracion" : 4.3
    }
  ]
}
>
```

Hay que observar que:

- Tanto «\$slice» como «\$sort» deben ir junto a un operador «\$each» y no pueden aparecer solos con un «\$push».
- «\$sort» también puede ser usado para ordenar elementos que no son documentos, en cuyo caso no hay que indicar ningún campo. En el siguiente ejemplo se insertan dos elementos y se ordena el conjunto de manera ascendente (figura 49):

**Figura 49.** Ejemplo de ordenación ascendente con \$sort.

```
> db.estudiantes.findOne()
{ "_id" : 2, "tests" : [ 89, 70, 89, 50 ] }
> db.estudiantes.findOne()
{ "_id" : 2, "tests" : [ 89, 70, 89, 50 ] }
> db.estudiantes.update( { "_id: 2 }, { $push: { tests: { $each: [ 40, 60 ] } }, $sort: 1 } )
> writeResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.estudiantes.findOne()
{ "_id" : 2, "tests" : [ 40, 50, 60, 70, 89, 89 ] }
>
```

También es posible ordenar un array completo especificando un array vacío junto al operador «\$each». En el siguiente ejemplo se ordena el array en orden descendente (figura 50):

**Figura 50.** Ejemplo de ordenación descendente con \$sort.

```
> db.estudiantes.findOne()
{ "_id" : 2, "tests" : [ 40, 50, 60, 70, 89, 89 ] }
> db.estudiantes.update( { "_id: 2 }, { $push: { tests: { $each: [ 1, $sort
: -1 } ] } })
> writeResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.estudiantes.findOne()
{ "_id" : 2, "tests" : [ 89, 89, 70, 60, 50, 40 ] }
```

- Si «\$slice» se usa con un array vacío, se aplica a los elementos actuales del array. Por ejemplo, en el siguiente ejemplo se reduce la longitud del array, y nos quedamos con el primer elemento (figura 51):

**Figura 51.** Ejemplo de reducción de la longitud de un array con \$slice.

```
> db.peliculas.findOne()
{
  "_id" : ObjectId("54d8d8030bc8bbbed882d4165"),
  "genero" : "historica",
  "top10" : [
    {
      "nombre" : "Los cañones del Navarone",
      "valoracion" : 6.6
    },
    {
      "nombre" : "El Cid campeador",
      "valoracion" : 4.3
    }
  ]
}
> db.peliculas.update({ "_id" : ObjectId("54d8d8030bc8bbbed882d4165") },
... { "$push": { "top10": { "$each": [ ], "$slice": 1 } } })
> writeResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.peliculas.findOne()
{
  "_id" : ObjectId("54d8d8030bc8bbbed882d4165"),
  "genero" : "historica",
  "top10" : [
    {
      "nombre" : "Los cañones del Navarone",
      "valoracion" : 6.6
    }
  ]
}
```

- Usar arrays como conjuntos.* Los arrays se pueden tratar como un conjunto añadiendo valores solo si no estaban ya. Para ello se

usa el operador «\$ne» junto al operador «\$push». Por ejemplo, si se quiere añadir un autor a una lista de citas pero solo en el caso de que no estuviera, se podría hacer de la siguiente manera (figura 52):

**Figura 52.** Ejemplo de uso del operador \$ne junto \$push.

```
< db.articulos.findOne()
<   "_id" : ObjectId("54d8e24b0bc8bbed882d4166"),
<     "autores_citados" : [
<       "Juan",
<       "Pablo"
<     ]
>
> db.articulos.update({ "autores_citados": { "$ne": "Pepe" } }, { "$push": { "autores_citados": "Pepe" } })
> writeResult = db.articulos.update({ "autores_citados": { "$ne": "Pepe" } }, { "$push": { "autores_citados": "Pepe" } })
> db.articulos.findOne()
<   "_id" : ObjectId("54d8e24b0bc8bbed882d4166"),
<     "autores_citados" : [
<       "Juan",
<       "Pablo",
<       "Pepe"
<     ]
>
```

Alternativamente, también es posible hacer la misma operación mediante el operador «\$addToSet». Por ejemplo, supóngase que se tiene un documento que representa a un usuario y se dispone de una clave que es un array de direcciones de correo electrónico, entonces, si se quiere añadir una nueva dirección sin que se produzcan duplicados, se puede hacer de la siguiente manera mediante «\$addToSet» (figura 53):

**Figura 53.** Ejemplo de uso del operador \$addToSet.

```
> db.usuarios.findOne()
{
  "_id" : ObjectId("54d8e4aa0bc8bbcd882d4167"),
  "nombre" : "Isabel",
  "emails" : [
    "isa@ejemplo.com",
    "isabel@mail.com",
    "isabels@yahoo.es"
  ]
}
> db.usuarios.update(
...   "_id" : ObjectId("54d8e4aa0bc8bbcd882d4167"),
...   {"$addToSet": {"emails": "isa@gmail.com"}}
)
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.usuarios.findOne()
{
  "_id" : ObjectId("54d8e4aa0bc8bbcd882d4167"),
  "nombre" : "Isabel",
  "emails" : [
    "isa@ejemplo.com",
    "isabel@mail.com",
    "isabels@yahoo.es",
    "isa@gmail.com"
  ]
}
>
```

Es posible utilizar «\$addToSet» junto al operador «\$each» para añadir valores múltiples únicos, lo cual no puede ser hecho con la combinación «\$ne» / «\$push». Por ejemplo, si quisiéramos añadir más de una dirección de correo electrónico, se podría hacer de la siguiente manera (figura 54):

**Figura 54.** Ejemplo de utilización conjunta de \$addToSet y \$each.

```
> db.usuarios.findOne()
{
  "_id" : ObjectId("54d8e4aa0bc8bbcd882d4167"),
  "nombre" : "Isabel",
  "emails" : [
    "isa@ejemplo.com",
    "isabel@mail.com",
    "isabels@yahoo.es",
    "isa@gmail.com"
  ]
}
> db.usuarios.update(
...   "_id" : ObjectId("54d8e4aa0bc8bbcd882d4167"),
...   {"$addToSet": {"emails": {"$each": ["isa@php.com", "isa@python.com", "isa@java.com"]}}}
)
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.usuarios.findOne()
{
  "_id" : ObjectId("54d8e4aa0bc8bbcd882d4167"),
  "nombre" : "Isabel",
  "emails" : [
    "isa@ejemplo.com",
    "isabel@mail.com",
    "isabels@yahoo.es",
    "isa@gmail.com",
    "isa@php.com",
    "isa@python.com",
    "isa@java.com"
  ]
}
>
```

- *Borrado de elementos.* Existen varias formas de eliminar elementos de un array dependiendo de la forma en la que se quieran gestionar. Si se quiere gestionar como si fuera una pila o una cola, se puede usar el operador «\$pop», que permite eliminar elementos del final del array (si toma el valor 1) o bien del principio del array (si toma el valor -1) (figura 55):

**Figura 55.** Ejemplo de uso de \$pop

```
> db.prueba.findOne()
< "_id" : 1, "scores" : [ 8, 9, 10 ] >
> db.prueba.update({ "_id": 1 }, { $pop: { "scores": -1 } })
> writeResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.prueba.findOne()
< "_id" : 1, "scores" : [ 9, 10 ] >
> db.prueba.update({ "_id": 1 }, { $pop: { "scores": 1 } })
> writeResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.prueba.findOne()
< "_id" : 1, "scores" : [ 9 ] >
```

Otra forma alternativa de eliminar elementos es especificando un criterio en vez de una posición en el array usando el operador «\$pull». Por ejemplo, si se tiene un conjunto de tareas que se tienen que realizar en un orden dado (figura 56):

**Figura 56.** Ejemplo de uso de \$pull.

```
> db.listas.findOne()
< "_id" : ObjectId("54d8eb3f0bc8bbcd882d4168"),
  "tareas" : [
    "desayunar",
    "coner",
    "merendar",
    "cenar"
  ]
> db.listas.update({ "_id" : ObjectId("54d8eb3f0bc8bbcd882d4168") }, { "$pull" : { "tareas" : "coner" } })
> writeResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.listas.findOne()
< "_id" : ObjectId("54d8eb3f0bc8bbcd882d4168"),
  "tareas" : [
    "desayunar",
    "merendar",
    "cenar"
  ]
>
```

Hay que observar que el operador «\$pull» elimina todas las coincidencias que encuentre en los documento, no solo la primera coincidencia. Por ejemplo, la eliminación del valor 1 en el siguiente array numérico lo dejará con un solo elemento (figura 57):

**Figura 57.** Ejemplo de vaciado de un array con \$pull.

```
> db.lista.findOne()
{
  "_id" : ObjectId("54d8ed800bc8bbcd882d4169"),
  "aciertos" : [
    1,
    2,
    1,
    1,
    1
  ]
}
> db.lista.update({"_id" : ObjectId("54d8ed800bc8bbcd882d4169")}, {"$pull": {"aciertos": 1}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.lista.findOne()
{
  "_id" : ObjectId("54d8ed800bc8bbcd882d4169"),
  "aciertos" : [ 2 ]
}
```

Hay que observar que los operadores de array solo pueden ser usados sobre claves con array de valores, y no es posible modificar valores escalares (para ello se usa «\$set» o «\$inc»).

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

**Figura 58.** Ejemplo manipulación de los valores de un array por su posición.

```

< db.blog.posts.findOne()
<   "_id" : ObjectId("4b329a216cc613d5ee930192"),
<   "contenido" : "el contenido",
<   "comentarios" : [
<     {
<       "comentario" : "buen post",
<       "autor" : "Juan",
<       "votos" : 8
<     },
<     {
<       "comentario" : "un poco corto",
<       "autor" : "Clara",
<       "votos" : 3
<     },
<     {
<       "comentario" : "no está mal",
<       "autor" : "Alicia",
<       "votos" : -1
<     }
<   ]
>
> db.blog.posts.update({"_id" : ObjectId("4b329a216cc613d5ee930192")}, {"$inc": {"comentarios.0.votos":1}})
> writeResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.blog.posts.findOne()
<   "_id" : ObjectId("4b329a216cc613d5ee930192"),
<   "contenido" : "el contenido",
<   "comentarios" : [
<     {
<       "comentario" : "buen post",
<       "autor" : "Juan",
<       "votos" : 9
<     },
<     {
<       "comentario" : "un poco corto",
<       "autor" : "Clara",
<       "votos" : 3
<     },
<     {
<       "comentario" : "no está mal",
<       "autor" : "Alicia",
<       "votos" : -1
<     }
<   ]
>

```

- *Mediante su posición.* En este caso, sus elementos son seleccionados como si se indexaran las claves de un documento. Por ejemplo, supóngase que se tiene un array con documentos embebidos tales como los comentarios a un post de un blog y se quiere incrementar el número de votos del primer comentario; entonces, se puede hacer tal como se muestra en figura 58.
- En algunas ocasiones no se conoce el índice del array que se quiere modificar sin antes consultar el documento. Para estos casos se dispone de un operador posicional «\$\$», que indica el elemento del array que encaja con la condición de búsqueda y actualiza ese elemento. Si en el ejemplo anterior se tiene un autor de un comentario que se llama «Juan» y se quiere actualizar a «Juanito», se puede hacer de la siguiente manera con el operador posicional (figura 59):

**Figura 59.** Ejemplo manipulación de los valores de un array mediante el operador posicional.

```
> db.blog.posts.update({ "comentarios.autor": "Juan"}, { "$set": { "comentarios.$.autor": "Juanito" } })
WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.blog.posts.findOne()
{
  "_id": ObjectId("4b329a216cc613d5ee930192"),
  "contenido": "el contenido",
  "comentarios": [
    {
      "comentario": "buen post",
      "autor": "Juanito",
      "votos": 1
    },
    {
      "comentario": "un poco corto",
      "autor": "Clara",
      "votos": 3
    },
    {
      "comentario": "no está mal",
      "autor": "Alicia",
      "votos": -1
    }
  ]
}
```

Hay que observar que el operador posicional solo actualiza la primera coincidencia, de manera que, si hubiera más de un comentario realizado por «Juan», solo se actualizaría el primer comentario.

## 6. Upsert

Un upsert es un tipo especial de actualización. Si no se encuentra ningún documento que coincida con el criterio de búsqueda, entonces se crea un nuevo documento que combina el criterio de búsqueda y la actualización. Si se encuentra un documento que encaja con la condiciones de búsqueda, será actualizado de forma normal. Este tipo de actualización puede ser muy útil para generar una colección si se tiene el mismo código para crear y actualizar documentos.

Considérese el ejemplo anterior sobre el almacenamiento del número de visitas de cada página de un sitio web. Sin un upsert, se podría intentar encontrar la URL e incrementar el número de visitas o crear un nuevo documento si la URL no existe. Por ejemplo, se podría implementar mediante un programa en JavaScript:

```
// Se cheque si existe alguna entrada en la página
blog = db.analisis.findOne ({url: “/blog”})
// Si existe, se actualiza, se añade uno y se almacena
if (blog) {
    blog.paginasvisitadas++;
    db.analisis.save (blog);
}
// En caso contrario, se crea un nuevo documento
else {
    db.analisis.save ({url: “/blog”, paginasvisitadas: 1})
}
```

Todo este código se podría reducir si se usa un upsert (tercer parámetro de un update) con las ventajas de ser una operación atómica y mucho más rápida:

```
db.analisis.update ({“url”: “/blog”}, {“$inc”: {“paginasvisitadas”: 1}}, true)
```

El nuevo documento es creado usando la especificación dada del documento y aplicando las modificaciones descritas. Por ejemplo, si se hace un upsert sobre la clave de un documento y se tiene especificada como modificación el incremento del valor de la clave, entonces se aplicará este incremento (figura 60).

**Figura 60.** Ejemplo de aplicación del upsert.

```
> db.users.update({rep : 25}, {"$inc" : {"rep" : 3}}, true)
WriteResult({
    "nMatched" : 0,
    "nUpserted" : 1,
    "nModified" : 0,
    "_id" : ObjectId("54da09f27f193a008b23b537")
})
> db.users.findOne({_id : ObjectId("54da09f27f193a008b23b537")}, "rep" : 28)
> =
```

El upsert crea un nuevo documento con valor 25 para la clave «rep» y a continuación lo incrementa en 3, generando un documento donde la clave «rep» toma el valor de 28. Si no se hubiera especificado la opción de upsert, entonces no habría encontrado ningún documento que encajara con las condiciones dadas y no habría hecho nada. Hay que observar que, si se vuelve a correr la misma actualización, pasará lo mismo, pues seguirá sin encontrar un documento que encaje con las condiciones de búsqueda, por lo que creará otro documento nuevamente.

A veces, un campo necesita ser inicializado cuando es creado pero no cambiado en actualizaciones posteriores. Para ello se usa el modificador «\$setOnInsert» junto a upsert activado (valor a cierto). Este modificador lo que hace es que inicializa el valor de una clave cuando el documento es insertado, pero, si vuelve a ejecutarse, no hace nada sobre la clave que ha inicializado en el documento que se ha insertado. En el ejemplo (figura 61), la primera vez crea un documento y actualiza el campo «creadoEn», pero, si se volviera a ejecutar la misma actualización, se podría ver que no se actualiza nuevamente el campo «creadoEn».

**Figura 61.** Ejemplo de uso del modificador \$setOnInsert.


The screenshot shows a terminal window titled 'C:\mongodb\bin\mongo.exe' running on Windows. The command entered is:

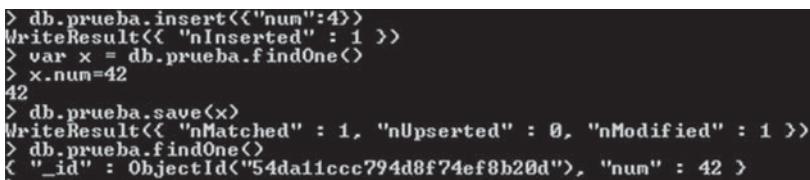
```

> db.usuarios.update({},{ "$setOnInsert": { "creadoEn": new Date() } }, true)
WriteResult({
    "nMatched": 0,
    "nUpserted": 1,
    "nModified": 0,
    "_id": ObjectId("56598940e71200f54df861ec")
})
> db.usuarios.findOne()
{
    "_id": ObjectId("56598940e71200f54df861ec"),
    "creadoEn": ISODate("2015-11-28T11:00:16.762Z")
}
>

```

Hay que observar que «\$setOnInsert» puede ser útil para la inicialización de contadores, colecciones que no usan «ObjectIds» o para crear rellenos en los documentos.

Relacionado con upsert está la función de la Shell «save», que permite insertar un documento si no existe y actualizarlo si existe. Toma como argumento un documento. Si el documento contiene un «\_id», entonces «save» hará un upsert, y en caso contrario lo insertará (figura 62):

**Figura 62.** Ejemplo de uso del comando «save» de la Shell.


```

> db.prueba.insert({ "num": 42 })
WriteResult({ "nInserted": 1 })
> var x = db.prueba.findOne()
> x.num = 42
42
> db.prueba.save(x)
WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.prueba.findOne()
{
    "_id": ObjectId("54da11ccc794d8f74ef8b20d"),
    "num": 42
}

```

El comando «save» está pensado para facilitar la modificación de documentos en la Shell. Sin el «save», habría que haber hecho un update de la siguiente forma (figura 63):

**Figura 63.** Actualización equivalente a «save» utilizando «update».

```
> db.prueba.update({ "_id": x._id }, x)
```

## 7. Actualización de múltiples documentos

Las actualizaciones por defecto solo modifican el primer documento que encaja con los criterios de búsqueda, de manera que, si existen más documentos que cumplen las condiciones, no se cambian. Para que los cambios se realicen en todos los documentos que coinciden, entonces hay que pasar como cuarto parámetro del update un valor true. Por ejemplo, considérese el caso de que se quiere dar un regalo a cada usuario en el día de su cumpleaños; en este caso se podría realizar una actualización múltiple de la siguiente manera:

```
db.users.update ({“cumpleaños”: “10/13/1978”},  
... {“$set”: {“regalo”: “[Feliz cumpleaños!]”}}, false, true)
```

Con esta expresión se añadiría el mensaje de «[Feliz cumpleaños!]» a todos los documentos en los que el campo «cumpleaños» tuviera el valor de 13 de octubre de 1978.

Para ver el número de documentos que han sido actualizados mediante una actualización múltiple se puede usar el comando `getLastError`, que retorna información acerca de la última operación que ha sido realizada. La clave `«n»` contiene el número de documentos que se han visto afectados por una actualización (figura 64).

**Figura 64.** Obtención del número de documentos actualizados.

```
> db.count.insert(<“x”:1>)  
WriteResult<< “nInserted” : 1 >>  
> db.count.update(<x : 1>, <$inc : {x : 1}>, false, true)  
WriteResult<< “nMatched” : 1, “nUpserted” : 0, “nModified” : 1 >>  
> db.runCommand(<getLastError : 1>)  
< “connectionId” : 4,  
  “updatedExisting” : true,  
  “n” : 1,  
  “syncMillis” : 0,  
  “writtenTo” : null,  
  “err” : null,  
  “ok” : 1  
>
```

Se puede observar que `n=1` indica que fue actualizado un documento y, por otra parte, `«updateExisting»` toma el valor de `«true»`, que indica que algún documento fue actualizado.

## 8. Eficiencia de las modificaciones

Algunos modificadores son más rápidos que otros. Así, por ejemplo, `«$inc»` modifica un documento variando unos pocos bytes, por lo que es muy eficiente; sin embargo, los modificadores de array pueden variar considerablemente el tamaño de un array, por lo que son lentos (`«$set»` puede modificar documentos de manera eficiente si no cambia el tamaño, pero en caso contrario tiene las mismas limitaciones que los operadores de array).

La eficiencia viene determinada por la forma en que se insertan los documentos. Cuando se empieza a insertar documentos, cada documento se pone junto a los previos en el disco, de manera que, si un documento crece de tamaño, no podrá colocarse en el sitio original y tendrá que ser movido a otra parte de la colección. Para ver cómo funciona se puede crear una colección con un par de documentos de manera que al documento que se encuentra en la mitad se le cambia de tamaño, y entonces será empujado al final de la colección (figura 65).

**Figura 65.** Creación de un par de documentos.

```
> db.coll.insert({ "x" : "a" })
WriteResult({ "nInserted" : 1 })
> db.coll.insert({ "x" : "b" })
WriteResult({ "nInserted" : 1 })
> db.coll.insert({ "x" : "c" })
WriteResult({ "nInserted" : 1 })
> db.coll.find()
{ "_id" : ObjectId("54d908950bc8bbcd882d416d"), "x" : "a" }
{ "_id" : ObjectId("54d908970bc8bbcd882d416e"), "x" : "b" }
{ "_id" : ObjectId("54d908980bc8bbcd882d416f"), "x" : "c" }
> db.coll.update({ "x" : "b" }, { $set: { "x" : "bbbbbbbbbbbbbbbbbbbbbbbbbb" } })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.coll.find()
{ "_id" : ObjectId("54d908950bc8bbcd882d416d"), "x" : "a" }
{ "_id" : ObjectId("54d908970bc8bbcd882d416f"), "x" : "c" }
{ "_id" : ObjectId("54d908980bc8bbcd882d416e"), "x" : "bbbbbbbbbbbbbbbbbbbbbbbbbb" }
>
```

Cuando MongoDB tiene que mover un documento, aplica un factor de relleno a la colección, que es la cantidad de espacio extra que deja alrededor de cada documento por si crecen en tamaño. En el ejemplo anterior se puede ver mediante el método stats () (figura 66):

**Figura 66.** Ejecución del método stats () .

```
> db.coll.stats()
{
  "ns" : "prueba.coll",
  "count" : 3,
  "size" : 208,
  "avgObjSize" : 69,
  "storageSize" : 8192,
  "numExtents" : 1,
  "nindexes" : 1,
  "lastExtentSize" : 8192,
  "paddingFactor" : 1,
  "systemFlags" : 1,
  "userFlags" : 1,
  "totalIndexSize" : 8176,
  "indexSizes" : {
    "_id_" : 8176
  }
}
> "ok" : 1
```

Inicialmente, el factor de relleno es 1, puesto que asigna exactamente el tamaño del documento por cada nuevo documento. Cuando se ejecuta varias veces, las actualizaciones van haciendo más grandes los documentos, de manera que el factor de relleno crece alrededor de 1,5, dado que a cada nuevo documento se

le asignará la mitad del tamaño como espacio libre para poder crecer. Si posteriores actualizaciones producen movimientos de documentos, entonces el factor de relleno continuará creciendo aunque no tan rápidamente como en el primer caso. Si no existen más movimientos, el factor de relleno irá decreciendo lentamente. El movimiento de documentos es lento, dado que hay que liberar el espacio que ocupaba el documento y escribirlo en un nuevo sitio. Por tanto, hay que intentar mantener el factor de relleno tan cerca de 1 como sea posible. En general no se puede modificar manualmente el factor de relleno salvo que se esté realizando una compactación de datos, pero sí es posible realizar un esquema que no dependa de que los documentos se hagan grandes arbitrariamente.

En el siguiente programa se va a mostrar la diferencia entre la actualizaciones que no producen variación del tamaño del documento (y por tanto son eficientes) y las actualizaciones que requieren mover documentos. El programa inserta una clave e incrementa su valor 10.000 veces (figura 67):

**Figura 67.** Programa que incrementa 1.000 veces el valor de una clave usando \$inc.

```
> db.ejemplo.insert({ "x": 1 })
> writeResult({ "nInserted": 1 })
> var funcionTiempo=function() {
... var comienzo=new Date().getTime();
... for (var i=0; i<10000; i++) {
...   db.ejemplo.update({}, { "$inc": { "x": 1 } });
...   db.getLastError();
... }
... var diferencia=new Date().getTime()-comienzo;
... print("Tiempo tomado en actualizar: "+ diferencia + " ns");
... }
> funcionTiempo()
Tiempo tomado en actualizar: 34509 ns
```

Como se puede observar se tarda casi 35 segundos. Sin embargo, si se cambia la llamada de actualización y se usa un «\$push», tarda casi 57 segundos (figura 68):

**Figura 68.** Programa que incrementa 1.000 veces el valor de una clave usando \$push.

```
> db.ejemplo.insert({ "x": 1 })
> var writeResult = db.ejemplo.findOne();
> var funcionTiempo = function() {
...   var comienzo = new Date().getTime();
...   for (var i=0; i<10000; i++) {
...     db.ejemplo.update({}, { "$push": { "x": 1 } });
...     db.getLastError();
...   }
...   var diferencia = new Date().getTime() - comienzo;
...   print("Tiempo tomado en actualizar: " + diferencia + " ms");
... }
> funcionTiempo()
tiempo tomado en actualizar: 56184 ms
>
```

En conclusión, si se usa «\$push» y modificadores de array con frecuencia, hay que tener en cuenta las ventajas y desventajas que tienen. En este sentido, si «\$push» ralentiza las modificaciones, entonces puede merecer la pena colocar los documentos embebidos en una colección separada, rellenarlos manualmente o bien utilizar alguna otra técnica de optimización. Por otra parte, MongoDB no es bueno reusando espacio vacío, por lo que los cambios de documentos pueden dar lugar a grandes espacios vacíos de datos, y producir mensajes de advertencia avisando de tal situación para que se lleve a cabo una compactación, dado que hay un alto nivel de fragmentación.

Por otra parte, si el esquema que se va a usar requiere muchos movimientos de documentos, inserciones y borrados, se puede mejorar el uso del disco usando la opción usePowerOf2Sizes, que se puede configurar con el comando collMod:<sup>1</sup>

```
db.runCommand ({ "collMod": collectionName, "usePowerOf2Sizes": true })
```

Con esta opción activada, todas las asignaciones de memoria que se hagan serán en bloques de tamaño de potencia de 2, pero

---

<sup>1</sup> Hay que observar que, si se ejecuta nuevamente el comando con la opción «usePowerOf2Sizes» a «false», se vuelve al modo normal.

la asignación de espacio será menos eficiente. Por ejemplo, si se utiliza esta configuración en colecciones en las que solo se inserta o se hacen actualizaciones que no producen variación del tamaño, estas operaciones se realizarán de una manera más lenta. Así mismo, los efectos de esta opción solo afectan a los registros asignados nuevos, no a colecciones previamente existentes.

Por último, hay que mencionar que mediante el comando «`getLastError`» se puede obtener información acerca de las actualizaciones, pero no es posible recuperar los documentos que fueron actualizados. En caso de ser necesario se podría usar el comando «`findAndModify`», que retorna un documento al estado antes de ser modificado y lo actualiza en una sola operación. Los campos que tiene el comando son los siguientes:

- `findAndModify`: es una cadena que representa el nombre de la colección.
- `query`: es una consulta sobre un documento.
- `sort`(optativo): criterio de ordenación de los resultados
- `update`: un modificador de documento (actualización que realizar). Si no aparece esta opción, aparece la opción de borrar.
- `remove`: es un booleano que especifica si el documento debe ser eliminado. Si no aparece esta opción, entonces aparece la opción de actualizar.
- `new`: es un booleano que especifica si el documento retornado debe ser el documento actualizado o bien el documento previo a la actualización (esta es la opción por defecto).
- `fields` (optativo): los campos del documento que retornar.
- `upsert`: es un booleano que especifica si la actualización se debe comportar como un «`upsert`» o no (la opción por defecto es a `false`).

El comando puede tener un «update» clave o bien un «remove» clave, pero no ambos. En este último caso indica que el documento que coincide con las condiciones de búsqueda debería ser eliminado de la colección. Si ningún documento encaja, el comando devolverá un error.

## 9. Ejercicios propuestos

Considerar una colección con documentos de MongoDB que representan información multimedia de la forma:

```
{
  "tipo": "libro",
  "titulo": "Java para todos",
  "ISBN": "987-1-2344-5334-8",
  "editorial": "Anaya",
  "Autor": ["Pepe Caballero", "Isabel Sanz", "Timoteo Marino"],
  "capitulos": [
    {"capitulo": 1, "titulo": "Primeros pasos en Java", "longitud": 20},
    {"capitulo": 2, "titulo": "Primeros pasos en Java", "longitud": 25}
  ]
}

{
  "tipo": "CD",
  "Artista": "Los piratas",
  "Titulo": "Recuerdos",
  "canciones": [
    {"cancion": 1, "titulo": "Adiós mi barco", "longitud": "3:20"},
    {"cancion": 2, "titulo": "Pajaritos", "longitud": "4:15"}
  ]
}

{
  "tipo": "DVD",
  "Titulo": "Matrix",
  "estreno": 1999,
  "actores": [
    "Keanu Reeves",
    "Carry-Anne Moss",
    "Laurence Fishburne",
    "Hugo Weaving",
    "Gloria Foster",
    "Joe Pantoliano"
  ]
}
```

Realizar las siguientes operaciones:

- 1)** Insertar los documentos dados en una base de datos llamada «media» en una única operación.
- 2)** Actualizar el documento que hace referencia a la película «Matrix», de manera que se cambia su estructura a:

```
{“tipo”: “DVD”,  
“Titulo”: “Matrix”,  
“estreno”: 1999,  
“genero”：“accion”  
}
```

- 3)** Considerar un nuevo documento para la colección media:

```
{“tipo”: “Libro”,  
“Titulo”: “Constantinopla”,  
“capitulos”:12,  
“leidos”:3  
}
```

Añadir el documento a la colección media y a continuación incrementar en 5 unidades el valor de la clave «leídos».

- 4)** Actualizar el documento referido a la película «Matrix» cambiando el valor de la clave «género» a «ciencia ficción».
- 5)** Actualizar el documento referido al libro «Java para todos» de manera que se elimine la clave «editorial».
- 6)** Actualizar el documento referido al libro «Java para todos» añadiendo el autor «María Sancho» al array de autores.
- 7)** Actualizar el documento referido a la película «Matrix» añadiendo al array «actores» los valores de «Antonio Banderas» y «Brad Pitt» en una única operación.
- 8)** Actualizar el documento referido a la película «Matrix» añadiendo al array «actores» los valores «Joe Pantoliano», «Brad Pitt» y «Natalie Portman» en caso de que no se encuentren, y si se encuentran no se hace nada.

**9)** Actualizar el documento referido a la película «Matrix» eliminando del array el primer y último actor.

**10)** Actualizar el documento referido a la película «Matrix» añadiendo al array actores los valores «Joe Pantoliano» y «Antonio Banderas».

**11)** Actualizar el documento referido a la película «Matrix» eliminando todas las apariciones en el array «actores» de los valores «Joe Pantoliano» y «Antonio Banderas».

**12)** Actualizar el documento referido al disco «Recuerdos» y añadir una nueva canción al array «canciones»:

```
{“cancion”:5,  
“titulo”: “El atardecer”,  
“longitud”: “6:50”  
}
```

**13)** Actualizar el documento referido al disco «Recuerdos» de manera que la canción «El atardecer» tenga asignado el número 3 en vez de 5.

**14)** Actualizar el documento referido al disco «Recuerdos» de manera que en una sola operación se cambia el nombre del artista a «Los piratillas» y se muestre el documento resultante.

**15)** Renombrar el nombre de la colección «media» a «multimedia».

## Bibliografía

- Banker, K.** (2011). *MongoDB in action*. Manning Publications Co.
- Chodorow, K.** (2013). *MongoDB: the definitive guide*. O'Reilly Media, Inc.
- Hows, D.; Membrey, P; Plugge, E.** (2014). *MongoDB Basics*. Apress.
- Membrey, P.; Plugge, E.; Hawkins, D.** (2010). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress.
- Dayley, B.** (2015). *Sams Teach Yourself NoSQL with MongoDB in 24 Hours*. Sams.
- Sitio oficial MongoDB, «MongoDB CRUD Operations». <<https://docs.mongodb.org/v2.6/core/crud-introduction/>>



## Capítulo IV

# Queryng en MongoDB

### 1. Introducción

Este capítulo se centra en las operaciones de consulta y recuperación de datos. Esta es una de las principales diferencias con respecto a las bases de datos relacionales en las que se dispone de un lenguaje estándar de consultas denominado «SQL». MongoDB no dispone de un lenguaje estándar, y facilita un conjunto de métodos que ofrecen capacidades similares a las que ofrece SQL en el mundo relacional. En el capítulo se revisan los principales métodos y operadores para realizar consultas, haciendo una referencia especial a los arrays y a los documentos embebidos.

### 2. El método `find()`

El método **find** se utiliza para realizar consultas que retornan un subconjunto de documentos de una colección (desde ningún documento hasta todos los documentos de la colección):

- El primer argumento especifica las condiciones que deben cumplir los documentos que se quieren recuperar.
- Una condición de búsqueda vacía (`{}`) encaja con todos los documentos de la colección.

- En caso de no especificar ninguna colección, entonces se toma por defecto la colección vacía ({}). Por ejemplo, la consulta db.c.find () recupera todos los documentos de la colección c.
- Cuando se añaden pares clave-valor a las condiciones de búsqueda se restringe la búsqueda. Esto funciona directamente para la mayoría de los tipos: números coinciden con números, booleanos con booleanos, cadenas con cadenas, etc.
- Si se quiere consultar un tipo simple, basta especificar el valor que se está buscando. Por ejemplo, si se quiere encontrar todos los documentos donde el valor de la edad es 27, se añade como condición de búsqueda el par clave-valor {"edad":27} a la condición de búsqueda (figura 69):

**Figura 69.** Ejemplo de uso de find.

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

- Cuando se quieren usar múltiples condiciones juntas se añaden los pares clave-valor que sean necesarios, que serán interpretados como «Condición1 AND Condición2 AND...AND Condición N ». Por ejemplo, si se quiere recuperar todos los usuarios con 27 años y que se llamen «Isabel», se realizaría la siguiente consulta (figura 70):

**Figura 70.** Ejemplo de uso de find con varias condiciones.

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

- A veces, cuando se recupera un documento, no es necesario recuperar todos los campos del documento, para ello se puede pasar un segundo argumento al método find para especificar qué campos se quieren recuperar. Por ejemplo, si se tiene una colección de usuarios y solo se quiere recuperar el nombre

del usuario y el email, entonces se podría realizar la siguiente consulta (figura 71):

**Figura 71.** Ejemplo de recuperación de campos concretos.

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

- Siempre que se recupera un documento, por defecto se recupera el campo «`_id`».
- También es posible especificar explícitamente qué pares clave-valor no se quiere recuperar en la consulta. Por ejemplo, se puede tener una colección que tenga documentos con diferentes claves pero en todos ellos no se quiere recuperar la clave «teléfono», entonces se podría realizar la siguiente consulta (figura 72):

**Figura 72.** Ejemplo de selección negativa de un campo.

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

También podría usarse para evitar recuperar la clave «`_id`» (figura 73):

**Figura 73.** Ejemplo de selección negativa de varios campos.

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

- Existe una limitación en cuanto al valor de la consulta, que debe ser una constante. No se puede hacer referencia al valor de otra clave en el documento. Por ejemplo, si se mantiene un inventario en el que existen dos claves, «`en_stock`» y «`num_vendidos`», no se podría comparar ambos valores en una consulta como la siguiente:

```
db.stock.find ({“en_stock”: “this. num_vendidos “})
```

Para este ejemplo se podría haber usado las claves «stock\_inicial» y «en\_stock», de manera que cuando alguien compra algo entonces se decrementa el valor de la clave «en\_stock». Y después se puede hacer una consulta simple para comprobar cuántos elementos están fuera del stock: db.stock.find ({“en\_stock”: 0}).

- Existe un método similar a `find ()` que es `findOne ()`, que permite recuperar un único documento que cumpla las condiciones especificadas.
- En las consultas se puede utilizar el valor «null», que representa que algo no existe. Sin embargo, este valor tiene un comportamiento extraño, dado que encaja consigo mismo. Por ejemplo, si tenemos una colección de la forma:

```
{“_id”: ObjectId (“4ba0f0dfd22aa494fd523621”), “y”: null}  
{“_id”: ObjectId (“4ba0f0dfd22aa494fd523622”), “y”: 1}  
{“_id”: ObjectId (“4ba0f148d22aa494fd523623”), “y”: 2}
```

Y se realizará la consulta `db.c.find ({“y”: null})` para buscar documentos cuyo valor de «y» fuera null, esperaríamos que devolviera:

```
{“_id”: ObjectId (“4ba0f0dfd22aa494fd523621”), “y”: null}
```

Sin embargo, null no solo encaja consigo mismo, sino que también encaja con «no existe», de manera que la consulta sobre una clave con el valor null retornará todos los documentos que carecen de esa clave. Así, por ejemplo:

```
> db.c.find ({“z”: null})
```

```
{“_id”: ObjectId (“4ba0f0dfd22aa494fd523621”), “y”: null}  
{“_id”: ObjectId (“4ba0f0dfd22aa494fd523622”), “y”: 1}  
{“_id”: ObjectId (“4ba0f148d22aa494fd523623”), “y”: 2}
```

De manera que si se quiere recuperar los documentos cuyas claves tomen el valor null, entonces hay que comprobar que su valor es null y que la clave existe con el condicional “\$exist”: db.c.find ({“z”: {"\$in": [null], “\$exists”: true}}).

### 3. Operadores condicionales: \$lt, \$lte, \$gt, \$gte

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

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

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

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

Una coincidencia exacta sobre la fecha es menos útil, puesto que las fechas son solo almacenadas con precisión de milisegun-

dos, y con frecuencia lo que se busca es comparar un día, semana o mes entero, lo que hace necesario una consulta sobre rangos.

También puede ser útil consultar los documentos en los que el valor de una clave no es igual a cierto valor, para lo que se usa el operador «\$ne», que representa «no igual». Por ejemplo, si se quiere recuperar los usuarios que no tienen por nombre «Pablo», se podría consultar de la siguiente manera:

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

Hay que observar que:

**1)** Se pueden expresar múltiples condiciones sobre una clave dada. Por ejemplo, si se quiere encontrar todos los usuarios que tienen una edad entre 20 y 30, se podría usar los operadores «\$gt» y «\$lt» sobre la clave «edad» de la siguiente manera:

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

**2)** No se pueden usar múltiples operadores de modificación sobre una clave dada como, por ejemplo, un modificador de documento de la forma {“\$inc”: {“edad”: 1}, “\$set”: {edad: 40}}, puesto que modifica la clave «edad» dos veces.

**3)** El operador «\$ne» puede ser usado con cualquier tipo.

**4)** Las claves \$prefijadas ocupan diferentes posiciones, así los condicionales (como «\$lt») son siempre claves de documento internas, mientras que los modificadores (como «\$inc») son siempre claves de documento externas.

## 4. Los operadores \$not, \$ and y \$or

El operador «\$not» es un metacondicional, es decir, que puede ser aplicado sobre otros criterios. Por ejemplo, considérese el operador modular «\$mod», que permite realizar consultas sobre valores de claves acerca de si el módulo de la división del primer valor dado entre el valor de la clave da como módulo el segundo valor dado. Por ejemplo, si queremos recuperar los documentos que tienen un valor para el identificador que vale 1 módulo 5 (es decir, 1, 6,11, etc.), entonces se podría hacer de la siguiente manera:

```
db.usuarios.find ({"id_num": {"$mod": [5, 1]}})
```

Sin embargo, si se quiere recuperar los usuarios con identificador 2,3,4,5,7,8,9,10,12, se podría usar el operador «\$not» de la siguiente manera:

```
db.usuarios.find ({"id_num": {"$not": {"$mod": [5, 1]}}})
```

Hay que observar que «\$not» puede ser útil en conjunción con expresiones regulares para encontrar todos los documentos que no encajan con un determinado patrón.

Existen dos posibilidades para realizar una consulta de tipo «AND»:

- El operador «\$and», que puede ser usado para consultar sobre un conjunto de valores dados sobre múltiples claves dadas.
- No usar ningún operador. Por defecto se recuperan los documentos que cumplen todas las condiciones especificadas.

Por ejemplo, la siguiente consulta, busca los documentos en los x es menor que 1 e igual a 4 utilizando el operador \$and:  
db.users.find ({"\$and": [{"x": {"\$lt": 1}}, {"x": 4}]}). En este caso, aunque parezca contradictorio, encajaría con dicha condición un array de la forma {"x", [0,4]}. Sin embargo, la forma más eficiente de hacerlo es sin utilizar «\$and», que no optimiza:  
db.users.find ({"x": {"\$lt": 1, "\$in": [4]}}).

De manera similar existen dos posibilidades para realizar una consulta de tipo «OR»:

- El operador «\$in», que puede ser usado para consultar sobre una variedad de valores para una clave dada.
- El operador «\$or», que puede ser usado para consultar sobre un conjunto de valores dados sobre múltiples claves dadas.

Hay que observar que:

**1)** Si se tiene más de un posible valor que encajar sobre una clave dada, es mejor usar un «\$in» sobre un array con los valores. Por ejemplo, si se quiere recuperar los documentos de personas que tienen un DNI autorizado (sean 725, 542 y 390 los DNI autorizados), se podría hacer de la siguiente manera:

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

**2)** «\$in» es más flexible y permite especificar criterios sobre diferentes tipos y valores. Por ejemplo, supóngase una base de datos donde se pueden usar tanto nombres de usuario como identificadores numéricos de usuario, se podría realizar una consulta de la siguiente manera:

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

**3)** Si el operador «\$in» aparece con un array con un único valor, entonces se comporta intentando hacer coincidir el valor. Por ejemplo, {"dni": {"\$in: [725]}} es equivalente a {"dni": 725}.

**4)** El operador opuesto a «\$in» es «\$nin», que retorna documentos que no coinciden con ninguno de los criterios dados en el array de valores. Por ejemplo, si se quiere recuperar todos los documentos de personas que tienen un DNI que no está autorizado, se podría hacer de la siguiente manera:

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

**5)** «\$or» toma un array con posibles criterios de coincidencia. Por ejemplo, si se quiere recuperar los documentos de personas que tiene 30 años de edad o que no son fumadores, se podría hacer de la siguiente manera:

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

**6)** «\$or» también puede contener otros condicionales. Por ejemplo, si se quiere recuperar los documentos de personas que tienen 30, 34 y 35 años o que no son fumadores, se podría hacer de la siguiente manera

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

**7)** El operador «\$or» siempre funciona, pero siempre que se pueda es mejor usar «\$in», dado que es más eficiente.

8) Hay que observar que no existe el operador «\$eq», pero se puede simular con el operador «\$in» con un único valor.

## 5. Expresiones regulares

Es un mecanismo muy útil para recuperar cadenas. MongoDB usa la librería Perl Compatible Regular Expression (PCRE) para gestionar las expresiones regulares, de manera que cualquier expresión regular permitida por PCRE es permitida por MongoDB. Por ejemplo, si se quiere encontrar todos los usuarios con el nombre «Isa» o «isa», se puede usar una expresión regular para hacer una coincidencia que no sea sensible a las mayúsculas o minúsculas:

```
db.usuarios.find ({"nombre": /isa/i})1
```

En el ejemplo anterior, si se quisiera recuperar no solo las posibles versiones en mayúscula o minúscula, sino también otras palabras que pudieran formarse a partir de «isa» como «isabel», entonces se podría ampliar la expresión regular con:

```
db.usuarios.find ({"nombre": /isa?/i})
```

También se pueden realizar consultas sobre prefijos de expresiones regulares como `/^joey/`.

---

1 El indicador «i» indica que se trata de una expresión regular «i».

## 6. Consultas sobre arrays

Las consultas sobre elementos de un array están diseñadas para comportarse de la misma forma que sobre valores escalares. Por ejemplo, si se tiene un array que representa una lista de frutas como, por ejemplo, db.comida.insert ({“fruta”: [“manzana”, “plátano”, “melocotón”]}), entonces la consulta db.comida.find ({“fruta”: “plátano”}) tendrá éxito sobre el documento que se ha insertado (figura 74).

**Figura 74.** Ejemplo de recuperación desde un array

```
> db.comida.insert({“fruta” : [“manzana”, “platano”, “melocoton”]})  
WriteResult({“nInserted” : 1 })  
> db.comida.find({“fruta” : “platano”})  
{ “_id” : ObjectId(“54db9d8b85a35b866e990e40”), “fruta” : [ “manzana”, “platano”  
“melocoton” ] }
```

La consulta es equivalente a si se tuviera un documento de la forma:

```
{“fruta”: “manzana”, “fruta”: “plátano”, “fruta”: “melocotón”}.
```

Los valores escalares en los documentos deben coincidir con cada cláusula que aparece en la consulta. Así, por ejemplo, en la consulta {“x”: {"\$gt": 10, "\$lt": 20}}, debería cumplirse a la vez que x es más grande que 10 y más pequeña que 20. Sin embargo, si un documento x fuera un array, entonces el documento encajaría si existe un elemento de x que encaja con cada uno de los criterios que aparecen en la consulta, pero cada criterio puede encajar con un elemento diferente del array. Así, por ejemplo, si se tuvieran los siguientes documentos en una colección:

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

Si se quieren encontrar todos los documentos donde x está entre 10 y 20, se podría construir la siguiente consulta: db.test.find ({“x”: {"\$gt": 10, "\$lt": 20}}), esperando que recuperase como resultado el documento {“x”: 15}. Sin embargo, recupera dos documentos:

```
{“x”: 15}  
{“x”: [5, 25]}
```

Aunque ni 5 ni 25 están entre 10 y 20, sin embargo, 25 coincide con la condición de ser más grande que 10, y 5 coincide con la condición de ser más pequeño que 20. Es por ello que las consultas sobre rangos en arrays no son muy útiles, pues un rango coincidirá con cualquier array de varios elementos. Sin embargo, existen varias formas de conseguir el comportamiento esperado:

- Se puede usar el operador «\$elemMatch» para forzar que se compare cada condición con cada elemento del array. Sin embargo, este operador no encajará con elementos que no sean arrays. Así, por ejemplo, db.test.find ({“x”: {"\$elemMatch": {"\$gt": 10, "\$lt": 20}}}) no devuelve ningún resultado cuando el documento {“x”: 15} cumpliría el criterio. El problema es que en ese documento x no es un array.
- Si se tiene un índice sobre el campo que se está consultando, se puede usar los métodos min () y max () para limitar el rango de índices considerado en la consulta para los valores de los

operadores «\$gt» y «\$lt». Así, por ejemplo, la consulta db.test.find ({“x”: {\$gt: 10, \$lt: 20}}).min ({“x”: 10}).max ({“x”: 20}) recupera el documento {“x”: 15}. De esta forma, solo se considera el índice desde 10 hasta 20, evitando las entradas 5 y 25. Hay que observar que solo se puede usar min () y max () cuando se tiene un índice sobre los campos en que se está consultando, por lo que se deberán pasar todos los campos del índice a min () y max ().

- Usar min () y max () cuando se consulta sobre documento que puede que incluya arrays es la mejor solución, puesto que, si se busca en los límites del índice con «\$gt» o «\$lt», buscará en cada entrada del índice y no solo en el rango.

A continuación se van a considerar los principales operadores para realizar consultas sobre arrays:

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

```
> db.comida.insert ({“_id”: 1, “fruta”: [“manzana”, “plátano”, “melocotón”]})  
> db.comida.insert ({“_id”: 2, “fruta”: [“manzana”, “pera”, “naranja”]})  
> db.comida.insert ({“_id”: 3, “fruta”: [“cereza”, “plátano”, “manzana”]})
```

Entonces se podría buscar todos los documentos que tienen a la vez «manzana» y «plátano» mediante una consulta de la forma db.comida.find ({fruta: {\$all: [“manzana”, “plátano”]} }) (figura 75):

**Figura 75.** Ejemplo de uso de \$all.

```
> db.comida.insert({ "_id" : 1, "fruta" : [ "manzana", "platano", "melocoton" ] })
> db.comida.insert({ "_id" : 2, "fruta" : [ "manzana", "pera", "naranja" ] })
> db.comida.insert({ "_id" : 3, "fruta" : [ "cereza", "platano", "manzana" ] })
> db.comida.find({ "fruta" : { "$all" : [ "manzana", "platano" ] } })
{
  "_id" : ObjectId("54db9d8bb85a35b866e998e40"),
  "fruta" : [ "manzana", "platano", "melocoton" ]
}
{
  "_id" : 1,
  "fruta" : [ "manzana", "platano", "melocoton" ]
}
{
  "_id" : 3,
  "fruta" : [ "cereza", "platano", "manzana" ]
}
```

Hay que observar:

- El orden no importa en la consulta. Así, por ejemplo, en el segundo resultado aparece «plátano» antes de «manzana».
- Cuando se usa «\$all» con un array de un solo elemento entonces es equivalente a no usar «\$all». Por ejemplo, {fruta: {\$all: ['manzana']}} es equivalente a {fruta: 'manzana'}.
- Se puede realizar una consulta para buscar una coincidencia exacta usando el array entero. Sin embargo, la coincidencia exacta no encajará un documento si alguno de los elementos no se encuentra o sobra. Por ejemplo, la siguiente consulta recupera el primer documento db.comida.find ({“fruta”: [“manzana”, “plátano”, “melocotón”]}); sin embargo, las consultas db.comida.find ({“fruta”: [“manzana”, “plátano”]}) y db.comida.find ({“fruta”: [“plátano”, “manzana”, “melocotón”]}) no recuperan el primer documento.
- También es posible consultar un elemento específico de un array usando la notación indexada clave «. índice», como, por ejemplo, con db.comida.find ({“fruta.2”: “melocotón”}). Hay que observar que los índices empiezan a contar desde cero, por lo que la consulta anterior buscaría que el tercer elemento del array tome el valor de «melocotón».

- *El operador «\$size».* Este operador permite consultar arrays de un tamaño dado. Por ejemplo, la siguiente consulta:  
db.comida.find ({"fruta": {"\$size": 3}}).

Un uso normal consiste en recuperar un rango de tamaños. El operador «\$size» no puede ser combinado con otro operador condicional, pero se puede añadir una clave «size» al documento, de manera que, cada vez que se añade un elemento al array, entonces se incrementa el valor de la clave «size». Por ejemplo, si la consulta original fuera de la forma db.comida.update(criterio, {"\$push" : {"fruta" : "fresa"}}, entonces se podría convertir en la siguiente consulta: db.comida.update(criterio, {"\$push": {"fruta": "fresa"}, "\$inc": {"size": 1}}). De esta forma, se pueden hacer consultas de la forma db.comida.find({"size": {"\$gt": 3}}).

Hay que observar que esta técnica no puede ser utilizar con el operador «\$addToSet».

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

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

Y si se quieren recuperar los 10 últimos comentarios, se podría hacer de la siguiente manera: db.blog.posts.findOne (criterio, {"comentarios": {"\$slice": -10}})

El operador «\$slice» también puede retornar elementos concretos de los resultados, para lo que es necesario espe-

cificar un valor que indica desde qué elemento se empieza a recuperar y otro valor que indica cuántos se van a considerar a partir del indicado. Así, por ejemplo, la consulta db.blog.posts.findOne ({“comentarios”: {“\$slice”: [23, 10]}}) se salta los 23 primeros elementos y recupera del 24 al 33. Si hubiera menos de 33 elementos en el array, se recuperan tantos como sea posible.

Hay que observar que, a menos que se indique lo contrario, todas las claves de un documento son recuperadas cuando se usa el operador «\$slice». Este comportamiento es diferente con respecto a otros especificadores de clave que suprimen las claves que no se menciona que deban recuperarse. Por ejemplo, si se tuviera el siguiente documento (figura 76):

**Figura 76.** Documento de ejemplo.

```
> db.blog.posts.findOne()
{
  "_id" : ObjectId("54dbaad9cd91cec01668dd98"),
  "titulo" : "un buen post",
  "contenido" : "...",
  "comentarios" : [
    {
      "nombre" : "juan",
      "email" : "juan@ejemplo.com",
      "contenido" : "estupendo"
    },
    {
      "nombre" : "isabel",
      "email" : "isabel@yahoo.es",
      "contenido" : "Excelente"
    }
  ]
}
```

Si se usa el operador «\$slice» para conseguir el último comentario, se haría de la siguiente manera (figura 77):

**Figura 77.** Ejemplo de uso de \$slice

```
> db.blog.posts.findOne({}, {"comentarios": {"$slice": -1}})
{
  "_id": ObjectId("54dbaad9cd91cec01668dd98"),
  "titulo": "un buen post",
  "contenido": "...",
  "comentarios": [
    {
      "nombre": "isabel",
      "email": "isabel@yahoo.es",
      "contenido": "Excelente"
    }
  ]
}
```

Se puede observar que se recupera, además del comentario, también las claves «*título*» y «*contenido*» aunque no se hayan especificado.

A veces se desconoce el índice del elemento que se quiere recuperar. En estos casos se puede utilizar el operador «\$». En el ejemplo anterior, si se quiere recuperar el comentario que ha realizado «juan», se podría hacer de la siguiente manera (figura 78):

**Figura 78.** Ejemplo de uso del operador posicional.

```
> db.blog.posts.find({"comentarios.nombre": "juan"}, {"comentarios.$": 1})
{
  "_id": ObjectId("54dbaad9cd91cec01668dd98"),
  "comentarios": [ {"nombre": "juan",
    "email": "juan@ejemplo.com",
    "contenido": "estupendo"} ]
}
```

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

## 7. Consultas sobre documentos embebidos

Considérese un documento de la siguiente forma (figura 79):

**Figura 79.** Documento de ejemplo.

```
> db.prueba.findOne()
{
    "_id" : ObjectId("54dbce969e303b82900a56bc"),
    "nombre" : {
        "nombre" : "Javier",
        "apellido" : "Sanz"
    },
    "edad" : 45
}>
```

Existen dos caminos para consultar sobre documentos embebidos:

- Consulta sobre un subdocumento entero. Funciona de la misma forma que una consulta normal y debe encajar exactamente con el subdocumento. Así, si por ejemplo se añadiera un nuevo campo en el subdocumento o se cambiara el orden de los campos, entonces ya no coincidirían con la búsqueda.
- Consulta sobre pares individuales clave-valor. Si se quiere recuperar un documento en el que el nombre tome el valor «Javier Sanz», se haría de la siguiente manera (figura 80):

**Figura 80.** Consulta sobre claves de un subdocumento.

```
> db.prueba.find({"nombre": {"$and": [{"nombre": "Javier", "apellido": "Sanz"}]}})
{
    "_id" : ObjectId("54dbce969e303b82900a56bc"),
    "nombre" : {
        "nombre" : "Javier",
        "apellido" : "Sanz"
    },
    "edad" : 45
}>
```

En general, es mejor realizar consultas sobre claves específicas de un documento embebido de manera que, si se producen cambios en la estructura de los documentos, las consultas no se vean

afectadas por los cambios. Para consultar sobre claves específicas de documentos embebidos también se puede usar la notación «.», como, por ejemplo:

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

Hay que observar que en los documentos de consultas el uso de la notación «.» tiene como significado alcanzar el interior del documento embebido. Es por ello que no se permite usar el carácter «.» dentro de los documentos que se van a insertar (por ejemplo, existen problemas cuando se quieren almacenar URL). Una forma de resolverlo consiste en realizar un reemplazamiento global antes de insertar o después de recuperar, sustituyendo un carácter que no es legal en una URL por el carácter «.».

Las coincidencias con documentos embebidos pueden complicarse cuando la estructura del documento se hace compleja. Por ejemplo, supóngase que se están almacenando posts de un blog y se quieren recuperar comentarios de Javier que fueron puntuados con al menos un 5, entonces se podría modelar el post de la siguiente manera (figura 81):

**Figura 81.** Ejemplo de documento para modelar un post de un blog.

```
> db.blog.findOne()
<
  "_id" : ObjectId("54dbda019e303b82900a56bd"),
  "contenido" : "...",
  "comentarios" : [
    {
      "autor" : "javier",
      "puntuacion" : 3,
      "comentario" : "bonito post"
    },
    {
      "autor" : "maria",
      "puntuacion" : 6,
      "comentario" : "terrible post"
    }
  ]
>
```

Ahora bien, con esta estructura no se puede consultar usando la expresión db.blog.find ({{“comentarios”: {“autor”: “javier”, “puntuación”: {"\$gte": 5}}}}), dado que los documentos embebidos deben encajar el documento entero y este no encaja con la clave «comentario».

Tampoco funcionaría la expresión db.blog.find ({{“comentarios.autor”: “javier”, “comentarios.puntuacion”: {"\$gte": 5}}}}, dado que la condición del autor podría encajar con un comentario diferente al comentario que encajaría con la condición de la puntuación. Así, esta consulta devolvería el documento anterior, dado que encajaría autor «Javier» con el primer comentario y puntuación «6» con el segundo comentario.

Para agrupar correctamente los criterios de búsqueda sin necesidad de especificar cada clave se puede usar «\$elemMatch». Este operador permite especificar parcialmente criterios para encajar con un único documento embebido en un array. Así, la consulta correcta sería:

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

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

## 8. Consultas \$where

Los pares clave-valor son bastante expresivos para realizar consultas, sin embargo, existen consultas que no pueden realizarse. Para estos casos se usan las cláusulas «\$where», que permiten

ejecutar códigos JavaScript como parte de la consulta. El caso más común de uso es para comparar los valores de dos claves en un documento. Por ejemplo, supóngase que se tienen documentos de la siguiente forma:

```
db.comida.insert({“manzana”:1,“plátano”:6,“melocotón”:3})  
db.comida.insert ({“manzana”: 8, “espinacas”: 4, “melón”: 4})
```

Si se quiere retornar documentos donde dos de los campos son iguales, como, por ejemplo, en el segundo documento, donde «espinacas» y «melón» tienen el mismo valor, entonces esto se puede hacer usando el operador «\$where» con un código JavaScript (figura 82):

**Figura 82.** Ejemplo de uso de \$where.

```
> db.comida.findOne()  
< _id : ObjectId("54dbe49e9e303b82900a56be"),  
  "manzana" : 1,  
  "platano" : 6,  
  "melocoton" : 3  
> db.comida.find({$where: function ()<( for (var a in this) <( for (var b in thi  
s) <( if (a != b && this[a] == this[b]) <( return true;);)> return false;);>}  
< _id : ObjectId("54dbe4ac9e303b82900a56bf"), "manzana" : 8, "espinacas" : 4,  
  "melon" : 4 >
```

Si la función retorna cierto, el documento será parte del conjunto resultado, y si retorna falso, no formará parte.

Hay que observar que:

- Las consultas «\$where» son más lentas que las consultas regulares. Cada documento tiene que ser convertido desde el formato BSON a un objeto JavaScript y entonces ejecutar la expresión «\$where».
- Los índices no pueden ser usados para satisfacer un «\$where».
- El uso del «\$where» debe usarse solo cuando no haya otra forma de hacer las consultas.

- Para aminorar la penalización del uso del «\$where» se puede usar en combinación con determinados filtros. En este sentido siempre que sea posible se usará un índice para filtrar basado en condiciones que no sean «\$where», de manera que la expresión «\$where» será usada solo para afinar los resultados.

Hay que tener cuidado con la seguridad cuando se ejecuta JavaScript en el servidor, puesto que si se hace incorrectamente podrían darse ataques basados en inyección de código. Por ejemplo, supóngase que se quiere imprimir «Hola nombre» siendo nombre el del usuario. Se podría escribir una función JavaScript de la forma `func = "function () {print ('Hola, "+nombre+"')}";`. Ahora, si nombre es una variable definida por el usuario, podría ser la cadena «`"); db.dropDatabase(); print("")`», de forma que el código que ejecutar sería `func = "function() { print("Hola, "); db.dropDatabase(); print(""); }";`. Si se ejecuta este código, se borraría la base de datos completa. Así pues, hay que asegurarse de no aceptar entradas y pasárlas directamente a mongod, o bien bloquear el uso de JavaScript ejecutando mongod con la opción `--noscripting`.

## 9. Configuración de las consultas

La base de datos retorna los resultados que genera el método `find()` usando un cursor, de manera que se puede controlar con gran detalle la salida de una consulta: limitar el número de resultados, saltarse un número de resultados, ordenar los resultados por alguna combinación de claves en algún tipo de orden y otras

operaciones. Para ello, la consulta debe ser configurada antes de ser enviada a la base de datos con alguna de las siguientes opciones:

- *LIMITAR el número de resultados.* Para conseguir limitar los resultados se encadena la función `limit()` sobre la llamada a `find()`. Por ejemplo, para retornar solo tres resultados se haría de la siguiente manera: `db.c.find().limit(3)`. Si existen menos de tres documentos que encajan con la consulta, entonces solo se retornan los documentos que encajan. La función `limit` solo establece un límite superior pero no un límite inferior.
- *Saltarse un número de resultados.* La función `skip()` permite saltarse resultados encadenándose a la llamada a `find()`. Así, por ejemplo, `db.c.find().skip(3)` se saltará los tres primeros documentos que encajen y retornará el resto de los resultados. Si existen menos de tres documentos, no retornará ningún documento.
- *Ordenar los resultados.* La función `sort()` se encadena a la llamada `find()` tomando como entrada un objeto formado por pares clave-valor, donde las claves son nombres de claves y los valores indican un sentido de la ordenación: `1`(ascendente) o `-1`(descendente). Cuando existe más de una clave, los resultados son ordenados en ese orden. Por ejemplo, para ordenar los resultados en orden ascendente de «nombre» y en orden descendente de «edad» se haría de la siguiente manera: `db.c.find().sort({nombre: 1, edad: -1})`. A veces se puede dar el caso de tener una clave con múltiples tipos, de manera que si se aplica la función `sort()`, entonces se ordenarán de acuerdo con un orden predefinido. Los valores del más pequeño al más grande son:

- Minimum value
- null
- Numbers (integers, longs, doubles)
- Strings
- Object/document
- Array
- Binary data
- Object ID
- Boolean
- Date
- Timestamp
- Regular expression
- Maximum value

Estos métodos se pueden combinar, por ejemplo, para paginar los resultados. Supóngase que se está consultando una base de datos de libros y se van a mostrar los resultados por páginas con un máximo de cincuenta resultados ordenados por precio de mayor a menor, entonces se puede hacer lo siguiente:

```
db.stock.find ({“desc”: “libros”}).limit (50).sort ({“precio”: -1}).
```

Cuando la persona quiera ver la página siguiente se tendrá que añadir un skip () a la consulta de manera que se salten los primeros cincuenta resultados:

```
db.stock.find ({“desc”: “libros”}).limit (50).skip (50).sort ({“precio”: -1})
```

El resultado de una consulta se puede asignar a una variable local (aquellas que se definen con la palabra reservada var), como, por ejemplo:

```
> var cursor = db.prueba.find();
```

Cuando se realiza la asignación `find()` no ha ejecutado la consulta y espera hasta que se solicitan los resultados para enviar la consulta, lo que permite encadenar opciones adicionales en cualquier orden sobre la consulta antes de que sea llevada a cabo. Por ejemplo, las siguientes expresiones son equivalentes (pues la consulta no ha sido ejecutada y solo se construye la consulta):

```
> var cursor = db.prueba.find().sort({“x”: 1}).limit(1).skip(10);
> var cursor = db.prueba.find().limit(1).sort({“x”: 1}).skip(10);
> var cursor = db.prueba.find().skip(10).limit(1).sort({“x”: 1});
```

Una vez creada la variable, se puede iterar a través de los resultados de la consulta usando el método `hasNext()`:

```
> while (cursor.hasNext()) {
... obj = cursor.next();
...
... }
```

Cuando se hace la llamada al método `cursor.hasNext()` es cuando la consulta es enviada al servidor y se comprueba si existe otro resultado más y en tal caso lo recupera. Se recuperan los cien primeros resultados o los primeros 4 MB de resultados (el valor que sea más pequeño), de manera que las siguientes llamadas a `next()` o `hasNext()` no tendrán que consultar el servidor. Después de haber procesado el primer conjunto de resultados, se contacta nuevamente con la base de datos y se recuperan más resultados con una petición `getMore`. Las peticiones `getMore` básicamente contienen un identificador para la consulta y piden a la base de

datos que si existen más resultados que retorne el siguiente conjunto si existe. Este proceso continua hasta que el cursor finaliza al no haber más resultados que recuperar.

La clase cursor también implementa la interfaz iterador de JavaScript, que puede usarse en un bucle de tipo forEach:

```
> var cursor = db.personas.find ();
> cursor.forEach (function(x) {
... print (x.nombre);
...});
```

Por último, podría ser necesario conseguir documentos al azar de una colección. Una forma de conseguirlo consiste en añadir una clave aleatoria extra a cada documento cuando se inserta en la colección usando la función Math.random () (que crea un número aleatorio entre 0 y 1). Por ejemplo:

```
> db.personas.insert ({“nombre”: “maría”, “aleatorio”: Math.
random ()})
> db.personas.insert ({“nombre”: “isabel”, “aleatorio”: Math.
random ()})
> db.personas.insert ({“nombre”: “juan”, “aleatorio”: Math.
random ()})
```

Cuando se quiere encontrar un documento aleatorio de una colección, entonces se puede calcular un número aleatorio y usarlo como criterio de la consulta:

```
> var aleatorio = Math.random ()
> resultado = db.personas.findOne ({“aleatorio”: {“$gt”:
aleatorio}})
```

```
> if (resultado == null) {  
... resultado = db.personas.findOne ({“aleatorio”: {"$lt": alea-  
torio}})  
... }
```

Hay que observar que si no existen documentos en la colección no se retorna ningún documento.

## 10. Otras características

- *Consistencia de las consultas.* Todas las consultas que retornan un único conjunto de resultados son automáticamente congeladas y no tienen problemas de inconsistencia. Sin embargo, cuando una consulta devuelve varios conjuntos de resultados y se realiza sobre una colección que puede cambiar a lo largo del proceso de recuperación de los conjuntos de resultados, entonces podría retornar el mismo resultado múltiples veces. Cuando se ejecuta el método `find ()`, el cursor retorna los resultados del principio de la colección y se mueve hacia la derecha. Se recuperan los primeros cien documentos y se procesan. Si al mismo tiempo se modifican los documentos recuperados y se vuelven a guardar en la base de datos, si el documento no tiene hueco para crecer en tamaño, entonces son recolocados al final de la colección. De esta forma, cuando el programa sigue recuperando documentos llegará un momento que llegue al final y en ese momento volverá a recuperar los documentos recolocados. La solución a este problema consiste en usar la opción `snapshot` en la consulta. De esta manera, cuando la consulta se ejecute, se utiliza el índice sobre `«_id»` para recorrer los resultados de

forma que se garantiza que solo se recuperará cada documento una única vez. Así, en vez de ejecutar db.prueba.find (), se ejecutaría db.prueba.find ().snapshot () .

- *Comandos administrativos.* Un tipo especial de consulta son las denominadas «database command», que llevan a cabo tareas administrativas tales como apagar el servidor, clonar bases de datos, contar los documentos de una colección o realizar agregaciones. Por ejemplo, la eliminación de una colección es realizada vía el comando «drop»:

```
db.runCommand ({“drop”: “test”});
```

Este comando es equivalente a la siguiente expresión db.test.drop () .

En general, los comandos pueden ejecutarse en forma de expresiones o bien usando el comando runCommand () . Para ver todos los comandos que existen se puede usar el comando db.listCommands () .

Otro comando útil es «getLastError», que comprueba el número de documentos que se actualizan:

```
> db.count.update ({x: 1}, {$inc: {x: 1}}, false, true)  
> db.runCommand ({getLastError: 1})
```

Respecto al funcionamiento de los comandos:

- Un comando siempre retorna un documento que contiene la clave «ok». Si esta vale 1, entonces se ha ejecutado con éxito, y si vale 0, entonces se ha producido algún fallo. En este último caso se añade una clave «errmsg», que contiene una cadena que explica la razón del fallo.

- Los comandos se implementan como un tipo especial de consulta que se realiza sobre la colección «\$cmd», de manera que runCommand toma el documento del comando y realiza la consulta equivalente. De manera que en el ejemplo anterior se estaría ejecutando la expresión db.\$cmd.findOne({“drop” : “test”});.
- Cuando MongoDB recibe una consulta sobre la colección «\$cmd», utiliza una lógica especial en vez del código normal para llevar a cabo las consultas.
- Algunos comandos requieren acceder como administrador y ser ejecutados en la base de datos admin, de manera que si se ejecuta en otra base de datos, devuelve un error del tipo access denied. Cuando se está trabajando sobre una base de datos que no es admin, y se necesita ejecutar un comando admin, se puede usar la función adminCommand en vez de runCommand.
- Los comandos son sensativos al orden, de manera que el nombre del comando debe ser el primer campo en el comando. Así, por ejemplo, {“getLastError”: 1, “w”:2} funcionará pero no así {“w”: 2, “getLastError”: 1}.

## 11. Ejercicios propuestos

Considerar una colección con documentos de MongoDB que representan información multimedia de la forma:

```
{
  "tipo": "libro",
  "titulo": "Java para todos",
  "ISBN": "987-1-2344-5334-8",
  "editorial": "Anaya",
  "Autor": ["Pepe Caballero", "Isabel Sanz", "Timoteo Marino"],
  "capitulos": [
    {"capitulo": 1,
     "titulo": "Primeros pasos en Java",
     "longitud": 20
    },
    {"capitulo": 2,
     "titulo": "Primeros pasos en Java",
     "longitud": 25
    }
  ]
}

{
  "tipo": "CD",
  "Artista": "Los piratas",
  "Titulo": "Recuerdos",
  "canciones": [
    {"cancion": 1,
     "titulo": "Adiós mi barco",
     "longitud": "3:20"
    },
    {"cancion": 2,
     "titulo": "Pajaritos",
     "longitud": "4:15"
    }
  ]
}

{
  "tipo": "DVD",
  "Titulo": "Matrix",
  "estreno": 1999,
  "actores": [
    "Keanu Reeves",
    "Carry-Anne Moss",
    "Laurence Fishburne",
    "Hugo Weaving",
    "Gloria Foster",
    "Joe Pantoliano"
  ]
}
```

Realizar las siguientes operaciones:

- 1)** Insertar los documentos dados en una base de datos llamada «media» en una única operación.
- 2)** Del documento que hace referencia a la película «Matrix» recuperar el array de actores.
- 3)** Del documento que hace referencia a la película «Matrix» recuperar todos los campos de información excepto el array de actores.
- 4)** Del documento que hace referencia a la película «Matrix» recuperar un único documento en el que aparezcan solo los campos tipo y título.
- 5)** Recuperar todos los documentos que sean de tipo «libro» y editorial «Anaya» mostrando solo el array «capítulos».
- 6)** Recuperar todos los documentos referidos a canciones que tengan una canción que se denomine «Pajaritos».

7) Recuperar todos los documentos en los que «Timoteo Marino» es autor de un libro.

8) Recuperar todos los documentos de la colección media ordenados de manera decreciente por el campo «tipo».

9) Recuperar todos los documentos de la colección media ordenados de manera decreciente por el campo «tipo». Mostrar solo dos resultados.

10) Recuperar todos los documentos de la colección «media» ordenados de manera decreciente por el campo «tipo». Saltarse el primer resultado.

11) Recuperar todos los documentos de la colección «media» ordenados de manera decreciente por el campo «tipo». Recuperar solo dos resultados y saltarse los dos primeros resultados.

12) Añadir los siguientes documentos a la colección media:

13) {“tipo”: “DVD”,           {“tipo”: “DVD”,  
“Titulo”: “Blade Runner”, “Titulo”：“Toy Story 3”,  
“estreno”:1982           “estreno”: 2010  
}                          }

Realizar las siguientes consultas:

- Recuperar los documentos sobre películas cuya fecha de estreno sea mayor que 2000. En los resultados no mostrar el array de actores.
- Recuperar los documentos sobre películas cuya fecha de estreno sea mayor o igual que 1999. En los resultados no mostrar el array de actores.
- Recuperar los documentos sobre películas cuya fecha de estreno sea menor que 1999. En los resultados no mostrar el array de actores.
- Recuperar los documentos sobre películas cuya fecha de estreno sea menor o igual que 1999. En los resultados no mostrar el array de actores.

- Recuperar los documentos sobre películas cuya fecha de estreno sea mayor o igual que 1999 y menor que 2010. En los resultados no mostrar el array de actores

**14)** Recuperar todos los documentos sobre libros de manera que el autor no sea «Camilo José Cela».

**15)** Recuperar todos los documentos sobre películas que se hayan estrenado en alguno de los años 1999, 2005 y 2006. En los resultados no mostrar el array de actores.

**16)** Recuperar todos los documentos sobre películas que no se hayan estrenado en los años 1999, 2005 y 2006. En los resultados no mostrar el array de actores.

**17)** Recuperar todos los documentos sobre películas que se hayan estrenado en los años 1999 y 2005 exactamente. En los resultados no mostrar el array de actores.

**18)** Recuperar todos los documentos sobre libros que hayan sido escritos por Pepe Caballero e Isabel Sanz y que además entre los títulos de sus capítulos haya alguno que se denomine «Búcles».

**19)** Recuperar todos los documentos que tomen en la clave «Título» el valor «Recuerdos» o que tome en la clave «estreno» el valor «1999», y que tome en la clave tipo «DVD».

**20)** Considerar el documento acerca de la película «Matrix» y recuperar del array de actores:

- Los 3 primeros actores.
- Los últimos 3 actores.
- 3 actores saltándose los 2 dos primeros actores.
- 4 actores saltándose los 5 últimos actores.

**21)** Recuperar los documentos referidos a películas tales que en su campo «estreno» tenga un valor par. No mostrar el array actores.

**22)** Recuperar los documentos referidos a películas tales que en su campo «estreno» tenga un valor impar. No mostrar el array actores.

- 23)** Recuperar todos los documentos referidos a canciones tales que el número de canciones es exactamente 2.
- 24)** Recuperar todos los documentos tales que tengan un array de actores.
- 25)** Recuperar todos los documentos tales que no tengan un array de actores.
- 26)** Considerar la siguiente tabla que asigna a cada tipo de datos BSON con un valor numérico:

**Figura 83.** Tabla de tipos de datos BSON.

#	Data Type	#	Data Type
-1	MinKey	11	Regular Expression
1	Double	13	JavaScript Code
2	Character string (UTF8)	14	Symbol
3	Embedded object	15	JavaScript Code with scope
4	Embedded array	16	32-bit integer
5	Binary Data	17	Timestamp
7	Object ID	18	64-bit integer
8	Boolean type	127	MaxKey
9	Date type	255	Min Key
10	Null type		

Recuperar todos los documentos que tienen un campo denominado «canciones» cuyo valor sea del tipo un documento embebido.

- 27)** Insertar el siguiente documento:
- ```
{“tipo”: “CD”,  
“Artista”: “Los piratas”,  
“Titulo”: “Recuerdos”,  
“canciones”: [
```

```
{“cancion”:1,  
“titulo”: “Adiós mi barco”,  
“longitud”: “3:20”},  
{“cancion”:3,  
“titulo”: “Pajaritos”,  
“longitud”: “4:15”}}}
```

**28)** Recuperar todos los documentos sobre discos en los que se dan exactamente las siguientes condiciones: existe una canción denominada «Pajaritos» y el número de canción es el 2.

**29)** Recuperar todos los documentos sobre discos en los que no se dan exactamente las siguientes condiciones: existe una canción denominada «Pajaritos» y el número de canción es el 2.

**30)** Encontrar los DVD que sean más antiguos que 1995.

**31)** Encontrar todos los documentos en los que en el campo «Artista» aparece la palabra «piratas».

**32)** Encontrar todos los documentos en los que en el campo «Artista» aparece la palabra «piratas» y además tienen un campo «Titulo».

## Bibliografía

- Banker, K.** (2011). *MongoDB in action*. Manning Publications Co.
- Chodorow, K.** (2013). *MongoDB: the definitive guide*. O'Reilly Media, Inc.
- Hows, D.; Membrey, P.; Plugge, E.** (2014). *MongoDB Basics*. Apress.
- Membrey, P.; Plugge, E.; Hawkins, D.** (2010). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress.
- Dayley, B.** (2015). *Sams Teach Yourself NoSQL with MongoDB in 24 Hours*. Sams.
- Sitio oficial MongoDB, «Query Documents».  
<<https://docs.mongodb.org/v2.6/tutorial/query-documents/>>



## Capítulo V

# La framework de agregación

### 1. Introducción

En este capítulo se va a describir la denominada «framework de agregación». Se trata de un conjunto de operaciones que permiten procesar los datos de forma que los resultados se obtienen mediante la combinación de sus valores.

Se caracteriza por que las operaciones agregadas se aplican sobre colecciones de documentos y devuelve los valores procesados en forma de uno o más documentos.

La framework funciona en tres modalidades diferentes, con sintaxis también distintas:

- Agregación mediante tuberías.
- Map Reduce.
- Operaciones de propósito único.

### 2. Agregación mediante tuberías

En esta modalidad de funcionamiento, los datos son procesados como si atravesaran tuberías, de forma que en cada tubería se aplica una determinada función y la salida de la misma es la entrada de la siguiente, en la que se realiza otra operación.

Estas tuberías pueden ser de varios tipos:

- Filtros que operan como consultas y transformaciones sobre documentos, modificando el documento de entrada.
- Otras tuberías sirven para agrupar y ordenar documentos basándose en campos específicos del mismo.
- Operaciones sobre arrays como cálculos de marcadores estadísticos, concatenadores de strings, etc.

La llamada a la framework se realiza mediante la función aggregate(), que recibe como argumento un array de operaciones separadas por comas.

La sintaxis general es la siguiente:

```
db.colección.aggregate( [ { <stage_1> }, ..., { <stage_n> } ] )
```

Donde <stage\_1>...<stage\_n> son cada una de las tuberías u operaciones aplicadas ordenadamente a los datos.

Algunas de las operaciones disponibles son:

**1) \$project:** se utiliza para modificar el conjunto de datos de entrada, añadiendo, eliminando o recalculando campos para que la salida sea diferente.

Excluir o incluir campos:

```
{“$project”: {<campo>: 1, <campo>: 1}}
```

Crear nuevos campos usando los valores de otros campos mediante la notación \$ y el nombre:

```
{“$project”: {<campo_nuevo>: “$otro_campo”, <campo_nuevo>: <expresion>} }
```

**2) \$match:** filtra la entrada para reducir el número de documentos, dejando solo los que cumplan las condiciones establecidas.

{“\$match”: <expresion>}

**3) \$limit:** restringe el número de resultados al número indicado.

{“\$limit”: <numero>}

**4) \$skip:** ignora un número determinado de registros, y devuelve los siguientes:

{“\$skip”: <positivo\_entero>}

**5) \$unwind:** desagrupa. Convierte un array para devolverlo separado en documentos. Convierte documentos con una lista de valores en un campo en un conjunto de documentos con un único valor en dicho campo.

{“\$unwind”: <expresion>}

**6) \$group:** agrupa documentos según una determinada condición. Se pueden elegir varios campos como identificador.

{ “\$group”: { “\_id”: { <campo1>: <expresion>, ... },  
<campo>: { <operador>: <expresion> }, ... } }

**7) \$sort:** ordena un conjunto de documentos según el campo especificado.

{“\$sort”: <expresion>}

**8) \$geoNear:** utilizado con datos geoespaciales, devuelve los documentos ordenados por proximidad según un punto geoespacial.

```
{“$geoNear”: <opciones>}
```

**9) \$redact:** trunca subdocumentos. Permite recorrer en profundidad cada campo del documento y decidir en cada nivel la acción que realizar: \$\$KEEP (devolver todos los campos y subdocumentos), \$\$DESCEND (devolver todos los campos y evaluar los niveles inferiores), \$\$PRUNE (no devolver ningún campo ni subdocumento).

```
{“$redact”: {“$cond”: {“if”: <condicion>, “then”: <accion>, “else”: <accion>}}}
```

**10) \$out:** Almacena los resultados en la colección indicada. Crea la colección si no existe, la reemplaza atómicamente si existe y si hay errores no hace nada.

```
{“$out”: <colección>}
```

Para realizar cálculos sobre los datos producidos se utilizan expresiones. Las expresiones son funciones que realizan una determinada operación sobre un grupo de documentos, un array o un campo en concreto.

### ***Operadores de agrupación***

Aquellos que se utilizan cuando se agrupa información con \$group.

- `$first`: devuelve el primer valor de un campo en un grupo. Si el grupo no está ordenado, el valor mostrado será impredecible.
- `$last`: devuelve el último valor de un campo en un grupo. Si el grupo no está ordenado, el valor mostrado será impredecible.
- `$max`: devuelve el valor más alto de un determinado campo dentro de un grupo.
- `$min`: devuelve el valor más pequeño de un determinado campo dentro de un grupo.
- `$avg`: calcula la media aritmética de los valores dentro del campo especificado y los devuelve.
- `$sum`: suma todos los valores de un campo y los devuelve.
- `$addToSet` y `$push`: se crea un campo array que incluirá todos los valores del campo que especifiquemos. La diferencia es que `$push` puede mostrar varias veces un elemento si este se repite en el grupo y `$addToSet` solo mostrará uno de ellos, independientemente de las veces que aparezca en el grupo.

## ***Operadores aritméticos***

Aquellos que pueden usarse con `$group` y otros como `$project`.

- `$add`: realiza la suma de un array de números.
- `$divide`: divide dos números.
- `$mod`: a partir de dos números calcula el resto producido al dividir el primero entre el segundo.
- `$multiply`: multiplica dos números.
- `$subtract`: a partir de dos números realiza la resta.
- Operadores de comparación
- Aquellos que se utilizan para comparar valores y devolver un resultado.
- `$cmp`: compara dos valores y devuelve un número entero como resultado. Devuelve -1 si el primer valor es menor que

el segundo, 0 si son iguales y 1 si el primer valor es mayor que el segundo.

- `$eq`: compara dos valores y devuelve true si son equivalentes.
- `$gt`: compara dos valores y devuelve true si el primero es más grande que el segundo.
- `$gte`: compara dos valores y devuelve true si el primero es igual o más grande que el segundo.
- `$lt`: compara dos valores y devuelve true si el primero es menor que el segundo.
- `$lte`: compara dos valores y devuelve true si el primero es igual o menor que el segundo.
- `$ne`: compara dos valores y devuelve true si los valores no son equivalentes.

### ***Operadores booleanos***

Aquellos que reciben parámetros booleanos y devuelven otro booleano como resultado.

- `$and`: devuelve true si todos los parámetros de entrada son true.
- `$or`: devuelve true si alguno de los parámetros de entrada es true.
- `$not`: devuelve el valor contrario al parámetro de entrada. Si el parámetro es true, devuelve false. Si el parámetro es false, devuelve true.

### ***Operadores condicionales***

Aquellos que se utilizan para verificar una expresión y, según el valor de esta expresión, devolver un resultado u otro.

- `$cond`: operador ternario que recibe tres expresiones. Si la primera es verdadera, devuelve la segunda. Si es falsa, devuelve la tercera.

- *\$ifNull*: operador que recibe dos parámetros y, en el caso de que el primero sea null, devuelve el segundo.

## ***Operadores de cadena***

Aquellos que se utilizan para realizar operaciones con strings.

- *\$concat*: concatena dos o más strings.
- *\$strcasecmp*: compara dos strings y devuelve un entero con el resultado. Devolverá un número positivo si el primer string es mayor, un número negativo si el segundo es mayor o un 0 en el caso de que sean iguales.
- *\$substr*: crea un substring con una cantidad determinada de caracteres.
- *\$toLowerCase*: convierte el string a minúsculas.
- *\$toUpperCase*: convierte el string a mayúsculas.

## ***Operadores de fecha***

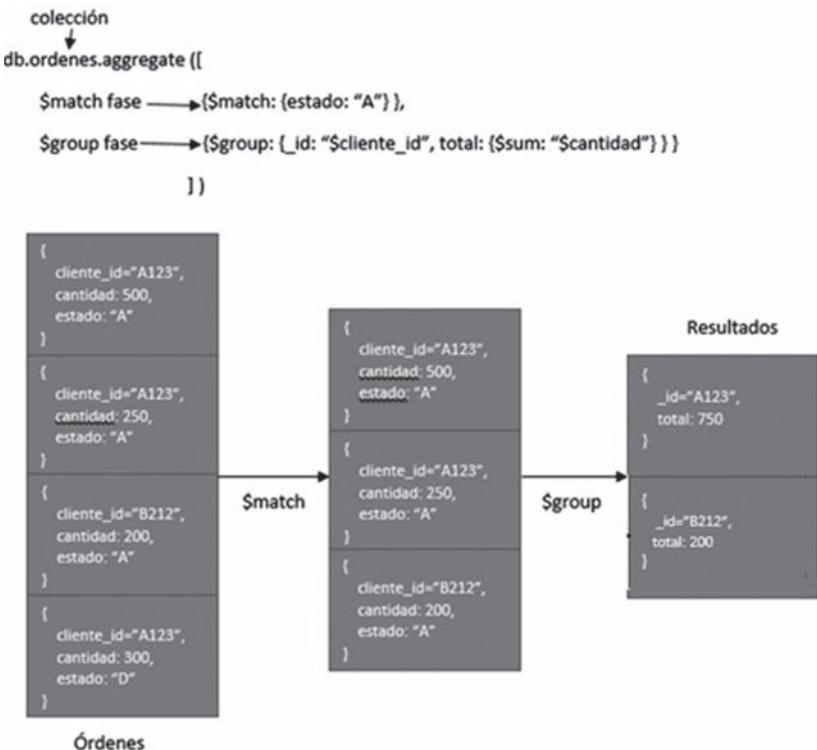
Aquellos que se utilizan para realizar operaciones con campos de tipo fecha.

- *\$dayOfYear*: convierte una fecha a un número entre 1 y 366.
- *\$dayOfMonth*: convierte una fecha a un número entre 1 y 31.
- *\$dayOfWeek*: convierte una fecha a un número entre 1 y 7.
- *\$year*: convierte una fecha a un número que representa el año (2000, 1998, etc.).
- *\$month*: convierte una fecha a un número entre el 1 y el 12.
- *\$week*: convierte una fecha a un número entre 0 y 53.
- *\$hour*: convierte una fecha a un número entre 0 y 23.
- *\$minute*: convierte una fecha a un número entre 0 y 59
- *\$second*: convierte una fecha a un número entre 0 y 59. Aunque puede ser 60 para contar intervalos.

- `$millisecond`: devuelve los milisegundos de la fecha, con un número entre 0 y 999.

Se va a considerar el siguiente ejemplo que aparece en la figura 84. El proceso que se realiza es el siguiente:

- En primer lugar se obtienen aquellos documentos de la colección «órdenes» en los que el campo «estado» es A mediante el operador \$match.
- A continuación, la salida de la operación anterior (documentos que cumplen la condición indicada) es ordenada mediante el operador \$group utilizando el campo «cliente\_id» de la salida anterior.
- Para cada tipo de documento se obtiene el total de cada tipo mediante el operador \$sum, que suma los valores del campo «cantidad» de los documentos agrupados, y los guarda en un nuevo campo denominado «total».

**Figura 84.** Ejemplo de procesamiento con la framework.

### 3. Map Reduce

En este modo de funcionamiento, la framework proporciona los métodos necesarios para realizar operaciones map-reduce. Map-reduce es una framework creado por Google, y pensada para realizar operaciones de forma paralela sobre grandes colecciones de datos.

Las operaciones map-reduce constan de tres partes:

- Una etapa de *map*, en la que se procesa cada documento y se emite uno o varios objetos por cada documento procesado. Se implementa mediante una función que mapea los datos de origen, de manera que, para cada dato de origen, se genera una tupla clave-valor, que son unidas en una lista que se pasa a la etapa *reduce*.
- Una etapa *reduce* en la que se combinan las salidas de la etapa anterior. Se implementa mediante una función *reduce*, que trata cada elemento de la lista de pares y realiza operaciones sobre ella para devolver un dato concreto.
- Una tercera etapa opcional, *finalize*, en la que se permite realizar algunas modificaciones adicionales a las salidas de la etapa *reduce*.

Hay que observar:

- Las funciones involucradas en map-reduce se implementan en JavaScript en la consola de mongo.
- Aunque la flexibilidad es total al poderse implementar cualquier tipo de función que se desee, en general este modo de funcionamiento es menos eficiente y más complejo que la agregación mediante tuberías. Además, en general se pueden conseguir los mismos resultados con ambos modos de funcionamiento.

La llamada a map-reduce se realiza mediante el método `mapReduce()`, que puede recibir un número variable de entradas. Existen dos sintaxis diferentes para invocarse:

- Primera forma de sintaxis:  
`db.runCommand (`  
    `{`

```
mapReduce: <colección>,
map: <función>,
reduce: <función>,
out: <salida>,
query: <documento>,
sort: <documento>,
limit: <número>,
finalize: <función>,
scope: <documento>,
jsMode: <booleano>,
verbose: <booleano>
}
)
```

- Segunda forma de sintaxis:  
db.coleccion.mapReduce (  
 mapFunction,  
 reduceFunction,  
 {  
 out: <salida>,  
 query: <documento>,  
 sort: <documento>,  
 limit: <número>,  
 finalize: <función>,  
 scope: <documento>,  
 jsMode: <booleano>,  
 verbose: <booleano>  
 }  
)

Los parámetros tienen el siguiente significado:

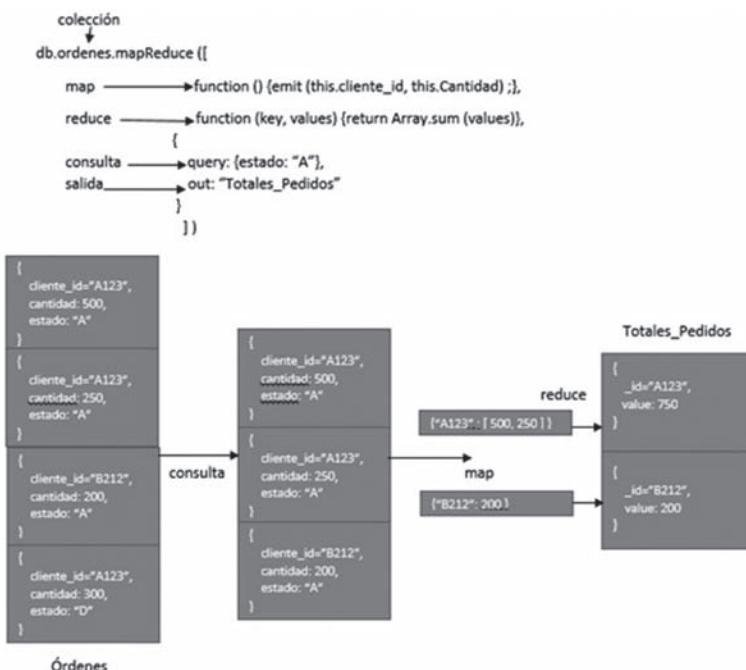
| Campo     | Tipo               | Descripción                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| mapReduce | Colección          | Nombre de la colección sobre la que se desea aplicar la operación map-reduce. Esta colección será filtrada utilizando el contenido de query antes de aplicarle la operación map.                                                                                                                                                                                                                   |
| map       | Función JavaScript | Una función JavaScript que asocia o mapea los valores con claves y emite pares clave valor.                                                                                                                                                                                                                                                                                                        |
| reduce    | Función JavaScript | Una función JavaScript que reduce los resultados del mapeo a un único objeto que contiene todos los valores asociados a una clave concreta.                                                                                                                                                                                                                                                        |
| out       | String o documento | Especifica dónde agrupar la salida de la operación map-reduce. Es posible especificar una colección o únicamente devolver los resultados en la consola.                                                                                                                                                                                                                                            |
| query     | Documento          | Opcional. Especifica el contenido de selección de los documentos de la colección a los que se les quiere aplicar la operación map-reduce.                                                                                                                                                                                                                                                          |
| sort      | Documento          | Opcional. Especifica el orden de los documentos de entrada sobre los que se desea aplicar la operación map-reduce. Se suele emplear para optimizar el proceso. Por ejemplo, si la clave de ordenación es la misma que la clave emitida por la etapa map, entonces la etapa reduce podrá realizarse en pocas operaciones. La clave de ordenación debe corresponderse con un índice de la colección. |
| limit     | Número             | Opcional. Especifica el número máximo de documentos de entrada a la función map.                                                                                                                                                                                                                                                                                                                   |
| finalize  | Función JavaScript | Opcional. Función JavaScript que determina qué operaciones aplicar a la salida de la función reduce.                                                                                                                                                                                                                                                                                               |
| scope     | Documento          | Opcional. Especifica variables globales que serán accesibles desde las funciones map y finalize.                                                                                                                                                                                                                                                                                                   |
| jsMode    | Booleano           | Opcional. Determina si transformar los datos resultantes de la función map a formato BSON. Por defecto es false.                                                                                                                                                                                                                                                                                   |
| verbose   | Booleano           | Opcional. Determina cuándo incluir información temporal sobre el proceso del tipo log. Por defecto es true.                                                                                                                                                                                                                                                                                        |

Así pues, el esquema general de map-reduce es el siguiente:

```
var mapFunction = function () {...};
var reduceFunction = function (key, values) {...};
db.runCommand (
{
  mapReduce: <colección-entrada>,
  map: mapFunction,
  reduce: reduceFunction,
  out: <colección-salida>,
  query: <consulta>
})
```

Vamos a considerar el siguiente ejemplo (figura 85):

**Figura 85.** Ejemplo de procesamiento con la framework.



El flujo que se sigue en la operación es el siguiente:

- Aplicación de la consulta de filtrado:{estado: “A”}, que genera documentos intermedios que contienen el campo «cliente\_id», el campo «cantidad» y el campo «estado» de todos aquellos documentos que cumplen la consulta de búsqueda.
- Ejecución de la función map function () {emit (this.cliente\_id, this.cantidad)}, que no modifica los documentos de partida y únicamente realiza un filtrado, que genera documentos que contienen una clave que coincide con los cliente\_id de entrada y cuyo valor es un array con las cantidades asociadas a esa clave.
- Ejecución de la función reduce function (key, values) {return Array.sum (values)}, que suma los valores de los arrays generados en la etapa anterior y genera documentos que están formados por el campo «\_id» con la clave del proceso y el campo «value» que contiene las sumas obtenidas.
- El operador out indica que los documentos de salida se almacenen en la colección Totales\_Pedidos

## 4. Operaciones de propósito único

Se trata de un conjunto de comandos de la framework de agregación para realizar operaciones usuales que llevan a cabo una única acción.

Está formado por las operaciones count (), distinct () y group () cuyo funcionamiento es el siguiente:

- db.collection.count (*<query>*): devuelve el número de documentos que cumplen la condición de búsqueda especificada en la query.
- db.collection.distinct (field, query): devuelve el número de documentos diferentes según el campo «*field*» de los documentos que cumplen la condición de búsqueda especificada en query.
- db.collection.group ({key, reduce, initial [, keyf] [, cond] [, finalize]}): esta operación toma el conjunto de documentos que cumplen la query y los agrupa en función de un determinado campo o campos. Devuelve un array de documentos con los resultados procesados para cada grupo.

Donde los argumentos de entrada son:

- key: indica el campo o campos que agrupar y devuelve un objeto clave que se usa como clave de agrupamiento.
- reduce: función que opera sobre los documentos durante la etapa de agrupamiento que puede devolver una suma o un conteo. Toma como argumentos el documento sobre el que aplicar la operación y el documento resultante para el grupo.
- initial: inicializa el documento resultante.
- keyf (Opcional): es una clave alternativa que especifica una función que crea un objeto clave para ser utilizado como clave de agrupamiento. Se usa en lugar del campo «*key*», para agrupar por campos procesados en lugar de por campos existentes en los documentos iniciales.
- cond (Opcional): es un criterio de selección de documentos que indica qué documentos de la colección deben ser procesados. Si se omite, entonces se procesarán todos los documentos de la colección en la operación de agrupamiento.

- **finalize (Opcional):** es una función que se ejecuta sobre cada elemento del conjunto en la etapa final.

Considerar la siguiente colección de documentos que se quieren agrupar por año, mes y día y calcular el precio total, la cantidad media y el número de documentos en cada grupo:

- `{ “_id” : 1, “item” : “abc”, “precio” : 10, “cantidad” : 2, “fecha” : ISODate(“2014-03-01T08:00:00Z”) }`
- `{ “_id” : 2, “item” : “jkl”, “precio” : 20, “cantidad” : 1, “fecha” : ISODate(“2014-03-01T09:00:00Z”) }`
- `{ “_id” : 3, “item” : “xyz”, “precio” : 5, “cantidad” : 10, “fecha” : ISODate(“2014-03-15T09:00:00Z”) }`
- `{ “_id” : 4, “item” : “xyz”, “precio” : 5, “cantidad” : 20, “fecha” : ISODate(“2014-04-04T11:21:39.736Z”) }`
- `{ “_id” : 5, “item” : “abc”, “precio” : 10, “cantidad” : 10, “fecha” : ISODate(“2014-04-04T21:23:13.331Z”) }`

Se puede calcular de la siguiente manera:

```
db.ventas.aggregate (
[ {
  $group: {
    _id: {mes: {$month: “$fecha”}, dia: {$dayOfMonth: “$fecha”}, año: {$year: “$fecha”}},
    precioTotal: {$sum: {$multiply: [“$precio”, “$cantidad”]}},
    cantidadMedia: {$avg: “$cantidad”},
    cantidad: {$sum: 1}
  }
}]
```

Se devuelven los documentos siguientes:

- { “\_id” : { “mes” : 3, “día” : 15, “año” : 2014 }, “precioTotal” : 50, “cantidadMedia” : 10, “cantidad” : 1 }
- { “\_id” : { “mes” : 4, “día” : 4, “año” : 2014 }, “precioTotal” : 200, “cantidadMedia” : 15, “cantidad” : 2 }
- { “\_id” : { “mes” : 3, “día” : 1, “año” : 2014 }, “precioTotal” : 40, “cantidadMedia” : 1.5, “cantidad” : 2 }

## 5. Ejercicios propuestos

Considerar una colección con documentos de MongoDB que representan información multimedia de la forma:

```
{ “tipo”: “CD”, { “tipo”: “DVD”,  
“Artista”: “Los piratas”, “Titulo”: “Matrix”,  
“TituloCanción”: “estreno”: 1999,  
“Recuerdos”, “actores”: [  
“canciones”: [ “Keanu Reeves”,  
“cancion”:1, “Carry-Anne Moss”,  
“titulo”: “Adiós mi barco”, “Laurence Fishburne”,  
“longitud”: “3:20” “Hugo Weaving”,  
}, “Gloria Foster”,  
“cancion”:2, “Joe Pantoliano”  
“titulo”: “Pajaritos”, ] }  
“longitud”: “4:15” }  
}  
]  
}
```

Realizar las siguientes operaciones:

**1)** Seleccionar los documentos de tipo «CD», de manera que solo se muestre en dichos documentos los campos «Artista», «TituloCanción», y un nuevo campo «TitulosCanciones», que contenga un array con las canciones del disco.

**2)** Añadir las siguientes películas:

```
{“tipo”: “DVD”,           {“tipo”: “DVD”,           {“tipo”: “DVD”,  
  “Titulo”: “Blade Runner”, “Titulo”: “Batman”, “Titulo”:  
  “Superman”,  
  “estreno”:1982   “estreno”: 1999   “estreno”: 1999  
 }           }           }
```

Seleccionar todos los documentos de tipo «DVD» y calcular cuántas películas hay de cada año de estreno, mostrando el año de estreno y el número de películas de cada año.

**3)** Seleccionar el documento sobre la película «Matrix» y crear un documento por cada uno de los actores que intervienen. En los documentos resultantes solo se mostrará el título y el actor.

**4)** Igual que la consulta anterior, pero se mostrará solo los tres últimos resultados ordenados por el nombre del actor.

**5)** Obtener las películas distintas que hay con respecto al título, los diferentes años de estrenos, y los diferentes tipos de documentos. Se realizará en tres consultas diferentes.

**6)** Agrupar los documentos por «Título», mostrando el título y el total que hay de cada grupo.

**7)** Añadir las siguientes películas:

```
{“tipo”: “DVD”,           {“tipo”: “DVD”,           {“tipo”: “DVD”,
```

“Título”: “Blade Runner”, “Título”: “Batman”, “Título”:  
“Superman”,  
“estreno”:1967                “estreno”: 1989                “estreno”: 1996  
}                              }                                      }

Repetir el caso anterior pero solo con los documentos que pertenecen a películas.

8) Obtener usando MapReduce la suma de los años de los estrenos de cada película. Es decir, debe obtenerse documentos de la forma: {“\_id”: “Batman”, “value”: {“TotalPelículas”:3988 }}.

## Bibliografía

- Banker, K.** (2011). *MongoDB in action*. Manning Publications Co.
- Chodorow, K.** (2013). *MongoDB: the definitive guide*. O'Reilly Media, Inc.
- Hows, D.; Membrey, P.; Plugge, E.** (2014). *MongoDB Basics*. Apress.
- Membrey, P.; Plugge, E.; Hawkins, D.** (2010). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress.
- Sitio oficial MongoDB, «Aggregation Introduction».

## Capítulo VI **Indexación**

### 1. Introducción

En este capítulo se tratan los índices. Un índice es una estructura de datos que mantiene información acerca de los valores de campos específicos en los documentos de una colección, y se utiliza para ordenar y clasificar rápidamente los documentos de una colección. De esta forma, asegura una búsqueda y recuperación rápida de datos de los documentos.

Esencialmente se puede pensar que un índice es una consulta predefinida que fue ejecutada y los resultados de la misma se almacenan; de esta forma, la consulta de información se hace rápida al no tener que recorrer la base de datos para recopilar esta información.

Cuando se crea un índice, aumenta la velocidad de las consultas, pero se reduce la velocidad de las inserciones y las eliminaciones debido a que el sistema debe mantener y actualizar el índice cada vez que se realiza una operación de escritura (inserción, actualización o borrado). Es por ello que generalmente es mejor añadir índices en las colecciones cuando el número de lecturas es mayor que el número de escrituras; de hecho, si hay más escrituras que lecturas, los índices pueden ser contraproducentes.

Por otra parte, cuando se tienen índices, de vez en cuando hay que borrar algunos o reconstruirlos debido a varias razones:

- Limpiar algunas irregularidades que aparecen en los índices.
- Aumento del tamaño de la base de datos.
- Espacio excesivo ocupado por los índices. Solo se pueden definir como máximo cuarenta índices por colección.

En general, los índices se usan con las consultas (`find`, `findOne`) y en las ordenaciones. Si se intentan realizar muchas ordenaciones sobre la información de una colección, se deberían añadir índices que correspondan con la especificación de la ordenación. Así, si se usa el comando `sort()` sobre una colección donde no existen índices sobre los campos que aparecen especificados en la ordenación, puede dar lugar a un error si se excede el tamaño máximo del buffer interno de ordenación.

El capítulo revisa la creación de índices simples y compuestos, los principales parámetros de configuración, la eliminación de índices, así como otros conceptos relacionados. En la última sección se tratan los índices geoespaciales que proporciona MongoDB.

## 2. Los índices en MongoDB

Toda la información sobre los índices se encuentra almacenada en la colección `system.indexes`. Esta colección gestiona todos los índices que han sido creados en todas las colecciones, así como los campos o elementos a los que hacen referencia. Se trata de una colección normal, por lo que puede operarse con ella con los comandos habituales. Por ejemplo, si se quieren listar los índices definidos hasta el momento sobre una base de datos determinada, se ejecuta el comando `indexes.find()` (figura 86):

**Figura 86.** Listado de los índices definidos en el sistema.

```
> db.system.indexes.find()
< "v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "blog.posts" >
< "v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "blog.autores" >
```

En el ejemplo se tiene una base de datos denominada «blog» que tiene dos colecciones: posts y autores. Aunque sobre las colecciones no se han definido índices por el usuario, sin embargo, sí existen dos índices que se han creado de forma automática sobre los campos «\_id» de cada colección. Los índices sobre el campo «\_id» son creados y borrados automáticamente por el sistema cada vez que se crea o se borra una colección.

Cuando se crea un índice sobre un elemento, entonces el sistema construye un índice en forma de árbol b, que es usado para localizar eficientemente los documentos. Si no existe ningún índice adecuado, entonces se recorren todos los documentos de la colección para encontrar los registros que satisfacen la consulta.

### 3. Creación de un índice simple

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

Por ejemplo, si se quiere crear un índice ascendente sobre el campo «Etiquetas», se haría de la siguiente manera (figura 87):

**Figura 87.** Creación de un índice ascendente.

```
> db.posts.createIndex({ "Etiquetas": 1 })
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
> db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "name" : "_id", "ns" : "blog.posts" }
{ "v" : 1, "key" : { "_id" : 1 }, "name" : "_id", "ns" : "blog.autores" }
{ "v" : 1, "key" : { "Etiquetas" : 1 }, "name" : "Etiquetas_1", "ns" : "blog.posts" }
```

Para indexar un campo de un documento embebido se usa la notación dot. Así, por ejemplo, si se tiene un campo contador dentro de un subdocumento «comentarios» sobre el que se quiere definir un índice, se haría de la siguiente manera (figura 88):

**Figura 88.** Definición de un índice a partir de un campo.

```
> db.posts.createIndex({ "comentarios.contador": 1 })
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 2,
  "numIndexesAfter" : 3,
  "ok" : 1
}
> db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "name" : "_id", "ns" : "blog.posts" }
{ "v" : 1, "key" : { "_id" : 1 }, "name" : "_id", "ns" : "blog.autores" }
{ "v" : 1, "key" : { "Etiquetas" : 1 }, "name" : "Etiquetas_1", "ns" : "blog.posts" }
{ "v" : 1, "key" : { "comentarios.contador" : 1 }, "name" : "comentarios.contador_1", "ns" : "blog.posts" }
```

Si se especifica un campo de un documento que es de tipo array, el índice incluirá todos los elementos del array como términos separados del índice, es decir, un índice multiclave. De esta forma, cada documento es enlazado a múltiples valores del índice.

Hay que observar que existe un operador especial para realizar consultas donde se selecciona solo aquellos documentos que tienen todos los términos que se especifican. Así, por ejem-

plo, en la base de datos blog, se tiene la colección posts con un campo denominado «Etiquetas», que representa las etiquetas que se le asocian a un post determinado. En este sentido, se podría definir una consulta que recuperase todos los artículos que tienen unas determinadas etiquetas como «teléfono», «app» (figura 89):

**Figura 89.** Consulta sobre documentos indexados.

```
> db.posts.find({ "Etiquetas": { "$all": [ "telefono", "app" ] } })
```

Sin un índice multiclave sobre el campo «Etiquetas», se habría tenido que consultar cada documento de la colección para ver si existe un término, y en tal caso a continuación chequear si ambos términos están presentes.

## 4. Creación de un índice compuesto

A primera vista crear un índice separado para cada campo que aparece en las consultas es una buena idea para hacerlas más eficientes, sin embargo, los índices tienen un impacto significativo sobre la adición y eliminación de datos de la base de datos, puesto que es necesario actualizarlos cada vez que se realiza una de estas operaciones.

Los índices compuestos son una buena forma de mantener bajo el número de índices que se tienen sobre una colección, lo que permite combinar múltiples campos en un único índice. Es por ello que se debe usar este tipo de índices siempre que sea posible. Para crear un índice compuesto se especifican varias claves en vez de una.

Existen dos tipos de índices compuestos: índices de subdocumentos e índices compuestos definidos por el usuario.

Se han definido algunas reglas para usar los índices compuestos en consultas que no usan todas las claves que componen el índice. La comprensión de estas reglas permite construir un conjunto de índices compuesto que cubren todas las consultas que se desean realizar sobre una colección sin tener un índice individual sobre cada elemento (por lo que se vira así el impacto sobre el rendimiento en las actualizaciones e inserciones).

Un contexto donde puede que los índices compuestos no sean una buena elección es cuando se usa el índice en una ordenación. En la ordenación no es bueno usar índices compuestos, salvo que la lista de términos y las direcciones de ordenación encajen exactamente con la estructura del índice. En estos casos, una elección mejor es usar índices simples individuales sobre cada campo.

## 5. Índices de subdocumentos

Se puede crear un índice compuesto usando un subdocumento entero, de manera que cada elemento del documento embedido se convierte en parte del índice. Por ejemplo, supóngase que se tiene un subdocumento autor con el nombre y el email en su interior, entonces se puede crear un índice compuesto con los términos autor.nombre y autor.email de la siguiente manera (figura 90):

**Figura 90.** Creación de un índice compuesto.

```
> db.posts.createIndex({ "Autor": 1 })
  "createdCollectionAutomatically" : false,
  "nunIndexesBefore" : 3,
  "nunIndexesAfter" : 4,
  "ok" : 1

> db.system.indexes.find()
  "_id" : 1, "key" : { "_id" : 1 }, "name" : "_id", "ns" : "blog.posts"
  "_id" : 1, "key" : { "_id" : 1 }, "name" : "_id", "ns" : "blog.autores"
  "_id" : 1, "key" : { "Etiquetas" : 1 }, "name" : "Etiquetas_1", "ns" : "blog.posts"
  "_id" : 1, "key" : { "comentarios.contador" : 1 }, "name" : "comentarios.contador_1", "ns" : "blog.posts"
  "_id" : 1, "key" : { "Autor" : 1 }, "name" : "Autor_1", "ns" : "blog.posts"
```

El único problema que existe con esta forma de crear índices compuestos es que se pierde la posibilidad de configurar el orden de las claves en el índice, puesto que no se puede configurar la dirección de cada uno de ellos.

Además, hay que observar que, cuando se realizan consultas exactas sobre documentos embebidos, el orden en que ocurren los campos debe encajar exactamente. En este ejemplo no se recupera lo mismo al cambiar el orden de los campos, pues se realiza un emparejamiento exacto en valores y orden (figura 91):

**Figura 91.** Recuperación sobre documentos indexados.

```
> db.posts.find({ "Autor": { "Nombre": "Pepito", "Email": "pepito@gmail.com" } })
  "_id" : ObjectId('55056bf10f5a8a1dfce89f22'), "Autor" : { "Nombre" : "Pepito", "Email" : "pepito@gmail.com" }
> db.posts.find({ "Autor": { "Email": "pepito@gmail.com", "Nombre": "Pepito" } })
=
```

## 6. Índices creados manualmente

Tal como se ha visto antes, cuando se usa un subdocumento como clave del índice, el orden de los elementos usados para construir el índice multiclave encaja con el orden en que apare-

cen en la representación interna del subdocumento. En muchos casos, esto no da el suficiente control sobre el proceso de creación de un índice. Para evitar esto y garantizar que la consulta usa un índice construido de la forma deseada, se necesita asegurar que se usa la misma estructura del subdocumento para crear el índice que la usada para realizar la consulta.

Para ello se crea un índice compuesto nombrando explícitamente todos los campos por los que se desea combinar el índice, y el orden de combinación. Así, por ejemplo, si se quiere crear un índice compuesto en el que los documentos primero se ordenan con respecto al campo «Email» y a continuación para cada valor del campo «Email» se ordena con respecto al campo «Nombre» (figura 92):

**Figura 92.** Documento con campos «Email» y «Nombre».

```
> db.posts.find({ "Autor": { "Email": "pepito@gmail.com", "Nombre": "Pepito" } })
```

Entonces, el índice se define de la siguiente manera (figura 93):

**Figura 93.** Creación de un índice compuesto.

```
> db.posts.createIndex({ "Autor.Email": 1, "Autor.Nombre": 1 })
{
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
}
```

El beneficio de esta aproximación es que se puede crear un índice sobre múltiples claves, pero con la ventaja de poder especificar cómo se quiere que sean indexados cada uno de los campos de forma descendente o ascendente. Sin embargo, en el caso de los subdocumentos se está limitando a que se ordenen solo de manera ascendente o descendente.

## 7. Especificar opciones sobre los índices

Se pueden especificar diversas opciones cuando se crea un índice, tales como la creación de índices únicos o permitir la indexación en segundo plano. Para ello, se especifican las opciones como parámetros de la función `createIndex()` de la siguiente manera:

```
db.Coleccion ({Campo: 1}, {opcion1: true, opcion2: true,...})
```

a) *Indexación en segundo plano.* Cuando se usa la función `createIndex()` para crear un índice la primera vez, entonces el servidor debe leer todos los datos de la colección y crear el índice especificado. Por defecto, esto se hace en primer plano y se llevan todas las operaciones necesarias sobre la colección especificada hasta crear el índice. Sin embargo, es posible realizar una indexación en segundo plano, de forma que las operaciones que realicen otras conexiones sobre la colección no son bloqueadas, las consultas no usarán el índice hasta que no se haya terminado de crear, y el servidor permitirá que continúen las operaciones de lectura y escritura. Una vez completada la construcción del índice, todas las consultas que requieran el uso del índice empezarán a usarlo.

Por ejemplo, si se quiere construir un índice en segundo plano sobre el campo «Nombre» de «Autor», se haría de la siguiente manera (figura 94):

**Figura 94.** Creación de un índice en segundo plano.

```
> db.posts.createIndex({ "Autor.Nombre":1 }, { "background":true })
{
    "createdCollectionAutomatically" : false,
    "nunIndexesBefore" : 2,
    "nunIndexesAfter" : 3,
    "ok" : 1
}
```

Hay que observar que la conexión que inicia el proceso de indexación se bloquea si la petición se realiza desde la Shell, de manera que no retornará nada hasta que la operación de indexación se haya completado. Sin embargo, si se abriera otra Shell al mismo tiempo, se puede comprobar que las consultas y actualizaciones sobre esa colección se realizan sin problemas mientras se está creando el índice. Este comportamiento difiere de cuando se crea un índice en el acto, donde las operaciones que se realizarán sobre una segunda Shell abierta, al mismo tiempo estarían bloqueadas.

Es posible finalizar el proceso de indexación cuando se ha colgado el sistema o bien está tomando demasiado tiempo. Para ello se invoca la función killOp (): db.killOp (id-operación). Cuando se invoca, la indexación parcial que se haya realizado se elimina automáticamente para evitar inconsistencias.

**b) Índices únicos.** Cuando se especifica la opción «unique», entonces se crea un índice donde todas las claves deben ser diferentes. De manera que el sistema retornará un error si se intenta insertar un documento donde la clave del índice coincide con la clave de un documento existente. Es útil para campos donde se quiere asegurar que no se repiten valores.

Sin embargo, si se quiere añadir un índice único a una colección ya existente con datos, hay que asegurarse de que no existen duplicaciones en las claves, pues de lo contrario fallará si cualquiera de las claves no es única.

Funciona con índices simples y compuestos, pero no así con índices para valores multiclave. En el caso de los índices compuestos, el sistema fuerza a la unicidad sobre la combinación de los valores en vez del valor individual para alguno o todos los valores de la clave.

Si un documento es insertado con un campo que falta y especificado como una clave única, entonces automáticamente se inserta el campo con el valor a null. Esto significa que solo se puede insertar un documento en el que falte un campo, pues nuevos valores nulos harán que se considere que la clave no es única.

En el siguiente ejemplo se crea un índice único sobre el campo «título» de la colección «posts» (figura 95):

**Figura 95.** Creación de un índice único sobre «título».

```
> db.posts.insert({ "titulo": "Prueba" })
> writeResult({ "inserted" : 1 })
> db.posts.createIndex({ "titulo": 1 }, { "unique": true })
{
    "createdCollectionAutomatically" : false,
    "nunIndexesBefore" : 1,
    "nunIndexesAfter" : 2,
    "ok" : 1
}
```

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

En el siguiente ejemplo se crea un índice único sobre el campo «título» de la colección «posts» y además se indica que se borren todos los documentos que pudieran hacer fallar la creación del índice debido a la duplicidad de valores del campo «título» (figura 96):

**Figura 96.** Creación de un índice único sobre «título» y borrado de todo.

```
> db.posts.createIndex({ "titulo": 1 }, { "unique": true, "dropdups": true })
{
    "createdCollectionAutomatically" : false,
    "nunIndexesBefore" : 1,
    "nunIndexesAfter" : 2,
    "ok" : 1
}
```

## 8. Eliminación de índices

Se puede elegir entre eliminar un índice concreto de una colección usando la función dropIndex (Especificación del índice) (figura 97):

**Figura 97.** Eliminación de un índice.

```
> db.posts.dropIndex({ "titulo": 1 })
{
    "nIndexesWas" : 2,
    "ok" : 1
}
```

También es posible eliminar todos los índices de una colección usando la función dropIndexes () (figura 98):

**Figura 98.** Eliminación de todos los índices.

```
> db.posts.dropIndexes()
{
    "nIndexesWas" : 2,
    "msg" : "non-_id indexes dropped for collection",
    "ok" : 1
}
```

## 9. Reindexación de una colección

Cuando se sospecha que un índice está dañado, se puede forzar la reindexación de la colección. Esto hace que se eliminen y se vuelvan a crear todos los índices sobre la colección especificada (figura 99):

**Figura 99.** Reindexación.

```
> db.posts.reIndex()
{
  "nIndexesWas" : 1,
  "nIndexes" : 1,
  "indexes" : [
    {
      "key" : {
        "_id" : 1
      },
      "name" : "_id_",
      "ns" : "blog.posts"
    }
  ],
  "ok" : 1
}
```

La salida muestra todos los índices que el comando ha reconstruido incluyendo las claves. Así, el nIndexWas indica cuántos índices existían antes de ejecutarse el comando y el nIndex indica el número de índices después de que se complete el comando. Cuando los valores de estos parámetros son diferentes, indica que hay algún problema en la recreación de algunos de los índices de la colección.

## 10. Selección de índices

Cuando se quiere ejecutar una consulta, el sistema crea un plan de ejecución que es una lista de los pasos que debe ejecutar para llevar a cabo la consulta. Cada consulta tiene múltiples

planes que producirán el mismo resultado. Sin embargo, cada plan puede tener elementos que sean más costosos de ejecutar que otros. Por ejemplo, un recorrido de todos los registros de una colección es una operación costosa y cualquier plan que lo incorpore será lento. Estos planes pueden incluir alternativamente listas de índices que usar para las operaciones de consulta y ordenación.

MongoDB usa un componente denominado «analizador de consultas», que toma una consulta y los objetivos de esta, y produce un conjunto de plantes de ejecución. La función `explain()` lista tanto el plan que se usará para la consulta así como los planes alternativos.

Asimismo existe otro componente denominado «optimizador de consultas», que tiene como función seleccionar qué plan de ejecución es el más adecuado para una consulta particular. Este componente no usa un método basado en costes para seleccionar el plan de ejecución, sino que ejecuta todos en paralelo, usa el que retorna los resultados más rápidamente, y finaliza el resto una vez el plan ganador ejecuta la última línea.

## 11. El comando `hint()`

El optimizador de consultas selecciona el mejor índice del conjunto de índices existentes para una colección usando los métodos antes comentados para encontrar el mejor o mejores índices para una consulta dada. Sin embargo, puede haber casos en los que el optimizador no haga la mejor elección. En estos casos es posibles forzar al optimizador de consultas a que use un índice dado usando el operador `hint()`.

Por ejemplo, supóngase que se tiene un índice sobre un sub-dокументo llamado «Autor» que tiene como campos «nombre» e «email» (figura 100):

**Figura 100.** Índice sobre un subdocumento.

```
> db.posts.createIndex({"Autor":1})
{
  "createdCollectionAutomatically" : true,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

Se podría usar `hint()` para forzar que el optimizador de consultas use el anterior índice (figura 101):

**Figura 101.** Utilización de `hint()` para forzar el optimizador.

```
> db.posts.find({"Autor":{"Nombre":"Juan","Email":"juan@gmail.com"}}).hint({"Autor":1})
```

También es posible usar `hint()` para forzar que una consulta no use índices, es decir, que se use el escaneo de la colección de documentos como medio para seleccionar los registros. Se haría de la siguiente manera (figura 102):

**Figura 102.** Utilización de `hint()` para forzar el no uso de índices.

```
> db.posts.find({"Autor":{"Nombre":"Juan","Email":"juan@gmail.com"}}).hint({$natural:1})
```

Existe un conjunto de funciones, `min()` y `max()`, que permiten establecer restricciones en las consultas, de manera que solo seleccione resultados que coincidan con claves de índices que se encuentren entre las claves mínima y máxima especificadas. Por tanto, se necesitará tener un índice para las claves especificadas. Se pueden combinar las dos funciones o bien usarlas de forma separada.

En el siguiente ejemplo se inserta un conjunto de documentos que tienen un campo «Año», se crea un índice ascendente sobre

dicho campo, y se recupera un conjunto de documentos usando las funciones max y min (figura 103):

**Figura 103.** Creación de un índice ascendente.

```
> db.posts.insert({Año:1995})
WriteResult({nInserted: 1})
> db.posts.insert({Año:1990})
WriteResult({nInserted: 1})
> db.posts.insert({Año:2000})
WriteResult({nInserted: 1})
> db.posts.insert({Año:2005})
WriteResult({nInserted: 1})
> db.posts.createIndex({Año:1})
{
    "createdCollectionAutomatically": false,
    "numIndexesBefore": 1,
    "numIndexesAfter": 2,
    "ok": 1
}
> db.posts.find().min({Año:1990}).max({Año:2001})
{
    "_id": ObjectId("5505f39fb83862235781d8d6"), "Año": 1990
}
{
    "_id": ObjectId("5505f398b83862235781d8d5"), "Año": 1995
}
{
    "_id": ObjectId("5505f3a6b83862235781d8d7"), "Año": 2000
}
```

Si no existe un índice, se producirá un error. También es posible especificar el índice que se desea usar mediante hint () (figura 104):

**Figura 104.** Especificación de un índice mediante hint () .

```
> db.posts.find().min({Año:1990}).max({Año:2001}).hint({Año:1})
{
    "_id": ObjectId("5505f39fb83862235781d8d6"), "Año": 1990
}
{
    "_id": ObjectId("5505f398b83862235781d8d5"), "Año": 1995
}
{
    "_id": ObjectId("5505f3a6b83862235781d8d7"), "Año": 2000
}
```

Hay que observar que el valor indicado por min () será incluido en los resultados, pero no así el valor max (), que será excluido. Por otra parte, se recomienda usar \$gt y \$lt en vez de min () y max (), dado que los primeros no requieren un índice. En este sentido, las funciones max () y min () son usadas principalmente con índices compuestos.

## 12. Optimización del almacenamiento de pequeños objetos

Los índices son la clave básica de la velocidad de las consultas de datos. Sin embargo, otro factor que afecta al rendimiento de la aplicación es el tamaño de los datos accedidos. A diferencia de los sistemas de bases de datos con esquema fijo, en MongoDB se almacenan todos los datos del esquema para cada registro en el propio registro. Así, para registros grandes con contenidos de datos grandes por campo, la relación de los datos del esquema para registrar los datos es baja; sin embargo, para registros pequeños con valores de datos pequeños, la relación puede crecer ampliamente.

Por ejemplo, considérese una aplicación de logging en la que se usa MongoDB. La tasa de escritura crea eventos de streaming como pequeños documentos de una colección de forma muy eficiente. Sin embargo, se podría optimizar la funcionalidad:

- En primer lugar se podría realizar como un trabajo por lotes las inserciones, realizando múltiples llamadas para insertar los documentos. Se puede usar esta llamada para colocar varios eventos en una colección al mismo tiempo. Esto resulta en un menor número de llamadas a la API de la base de datos.
- Otra opción consiste en reducir el tamaño de los nombres de los campos. Si se tienen nombres de campos más pequeños, se pueden mantener más registros de eventos en memoria antes de tener que eliminarlos del disco. Esto hace el sistema más eficiente.

Por ejemplo, asumiendo que se tiene una colección que es usada para iniciar la sesión de tres campos: un campo temporal,

un contador y una cadena de cuatro caracteres usada para indicar la fuente de los datos. El tamaño total de almacenamiento de los datos será el siguiente (figura 105):

**Figura 105.** Tamaño total de almacenamiento.

| Field     | Size     |
|-----------|----------|
| Timestamp | 8 bytes  |
| Integer   | 4 bytes  |
| string    | 4 bytes  |
| Total     | 16 bytes |

Si se usa ts, n y src para los nombres de los campos, el tamaño total de los nombres de los campos es 6 bytes. Esto es un valor relativamente pequeño comparado con el tamaño de los datos. Sin embargo, si se usaran como nombres «CuandoOcurreEvento», «NúmeroEventos» y «FuenteEventos», el tamaño total de los nombres de los campos es de 48 bytes, o tres veces el tamaño de los datos. Si se escribe 1Tr de datos en una colección, se necesitaría almacenar 750 GB para los nombres de los campos frente a los 250 GB de los datos.

Esto hace gastar espacio de disco y afecta al rendimiento del sistema, incluyendo el tamaño del índice, el tiempo de transferencia de los datos, y probablemente el uso de la RAM para cachear los archivos de datos.

Otra recomendación para las aplicaciones de logging es que se debe evitar añadir índices sobre las colecciones cuando se escriben registros, puesto que los índices toman tiempo y recursos para mantenerlos. En vez de ello, se deberían añadir los índices inmediatamente antes de empezar a analizar los datos. Asimismo se debería considerar usar un esquema que divida los eventos de stream en múltiples colecciones. Por ejemplo, se podría escribir

cada evento de un día en una colección separada. Colecciones más pequeñas requieren menos tiempo para indexar y analizar.

## 13. Índices geoespaciales

Además de los índices normales, existen índices geoespaciales de dos dimensiones que están diseñados para funcionar como un camino adicional con consultas basadas en la localización. Por ejemplo, se puede usar esta característica para encontrar un número de ítems cercanos a la actual localización o refinar una búsqueda que quiere encontrar un número específico de restaurantes cercanos a la localización actual.

Un documento al que se le quiere añadir información geoespacial debe contener un subobjeto o array donde los dos primeros elementos contengan las coordenadas x e y, como, por ejemplo: {loc: {lat: 52.033475, long: 5.099222}}.

Una vez que la información anterior es añadida al documento, entonces se puede crear el índice usando la función `createIndex()` con el parámetro «`2d`» (figura 106):

**Figura 106.** Creación de un índice geoespacial.

```
> db.lugares.insert({ loc : { lat : 52.033475, long: 5.099222 } })
> writeResult({ "nInserted" : 1 })
> db.lugares.createIndex({loc:"2d"})
<
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
```

El parámetro `2d` indica que se está indexando una coordenada o alguna información bidimensional. Por defecto, se asume que se indica una clave «`latitud/longitud`» y se usa un rango desde

-180 a 180. Sin embargo, se puede sobrescribir estos valores usando los parámetros min/max (figura 107):

**Figura 107.** Sobrescritura de los rangos de valores del índice.

```
> db.lugares.createIndex({loc:"2d"},{min:-500,max:500})
{
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
}
```

Hay que observar que, una vez definidos los límites, no se pueden insertar valores que sean exactamente los valores de los límites.

También se pueden expandir los índices geoespaciales usando valores de claves secundarias (o claves compuestas), lo que puede ser útil si se consulta sobre múltiples valores, como, por ejemplo, por una localización (información geoespacial) y una categoría (ordenación ascendente) (figura 108):

**Figura 108.** Creación de un índice geoespacial.

```
> db.lugares.createIndex({loc:"2d", "category":1})
{
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
}
```

Una vez que se han añadido datos a la colección y se ha creado un índice, se pueden crear consultas que aprovechan la información geoespacial. Por ejemplo, se podría buscar un valor exacto (figura 109):

**Figura 109.** Búsqueda del valor exacto.

```
> db.lugares.find({loc:[52,6]})
```

La consulta no devuelve nada, puesto que es demasiado específica. Sin embargo, se podría hacer una consulta que

buscara documentos que contuvieran información cercana a un valor dado, para lo que se puede usar el modificador \$near (figura 110):

**Figura 110.** Utilización de \$near.

```
> db.lugares.find({loc:{$near:[52,6]}})
{ "_id" : ObjectId("5506dc612846a088c272dd5f"), "loc" : { "lat" : 52.033475, "lo
ng" : 6.099222 } }
{ "_id" : ObjectId("5506d89c2846a088c272dd5e"), "loc" : { "lat" : 52.033475, "lo
ng" : 5.099222 } }
```

Este operador hace que la función find () busque cualquier documento cercano a las coordenadas 52 y 6, y los resultados son ordenados por su distancia desde el punto especificado al operador \$near. La salida por defecto estará limitada a cien resultados, aunque este valor es configurable (figura 111):

**Figura 111.** Limitación de la salida de una búsqueda con \$near.

```
> db.lugares.find({loc:{$near:[52,6]}}, limit(200))
{ "_id" : ObjectId("5506dc612846a088c272dd5f"), "loc" : { "lat" : 52.033475, "lo
ng" : 6.099222 } }
{ "_id" : ObjectId("5506d89c2846a088c272dd5e"), "loc" : { "lat" : 52.033475, "lo
ng" : 5.099222 } }
```

Hay que observar que no existe correlación entre el número de resultados retornados y el tiempo que la consulta tarda en ejecutarse.

Además del operador \$near, existe el operador \$within, que sirve para encontrar ítems dentro de una forma particular, como, por ejemplo, \$box, que representa un rectángulo, o bien \$center, que representa un círculo.

Para usar \$box es necesario especificar las esquinas inferior izquierda y superior derecha del rectángulo y almacenar estos valores en una variable (figura 112):

**Figura 112.** Utilización de \$box.

```
> box=[[40,60],[4,8]]
[ 40, 60 ], [ 4, 8 ]
> db.lugares.find({loc:{$within:{$box:box}}})
```

Para usar \$center es necesario especificar el centro y el radio del círculo y almacenar estos valores en una variable (figura 113):

**Figura 113.** Utilización de \$center.

```
> db.lugares.find({loc:{$within:{$center:[center,radius]}}})
{ "_id" : ObjectId("5506dc612846a088c272dd5f"), "loc" : { "lat" : 52.033475,
"long" : 6.099222 } }
```

Por defecto, la función find () es ideal para ejecutar consultas. Sin embargo, también existe la función geoNear (), que funciona como find (), pero también muestra la distancia desde un punto especificado por cada uno de los ítems del resultado. Además, la función incluye algún diagnóstico adicional. En el siguiente ejemplo se van a buscar los dos resultados más cercanos a una posición especificada (figura 114):

**Figura 114.** Búsqueda de los dos resultados más cercanos.

```
> db.runCommand({geoNear:"lugares", near:{52,6},num:2})
{
  "results" : [
    {
      "dis" : 0.10471666968062066,
      "obj" : {
        "_id" : ObjectId("5506dc612846a088c272dd5f"),
        "loc" : {
          "lat" : 52.033475,
          "long" : 6.099222
        }
      }
    },
    {
      "dis" : 0.9813997897220965,
      "obj" : {
        "_id" : ObjectId("5506d89c2846a088c272dd5e"),
        "loc" : {
          "lat" : 52.033475,
          "long" : 6.099222
        }
      }
    }
  ],
  "stats" : {
    "nscanned" : NumberLong(2),
    "nobjectsLoaded" : NumberLong(2),
    "avgDistance" : 0.5020592297907928,
    "maxDistance" : 0.9813997897220965,
    "time" : 1
  },
  "ok" : 1
}
```

## Bibliografía

- Banker, K.** (2011). *MongoDB in action*. Manning Publications Co.
- Chodorow, K.** (2013). *MongoDB: the definitive guide*. O'Reilly Media, Inc.
- Hows, D.; Membrey, P.; Plugge, E.** (2014). *MongoDB Basics*. Apress.
- Membrey, P.; Plugge, E.; y Hawkins, D.** (2010). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress.
- Sitio oficial MongoDB, «Aggregation Introduction». <<https://docs.mongodb.org/v2.6/core/aggregation-introduction/>>



## Capítulo VII **Replicación**

### 1. Introducción

En este capítulo se va a describir el proceso de replicación de información en MongoDB. La replicación es el proceso por el que se sincronizan los datos almacenados en la base de datos a través de múltiples servidores. Las ventajas que ofrece la replicación son:

- Proporciona redundancia e incrementa la disponibilidad de los datos.
- Si los datos están distribuidos en diferentes nodos de la red y en múltiples copias, la base de datos queda protegida frente a pérdidas o fallos en un único servidor.
- Incrementa el rendimiento en los procesos de lectura ya que los clientes pueden realizar dichas peticiones a diferentes servidores en paralelo y seleccionar en cada caso el más cercano con el objetivo de reducir tiempos de latencia.

A lo largo del capítulo se introducirá el concepto de conjunto de replicación, se describirá el proceso de replicación y la configuración de los nodos secundarios. Y a continuación se explicará con detalle cómo crear un conjunto de replicación y las labores de mantenimiento de este.

## 2. Replica sets

Un conjunto de replicación (replica set) es un conjunto de instancias de mongod que almacenan el mismo conjunto de datos. Una de las instancias, denominada «primaria», recibe y gestiona todas las operaciones de escritura. El resto de las instancias, denominadas «secundarias», realizan las operaciones indicadas por la primaria de forma que todas tengan el mismo conjunto de datos almacenado.

Cada replica set solo puede contener un nodo primario, y dado que es el único que puede recibir peticiones de escritura, se asegura la consistencia estricta de las operaciones de lectura realizadas a dicho nodo. Además, cada replica set puede tener uno o más secundarios. En general, conviene tener un número impar de nodos en total con un mínimo de tres.

En general, las lecturas se realizan sobre el nodo primario, pero es posible especificar un nodo secundario para ello. En ese caso, es posible que los datos obtenidos no sean los mismos que los almacenados en el nodo primario. En caso de que el nodo primario no esté disponible se reelige un nuevo primario de entre los secundarios disponibles.

## 3. El proceso de replicación

El nodo primario mantiene una serie de logs, denominados «oplogs», con todos los cambios que se han realizado sobre los datos almacenados en su conjunto de replicación. Cada nodo secundario replica el oplog del nodo primario y aplica las operaciones pendientes contenidas en el mismo a sus propios datos, de forma que replican el nodo primario.

La aplicación de las operaciones del primario en los secundarios se realiza de manera asíncrona (figura 115), de forma que el conjunto de replicación puede seguir funcionando. La única anomalía es que puede que algunos secundarios devuelvan datos no actualizados.

**Figura 115.** Proceso de replicación



## 4. Configuración de nodos secundarios

En los nodos secundarios pueden ser configurados algunos parámetros:

- *Prioridad*
  - Cada nodo tiene un parámetro denominado «prioridad», que permite establecer qué preferencia tiene para ser elegido primario en las elecciones.
  - Un nodo tiene prioridad 0 nunca será elegido como primario.

- *Visibilidad*

- Cada nodo tiene una propiedad hidden, que, en caso de estar activa, hace que un nodo mantenga la información replicada como el resto, pero no es visible a las aplicaciones, que no pueden acceder al mismo.
- Deben tener prioridad 0, aunque sí pueden participar en las elecciones de los nodos primarios.

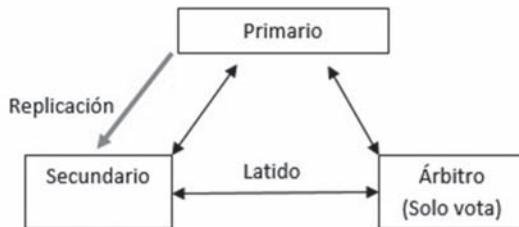
Se pueden distinguir varios tipos de nodos (figura 116):

**1) Nodos retardados**

- Es un tipo de nodo secundario que mantiene copias de los estados anteriores de la base de datos, de forma que puedan usarse en caso de una operación de recuperación ante fallos.
- Es posible configurar el tiempo de retardo (slaveDelay).
- Deben tener prioridad 0 y estar ocultos, pero participan en las elección de nuevos nodos primarios.

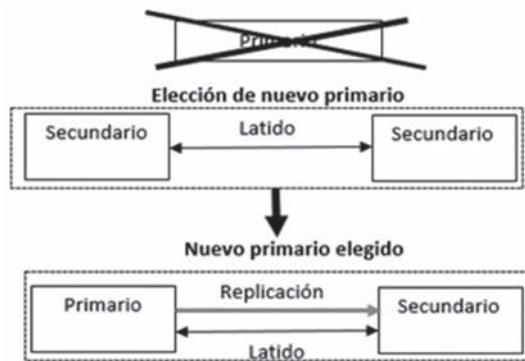
**2) Nodos árbitros**

- Es un nodo que no contiene copias de los datos y nunca se puede convertir en primario.
- Su función es votar en las elecciones de los primarios para desempatar los resultados, de forma que no sea necesario mantener un conjunto de nodos impar si no es necesario, evitando la sobrecarga que produce la replicación de un nodo adicional.
- Un nodo árbitro no debe actuar como árbitro en el conjunto de replicación al que pertenece.

**Figura 116.** Tipos de nodos secundarios y sus relaciones.

## 5. Recuperación ante un fallo del primario

Cuando el nodo primario no se comunica con el resto de los nodos secundarios durante más de 10 segundos, entonces se inicia el proceso para elegir un nuevo primario (figura 117). No se permiten escrituras hasta que no se elija uno nuevo.

**Figura 117.** Inicio del proceso de elección de un nuevo primario.

El nuevo primario debe cumplir las siguientes características:  
Se elige entre los nodos secundarios.

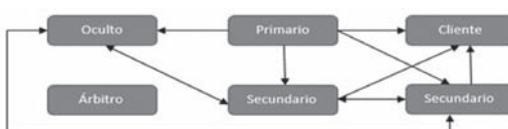
- Tiene la mayor prioridad. Los nodos con prioridad 0 no son elegibles.
- Puede ver a una mayoría de servidores que tengan voto.
- Está al día con los datos.

Un ReplicaSet puede tener hasta cincuenta miembros, pero solo pueden tener voto hasta siete miembros. Cuando el servidor primario cae, pueden perderse cambios si estos no han sido transferidos a ningún otro miembro del ReplicaSet. Otro miembro ejercerá de primario y, por tanto, no conocerá algunos de los cambios producidos. Cuando el servidor caído vuelve a conectar, debe descartar los cambios no propagados y ponerse al día con el primario. Los cambios descartados se pueden ver con bsondump y restaurarse manualmente con mongorestore.

Así, el proceso completo de replicación se puede resumir de la siguiente manera:

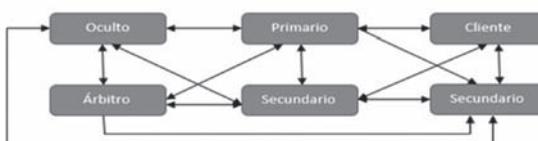
**1) Transferencia de datos** (figura 118).

**Figura 118.** Transferencia de datos.



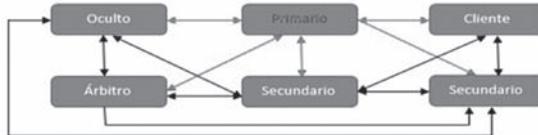
**2) Comprobación de estado** (figura 119).

**Figura 119.** Comprobación de estado.



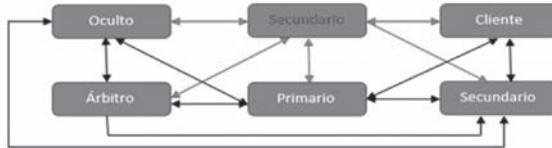
3) *Caída del servidor primario* (figura 120).

**Figura 120.** Caída del servidor primario.



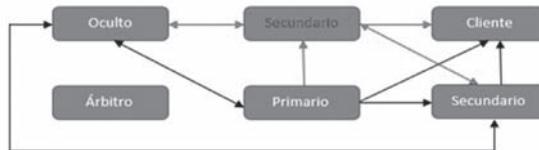
4) *Elección del nuevo servidor primario* (figura 121).

**Figura 121.** Elección del nuevo servidor primario.



5) *Transferencia de datos con el nuevo primario* (figura 122).

**Figura 122.** Transferencia de datos con el nuevo primario.



6) *Creación de un replica set*. Para crear un replica set se siguen los pasos:

- Comenzar una instancia del servidor con la opción `--replSet` para especificar un nuevo replica set en un puerto concreto y sobre un camino a un directorio de una base de datos existente:

```
mongod --port 27017 --dbpath /data/n1 --replSet rs0
```

- Conectarse con mongo.
- Usar la instrucción rs.initiate() para inicializar el replica set (figura 123):

**Figura 123.** Instrucción rs.initiate () .

```
> rs.initiate()
{
    "info2" : "no configuration explicitly specified -- making one",
    "me" : "Antonio:27017",
    "info" : "Config now saved locally. Should come online in about a minute.",
    "ok" : 1
}
```

Para ver la configuración del replica set se teclea rs.conf () (figura 124):

**Figura 124.** Instrucción rs.config () .

```
> rs.conf()
{
    "_id" : "rs0",
    "version" : 1,
    "members" : [
        {
            "id" : 0,
            "host" : "Antonio:27017"
        }
    ]
}
> rs0:PRIMARY>
```

Para chequear el estado del replica set se teclea rs.status () (figura 125):

**Figura 125.** Instrucción rs.status () .

```
rs0:PRIMARY> rs.status()
{
    "set" : "rs0",
    "date" : ISODate("2015-04-20T11:09:31Z"),
    "myState" : 1,
    "members" : [
        {
            "id" : 0,
            "name" : "Antonio:27017",
            "health" : 1,
            "state" : 1,
            "stateStr" : "PRIMARY",
            "uptime" : 249,
            "optime" : Timestamp(1429527950, 1),
            "optimeDate" : ISODate("2015-04-20T11:05:50Z"),
            "electionTime" : Timestamp(1429527951, 1),
            "electionDate" : ISODate("2015-04-20T11:05:51Z"),
            "self" : true
        }
    ],
    "ok" : 1
}
```

Ahora se va expandir el replica set con dos nuevos nodos secundarios. Para ello:

- Se abren dos nuevas instancias de mongod de la siguiente manera:

```
mongod --port 27018 --dbpath /data/n2 --replSet rs0
mongod --port 27019 --dbpath /data/n3 --replSet rs0
```

- Desde la consola de la primera instancia de mongod se introduce (figura 126):

```
rs.add ("hostname: puerto")
rs.add ("hostname: puerto")
```

**Figura 126.** Añadir nodos.

```
rs0:PRIMARY> rs.add<"Antonio:27018">
{
  "ok" : 1
}
rs0:PRIMARY> rs.add<"Antonio:27019">
{
  "ok" : 1
}
rs0:PRIMARY> =
```

- A continuación consultamos la configuración (figura 127):

**Figura 127.** Consulta de la configuración.

```
rs0:PRIMARY> rs.config()
{
  "_id" : "rs0",
  "version" : 3,
  "members" : [
    {
      "_id" : 0,
      "host" : "Antonio:27017"
    },
    {
      "_id" : 1,
      "host" : "Antonio:27018"
    },
    {
      "_id" : 2,
      "host" : "Antonio:27019"
    }
  ]
}
```

- Ahora consultamos el estado (figura 128):

**Figura 128.** Consulta del estado con rs.status () .

```
rs0:PRIMARY> rs.status()
{
  "set" : "rs0",
  "date" : ISODate("2015-04-20T14:12:46Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "Antonio:27017",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 421,
      "optime" : Timestamp(1429538988, 1),
      "optimeDate" : ISODate("2015-04-20T14:09:48Z"),
      "electionTime" : Timestamp(1429538776, 1),
      "electionDate" : ISODate("2015-04-20T14:06:16Z"),
      "self" : true
    },
    {
      "_id" : 1,
      "name" : "Antonio:27018",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 191,
      "optime" : Timestamp(1429538988, 1),
      "optimeDate" : ISODate("2015-04-20T14:09:48Z"),
      "lastHeartbeat" : ISODate("2015-04-20T14:12:44Z"),
      "lastHeartbeatRecv" : ISODate("2015-04-20T14:12:46Z"),
      "pingMs" : 4,
      "syncingTo" : "Antonio:27017"
    },
    {
      "_id" : 2,
      "name" : "Antonio:27019",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 178,
      "optime" : Timestamp(1429538988, 1),
      "optimeDate" : ISODate("2015-04-20T14:09:48Z"),
      "lastHeartbeat" : ISODate("2015-04-20T14:12:44Z"),
      "lastHeartbeatRecv" : ISODate("2015-04-20T14:12:45Z"),
      "pingMs" : 1,
      "syncingTo" : "Antonio:27017"
    }
  ],
  "ok" : 1
}
```

Si se quiere añadir un árbitro se siguen los pasos anteriores, pero cuando se añade el nodo al replica set se introduce rs.add (“Hostname: puerto”, true) (figura 129):

**Figura 129.** Adición de un nodo árbitro.

```
rs0:PRIMARY> rs.add("Antonio:27018",true)
{ "ok" : 1 }
```

Si se consulta el estado del replica set (figura 130):

**Figura 130.** Estado del replica set.

```
{
  "_id" : 1,
  "name" : "Antonio:27018",
  "health" : 1,
  "state" : 7,
  "stateStr" : "ARBITER",
  "uptime" : 51,
  "lastHeartbeat" : ISODate("2015-04-20T14:46:11Z"),
  "lastHeartbeatRecv" : ISODate("2015-04-20T14:46:11Z"),
  "pingMs" : 1
}
```

**7) Configuración de un replica set.** Para configurar un replica set se siguen los pasos siguientes:

- Se recupera la configuración del replica set en una variable (figura 131)

**Figura 131.** Recuperación de la configuración del replica set.

```
rs0:PRIMARY> config=rs.config<>
{
  "_id" : "rs0",
  "version" : 3,
  "members" : [
    {
      "_id" : 0,
      "host" : "Antonio:27017"
    },
    {
      "_id" : 1,
      "host" : "Antonio:27018"
    },
    {
      "_id" : 2,
      "host" : "Antonio:27019"
    }
  ]
}
```

- Nodo retardado:
  - Para configurar un nodo retardado es necesario establecer la prioridad del nodo a 0, indicar que es oculto, definir un tiempo de retardo y cantidad de votos en las elecciones (figura 132).

**Figura 132.** Configuración de un nodo retardado.

```
rs0:PRIMARY> configu.members[1].priority=0
0
rs0:PRIMARY> configu.members[1].hidden=true
true
rs0:PRIMARY> configu.members[1].slaveDelay=3000
3000
rs0:PRIMARY> configu.members[1].votes=3
3
```

- A continuación se reconfigura el replica set con el comando rs.reconfig (configuración) en la figura 133.

**Figura 133.** Aplicación del comando rs.reconfig () .

```
rs0:PRIMARY> rs.reconfig(config)
2015-04-28T22:45:12.149+0200  DBClientCursor::init call() failed
2015-04-28T22:45:12.164+0200  Trying reconnect to 127.0.0.1:27012 <127.0.0.1> failed
2015-04-28T22:45:12.168+0200  reconnect 127.0.0.1:27012 <127.0.0.1> ok
reconnected to server after rs command (which is normal)

rs0:PRIMARY> rs.config()
{
  "_id" : "rs0",
  "version" : 4,
  "members" : [
    {
      "_id" : 0,
      "host" : "Antonio:27017"
    },
    {
      "_id" : 1,
      "host" : "Antonio:27018",
      "votes" : 3,
      "priority" : 0,
      "slaveDelay" : 3000,
      "hidden" : true
    },
    {
      "_id" : 2,
      "host" : "Antonio:27019"
    }
  ]
}
```

- Hay que observar que las acciones realizadas son equivalentes a si se hubieran realizado cuando se ha añadido el nodo al replica set:

```
rs.add ({“_id”: 1, “host”: “Antonio: 27018”, “votes”: 3, “priority”: 0, “slaveDelay”: 3000, “hidden”: true})
```

- Configurar un nodo secundario como un árbitro sobre el mismo puerto:
  - En primer lugar hay que desconectar el secundario del replica set.
  - Se elimina el secundario del replica set con el método rs.remove (<hostname: puerto>) en la figura 134.

**Figura 134.** Eliminación de un nodo secundario.

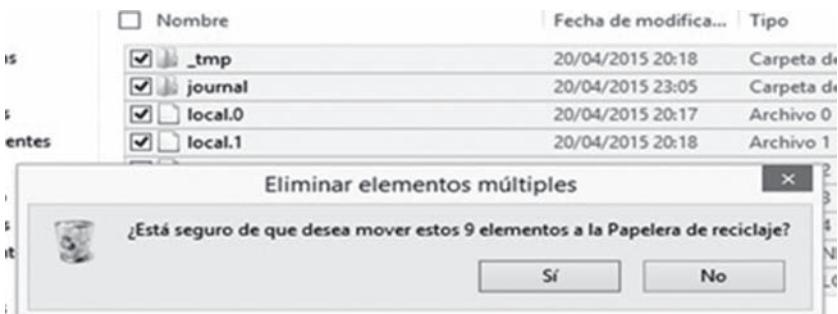
```
rs0:PRIMARY> rs.remove("Antonio:27019")
2015-04-20T23:06:32.985+0200 DBCClientCursor::init call() failed
2015-04-20T23:06:32.993+0200 Error: error doing query: failed at src/mongo/shell
query.js:81
2015-04-20T23:06:33.003+0200 trying reconnect to 127.0.0.1:27017 <127.0.0.1> fai
led
2015-04-20T23:06:33.013+0200 reconnect 127.0.0.1:27017 <127.0.0.1> ok
```

- Se comprueba que se ha eliminado del replica set (figura 135):

**Figura 135.** Comprobación de la eliminación del secundario.

```
rs0:PRIMARY> rs.config()
{
    "_id" : "rs0",
    "version" : 5,
    "members" : [
        {
            "_id" : 0,
            "host" : "Antonio:27017"
        },
        {
            "_id" : 1,
            "host" : "Antonio:27018",
            "votes" : 3,
            "priority" : 0,
            "slaveDelay" : 3000,
            "hidden" : true
        }
    ]
}
```

- Se elimina el fichero de la base de datos (figura 136):

**Figura 136.** Eliminación del fichero de la base de datos.

- Se ejecuta la instancia nuevamente sobre el mismo puerto:

```
mongod --port 27019 --dbpath /data/n3 --replSet rs0
```

- Se añade el nodo como un árbitro (figura 137):

**Figura 137.** Adición de un nodo árbitro.

```
rs0:PRIMARY> rs.addArb("Antonio:27019")
{
  "ok" : 1
}
```

- Se comprueba que la clave-valor se ha realizado correctamente con rs.config () en la figura 138:

**Figura 138.** Comprobación de la operación de adición.

```
rs0:PRIMARY> rs.config()
{
  "_id" : "rs0",
  "version" : 6,
  "members" : [
    {
      "id" : 0,
      "host" : "Antonio:27017"
    },
    {
      "id" : 1,
      "host" : "Antonio:27018",
      "votes" : 3,
      "priority" : 0,
      "slaveDelay" : 3000,
      "hidden" : true
    },
    {
      "id" : 2,
      "host" : "Antonio:27019",
      "arbiterOnly" : true
    }
  ]
}
```

**8) Eliminación de miembros de un replica set.** Se van a considerar dos casos diferentes.

- En caso de que la instancia no sea el nodo primario:
  - Se para la instancia correspondiente.
  - Desde la instancia del nodo primario del replica set se introduce el comando:

```
rs.remove (<Hostname: puerto>)
```

- Si la instancia es el nodo primario:
  - Primero hay que obligarle a dejar de ser nodo primario, para lo que usamos el comando rs.stepDown(Cantidad\_Tiempo) (figura 139):

**Figura 139.** Ejecución del comando rs.stepDown () .

```

rs0:PRIMARY> rs.stepDown()
2015-04-21T00:12:52.517+0200 DBClientCursor::init call() failed
2015-04-21T00:12:52.531+0200 Error: error doing query: failed at src/mongo/shell
    /query.js:81
2015-04-21T00:12:52.534+0200 trying reconnect to 127.0.0.1:27017 <127.0.0.1> failed
2015-04-21T00:12:52.544+0200 reconnect 127.0.0.1:27017 <127.0.0.1> ok
rs0:SECONDARY> rs.status()
{
  "set" : "rs0",
  "date" : ISODate("2015-04-20T22:13:01Z"),
  "myState" : 2,
  "syncingTo" : "Antonio:27018",
  "members" : [
    {
      "_id" : 0,
      "name" : "Antonio:27018",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 167,
      "optime" : Timestamp(1429562877, 1),
      "optimeDate" : ISODate("2015-04-20T22:11:17Z"),
      "infoMessage" : "syncing to: Antonio:27018",
      "self" : true
    }
  ]
}

```

- Identificamos el nuevo primario (figura 140).

**Figura 140.** Identificación del nuevo primario.

```

{
  "_id" : 1,
  "name" : "Antonio:27018",
  "health" : 1,
  "state" : 1,
  "stateStr" : "PRIMARY",
  "uptime" : 160,
  "optime" : Timestamp(1429562877, 1),
  "optimeDate" : ISODate("2015-04-20T22:11:17Z"),
  "lastHeartbeat" : ISODate("2015-04-20T22:13:43Z"),
  "lastHeartbeatRecv" : ISODate("2015-04-20T22:13:44Z"),
  "pingMs" : 8,
  "electionTime" : Timestamp(1429562974, 1),
  "electionDate" : ISODate("2015-04-20T22:12:54Z")
}

```

- Nos conectamos al nuevo primario (figura 141):

**Figura 141.** Conexión al nuevo primario.

```

C:\mongodb\bin>mongo -port 27018
MongoDB shell version: 2.6.7
connecting to: 127.0.0.1:27018/test
rs0:PRIMARY>

```

- Se elimina el nodo anterior que era primario desde la instancia del actual primario (figura 142):

**Figura 142.** Eliminación del anterior primario.

```
rs0:PRIMARY> rs.remove("Antonio:27017")
2015-04-21T00:18:44.282+0200  DBClientCursor::init call() failed
2015-04-21T00:18:44.214+0200  Error: error doing query: failed at src/mongo/shell/query.js:81
2015-04-21T00:18:44.219+0200  trying reconnect to 127.0.0.1:27018 <127.0.0.1> failed
2015-04-21T00:18:44.233+0200  reconnect 127.0.0.1:27018 <127.0.0.1> ok
rs0:PRIMARY> rs.config()
{
    "_id" : "rs0",
    "version" : 4,
    "members" : [
        {
            "_id" : 1,
            "host" : "Antonio:27018"
        },
        {
            "_id" : 2,
            "host" : "Antonio:27019"
        }
    ]
}
```

- Se puede acceder al antiguo nodo y consultar su estado para ver el estado en el que se encuentra (figura 143):

**Figura 143.** Comprobación de la operación de eliminación.

```
C:\mongodb\bin>nmongo --port 27017
MongoDB shell version: 2.6.7
connecting to: 127.0.0.1:27017/test
rs0:REMOVED> rs.status()
{
    "set" : "rs0",
    "date" : ISODate("2015-04-20T22:20:39Z"),
    "nyState" : 10,
    "members" : [
        {
            "_id" : 0,
            "name" : "Antonio:27017",
            "health" : 1,
            "state" : 10,
            "stateStr" : "REMOVED",
            "uptime" : 625,
            "optime" : Timestap(1429568324, 1),
            "optimeDate" : ISODate("2015-04-20T22:18:44Z"),
            "infoMessage" : "sync source problem: 10278 dbclient error communicating with server: Antonio:27018",
            "self" : true
        }
    ],
    "ok" : 1
}
```

**9)** *Determinación de si un nodo es primario.* Para determinar si un nodo es primario, se usa el comando db.isMaster () (figura 144).

**Figura 144.** Determinación de si un nodo es primario.

```

rs0:PRIMARY> db.isMaster()
{
  "setName" : "rs0",
  "setVersion" : 3,
  "isMaster" : true,
  "secondary" : false,
  "hosts" : [
    "Antonio:27017",
    "Antonio:27019",
    "Antonio:27018"
  ],
  "primary" : "Antonio:27017",
  "me" : "Antonio:27017",
  "arbiterOnly" : false,
  "maxMessageSizeBytes" : 16222216,
  "maxWriteBatchSize" : 1000,
  "localTime" : ISODate("2015-04-21T07:12:27.00Z"),
  "maxWireVersion" : 2,
  "minWireVersion" : 0,
  "ok" : 1
}

```

**10)** *Configuración de los miembros de un replica set.* Cuando se quiere configurar un replica set hay dos formas:

- La primera forma es utilizar la configuración por defecto mediante los métodos rs.initiate () y rs.add () .
- La segunda forma consiste en crear el documento de configuración. El documento está formado por (figura 145):
  - Campo «\_id», que es el nombre del replica set.
  - Campo «version», que es número creciente que indica distintas revisiones del replica set.
  - Campo «members», que describe la configuración de los diferentes miembros del replica set. Está formado por:
    - Campo «\_id»: identificador numérico de un miembro del replica set.
    - Campo «host»: nombre del host y del puerto.
    - Campo «arbiterOnly»: booleano que idéntica un arbitro
    - Campo «buildIndexes»: booleano que indica si existen índices construidos sobre ese miembro.
    - Campo «hidden»: indica si es un miembro oculto.

- Campo «`priority`»: indica la elegibilidad del miembro para ser primario. Es un número entre 0 y 1.000.
  - Campo «`ttags`»: es un documento que mapea claves con valores relativos a los miembros del replica set.
  - Campo «`slaveDelay`»: tiempo en milisegundos que representa el retardo respecto a la aplicación de los cambios del primario.
  - Campo «`votes`»: número de votos de un miembro. Es un número entre 0 y 1. En el caso de los árbitros es exactamente 1.
- 
- Campo «`settings`»: son opciones de configuración que se aplican al replica set entero. Está formado por:
    - Campo «`chainingAllowed`»: booleano que indica si está permitido que los secundarios puedan replicar datos de otros secundarios.
    - Campo «`getLastErrorDefaults`»: es un documento que especifica la confirmación del éxito de una escritura de un replica set.
    - Campo «`getLastErrorModes`»: es un documento usado para confirmar el éxito de una operación de escritura mediante el uso de etiquetas.
    - Campo «`heartbeatTimeoutSecs`»: tiempo en segundos que se espera a tener señales de vida de un miembro antes de considerarlo innaccesible.

**Figura 145.** Documento de configuración.

```
{  
    _id: <cadena>,  
    version: <entero>,  
    members: [  
        {  
            _id: <entero>,  
            host: <cadena>,  
            arbiterOnly: <booleano>,  
            buildIndexes: <booleano>,  
            hidden: <booleano>,  
            priority: <número>,  
            tags: <documento>,  
            slaveDelay: <entero>,  
            votes: <número>  
        },  
        ...  
    ],  
    settings: {  
        getLastErrorHandlerDefaults: <documento>,  
        chainingAllowed: <booleano>,  
        getLastErrorHandlerModes: <documento>,  
        heartbeatTimeoutSecs: <entero>  
    }  
}
```

**11)** *Convertir un nodo de un replica set en independiente.* Para que un nodo se convierta en una instancia independiente del replica set:

- Se elimina del replica set mediante rs.remove () .
- Se reinicia la instancia.
- Se elimina la base de datos local.

## Bibliografía

- Banker, K.** (2011). *MongoDB in action*. Manning Publications Co.
- Chodorow, K.** (2013). *MongoDB: the definitive guide*. O'Reilly Media, Inc.
- Hows, D.; Membrey, P.; Plugge, E.** (2014). MongoDB Basics. Apress.
- Membrey, P.; Plugge, E.; Hawkins, D.** (2010). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress.
- Sitio oficial MongoDB, «Replication Introduction». <<https://docs.mongodb.org/v2.6/core/replication-introduction/>>

## Capítulo VIII **Sharding**

### 1. Introducción

En este capítulo se va a describir el sharding, que es el mecanismo que implementa MongoDB para realizar un procesamiento distribuido de los datos. La necesidad de manejar conjuntos de datos extremadamente grandes se encuentra con problemas tales como limitaciones de la CPU en cuanto a las consultas que se pueden procesar y falta de espacio en memoria para el almacenamiento de índices. Hay dos formas de solucionar estas limitaciones:

- Escalado vertical: mejores máquinas: caro y limitado.
- Escalado horizontal: más máquinas: baratas e «ilimitadas».

En una base de datos distribuida, los datos se parten en conjuntos disjuntos y cada conjunto se coloca en un servidor. Hay un balanceo de carga:

- Cada servidor maneja menos datos.
- Las consultas pueden afectar a uno o varios servidores.
- Las escrituras solo afectan a un servidor.

Sin embargo, se produce una penalización por gestión de comunicación entre servidores.

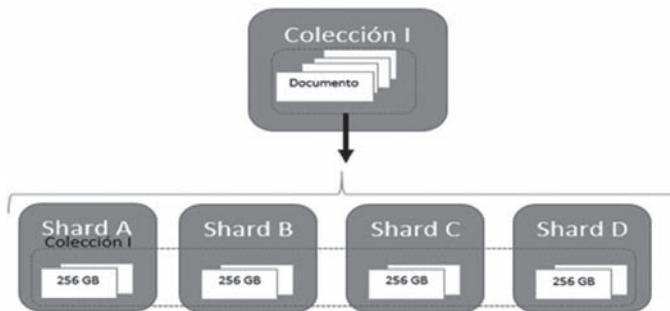
A lo largo del capítulo se van a definir los principales conceptos relacionados con el procesamiento distribuido de datos que se puede

realizar con MongoDB mediante sharding. Asimismo se va describir con detalle cómo crear un clúster, gestionarlo y mantenerlo.

## 2. Sharding

El sharding es la forma en la que MongoDB implementa el escalado horizontal. Se distribuyen los datos de una base de datos a lo largo de múltiples máquinas (figura 146).

**Figura 146.** Distribución de datos en shards.



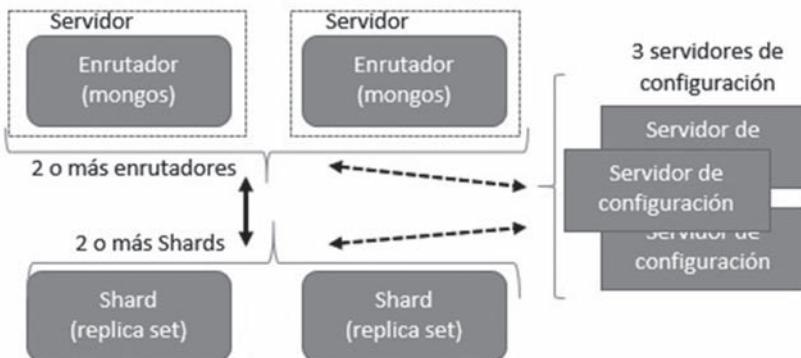
Para realizar la distribución de los datos es necesario:

- Indicar criterio para separar los datos.
- Montar infraestructura de servidores.
- Configurar la distribución automática de los datos.
- Configurar la distribución automática de las consultas y modificaciones.

En MongoDB se realiza esta distribución a través de lo que se denomina «sharded clúster», un conjunto de máquinas donde

cada una de ellas actúa de una determinada forma. Existen tres tipos de máquinas en un sharded clúster son (figura 147):

- *Shards:*
  - Son los elementos en los que se organizan y se almacenan los datos.
  - Pueden contener más de un nodo y en general suelen ser replica set.
- Enrutadores de consultas:
  - No almacenan datos.
  - Actúan como interfaces con las aplicaciones cliente y redirigen las operaciones solicitadas a los shards donde se encuentran almacenados los datos implicados.
  - Tras obtener dichos datos, estos son devueltos por el rúter a la aplicación cliente.
  - Pueden existir varios enrutadores.
- Servidores de configuración:
  - Almacena los metadatos del clúster, y se utilizan exactamente tres.
  - Contienen la información sobre el shard donde se encuentra cada subconjunto de datos, de forma que los enrutadores lo consultan para saber dónde deben redirigir la solicitud.

**Figura 147.** Máquinas de un sharded clúster.

### 3. Clave de sharding

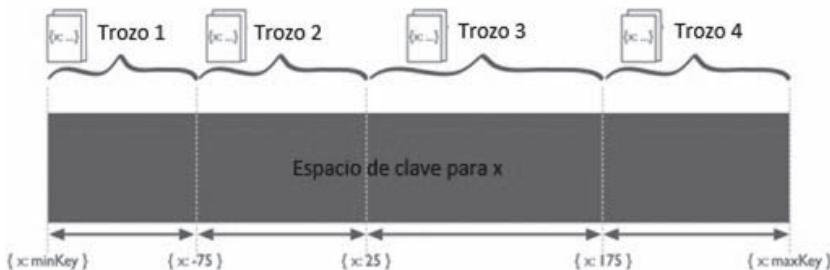
La distribución de los documentos a través de los shard se hace usando una shard key o clave de sharding. Una clave de sharding es un campo indexado o un campo compuesto indexado que existe en todos los documentos de la colección. La clave se divide en trozos y se distribuye cada trozo en los shards. La clave de sharding debe cumplir las siguientes características:

- Disponer de una cardinalidad suficiente de forma que el rango total de valores sea lo suficientemente grande como para poder dividirlo apropiadamente entre los shards.
- Se debe evitar que la clave no sea distribuida uniformemente entre todos los shards, ya que se corre el riesgo de sobrecargar algunos shards.

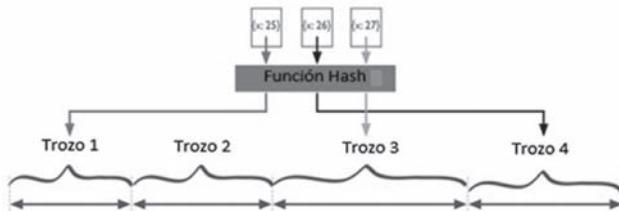
Existen dos formas de dividir los trozos:

- Particionamiento basados en rangos (figura 148):
  - Se dividen los datos en rangos determinados por los valores de la shard key.
  - Por ejemplo, si se considera una clave numérica, se podrían representar en un eje los valores de la shard key, de manera que MongoDB partitiona los datos en rangos que no se solapan en denominados «trozos», donde cada trozo es un rango de valores desde un valor mínimo hasta un valor máximo.
  - Dado un rango, los documentos con valores cercanos a un valor de la clave de sharding estarán en el mismo trozo y, por tanto, en el mismo shard.

**Figura 148.** Particionamiento basado en rangos.



- Particionamiento basado en hash (figura 149):
  - Se computa un hash del valor del campo, y se usan estos hashes para crear los trozos.
  - En este caso, dos documentos con valores cercanos de shard key no tienen por qué estar en el mismo trozo, lo que asegura una distribución aleatoria de la colección en el clúster.

**Figura 149.** Particionamiento basado en hash.

Observaciones:

- Los servidores de configuración mapean los rangos de claves de sharding con los shards.
- Se suelen implementar índices sobre las shard keys, o bien de forma única o bien que comiencen por ellas, ya que se agiliza el proceso de inserción de datos.
- Las claves son inmutables de manera que tras una inserción no se puede modificar su valor.

El particionamiento basado en rangos soporta de forma más eficiente las consultas sobre rangos. Así, dada una consulta de rango sobre una shard key, el enrutador puede fácilmente determinar qué trozo solapa el rango y enrutar en la consulta solo aquellos shards que contienen esos trozos. Sin embargo, el particionamiento basado en rangos puede dar lugar a una distribución desigual de los datos. Por ejemplo, si la shard key es un campo que se incrementa de manera lineal, tal como el tiempo, entonces todas las peticiones para un rango de tiempo dado irán al mismo trozo, y por tanto al mismo «shard». En esta situación, un pequeño conjunto de shards puede que reciba la mayoría de las peticiones y el sistema no escale bien.

El particionamiento basado en hash, por el contrario, asegura una distribución equilibrada de datos a expensas de sacrificar la

eficiencia de las consultas sobre rangos. Los valores de clave hasheados dan lugar a una distribución aleatoria de los datos a través de los rangos y, por tanto, de los shards. Pero las distribuciones aleatorias de los datos hacen que probablemente una consulta de rangos sobre una clave de sharding no pueda dirigirse a unos pocos shards, sino que tendrá que visitar cada shard para obtener el resultado.

## 4. Configuración de la distribución de datos

Es posible dirigir el balanceo de los datos asociando etiquetas a los rangos de las claves de sharding, y asignar las etiquetas a shards concretos. De esta forma, el balanceador migra los datos etiquetados a los shards apropiados y asegura que el clúster obliga a la distribución que se ha descrito en las etiquetas.

La adición de nuevos datos o la adición de nuevos servidores puede resultar en una distribución no balanceada dentro del clúster, tal que un shard particular contenga un número significativo de trozos más que otro «shard» o un tamaño de trozo más grande que otros tamaños de trozos. Existen dos formas de asegurar el balanceo de los datos, mediante el splitting y el balanceador.

El Splitting se caracteriza por (figura 150):

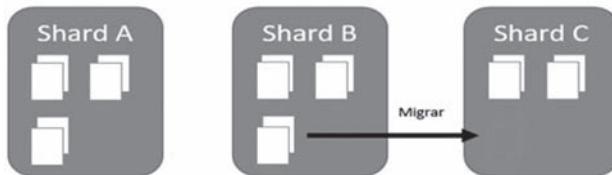
- Es un proceso que gestiona los trozos cuando crecen demasiado. De manera que cuando un trozo crece por encima de un umbral especificado, se divide el trozo por la mitad.
- Las inserciones y las actualizaciones desencadenan los splits.
- Es un cambio eficiente de metadatos.
- No migra datos o afecta a los shards.

**Figura 150.** Esquema del splitting.



El balanceador se caracteriza por:

- Es un proceso que gestiona las migraciones de trozos. Puede ejecutarse desde cualquiera de los rúteres del clúster.
- Cuando la distribución de los datos no es equilibrada, el balanceador migra los trozos del shard que tiene el mayor número de trozos al shard que tiene el menor número de trozos, hasta que la colección se balancea. Por ejemplo, si una colección tiene 100 trozos sobre un shard y 50 sobre otro, el balanceador migra trozos del primero al segundo.
- El shard gestiona la migración de los trozos como una operación de segundo plano entre el shard origen y el shard destino. Así, durante el proceso de migración, el shard destino recibe todos los documentos actuales en el trozo desde el shard origen. A continuación, el shard destino captura y aplica todos los cambios hechos a los datos durante el proceso de migración. Y finalmente los metadatos respecto a la localización del trozo son actualizados.
- Si durante el proceso de migración se produce algún error, el balanceador aborta el proceso dejando el trozo sin intercambiar en el shard origen. El sistema elimina los datos del trozo del shard origen cuando el proceso de migración se realiza con éxito.

**Figura 151.** Esquema del balanceamiento.

Hay que observar que:

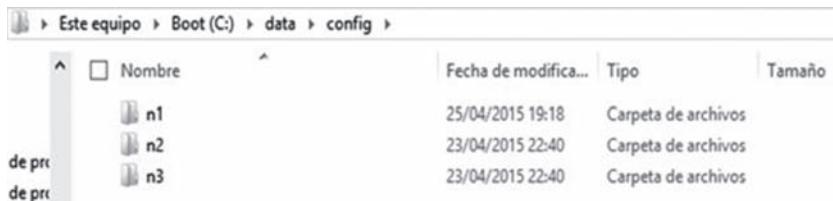
- Cuando se añade un shard al clúster se crea una situación de desbalanceo, puesto que el nuevo shard no tiene trozos. Así, mientras que se produce el proceso de migración al nuevo shard, puede pasar un tiempo hasta que el clúster esté balanceado.
- De la misma forma, cuando se elimina un shard, el balanecedor migrar todos los trozos de un shard a otros shards. Después de migrar todos los datos y actualizar los metadatos, se puede eliminar el shard de manera segura.

## 5. Creación de un clúster

Para crear un clúster se siguen los pasos:

### *Servidores de configuración*

Dado que hay que crear tres servidores de configuración, se crean tres directorios para cada uno de los servidores (figura 152):

**Figura 152.** Directories para los servidores.


| Nombre | Fecha de modifica... | Tipo                |
|--------|----------------------|---------------------|
| n1     | 25/04/2015 19:18     | Carpeta de archivos |
| n2     | 23/04/2015 22:40     | Carpeta de archivos |
| n3     | 23/04/2015 22:40     | Carpeta de archivos |

- Se crea cada servidor de configuración, para lo que por cada servidor de configuración se abre una instancia de mongod en un puerto con la opción --port, se indica el directorio dónde se almacena los datos con la opción --dbpath, y se indica la opción --configsvr (figuras 153, 154 y 155):

**Figura 153.** Instancia mongod para el servidor 1.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 22023 --dbpath /data/config/n2 --configsvr
2015-04-25T19:24:16.627+0200 I JOURNAL [initandlisten] journal dir=/data/config/n2\journal
2015-04-25T19:24:16.639+0200 I JOURNAL [initandlisten] recover : no journal files present, no recovery needed
2015-04-25T19:24:16.734+0200 I JOURNAL [durability] Durability thread started
2015-04-25T19:24:16.761+0200 I JOURNAL [journal writer] Journal writer thread s
```

**Figura 154.** Instancia mongod para el servidor 2.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27024 --dbpath /data/config/n3 --configsvr
2015-04-25T19:24:01.714+0200 I JOURNAL [initandlisten] journal dir=/data/config/n3\journal
2015-04-25T19:24:01.726+0200 I JOURNAL [initandlisten] recover : no journal files present, no recovery needed
2015-04-25T19:24:01.929+0200 I JOURNAL [durability] Durability thread started
2015-04-25T19:24:01.932+0200 I JOURNAL [journal writer] Journal writer thread s
```

**Figura 155.** Instancia mongod para el servidor 3.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27022 --dbpath /data/config/n1 --configsvr
2015-04-25T19:22:40.515+0200 I JOURNAL [initandlisten] journal dir=/data/config/n1\journal
2015-04-25T19:22:40.525+0200 I JOURNAL [initandlisten] recover : no journal files present, no recovery needed
2015-04-25T19:22:40.711+0200 I JOURNAL [initandlisten] preallocateIsFaster=true
2.4
```

## Enrutadores

- Para crear un enrutador se abre una instancia de mongos (figura 156). Este programa se puede encontrar en el directorio bin de la instalación de MongoDB. Se invoca indicando el puerto donde se va a ejecutar con la opción --port, se listan los servidores de configuración indicando el nombre del equipo y el puerto donde se ejecutan, y se usa la opción --configdb para indicar que es un enrutador.

**Figura 156.** Creación de un enrutador.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongos --configdb localhost:27022,localhost:27023,localhost:27024 --port 27021
2015-04-25T19:37:10.125+0200 I SHARDING [mongosMain] MongoS version 3.0.2 starting: pid=4600 port=27021 64-bit host=Antonio <--help for usage>
2015-04-25T19:37:10.133+0200 I CONTROL [mongosMain] db version v3.0.2
2015-04-25T19:37:10.133+0200 I CONTROL [mongosMain] git version: 6201872043ecbb
:b4fccc169b5482dcf385fc464f
2015-04-25T19:37:10.133+0200 I CONTROL [mongosMain] OpenSSL version: OpenSSL 1.0.2n-fips 19 Mar 2015
```

Hay que observar que también es posible en este paso indicar el tamaño de los trozos con la opción --chunkSize. Si no se indica esta opción, por defecto toma el valor de 64 MB.

## Shards

Se van a crear, por ejemplo, dos shards, por lo que es necesario disponer de dos directorios para cada uno de los servidores (figura 157):

**Figura 157.** Creación de directorios para los shards.

| Este equipo > Boot (C:) > data > shards > |                      |                     |        |  |
|-------------------------------------------|----------------------|---------------------|--------|--|
| Nombre                                    | Fecha de modifica... | Tipo                | Tamaño |  |
| n1                                        | 25/04/2015 19:18     | Carpeta de archivos |        |  |
| n2                                        | 25/04/2015 19:18     | Carpeta de archivos |        |  |

- Para crear un shard se abre una instancia de mongod indicando el puerto donde se va a ejecutar con la opción –port, se indica el directorio donde se almacenan los datos con la opción –dbpath, y se indica la opción --shardsvr (figuras 158 y 159):

**Figura 158.** Instancia de mongod para shard 1.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27026 --dbpath /data/shard
s/n1 --shardsvr
2015-04-25T19:45:36.041+0200 I JOURNAL [initandlisten] journal dir=/data/shards
/n1\journal
2015-04-25T19:45:36.053+0200 I JOURNAL [initandlisten] recover : no journal fil
es present, no recovery needed
2015-04-25T19:45:36.324+0200 I JOURNAL [durability] Durability thread started
2015-04-25T19:45:36.328+0200 I JOURNAL [journal writer] Journal writer thread s
tarted
```

**Figura 159.** Instancia de mongod para shard 2.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27027 --dbpath /data/shard
s/n2 --shardsvr
2015-04-25T19:46:07.010+0200 I JOURNAL [initandlisten] journal dir=/data/shards
/n2\journal
2015-04-25T19:46:07.050+0200 I JOURNAL [initandlisten] recover : no journal fil
es present, no recovery needed
2015-04-25T19:46:07.330+0200 I JOURNAL [durability] Durability thread started
2015-04-25T19:46:07.333+0200 I JOURNAL [journal writer] Journal writer thread s
tarted
```

A continuación se agregan los shards al clúster (figura 160). Para ello se inicia una instancia de mongo en el puerto donde se está ejecutando el enrutador, y cada shard se añade con el comando sh.addShard (<Nombre del equipo: puerto>), indicando el equipo y el puerto donde se ejecuta el shard.

**Figura 160.** Agregación de los shards al clúster.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongo --port 27021
MongoDB shell version: 3.0.2
connecting to: 127.0.0.1:27021/test
mongos> sh.addShard("localhost:27026")
{
  "shardAdded" : "shard0000",
  "ok" : 1
}
mongos> sh.addShard("localhost:27027")
{
  "shardAdded" : "shard0001",
  "ok" : 1
}
```

Se puede consultar el estado del clúster mediante el método sh.status() (figura 161).

**Figura 161.** Consulta del estado del clúster.

```

mongos> sh.status()
--- Sharding Status ---
shardingVersion: {
  "id": "553hd0c9dabfed166e263fea",
  "minCompatibleVersion": 5,
  "currentVersion": 6,
  "clusterId": ObjectID("553hd0c9dabfed166e263fea")
}
shards: [
  {
    "_id": "shard0000",
    "host": "localhost:27026"
  },
  {
    "_id": "shard0001",
    "host": "localhost:27027"
  }
]
balancer:
  Currently enabled: yes
  Currently running: no
  Failed balancer rounds in last 5 attempts: 0
  Migration Results for the last 24 hours:
    No recent migrations
databases: [
  {
    "_id": "admin",
    "partitioned": false,
    "primary": "config"
  }
]

```

### *Activar el sharding sobre una base de datos y una colección*

- Desde el terminal de la Shell donde se ejecuta la instancia del enrutador, se selecciona la base de datos que se va a distribuir, y se invoca el método sh.enablesharding sobre la misma (figura 162):

**Figura 162.** Invocación del método sh.enablesharding.

```

mongos> use testDB
switched to db testDB
mongos> sh.enableSharding("testDB")
{"ok": 1}

```

- A continuación se indica la colección de la base de datos que va a ser distribuida (figura 163). Para ello se usa el método sh.shardCollection (), que toma como parámetros la colección especificada como base\_datos.colección y la clave de distribución. La clave de distribución puede ser la especificación de un índice simple o compuesto de la forma {campo1:1,..., campo n: 1} o bien una clave hasheada que se especifica como {campo:"hashed"}.

**Figura 163.** Indicación de la colección que distribuir.

```

mongos> sh.shardCollection("testDB.prueba", {"_id": "hashed"})
{
  "collectionsharded": "testDB.prueba",
  "ok": 1
}

```

- El sistema habrá distribuido los documentos y puede consultarse su distribución mediante el método getShardDistribution () aplicado sobre la colección (figura 164).

**Figura 164.** Consulta sobre la distribución realizada.

```
mongos> db.prueba.getShardDistribution()
{
  "shards": [
    {
      "shard": "shard0000",
      "at": "localhost:27026",
      "data": 2.29MB,
      "docs": 50110,
      "chunks": 2,
      "estimated docs per chunk": 25055
    },
    {
      "shard": "shard0001",
      "at": "localhost:27027",
      "data": 2.28MB,
      "docs": 49891,
      "chunks": 2,
      "estimated docs per chunk": 24945
    }
  ],
  "totals": {
    "data": 4.57MB,
    "docs": 100001,
    "chunks": 4
  },
  "status": {
    "shard": "shard0000",
    "contains": "50.1x data, 50.1x docs in cluster, avg obj size on shard": 48B
  },
  "shard": "shard0001",
  "contains": "49.89% data, 49.89% docs in cluster, avg obj size on shard": 48B
}
```

- Se pueden consultar aspectos del clúster usando el comando sh.status () (figura 165).

**Figura 165.** Consulta sobre el estado del clúster.

```
mongos> sh.status()
--- Sharding Status ---
sharding version: {
  "id": 1,
  "minCompatibleVersion": 5,
  "currentVersion": 6,
  "clusterId": ObjectId("553bd0c9dabfed166e263fea")
}
shards: [
  {
    "_id": "shard0000",
    "host": "localhost:27026"
  },
  {
    "_id": "shard0001",
    "host": "localhost:27027"
  }
]
balancer:
  Currently enabled: yes
  Currently running: no
  Failed balancer rounds in last 5 attempts: 0
  Migration Results for the last 24 hours:
    1 : Success
    1 : Failed with error 'migration already in progress', from shard0000 to shard0001
databases:
  {
    "_id": "admin",
    "partitioned": false,
    "primary": "config"
  },
  {
    "_id": "testDB",
    "partitioned": true,
    "primary": "shard0000"
  },
  testDB.prueba
    shard key: { "_id": "hashed" }
    chunks:
      shard0000 2
      shard0001 2
    < "_id": { "$minKey": 1 } > --> < "_id": NumberLong("-4611686018427387902") >
    on : shard0000 Timestamp(2, 2)
    < "_id": NumberLong("-4611686018427387902") > --> < "_id": NumberLong("4611686018427387902") >
    on : shard0000 Timestamp(2, 3)
    < "_id": NumberLong(0) > --> < "_id": NumberLong("4611686018427387902") >
    on : shard0001 Timestamp(2, 4)
    < "_id": NumberLong("4611686018427387902") > --> < "_id": NumberLong("4611686018427387902") >
    on : shard0001 Timestamp(2, 5)
  }
```

Hay que observar que:

- En la parte inferior del documento aparecen los rangos que ha tomado cada shard sobre la clave de sharding `_id` hasheadas para distribuir los documentos.
- También en el documento se hace explícito que existen cuatro trozos, dos en cada shard. Un trozo está delimitado por un rango definido sobre la clave de sharding. Pueden ocurrir varias situaciones. Si un shard tiene un trozo que supera los 64 MB o el especificado, se produce una separación o splitting que divide el trozo en dos. Otra situación se da cuando un shard tiene varios trozos más en comparación con otros shards. En este caso, se produce un proceso de migración de trozos en los extremos del rango de la clave de sharding de un shard a otro shard.

## 6. Adición de shards a un clúster

Para añadir un nuevo shard se siguen los mismos pasos que se hicieron para añadir shards al clúster original:

- Se crea un directorio nuevo para el nuevo shard (figura 166):

**Figura 166.** Creación de directorio para el shard.

| Este equipo > Boot (C:) > data > shards > |                      |                     |        |
|-------------------------------------------|----------------------|---------------------|--------|
| Nombre                                    | Fecha de modifica... | Tipo                | Tamaño |
| n1                                        | 25/04/2015 21:54     | Carpeta de archivos |        |
| n2                                        | 25/04/2015 21:54     | Carpeta de archivos |        |
| <input checked="" type="checkbox"/> n3    | 25/04/2015 22:23     | Carpeta de archivos |        |

- Para crear un shard se abre una instancia de mongod indicando el puerto donde se va a ejecutar con la opción –port, se indica el directorio donde se almacenan los datos con la opción –dbpath, y se indica la opción --shardsvr (figura 167).

**Figura 167.** Instancia de mongod para el nuevo shard.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27029 --dbpath /data/shard
/n3 --shardsvr
2015-04-25T22:24:48.820+0200 I JOURNAL [initandlisten] journal dir=/data/shards
/n3\journal
2015-04-25T22:24:48.833+0200 I JOURNAL [initandlisten] recover : no journal fil
es present, no recovery needed
2015-04-25T22:24:49.595+0200 I JOURNAL [initandlisten] preallocateIsFaster=true
13.7
2015-04-25T22:24:49.828+0200 I JOURNAL [durability] Durability thread started
2015-04-25T22:24:49.833+0200 I JOURNAL [journal writer] Journal writer thread s
tarted
```

- A continuación se agrega el shard al clúster (figura 168). Para ello, se inicia una instancia de mongo en el puerto donde se está ejecutando el enrutador, y el shard se añade con el comando sh.addShard («Nombre del equipo: puerto») indicando el equipo y puerto donde se ejecuta el shard.

**Figura 168.** Agregación del nuevo shard al clúster.

```
mongos> sh.addShard("localhost:27029")
{
  "shardAdded" : "shard0002",
  "ok" : 1
}
```

- Se puede consultar el estado del clúster mediante el método sh.status () (figura 169).

**Figura 169.** Consulta sobre el estado del clúster.

```

mongos> sh.addShard("localhost:27029")
{
  "shardAdded" : "shard0002", "ok" : 1
}
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
    "id" : 1,
    "minCompatibleVersion" : 5,
    "currentVersion" : 6,
    "clusterId" : ObjectId("553bd8c9dabfed166e263fea")
}
  shards: [
    {
      "_id" : "shard0000", "host" : "localhost:22826"
    },
    {
      "_id" : "shard0001", "host" : "localhost:22827"
    },
    {
      "_id" : "shard0002", "host" : "localhost:27029"
    }
  ]
  balancer:
    Currently enabled: yes
    Currently running: yes
    Balancer lock taken at Sat Apr 25 2015 22:27:26 GMT+0200 (Hora de verano romana) by Antonio:27021:1429983430:41:Balancer:41
    Failed balancer rounds in last 5 attempts: 0
    Migration Results for the last 24 hours:
      2 : Success
      1 : Failed with error 'migration already in progress', from shard0000 to shard0001
  databases:
    {
      "_id" : "admin", "partitioned" : false, "primary" : "config"
    },
    {
      "_id" : "testDB", "partitioned" : true, "primary" : "shard0000"
    }
      testDB.prueba
        shard key: { "_id" : "hashed" }
        chunks:
          shard0000 1
          shard0001 2
          shard0002 1
        < "_id" : { "$minKey" : 1 } > -->> < "_id" : NumberLong("-4611686018427307902") > on : shard0002 Timestamp(3, 0)
        < "_id" : NumberLong("-4611686018427307902") > -->> < "_id" : NumberLong(0) > on : shard0000 Timestamp(3, 1)
        < "_id" : NumberLong(0) > -->> < "_id" : NumberLong("4611686018427307902") > on : shard0001 Timestamp(2, 4)
        < "_id" : NumberLong("4611686018427307902") > -->> < "_id" : { "$maxKey" : 1 } > on : shard0001 Timestamp(2, 5)
    }
}

```

También es posible consultar cada uno de los shards para ver cuántos documentos hay en cada shard (figuras 170, 171 y 172):

**Figura 170.** Consulta sobre el número de documentos en el shard 1.

```

C:\Program Files\MongoDB\Server\3.0\bin>mongo --port 27029
MongoDB shell version: 3.0.2
connecting to: 127.0.0.1:27029/test
> use testDB
switched to db testDB
> db.prueba.count()
25041

```

**Figura 171.** Consulta sobre el número de documentos en el shard 2.

```

C:\Program Files\MongoDB\Server\3.0\bin>mongo --port 27026
MongoDB shell version: 3.0.2
connecting to: 127.0.0.1:27026/test
> use testDB
switched to db testDB
> db.prueba.count()
25069

```

**Figura 172.** Consulta sobre el número de documentos en el shard 3.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongo --port 27027
MongoDB shell version: 3.0.2
connecting to: 127.0.0.1:27027/test
> use testDB
switched to db testDB
> db.prueba.count()
49891
```

Otra información que se puede extraer es desde el terminal donde se ejecuta la instancia del enrutador. Se puede acceder a la base de datos admin y consultar los shards que actualmente están registrados usando el comando listShards (figura 173):

**Figura 173.** Listado de los shards.

```
mongos> use admin
switched to db admin
mongos> db.runCommand( { listShards : 1 } )
{
  "shards" : [
    {
      "_id" : "shard0000",
      "host" : "localhost:27026"
    },
    {
      "_id" : "shard0001",
      "host" : "localhost:27027"
    },
    {
      "_id" : "shard0002",
      "host" : "localhost:27029"
    }
  ],
  "ok" : 1
}
```

Por último, se puede conocer qué bases de datos se encuentran distribuidas desde el terminal donde se ejecuta la instancia del enrutador accediendo a la base de datos config y consultando sobre la colección databases el campo «partitioned» (figura 174).

**Figura 174.** Consulta de las bases de datos distribuidas.

```
mongos> use config
switched to db config
mongos> db.databases.find( { "partitioned": true } )
{ "_id" : "testDB", "partitioned" : true, "primary" : "shard0000" }
```

## 7. Eliminación de un shard de un clúster

Para eliminar un shard hay que conectarse al terminal en el que se ejecuta la instancia del enrutador y se siguen los pasos siguientes:

- Asegurarse de que el balanceador está activo (figura 175).

**Figura 175.** Consulta del estado del balanceador.

```
mongos> sh.getBalancerState()
true
```

- Determinar el nombre del shard que se va a eliminar mediante el método adminCommand sobre el campo «listShards» (figura 176).

**Figura 176.** Determinación del nombre del shard

```
mongos> db.adminCommand( { listShards: 1 } )
{
  "shards" : [
    {
      "_id" : "shard0000",
      "host" : "localhost:27026"
    },
    {
      "_id" : "shard0001",
      "host" : "localhost:27027"
    },
    {
      "_id" : "shard0002",
      "host" : "localhost:27029"
    }
  ],
  "ok" : 1
}
```

- Eliminar los trozos del shard mediante el método removeShard invocado desde la base de datos admin (figura 177). Alternativamente, se podría usar el comando adminCommand si se ejecuta desde otra base de datos.

**Figura 177.** Eliminar trozos del shard.

```
mongos> use admin
switched to db admin
mongos> db.runCommand({ removeShard: "localhost:27029" })
{
    "msg" : "draining started successfully",
    "state" : "started",
    "shard" : "shard0002",
    "ok" : 1
}
```

Durante el proceso de eliminación, los trozos que contiene ese shard y las bases de datos que tienen como primario al shard son migrados a otros shards.

- Se puede comprobar el estado de la eliminación ejecutando sucesivamente el mismo método (figura 178). El proceso finaliza cuando el campo «state» toma el valor «completed».

**Figura 178.** Comprobación del estado de la eliminación.

```
mongos> db.runCommand({ removeShard: "localhost:27029" })
{
    "msg" : "removeshard completed successfully",
    "state" : "completed",
    "shard" : "shard0002",
    "ok" : 1
}
```

Además, se consulta el número de shards (figura 179).

**Figura 179.** Consulta del número de shards.

```
mongos> db.adminCommand({ listShards: 1 })
{
    "shards" : [
        {
            "_id" : "shard0000",
            "host" : "localhost:27026"
        },
        {
            "_id" : "shard0001",
            "host" : "localhost:27027"
        }
    ],
    "ok" : 1
}
```

En este punto se pueden eliminar los archivos que almacenaban, dado que ya han sido migrados a otros shards.

- En un clúster, una base de datos con colecciones no distribuidas almacena esas colecciones solo en un único shard. Ese shard se convierte en el shard primario para esa base de datos (diferentes bases de datos pueden tener diferentes shards primarios), es por ello que un shard puede ser primario para una o más bases de datos en el clúster. Cuando se quiere eliminar un shard que es primario, se tienen que seguir los siguientes pasos:

**1)** Determinar si el shard que se quiere eliminar es primario para alguna base de datos, para lo que se usa el comando sh.status (). En el documento (figura 180) que se genera, el campo «bases de datos» lista cada base de datos y su shard primario. En el siguiente ejemplo se puede ver que la base de datos «testDB» tiene como primario «shard0000».

**Figura 180.** Documento generado con sh.status () .

```
databases:
  {
    "_id": "admin", "partitioned": false, "primary": "config" }
  {
    "_id": "testDB", "partitioned": true, "primary": "shard0000" }
```

**2)** Para mover la base de datos a otro shard, se usa el comando movePrimary (figura 181). Por ejemplo, si se migra la base de datos «testDB» del «shard0000» al «shard0001», se ejecuta el comando:

**Figura 181.** Movimiento de la base de datos a otro shard.

```
mongos> db.runCommand({movePrimary: "testDB", to: "shard0001"})
{
  "primary" : "shard0001:localhost:27027",
  "ok" : 1 }
```

Este comando no retorna nada hasta que el sistema haya completado el movimiento de todos los datos, lo cual puede llevar cierto tiempo.

## 8. Estado de un clúster

El método sh.status () muestra un informe formateado de la configuración del sharding y la información respecto a los trozos existentes en un sharded clúster. El comportamiento por defecto es suprimido de los detalles de la información de los trozos si el número total de trozos es mayor o igual que 20. Las principales secciones del documento que genera son:

- La sección sharding version (figura 182) muestra información sobre la base de datos configura:

**Figura 182.** Sección sharding version.

```
sharding version: {
  "_id": <número>,
  "minCompatibleVersion": <número>,
  "currentVersion": <número>,
  "clusterId": <Identificador de objeto>
}
```

La sección shards lista información sobre los shards (figura 183). Para cada shard, la sección muestra el nombre, el host y las etiquetas asociadas si existe alguna.

**Figura 183.** Sección sharding version.

```

shards:
  {"_id": <Nombre shard 1>,
   "host": <cadena>,
   "tags": [<cadena>...]
  }
  {"_id": <Nombre shard 2>,
   "host": <cadena>,
   "tags": [<cadena>...]
  }
  ...

```

- La sección balancer (figura 184) lista información acerca del estado del balanceador. Esto da una idea de cómo está funcionando el balanceador.

**Figura 184.** Sección balancer.

```

balancer:
  Currently enabled: yes
  Currently running: yes
    Balancer lock taken at Wed Dec 10 2014 12:00:16 GMT+1100 (AEDT) by
      Pixl.local:27017:1418172757:16807: Balancer: 282475249
  Collections with active migrations:
    test.t2 started at Wed Dec 10 2014 11:54:51 GMT+1100 (AEDT)
  Failed balancer rounds in last 5 attempts: 1
  Last reported error: tag ranges not valid for: test.t2
  Time of Reported error: Wed Dec 10 2014 12:00:33 GMT+1100 (AEDT)
  Migration Results for the last 24 hours:
    96: Success
    15: Failed with error 'ns not found, should be impossible', from
        shard01 to shard02

```

- La sección bases de datos (figura 185) lista información sobre las bases de datos. Para cada base de datos, la sección muestra el nombre si la base de datos ha permitido sharding, y el shard primario para la base de datos.

**Figura 185.** Sección bases de datos.

```

databases:
  {“_id”: <Nombre base de datos 1>,
   “partitioned”: <Booleano>,
   “primary”: <Cadena>
  }
  {“_id”: < Nombre base de datos 2>,
   “partitioned”: <Booleano>,
   “primary”: <Cadena>
  }
  ...
  ...

```

- La sección sharded (figura 186) proporciona información sobre los detalles del sharding para las colecciones distribuidas. Para cada colección distribuida, la sección muestra la clave de sharding, el número de trozos por cada shard, la distribución de documentos a través de los trozos y la información de etiquetas si existe alguna para los rangos de la clave de sharding.

**Figura 186.** Sección sharded.

```

<nombre_base de datos>.<colección>
shard key: {<shard key>: <1 or hashed>}
chunks:
  <shard nombre1> <número de trozos>
  <shard nombre2> <número de trozos>
  ...
  {<shard key>: <min rango1>} --> {<shard key>: <max rango1>} on: <Nombre shard>
  <last modified timestamp>
  {<shard key>: <min rango2>} --> {<shard key>: <max rango2>} on: <Nombre shard>
  <last modified timestamp>
  ...
tag: <tag1> {<shard key>: <min rango1>} --> {<shard key>: <max rango1>}
  ...

```

La definición detallada de cada campo se puede consultar en: <<http://docs.mongodb.org/manual/reference/method/sh.status>>.

## 9. Conversión de un replica set en un sharded clúster replicado

Se va a convertir un replica set formado por tres miembros a un sharded clúster con dos shards. Cada shard es un replica set con tres miembros independientes. El procedimiento es el siguiente:

- Crear un replica set de tres miembros e insertar los datos dentro de una colección. Para los elementos del replica set se han creado directorios en /data/repl.

1) Se crea el primer nodo del replica set (figura 187).

**Figura 187.** Creación del primer nodo del replica set

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27017 --dbpath /data/repl/
n1 --replSet rs0
2015-04-26T12:59:11.254+0200 I JOURNAL [initandlisten] journal dir=/data/repl/n
1\journal
2015-04-26T12:59:11.265+0200 I JOURNAL [initandlisten] recover : no journal fil
es present, no recovery needed
2015-04-26T12:59:11.554+0200 I JOURNAL [durability] Durability thread started
2015-04-26T12:59:11.562+0200 I JOURNAL [journal writer] Journal writer thread s
tarted
2015-04-26T12:59:11.565+0200 I JOURNAL [durability] Durability thread joined
2015-04-26T12:59:11.566+0200 I JOURNAL [journal writer] Journal writer thread jo
ined
```

2) Se conecta a mongo a través del puerto 27017, y se inicia el replica set con rs.initiate (figura 188).

**Figura 188.** Iniciación del replica set.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongo --port 27017
MongoDB shell version: 3.0.2
connecting to: 127.0.0.1:27017/test
> rs.initiate()
{
  "info2" : "no configuration explicitly specified -- making one",
  "me" : "Antonio:27017",
  "ok" : 1
}
rs0:OTHER>
```

- 3) Se abren dos nuevas instancias de mongod para los restantes nodos (figuras 189 y 190).

**Figura 189.** Apertura de nueva instancia de mongod para un nuevo nodo.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27018 --dbpath /data/repl/n2 --replSet rs0
2015-04-26T13:12:17.751+0200 I JOURNAL [initandlisten] journal dir=/data/repl/n2\journal
2015-04-26T13:12:17.762+0200 I JOURNAL [initandlisten] recover : no journal files present, no recovery needed
2015-04-26T13:12:17.939+0200 I JOURNAL [durability] Durability thread started
2015-04-26T13:12:17.943+0200 I JOURNAL [journal writer] Journal writer thread started
```

**Figura 190.** Apertura de nueva instancia de mongod para un nuevo nodo.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27019 --dbpath /data/repl/n3 --replSet rs0
2015-04-26T13:13:03.082+0200 I JOURNAL [initandlisten] journal dir=/data/repl/n3\journal
2015-04-26T13:13:03.094+0200 I JOURNAL [initandlisten] recover : no journal files present, no recovery needed
2015-04-26T13:13:03.298+0200 I JOURNAL [durability] Durability thread started
2015-04-26T13:13:03.304+0200 I JOURNAL [journal writer] Journal writer thread started
```

- 4) A continuación se añaden al replica set desde el terminal donde se ejecuta mongo (figura 191).

**Figura 191.** Adición de los nuevos nodos al replica set.

```
rs0:OTHER> rs.add("Antonio:27018")
{
  "ok" : 1
}
rs0:PRIMARY> rs.add("Antonio:27019")
{
  "ok" : 1
}
rs0:PRIMARY>
```

- 5) Se crea una base de datos y se rellena con datos (figura 192).

**Figura 192.** Creación de una base de datos.

```
rs0:PRIMARY> use testDB
switched to db testDB
rs0:PRIMARY> var productos=["mesas","sillas"]
rs0:PRIMARY> var colores=['azul','naranjo','negro','rosa','rojo','blanco','anarillo']
rs0:PRIMARY> var fechafabricacion=new Date()
rs0:PRIMARY> fechafabricacion.setYear(2000)
956742924050
rs0:PRIMARY> for (var i=0;i<999999;i++) {
... db.ensayo.insert({producto: productos[Math.floor(Math.random()*productos.length)],
... color:colores[Math.floor(Math.random()*colores.length)],
... fechafabricacion:fechafabricacion}});
... writeResult({ "nInserted" : 1 })
rs0:PRIMARY> rs.status()
```

- Empezar los servidores de configuración y el enrutador. Para los servidores de configuración se han creado directorios en /data/configura:

**1)** Se activa cada servidor de configuración (figuras 193, 194 y 195). Para que funcione en una única máquina, hay que darlo de alta en la misma IP de la máquina en la que se ha dado de alta el replica set:

**Figura 193.** Activación del servidor 1.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27022 --dbpath /data/config/n1 --configsvr --bind_ip 192.168.0.10
2015-04-26T14:16:44.137+0200 I JOURNAL [initandlisten] journal dir=/data/config/n1\journal
2015-04-26T14:16:44.148+0200 I JOURNAL [initandlisten] recover : no journal files present, no recovery needed
2015-04-26T14:16:44.225+0200 I JOURNAL [durability] Durability thread started
2015-04-26T14:16:44.228+0200 I JOURNAL [journal writer] Journal writer thread started
```

**Figura 194.** Activación del servidor 2.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27023 --dbpath /data/config/n2 --configsvr --bind_ip 192.168.0.10
2015-04-26T14:17:23.941+0200 I JOURNAL [initandlisten] journal dir=/data/config/n2\journal
2015-04-26T14:17:23.953+0200 I JOURNAL [initandlisten] recover : no journal files present, no recovery needed
2015-04-26T14:17:24.088+0200 I JOURNAL [durability] Durability thread started
2015-04-26T14:17:24.091+0200 I JOURNAL [journal writer] Journal writer thread started
```

**Figura 195.** Activación del servidor 3.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27024 --dbpath /data/config/n3 --configsvr --bind_ip 192.168.0.10
2015-04-26T14:17:51.481+0200 I JOURNAL [initandlisten] journal dir=/data/config/n3\journal
2015-04-26T14:17:51.494+0200 I JOURNAL [initandlisten] recover : no journal files present, no recovery needed
2015-04-26T14:17:51.615+0200 I JOURNAL [durability] Durability thread started
2015-04-26T14:17:51.620+0200 I JOURNAL [journal writer] Journal writer thread started
```

- 2)** Se activa el enrutador en la misma dirección IP (figura 196):

**Figura 196.** Activación del enrutador.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongos --configdb 192.168.0.10:27022,192
.168.0.10:27023,192.168.0.10:27024 --port 27021 --bind_ip 192.168.0.10
2015-04-26T14:18:38.528+0200 I SHARDING [mngosMain] MongoS version 3.0.2 starti
ng: pid=4136 port=27021 64-bit host=Antonio (<--help for usage)
2015-04-26T14:18:38.535+0200 I CONTROL [mngosMain] db version v3.0.2
2015-04-26T14:18:38.536+0200 I CONTROL [mngosMain] git version: 6201872043echb
:8a4cc169b5482dcf385fc464f
```

- Añadir el replica set inicial como un shard (figura 197). Se ejecuta una instancia de mongo en la IP y puerto donde se ejecuta el enrutador para dar de alta el replica set como un shard del clúster.

**Figura 197.** Adición del replica set como un shard.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongo --port 27021 --host 192.168.0.10
MongoDB shell version: 3.0.2
connecting to: 192.168.0.10:27021/test
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
    "_id" : 1,
    "minCompatibleVersion" : 5,
    "currentVersion" : 6,
    "clusterId" : ObjectId("553cd7a4cc5fccf447384a68")
}
shards:
  balancer:
    Currently enabled: yes
    Currently running: no
    Failed balancer rounds in last 5 attempts: 0
    Migration Results for the last 24 hours:
      No recent migrations
  databases:
    < "_id" : "admin", "partitioned" : false, "primary" : "config" >
mongos> sh.addShard("rs0/Antonio:27017, Antonio:27018, Antonio:27019")
{
  "ok" : 0,
  "errmsg" : "in seed list rs0/Antonio:27017, Antonio:27018, Antonio:27019
, host Antonio:27018 does not belong to replica set rs0"
}
mongos> sh.addShard("rs0/Antonio:27017,Antonio:27018,Antonio:27019")
< "shardAdded" : "rs0", "ok" : 1 >
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
    "_id" : 1,
    "minCompatibleVersion" : 5,
    "currentVersion" : 6,
    "clusterId" : ObjectId("553cd7a4cc5fccf447384a68")
}
shards:
  < "_id" : "rs0", "host" : "rs0/Antonio:27017,Antonio:27018,Antonio:27019" >
  balancer:
    Currently enabled: yes
    Currently running: no
    Failed balancer rounds in last 5 attempts: 0
    Migration Results for the last 24 hours:
      No recent migrations
  databases:
    < "_id" : "admin", "partitioned" : false, "primary" : "config" >
    < "_id" : "testDB", "partitioned" : false, "primary" : "rs0" >
```

- Crear un segundo shard y añadirlo al clúster. Se crea otro replica set de tres miembros. Para los elementos del replica set se han creado directorios en /data/repl:

1) Se crea el primer nodo del replica set (figura 198).

**Figura 198.** Creación del primer nodo del replica set.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27013 --dbpath /data/repl/r1
2015-04-26T14:45:29.402+0200 I JOURNAL [initandlisten] journal dir=/data/repl/r1\journal
2015-04-26T14:45:29.414+0200 I JOURNAL [initandlisten] recover : no journal files present, no recovery needed
2015-04-26T14:45:29.618+0200 I JOURNAL [durability] Durability thread started
2015-04-26T14:45:29.627+0200 I JOURNAL [journal writer] Journal writer thread started
```

2) Se conecta a mongo a través del puerto 27017, y se inicia el replica set con rs.initiate (figura 199).

**Figura 199.** Inicialización del replica set.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongo --port 27013
MongoDB shell version: 3.0.2
connecting to: 127.0.0.1:27013/test
> rs.initiate()
{
    "info2" : "no configuration explicitly specified -- making one",
    "ae" : "Antonio:27013",
    "ok" : 1
}
rs1:OTHER>
```

3) Se abren dos nuevas instancias de mongod para los restantes nodos (figuras 200 y 201).

**Figura 200.** Instancia de mongod para nodo nuevo.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27014 --dbpath /data/repl/r5
2015-04-26T14:48:18.457+0200 I JOURNAL [initandlisten] journal dir=/data/repl/r5\journal
2015-04-26T14:48:18.469+0200 I JOURNAL [initandlisten] recover : no journal files present, no recovery needed
2015-04-26T14:48:18.624+0200 I JOURNAL [durability] Durability thread started
2015-04-26T14:48:18.627+0200 I JOURNAL [journal writer] Journal writer thread started
```

**Figura 201.** Instancia de mongod para nodo nuevo.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongod --port 27015 --dbpath /data/repl/rs1
2015-04-26T14:49:21.348+0200 I JOURNAL [initandlisten] journal dir=/data/repl/rs1\journal
2015-04-26T14:49:21.368+0200 I JOURNAL [initandlisten] recover : no journal file present, no recovery needed
2015-04-26T14:49:21.612+0200 I JOURNAL [durability] Durability thread started
2015-04-26T14:49:21.617+0200 I JOURNAL [journal writer] Journal writer thread started
```

- 4) A continuación se añaden al replica set desde el terminal donde se ejecuta mongo (figura 202).

**Figura 202.** Adición de los nuevos nodos al replica set.

```
rs1:OTHER> rs.add("Antonio:27014")
{
  "ok" : 1
}
rs1:PRIMARY> rs.add("Antonio:27015")
{
  "ok" : 1
}
rs1:PRIMARY>
```

- 5) Añadir el replica set inicial como un shard (figura 203).

**Figura 203.** Adición del replica set como un shard.

```
mongos> sh.addShard("rs1/Antonio:27013,Antonio:27014,Antonio:27015")
{
  "shardAdded" : "rs1",
  "ok" : 1
}
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
    "id" : 1,
    "minCompatibleVersion" : 5,
    "currentVersion" : 6,
    "clusterId" : ObjectId("553cd2a4cc5fccc447384a68")
  }
  shards: [
    {
      "_id" : "rs0",
      "host" : "rs0/Antonio:27017,Antonio:27018,Antonio:27019"
    },
    {
      "_id" : "rs1",
      "host" : "rs1/Antonio:27013,Antonio:27014,Antonio:27015"
    }
  ]
  balancer:
    Currently enabled: yes
    Currently running: no
    Failed balancer rounds in last 5 attempts: 0
    Migration Results for the last 24 hours:
      No recent migrations
  databases:
    {
      "_id" : "admin",
      "partitioned" : false,
      "primary" : "config"
    },
    {
      "_id" : "testDB",
      "partitioned" : false,
      "primary" : "rs0"
    }
```

- Distribuir la colección deseada.

- 1)** Desde el terminal de la Shell donde se ejecuta la instancia del enrutador se selecciona la base de datos que se va a distribuir, y se invoca el método sh.enablesharding sobre la misma (figura 204).

**Figura 204.** Invocación del método sh.enablesharding

```
mongos> use testDB
switched to db testDB
mongos> sh.enableSharding("testDB")
{
  "ok" : 1
}
mongos> _
```

- 2)** Como la colección que se va a distribuir no es vacía, antes de ello se crea un índice sobre el campo que se va a usar como clave de distribución (figura 205).

**Figura 205.** Creación de un índice sobre la clave de distribución.

```
mongos> use testDB
switched to db testDB
mongos> db.ensayo.createIndex({"_id":1})
{
  "rav" : {
    "rs0/Antonio:27017,Antonio:27018,Antonio:27019" : {
      "createdCollectionAutomatically" : false,
      "nunIndexesBefore" : 1,
      "nunIndexesAfter" : 1,
      "note" : "all indexes already exist",
      "ok" : 1,
      "$gleStats" : {
        "lastOpTime" : Timestamp(0, 0),
        "electionId" : ObjectId("553cc5f1636ae765207f913"
      }
    }
  },
  "ok" : 1
}
```

- 3)** A continuación se indica la colección de la base de datos que va a ser distribuida (figura 206). Para ello, se usa el método sh.shardCollection (), que toma como parámetros la colección especificada como base\_datos.colección y la clave de distribución. La clave de distribución puede ser la especificación de un índice simple o compuesto de la forma {campo1:1,...,

campo n: 1} o bien una clave hasheada que se especifica como {campo:"hashed"}.

**Figura 206.** Indicación de la colección que debe ser distribuida.

```
mongos> sh.shardCollection("testDB.ensayo", {"_id":1})
{ "collectionsharded" : "testDB.ensayo", "ok" : 1 }
```

- 4) El sistema habrá distribuido los documentos y puede consultarse su distribución mediante el método getShardDistribution () aplicado sobre la colección.

## Bibliografía

- Banker, K.** (2011). *MongoDB in action*. Manning Publications Co.
- Chodorow, K.** (2013). *MongoDB: the definitive guide*. O'Reilly Media, Inc.
- Hows, D.; Membrey, P.; Plugge, E.** (2014). *MongoDB Basics*. Apress.
- Membrey, P.; Plugge, E.; Hawkins, D.** (2010). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress.
- Sitio oficial MongoDB. «Sharding Introduction». <<https://docs.mongodb.org/v2.6/core/sharding-introduction/>>



## Capítulo IX **Optimización**

### 1. Introducción

En este capítulo se hace una introducción a los mecanismos de optimización de consultas que posee MongoDB. De forma similar a las bases de datos relacionales, en MongoDB se dispone de dos herramientas para optimizar el rendimiento de una consulta:

- MongoDB Profiler.
- Explain().

La principal diferencia entre ambas herramientas es que MongoDB Profiler permite encontrar aquellas consultas que no tienen un buen rendimiento y seleccionar las consultas candidatas para ser analizadas, mientras que explain() permite investigar una única consulta para determinar lo bien que funciona.

En las secciones del capítulo se profundizará sobre ambas herramientas y los conceptos relacionados.

## 2. MongoDB Profiler

Almacena información estadística y detalles del plan de ejecución para cada consulta que encaja con los criterios de activación. Algunas de sus características son las siguientes:

- Se puede habilitar para cada base de datos de forma separada.
- Una vez habilitada, se inserta un documento con información acerca del rendimiento y sobre detalles de ejecución para cada consulta enviada en una colección denominada «`system.profiler`» de tipo capped-collection (es una colección de tamaño fijo que, una vez llega a su mayor tamaño, entonces sobrescribe la información).
- La colección se puede consultar usando los comandos normales.

Para activar MongoDB Profiler basta conectarse a Mongo y teclear (figura 207):

**Figura 207.** Activación de MongoDB Profiler.

```
> db.setProfilingLevel(2)
{ "was" : 0, "slowms" : 100, "ok" : 1 }
```

Y para desactivarlo (figura 208):

**Figura 208.** Desactivación de MongoDB Profiler.

```
> db.setProfilingLevel(0)
{ "was" : 2, "slowms" : 100, "ok" : 1 }
```

Hay que observar que:

- En el nivel de perfil 2 se controlan todas las consultas.

- En el nivel de perfil 0 no se controla ninguna consulta.

También se puede configurar para usarlo solo con consultas que excedan un tiempo de ejecución especificado. En el siguiente ejemplo solo se guarda información de consultas que superan más de la mitad de un segundo (figura 209):

**Figura 209.** Configuración para salvar consultas por su duración

```
> db.setProfilingLevel(1,500)
{ "was" : 0, "slowms" : 100, "ok" : 1 }
```

En el nivel de perfil 1 se puede proporcionar un valor máximo de tiempo de ejecución de una consulta en milisegundos, de forma que si la consulta supera esa cantidad de tiempo, se guarda la información de ejecución de la misma y, en caso contrario, es ignorada. Se puede consultar el nivel de perfil actual (figura 210).

**Figura 210.** Consulta del nivel de perfil actual.

```
> db.getProfilingLevel()
0
```

Y de la misma forma se puede consultar el límite actual para realizar el seguimiento (figura 211):

**Figura 211.** Consulta del límite actual para el seguimiento.

```
> db.getProfilingStatus()
{ "was" : 0, "slowms" : 500 }
```

Cuando se establece un perfil, el sistema devuelve un documento con tres campos:

- «was»: indica el nivel de perfil previo.
- «slowms»: indica la cantidad de tiempo que debe superar una operación para que se realice un seguimiento sobre esta.
- «ok»: indica que la operación se ha realizado con éxito.

Se puede configurar un nivel de perfil para toda una instancia de mongo realizando una llamada con los parámetros profile y slowns:

```
mongod --profile=1 --slowms=15
```

A continuación en la tabla 2 se muestra un ejemplo de documento de la colección system.profile:

|                                                                                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                                                             |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>{   "ts": ISODate ("2012-12-10T19:31:28.977Z"),   "op": "update",   "ns": "social.users",   "query": {     "name": "j.r."   },   "updateobj": {     "\$set": {       "likes": [         "basketball",         "trekking"       ]     }   },   "nscanned": 8,   "scanAndOrder": true,   "moved": true,   "nmoved": 1, }</pre> | <pre>"nupdated": 1, "keyUpdates": 0, "numYield": 0, "lockStats": {   "timeLockedMicros": {     "r": NumberLong (0),     "w": NumberLong (258)   },   "timeAcquiringMicros": {     "r": NumberLong (0),     "w": NumberLong (7)   } }, "millis": 0, "client": "127.0.0.1", "user": ""}</pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Tabla 2. Ejemplo de documento de la colección system.profile.

Para cada tipo de operación, MongoDB Profiler rellena información del siguiente tipo:

| Información                              | Significado                                                                                                                                    |
|------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>system.profile.ts</code>           | Cuándo se ejecutó la operación.                                                                                                                |
| <code>system.profile.op</code>           | El tipo de operación: insert, query, update, remove, getmore, command.                                                                         |
| <code>system.profile.ns</code>           | Nombre de la operación en el formato: <u>base</u><br><u>datos.colección</u> .                                                                  |
| <code>system.profile.query</code>        | Documento de consulta usado.                                                                                                                   |
| <code>system.profile.command</code>      | Operación de comando.                                                                                                                          |
| <code>system.profile.updateobj</code>    | El documento de actualización usado durante la operación.                                                                                      |
| <code>system.profile.cursorid</code>     | El id del cursor accedido durante una operación de tipo getmore.                                                                               |
| <code>system.profile.ntoreturn</code>    | Número de documentos que se retornan como resultado de la operación.                                                                           |
| <code>system.profile.ntoskip</code>      | Número de documentos especificados en el método skip().                                                                                        |
| <code>system.profile.nscanned</code>     | Número de documentos escaneados en el índice para llevar a cabo la operación.                                                                  |
| <code>system.profile.scanAndOrder</code> | Si toma el valor de «true», indica si el sistema no puede usar el orden de los documentos en el índice para retornar los resultados ordenados. |
| <code>system.profile.moved</code>        | Aparece con el valor de true en caso de que la operación de actualización haya movido uno o más documentos a una nueva localización en disco.  |
| <code>system.profile.nmoved</code>       | Número de documentos que la operación movió en el disco                                                                                        |
| <code>system.profile.nupdated</code>     | Número de documentos actualizados en la operación.                                                                                             |
| <code>system.profile.keyUpdates</code>   | Número de claves del índice que la actualización cambió en la operación.                                                                       |
| <code>system.profile.numYield</code>     | El número de veces que la operación cedió su turno a otras operaciones para que se completen antes que ella.                                   |

| Información                                               | Significado                                                                                                                                                                                                                                                                     |
|-----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>system.profile.lockStats</code>                     | Tiempo en microsegundos que la operación espera a adquirir y mantener un bloqueo. Puede retornar: R(bloqueo global de lectura), W(Bloqueo global de escritura), r(bloqueo de lectura de una base de datos específica) y w(bloqueo de escritura de una base de datos específica) |
| <code>system.profile.lockStats.timeLockedMicros</code>    | Tiempo en microsegundos que la operación mantiene un bloqueo específico.                                                                                                                                                                                                        |
| <code>system.profile.lockStats.timeAcquiringMicros</code> | Tiempo en microsegundos que la operación espera para adquirir un bloqueo específico.                                                                                                                                                                                            |
| <code>system.profile.nreturned</code>                     | Número de documentos retornados por la operación.                                                                                                                                                                                                                               |
| <code>system.profile.responseLength</code>                | Longitud en bytes del documento resultante de la operación.                                                                                                                                                                                                                     |
| <code>system.profile.millis</code>                        | Tiempo en milisegundos desde el comienzo de la operación hasta la finalización.                                                                                                                                                                                                 |
| <code>system.profile.client</code>                        | IP de la conexión cliente que origina la operación.                                                                                                                                                                                                                             |
| <code>system.profile.user</code>                          | Usuario autenticado que ejecuta la operación.                                                                                                                                                                                                                                   |

Algunos ejemplos de operaciones típicas sobre la colección:

- Retornar los 10 logs más recientes:  
`db.system.profile.find().limit (10).sort ({ts: -1}).pretty ()`
- Retornar todas las operaciones excepto las operaciones command:  
`db.system.profile.find ({op: {$ne: "command"} }).pretty ()`
- Retornar las operaciones para una colección particular:  
`db.system.profile.find ({ns: "mydb.test"} ). pretty ()`
- Retornar operaciones más lentas que 5 milisegundos:  
`db.system.profile.find ({millis :{ $gt: 5} }).pretty ()`
- Retornar información de un cierto rango de tiempo:  
`db.system.profile.find (`

```
{  
  ts: {  
    $gt: new ISODate  
    ("2012-12-09T03:00:00Z"),  
    $lt: new ISODate  
    ("2012-12-09T03:40:00Z")  
  }  
}.pretty()
```

- Mostrar las cinco operaciones más recientes que tardaron al menos 1 milisegundo:  
show profile

Por último, cuando se quiere cambiar el tamaño de la colección system.profile se debe:

- Deshabilitar el perfil:  
db.setProfilingLevel (0)
- Borrar la colección system.profile:  
db.system.profile.drop ()
- Crear una nueva colección system.profile:  
db.createCollection ("system.profile", {capped: true, size: 4000000})
- Habilitar nuevamente el perfil:  
db.setProfilingLevel (1)

### 3. Explain()

Retorna información sobre el plan de ejecución de las operaciones: aggregate (), count (), find (), group (), remove () y update ().

Para usar explain () se usa la estructura:

```
db.colección.explain ()<método (...)>
```

Por ejemplo:

```
db.productos.explain ()remove ({“categoría”: “frutas”}, {“unidades”: true})
```

El método tiene un parámetro opcional denominado «verbosity» que determina la cantidad de información que se retorna. Puede tomar los valores:

**1) queryPlanner.** Es el valor por defecto.

- Se ejecuta el optimizador de consultas para elegir el plan de ejecución.
- Se retorna el plan de ejecución ganador.

**2) executionStats**

- Se ejecuta el optimizador de consultas para elegir el plan de ejecución, ejecuta el plan de ejecución ganador, y retorna las estadísticas de la ejecución y el plan de ejecución ganador.
- En operaciones de escritura, retorna información sobre las operaciones de actualización o borrado que serían realizadas, pero no las aplica a la base de datos.
- No proporciona información sobre los planes rechazados.

### 3) allPlansExecution

- Ejecuta el optimizador de consultas para elegir el plan ganador, y a continuación lo ejecuta.
- Retorna estadísticas tanto sobre el plan de ejecución ganador como del resto de los planes considerados por el optimizador (información parcial), así como el plan de ejecución ganador.
- En operaciones de escritura, retorna información sobre las operaciones de actualización o borrado que serían realizadas, pero no las aplica a la base de datos.

Hay que observar que db.collection.explain () .find () permite encadenar cualquier modificador a continuación de find ():

```
db.productos.explain ("executionStats").find (  
    {cantidad: {$gt: 50}, categoría: "frutas"}  
).sort ({cantidad: -1}).hint ({categoría: 1, cantidad:-1})
```

## 4. El valor queryPlanner

Es el plan de ejecución que ha seleccionado el optimizador de consultas como el que aparece en la tabla 3:

```
{  
  "queryPlanner": {  
    "plannerVersion": <int>,  
    "namespace": <string>,  
    "indexFilterSet": <boolean>,  
    "parsedQuery": {  
      ...  
    },  
    "winningPlan": {  
      "stage": <STAGE1>,  
      ...  
      "inputStage": {  
        "stage": <STAGE2>,  
        ...  
        "inputStage": {  
          ...  
        }  
      }  
    },  
    "rejectedPlans": [  
      <Plan candidato 1>,  
      ...  
    ]  
  }  
}
```

Tabla 3. Plan de ejecución seleccionado por el optimizador.

Algunas características:

- La información que se devuelve en el plan de ejecución se muestra como un árbol de etapas, de manera que en cada etapa pasa sus resultados (documentos o claves de índice) al nodo padre.
- Los nodos hoja acceden a la colección o a los índices.
- Los nodos internos manipulan los documentos o las claves del índice que resultan de los nodos hijo.

- El nodo raíz es la etapa final desde la que se deriva el conjunto resultado.

En cada etapa se describe la operación:

- COLLSCAN para un recorrido de la colección.
- IXSCAN para un recorrido de las claves de los índices
- FETCH para recuperar documentos.
- SHARD\_MERGE para mezclar resultados de shards.

Los elementos que aparecen son:

| Información                                              | Significado                                                                                                             |
|----------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| <code>explain.queryPlanner.namespace</code>              | Indica sobre qué colección se ha realizado la consulta en el formato: <code>base_datos.colección</code> .               |
| <code>explain.queryPlanner.indexFilterSet</code>         | Es un booleano que especifica si se aplica un filtro de índice.                                                         |
| <code>explain.queryPlanner.winningPlan</code>            | Es un documento que detalla el plan de ejecución que ha sido seleccionado. Se presenta como un árbol de etapas o fases. |
| <code>explain.queryPlanner.winningPlan.stage</code>      | Cadena que representa el nombre de una etapa. Cada etapa contiene información específica.                               |
| <code>explain.queryPlanner.winningPlan.inputStage</code> | Un documento que describe una etapa hijo, que proporciona los documentos o claves de índices a su padre.                |
| <code>explain.queryPlanner.winningPlan.inutStages</code> | Un array de documentos describiendo las etapas hijo. Proporciona los documentos o claves de índices a su padre.         |
| <code>explain.queryPlanner.rejectedPlans</code>          | Array de planes candidatos considerados y rechazados por el optimizador                                                 |

## 5. El valor executionStats

Contiene información estadística de la ejecución completa del plan ganador como la que aparece en la tabla 4. En caso de operaciones de escritura, mostraría las modificaciones que llevaría a cabo, pero sin aplicarlas sobre la base de datos.

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>"executionStats": {   "executionSuccess": &lt;boolean&gt;,   "nReturned": &lt;int&gt;,   "executionTimeMillis": &lt;int&gt;,   "totalKeysExamined": &lt;int&gt;,   "totalDocsExamined": &lt;int&gt;,   "executionStages": {     "stage": &lt;STAGE1&gt;     "nReturned": &lt;int&gt;,     "executionTimeMillisEstimate" : &lt;int&gt;,     "works" : &lt;int&gt;,     "advanced" : &lt;int&gt;,     "needTime": &lt;int&gt;,     "needYield": &lt;int&gt;,     "isEOF": &lt;boolean&gt;,     ...   } }</pre> | <pre>"inputStage": {   "stage" : &lt;STAGE2&gt;,   ...   "nReturned" : 0,   "executionTimeMillisEstimate" : &lt;int&gt;,   ...   "inputStage": {     ...   } }, "allPlansExecution": [   {&lt;partial executionStats1&gt; },   {&lt;partial executionStats2&gt; },   ... ]</pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Tabla 4. Información estadística.

Los elementos que aparecen son:

| Información                                       | Significado                                                                                                        |
|---------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| <u>explain.executionStats.nReturned</u>           | Número de documentos que encajan con la condición de consulta.                                                     |
| <u>explain.executionStats.executionTimeMillis</u> | Tiempo total de milisegundos que se requiere para la ejecución de la consulta de acuerdo con el plan seleccionado. |
| <u>explain.executionStats.totalKeysExamined</u>   | Número de entradas del índice escaneadas.                                                                          |

| Información                                           | Significado                                                                 |
|-------------------------------------------------------|-----------------------------------------------------------------------------|
| <code>explain.executionStats.totalDocsExamined</code> | Número de documentos escaneados.                                            |
| <code>explain.executionStats.executionStages</code>   | Detalles de la ejecución completa del plan ganador como un árbol de etapas. |
| <code>explain.executionStats.allPlansExecution</code> | Contiene información parcial de la ejecución del plan ganador.              |

## 6. Server Info

Es información que retorna de la instancia de Mongo donde se ejecuta:

```
“serverInfo”: {
    “host”: <cadena>,
    “port”: <entero>,
    “version”: <cadena>,
    “gitVersion”: <cadena>
}
```

En el caso de colecciones distribuidas, se retorna información acerca de cada shard al que se ha accedido (tabla 5).

|                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                    |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>{   "queryPlanner": {     ...     "winningPlan": {       ...       "shards": [         {           "shardName": &lt;shard&gt;,           &lt;Información del planificador de consultas para el shard&gt;           &lt;serverInfo para el shard&gt;         },         ...       ],     },   } }</pre> | <pre>"executionStats": {   ...   "executionStages": {     ...     "shards": [       {         "shardName": &lt;Nombre&gt;,         &lt;Estadísticas de ejecución para el shard&gt;       },       ...     ]),     "allPlansExecution": [       {         "shardName": &lt;Nombre&gt;,         "allPlans": [...]       },       ...     ]}} }</pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Tabla 5 Información acerca de cada shard.

Incluye la siguiente información:

| Información                                                          | Significado                                                                          |
|----------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| <u><a href="#">explain.queryPlanner.winningPlan.shards</a></u>       | Array de documentos que contiene queryPlanner y serverInfo para cada shard accedido. |
| <u><a href="#">explain.executionStats.executionStages.shards</a></u> | Array de documentos que contiene executionStats para cada shard accedido.            |

## 7. El optimizador de consultas

Este componente procesa las consultas y elige el plan de ejecución más eficiente para una consulta teniendo en cuenta los índices existentes. Entonces, el sistema de consulta usa ese plan cada vez que ejecuta la consulta. Solo almacena los planes para aquellas consultas que tienen más de un plan viable.

Ocasionalmente, reevalúa los planes de ejecución cuando el contenido de la colección cambia para asegurar la optimalidad de los planes de ejecución. También es posible especificar qué índices debe considerar el optimizador mediante los «Index Filters».

Para crear un nuevo plan de ejecución:

- Se ejecuta la consulta contra varios índices candidatos en paralelo.
- Se registran las coincidencias en un buffer de resultados común:
  - Si los planes candidatos incluyen solo planes de consulta ordenados, existe un único buffer de resultados común.
  - Si los planes candidatos incluyen solo planes de consulta desordenados, existe un único buffer de resultados común.
  - Si los planes de consulta incluyen tanto planes de consulta ordenados como desordenados, existen dos buffers de resultados comunes, uno para los planes ordenados y otro para los planes no ordenados.
- Si un índice retorna un resultado ya retornado por otro índice, el optimizador salta la coincidencia duplicada. En el caso de dos buffers, ambos buffers son saltados.
- Se para el testeo de los planes candidatos y se selecciona un índice cuando uno de los siguientes eventos ocurre:
  - Un plan de consulta no ordenado ha retornado todos los resultados que se buscaban.
  - Un plan de consulta ordenado ha retornado todos los resultados que se buscaban.
  - Un plan de consulta ordenado ha retornado un tamaño umbral de resultados que se buscaban. Por defecto son 101.

El índice seleccionado se convierte en el índice especificado en el plan de ejecución, de manera que en futuras iteraciones de esta consulta o consultas con el mismo patrón de consulta se usará ese índice. El patrón de consulta se refiere a las condiciones de consulta que difieren solo en los valores tal como:

```
db.inventario.find ({tipo: "comida"})
db.inventario.find ({“tipo”:"limpieza"})
```

Cuando las colecciones cambian, el optimizador borra el plan de ejecución y reevalúa después de se produce alguno de los siguientes eventos:

- La colección recibe más de 1.000 operaciones de escritura.
- El reIndex reconstruye el índice.
- Se añade o se borra un índice.
- Se reinicia el proceso mongod

## 8. Index Filter

Determina qué índice va a evaluar el optimizador para una consulta dada.

Se caracteriza por:

- Una consulta consiste en una combinación de ordenaciones, proyecciones y condiciones de búsqueda. Si existe un index filter para una consulta dada, el optimizador solo considera aquellos índices especificados en el filtro.

- Cuando un index filter existe para una consulta, se ignora hint () (sirve para forzar el uso de índice dado). Para saber si se aplica un index filter a una consulta, se puede consultar el campo «indexFilterSet» en el documento devuelto por explain () .
- Index filters solo afecta a aquellos índices que el optimizador evalúa, pero el optimizador podría seleccionar un escaneo de la colección como plan ganador para una consulta dada.
- Los index filter existen durante el procesamiento en el servidor pero no persisten después. Además, es posible eliminarlos manualmente.
- Hay que tener cuidado con esta herramienta, pues sobrescribe el comportamiento del optimizador y del método hint () .

## 9. Observaciones

- El operador \$explain () está deprecado y sustituido por el nuevo método explain () . Se usaba de la siguiente manera:

```
db.coleccion.find()._addSpecial("$explain", 1)  
db.coleccion.find({$query: {}, $explain: 1})  
db.coleccion.find().explain()
```

- Se van a mostrar algunos de los cambios entre ambas versiones:

**1)** Cuando el planificador selecciona un recorrido de una colección en la información del explain se incluye una fase de COLLSCAN. En la versión anterior aparece el término «BasicCursor».

**2)** Si el planificador selecciona un índice, el explain incluye una fase de IXSCAN incluyendo información como dirección, patrón de índice de clave, etc. En la versión anterior aparece el término BtreeCursor <index name>[<direction>].

**3)** Cuando un índice cubre una consulta, no es necesario examinar los documentos de la colección para retornar los resultados. Además, en la información del explain existe una etapa IXSCAN que no tiene un descendiente de una etapa FETCH y en la información de executionStats el campo «totalDocsExamined» toma el valor de 0. En las versiones anteriores, retorna el campo «indexOnly» para indicar si el índice cubre la consulta.

**4)** Para un plan de intersección de índices, el resultado incluirá o bien una etapa de AND\_SORTED o una etapa de AND\_HAS con un array de inputStages que detallan los índices. En las versiones anteriores, se retorna el campo cursor con el valor del Complex Plan para las intersecciones de índices.

**5)** Si se usan índices para una \$or expresión, el resultado incluirá la etapa OR con un array de inputStages que detallan los índices. En las versiones previas se retorna el array clauses que detalla los índices.

```
{  
  "stage": "OR",  
  "inputStages": [  
    {  
      "stage": "IXSCAN",  
      ...  
    },  
    {  
      ...  
    }  
  ]  
}
```

```
    "stage": "IXSCAN",
    ...
},
...
]
}
```

- 6)** Si se usa un recorrido de índice para obtener la solicitud ordenada, el resultado no incluirá una etapa SORT. En caso contrario, si no se puede usar un índice para ordenar, aparecerá la etapa SORT. En las versiones previas se retornaba el campo «scanAndOrder» para especificar si se usaba el orden del índice para retornar ordenados los resultados.

## 10. Índice que cubre una consulta

Se dice que un índice cubre una consulta cuando se cumple lo siguiente:

- Todos los campos en la consulta son parte del índice.
- Todos los campos retornados están en el mismo índice.

Por ejemplo, considérese la colección inventory sobre la que se ha definido un índice sobre sus campos Type e item:

```
db.inventario.createIndex ({type: 1, ítem: 1})
```

El índice cubrirá la siguiente operación, que hace consultas sobre los campos Type e item, y retorna valores del campo «item»:

```
db.inventario.find (   
    {tipo: "comida", ítem: /^c/},  
    {ítem: 1, _id:0}  
)
```

Hay que observar que en los resultados retornados se debe explícitamente indicar «\_id» a cero para excluirlo, pues el índice no incluye el campo «\_id».

La consulta de un índice es mucho más rápida que consultar los documentos, pues los índices son más pequeños que los documentos que indexan y se encuentran en la memoria RAM o localizados secuencialmente en el disco. Un índice no cubre una consulta si:

- Alguno de los campos indexados en alguno de los documentos en la colección incluye un array. Si un campo indexado es un array, el índice se convierte en un índice multiclave y no puede soportar cubrir una consulta.
- Alguno de los campos indexados en el predicado de la consulta o retornados en la proyección son campos de documentos embebidos. Por ejemplo, considérese una colección de usuarios con documentos de la siguiente manera:

```
{_id: 1, usuario: {login: "prueba"}}
```

Por ejemplo, considérese la colección que tiene el siguiente índice:

```
{"usuario.login": 1}
```

Este índice no cubre la consulta, sin embargo, la consulta usa ese índice para recuperar los documentos.

```
db.usuarios.find ({“usuario.login”:”prueba”}, {“usuario.login”:1, id: 0})
```

Un índice no puede cubrir una consulta sobre un entorno distribuido si el índice no contiene la clave de distribución con la excepción para el índice «`_id`». En este caso, si una consulta sobre una colección distribuida solo especifica una condición sobre el campo «`_id`», el índice «`_id`» puede cubrir la consulta.

## 11. Selectividad de las consultas

Se refiere a lo bien que el predicado excluye o filtra documentos de la colección. Este factor es importante, pues puede determinar si una consulta puede usar índices efectivamente o incluso usar índices totalmente.

Las consultas más selectivas encajan un porcentaje menor de documentos. Por ejemplo, una coincidencia por igualdad sobre el campo único es muy selectiva, pues, como mucho, puede encajar un único documento.

Las consultas menos selectivas encajan un porcentaje más grande de documentos, y no pueden usar índices efectivamente o en su totalidad.

Por ejemplo, los operadores `$nin` y `$ne` no son muy selectivos, puesto que a menudo coinciden con una amplia proporción de un índice. Como resultado, en muchos casos, una consulta con `$nin` o `$ne` no resulta mejor realizarla con un índice que realizarla escaneando todos los documentos de la colección.

La selectividad de las expresiones regulares depende de las expresiones en sí mismas.

## 12. Ejercicios propuestos

a) Ejecuta el siguiente bucle:

**Figura 212.** Bucle

```
> for ($i=1; $i <100;$i++) { db.prueba.insert({"_id":ObjectId(), "producto":"mesas", "num-a":$i,"num-b":$i*2,"num-c":$i*3}); db.prueba.insert({"_id":ObjectId(), "producto":"sillas", "num-a":$i*3, "num-b":$i, "num-c":$i*2});}
```

Realizar las siguientes operaciones:

- 1) Encontrar el número de productos que cumplen que «\$num-a» es mayor que «\$num-b».
- 2) Con respecto a la consulta anterior, obtener el plan de ejecución que ha usado el optimizador de consultas.
  - ¿Cómo ha llevado a cabo la operación?
  - ¿Cuántos documentos ha tenido que consultar?
  - ¿Cuánto tarda en realizar la consulta?
  - Obtén las estadísticas de ejecución de la consulta.

b) Ejecuta el siguiente código en MongoDB:

**Figura 213.** Código ejemplo 1.

```
> var productos=["mesas","sillas"]
> var colores=["azul","marron","negro","rosa","rojo","blanco","amarillo"]
> var fechafabricacion=new Date()
> fechafabricacion.setYear(2000)
55295304738
```

**Figura 214.** Código ejemplo 2.

```
> for (var i=0;i<90000;++i){ db.ensayo.insert({ producto: productos[Math.floor(Math.random()*productos.length)], color:colores[Math.floor(Math.random()*colores.length)], fechafabricacion:fechafabricacion})}
```

Realizar las siguientes operaciones:

- 1)** Muestra el plan de ejecución de la consulta que recupera 30 resultados de mesas de color rojo ordenadas por fecha de fabricación. ¿Qué puedes decir de la ejecución en cuanto a tiempo empleado y cómo ha llevado a cabo la operación?
- 2)** Busca una estrategia para mejorar el tiempo de ejecución y el número de elementos que son necesarios revisar.

## Bibliografía

- Banker, K.** (2011). *MongoDB in action*. Manning Publications Co.
- Chodorow, K.** (2013). *MongoDB: the definitive guide*. O'Reilly Media, Inc.
- Hows, D.; Membrey, P.; Plugge, E.** (2014). *MongoDB Basics*. Apress.
- Membrey, P.; Plugge, E.; Hawkins, D.** (2010). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress.
- Sitio oficial MongoDB, «Optimization Strategies for MongoDB». [<https://docs.mongodb.org/v2.6/administration/optimization/>](https://docs.mongodb.org/v2.6/administration/optimization/)

## Capítulo X

# GridFS

### 1. GridFS

Es un protocolo para almacenar y recuperar archivos que exceden el tamaño máximo de documentos de 16 MB. En vez de almacenar un archivo en un único documento, se divide el archivo en partes y se almacena cada una como un archivo separado. Por defecto, cada trozo está limitado a 255 k.

Se usan dos colecciones para almacenar los archivos, una para almacenar los archivos de trozos y otro para almacenar archivos de metadatos asociados.

Cuanto se realiza una consulta a un almacén GridFS por un archivo, se reensamblan los trozos necesarios. Se pueden realizar consultas sobre los archivos almacenados, así como acceder a información de secciones arbitrarias de los archivos.

Asimismo con GridFS es posible almacenar archivos para los que se tenga acceso sin tener que cargar el archivo entero en memoria.

Las ventajas de usar GridFS son las siguientes:

- Ofrece una arquitectura simple para gestionar en un único sistema todos los tipos de datos, tanto documentos como archivos nativos.
- Los documentos de metadatos pueden ser expresados usando una estructura de documento flexible y rica, y se pueden recuperar recuperar los documentos usando el lenguaje de consultas de MongoDB.

- Se puede conseguir alta disponibilidad (con replica sets) y escalabilidad (con sharded clúster) tanto para archivos binarios como para archivos estructurados.
- No existen restricciones sobre los sistemas de archivos con respecto al número de documentos por directorio o reglas de nombrado de archivos.
- Ofrece un modelo de seguridad consistente para autenticación y acceso autorizado a los archivos y metadatos.

## 2. Mongofile

Para almacenar y recuperar archivos usando GridFS se usa o bien un driver de MongoDB o la herramienta de línea de comandos mongofiles. Para invocar mongofiles se sigue la estructura:

mongofiles opciones comandos nombre\_archivo

Donde:

- Opciones: controlan el comportamiento de mongofiles.
- Comandos: determinan la acción que realizar por mongofiles.
- Se especifica un nombre de archivo o sistema de archivos local o un objeto GridFS.

Hay que observar:

- Cuando se usan replica sets, mongofiles solo puede leer del nodo primario del conjunto.

- Cuando se conecta una instancia del servidor que tiene activada la autorización (opción `--auth`), deben usarse las opciones `--username` y `--password`. Además, el usuario debe poseer los roles «`read`» para acceder a la base de datos usando `list`, `search` o `get`, y el role «`readWrite`» para poder acceder a la base de datos usando los comandos `put` o `delete`.

Las principales opciones de mongofiles son:

- `db`: especifica el nombre de la base de datos en la que se ejecuta mongofiles.
- `local`: especifica el nombre en el sistema de archivos local del archivo que es usado en las operaciones `get` y `put`.
- `replace`: reemplaza los objetos GridFS por el archivo local especificado en vez de añadir un objeto adicional con el mismo nombre.
- `host`: especifica el nombre de la máquina en la que se ejecuta el servidor.
- `port`: especifica el puerto en el que se ejecuta el servidor.
- `username`: especifica el nombre de usuario con el que se identifica en un servidor con autorización activada.
- `password`: especifica el password asociado al usuario con el que se identifica en un servidor con autorización activada.

Los principales comandos de mongofiles son:

- `list <prefijo>`: lista los archivos del almacén GridFS. En caso de especificar un prefijo, limita la lista de ítems retornados a los archivos que empiezan con el prefijo dado.
- `search <string>`: lista los archivos del almacén GridFS con los nombres que encajan con la cadena dada.

- put <nombre\_archivo>: copia el archivo especificado del sistema local de archivos dentro del almacen GridFS. Este nombre será el nombre del objeto en GridFS y se asume que es el mismo que tiene en el sistema de archivos local. Si fueran nombres diferentes, se usa la opción --local.
- get <nombre\_archivo>: copia el archivo especificado del almacen GridFS al sistema de archivos local. Este nombre será el nombre del objeto en GridFS y se asume que es el mismo que tiene en el sistema de archivos local. Si fueran nombres diferentes, se usa la opción --local.
- delete <nombre\_archivo>: borra el archivo especificado del almacen GridFS.

### 3. Colecciones GridFS

Los archivos en GridFS se almacenan en dos colecciones:

- La colección fs.chunks almacena los trozos binarios en que se divide un fichero que se almacena en GridFS.

```
{ "id" : <ObjectId>,
  "files_id": <ObjectId>,
  "n": <número>,
  "data": <binario>}
```

Los campos que contiene son:

- «\_id»: identificador único del chunk.

- «files\_id»: el «\_id» del documento padre que aparece en la colección files.
- «n»: número de secuencia del chunk. Empieza desde 0.
- «data»: almacenamiento del chunk como un archivo binario BSON.
- La colección fs.files almacena los metadatos de los archivos.

Cada documento representa a un archivo del almacen GridFS con la siguiente estructura:

```
{“_id”: <ObjectId>,
  “length”: <número>,
  “chunkSize”: <número>,
  “uploadDate”: <fecha>,
  “md5”: <hash>,
  “filename”: <nombre>,
  “contentType”: <tipo de contenido>,
  “aliases”: <array de cadenas>,
  “metadata”: <Objeto de datos>,
}
```

Los campos que contiene son:

- «\_id»: identificador único del documento.
- «length»: tamaño del documento en bytes.
- «chunkSize»: tamaño de cada trozo, que es usado para dividir el documento en trozos. Por defecto, son 255 k.
- «uploadDate»: fecha de almacenamiento en GridFS.
- «md5»: un hash md5.
- «filename»: nombre del documento.
- «contentType»: tipo mime del documento.

- «aliases»: un array de alias.
- «metadata»: información adicional que se quiere almacenar.

Hay que observar:

- Cuando se ejecuta el comando `put` más de una vez sobre un mismo archivo, se generan nuevos documentos que son idénticos (metadatos y contenidos), salvo por el «`_id`». Aunque por el comportamiento podría pensarse que MongoDB actualizará el documento original, no lo hace y crea uno nuevo, pues no tiene la seguridad de que sean exactamente iguales. Si se quiere actualizar un documento, hay que hacerlo usando su «`_id`».
- El tamaño del archivo que se almacena es una información útil por varios motivos. En primer lugar, GridFS descompone un archivo en trozos (por defecto 255 k cada trozo, aunque puede ser modificado este valor), por lo que para saber cuántos trozos se van a crear es necesario conocer tanto el tamaño del trozo como el tamaño total del archivo. En segundo lugar, para buscar un dato en una posición determinada es necesario saber en qué trozo estará.
- La colección `files` es una colección normal, de forma que se pueden añadir nuevas claves y valores que sean necesarios. Así, cada documento puede necesitar unos metadatos diferentes.

## 4. Índices GridFS

GridFS utiliza un índice único y compuesto sobre la colección `chunks` para los campos «`files_id`» y «`n`», que contienen el «`_id`»

de un trozo del documento padre y el número de la secuencia entre los trozos en que se ha dividido el documento. La numeración de todos los trozos comienza con 0. Este índice permite una recuperación eficiente de los trozos de un archivo en particular ordenados desde 0 hasta n usando los valores de «files\_id» y «n», lo que permite ensamblarlos en él para generar el documento original. En caso de que no se haya creado automáticamente el índice, se puede crear con el comando:

```
db.fs.chunks.createIndex ({files_id: 1, n: 1}, {unique: true});
```

## 5. Carga de un documento en GridFS

Para ello nos situamos en el directorio bin de la instalación de MongoDB, y ejecutamos desde la línea de comandos el programa mongofiles con el comando put y el archivo que deseamos guardar. En el siguiente ejemplo (figura 215) se va almacenar un documento de Word denominado «prueba.docx» que se encuentra en c: /.

**Figura 215.** Almacenamiento de un documento Word.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongofiles put c:/prueba.docx
2015-04-27T09:22:45.753+0200      connected to: localhost
added file: c:/prueba.docx
```

A continuación se puede listar el contenido que tiene GridFS mediante el comando list (figura 216).

**Figura 216.** Listado del contenido de GridFS

```
C:\Program Files\MongoDB\Server\3.0\bin>mongofiles list
2015-04-27T09:26:02.379+0200      connected to: localhost
c:/prueba.docx  673416
```

Para poder ver el contenido del archivo cargado, se abre un terminal de mongo, y se puede consultar tanto la colección fs.files como la colección fs.chunks como si fueran colecciones normales (figura 217).

**Figura 217.** Consulta de la colección fs.files.

```
> db.fs.files.find()
{ "_id" : ObjectId("553dfa4cb99cc50e58000001"), "chunkSize" : 261120, "uploadDate" : ISODate("2015-04-27T08:58:53.013Z"), "length" : 673416, "md5" : "50deb160da5e660a55ce2f51d346fdec", "filename" : "c:/prueba.docx" }
```

Para consultar la colección fs.chunks se seleccionan todos los campos menos el campo «data», ya que, al contener una gran cantidad de datos, guarda los datos en binario. En el ejemplo se ve que el documento se ha guardado en 3 trozos (figura 218).

**Figura 218.** Trozos en los que se ha guardado el documento

```
> db.fs.chunks.find(<>, { "_id": 1, "files_id": 1, "n": 1 })
{ "_id" : ObjectId("553dfbd1b99cc50964000001"), "files_id" : ObjectId("553dfbd1b99cc50964000002"), "n" : 0 }
{ "_id" : ObjectId("553dfbd1b99cc50964000003"), "files_id" : ObjectId("553dfbd1b99cc50964000001"), "n" : 1 }
{ "_id" : ObjectId("553dfbd1b99cc50964000004"), "files_id" : ObjectId("553dfbd1b99cc50964000001"), "n" : 2 }
```

Hay que observar es posible guardar un fichero en una base de datos determinada, indicando la opción -d y la base datos (figuras 219 y 220):

**Figura 219.** Almacenamiento en una base de datos determinada.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongofiles put -d test1
2015-04-27T11:27:45.304+0200      connected to: localhost
added file: c:/prueba.docx
```

**Figura 220.** Listado de los elementos de la base de datos.

```
> use testDB
switched to db testDB
> show collections
fs.chunks
fs.files
system.indexes
```

## 6. Búsqueda y recuperación de archivos

Se pueden buscar archivos en el almacen utilizando el comando search. Este comando puede ir acompañado de un prefijo que se usará para recuperar todos los archivos que comienzan por ese prefijo (figura 221).

**Figura 221.** Búsqueda con el comando search.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongofiles search pr
2015-04-27T11:16:41.535+0200      connected to: localhost
c:/prueba.docx  673416
```

También se pueden recuperar los archivos y almacenarlos en un directorio mediante el comando get indicando el directorio de destino (figura 222).

**Figura 222.** Recuperación de documentos con get.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongofiles get c:/prueba.docx
2015-04-27T11:19:54.527+0200      connected to: localhost
finished writing to: c:/prueba.docx
```

## 7. Eliminación de archivos

Se pueden borrar los archivos de un almacen con el comando delete. Borra todos los archivos que tengan el nombre que se le proporciona (figura 223).

**Figura 223.** Borrado de documentos.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongofiles delete c:/prueba.docx
2015-04-27T11:24:17.414+0200      connected to: localhost
successfully deleted all instances of 'c:/prueba.docx' from GridFS
```

## Bibliografía

- Banker, K.** (2011). *MongoDB in action*. Manning Publications Co.
- Chodorow, K.** (2013). *MongoDB: the definitive guide*. O'Reilly Media, Inc.
- Hows, D.; Membrey, P.; Plugge, E.** (2014). *MongoDB Basics*. Apress.
- Membrey, P.; Plugge, E.; Hawkins, D.** (2010). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress.
- Sitio oficial MongoDB, «GridFS». <<https://docs.mongodb.org/manual/core/gridfs/>>

## Capítulo XI

# Operaciones de administración

### 1. Introducción

En este capítulo se realiza un repaso de las principales operaciones de administración que se pueden realizar sobre MongoDB.

### 2. Realizar backups del servidor

Para realizar un backup de la base de datos se utiliza el comando mongodump, y para ello:

- Es necesario que esté ejecutándose el servidor sobre el que se va a realizar el backup (figura 224).

**Figura 224.** Ejecución del servidor mongod.



```
C:\mongodb\bin\mongod.exe --help for help and startup options
2015-04-24T10:21:10.506+0200 [initandlisten] MongoDB starting : pid=5988 port=27017 dbpath=\data\db\ 64-bit host=antonio
2015-04-24T10:21:10.595+0200 [initandlisten] targetMinOS: Windows 7/Windows Server 2008 R2
2015-04-24T10:21:10.595+0200 [initandlisten] db version v2.6.7
2015-04-24T10:21:10.595+0200 [initandlisten] git version: a7d59ad22c382de82e9cb9
2015-04-24T10:21:10.596+0200 [initandlisten] build info: windows sys.getwindowsversion(major=6, minor=1, build=2601, platform=2, service_pack='Service Pack 1'), BOOST_LIB_VERSION=1_49
2015-04-24T10:21:10.597+0200 [initandlisten] allocator: system
2015-04-24T10:21:10.597+0200 [initandlisten] options: { config: "C:\data\mongod.conf" }
2015-04-24T10:21:10.740+0200 [initandlisten] journal dir=\data\dh\journal
2015-04-24T10:21:10.744+0200 [initandlisten] recover : no journal files present, no recovery needed
2015-04-24T10:21:10.802+0200 [initandlisten]
2015-04-24T10:21:10.803+0200 [initandlisten] ** WARNING: mongod started without --replSet yet 1 documents are present in local.system.replset
2015-04-24T10:21:10.803+0200 [initandlisten] **          Restart with --replSet unless you are doing maintenance and no other clients are connected.
2015-04-24T10:21:10.803+0200 [initandlisten] **          The TTL collection monitor will not start because of this.
2015-04-24T10:21:10.804+0200 [initandlisten] **          For more info see http://dochub.mongodb.org/core/ttlcollections
```

- Se abre un terminal de comandos y se ejecuta el programa mongodump, que se encuentra en el directorio bin en el que está instalado MongoDB (figura 225).

**Figura 225.** Ejecución de mongodump.

```

C:\> Símbolo del sistema
C:\Users\asus\asus>cd ..
C:\Users> cd ..
C:\> cd mongodb
C:\mongodb>cd bin
C:\mongodb\bin>mongodump
connected to: 127.0.0.1
2015-04-22T11:08:00.35+0200 all dbs
2015-04-22T11:08:00.35+0200 DATABASE: test to dump\test
2015-04-22T11:08:00.35+0200 test.system.indexes to dump\test\system.indexes
bson
2015-04-22T11:08:00.35+0200 12 documents
2015-04-22T11:08:00.35+0200 test.users to dump\test\users.bson
2015-04-22T11:08:00.35+0200 1 documents
2015-04-22T11:08:00.35+0200 Metadata for test.users to dump\test\users.netad
2015-04-22T11:08:00.35+0200 test.count to dump\test\count.bson
2015-04-22T11:08:00.35+0200 1 documents
2015-04-22T11:08:00.35+0200 Metadata for test.count to dump\test\count.netad
2015-04-22T11:08:00.35+0200 atav.json
2015-04-22T11:08:00.35+0200 9.31+0200
2015-04-22T11:08:00.35+0200 test.media to dump\test\media.bson
2015-04-22T11:08:00.35+0200 8 documents
2015-04-22T11:08:00.35+0200 atav.json
2015-04-22T11:08:00.35+0200 Metadata for test.media to dump\test\media.netad
2015-04-22T11:08:00.35+0200 test.snap_reduce_result to dump\test\nap_reduce_r
2015-04-22T11:08:00.35+0200 5 documents
2015-04-22T11:08:00.35+0200 Metadata for test.snap_reduce_result to dump\test
2015-04-22T11:08:00.35+0200 snap_reduce_result.metadata.json
2015-04-22T11:08:00.35+0200 test.resultado to dump\test\resultado.bson
2015-04-22T11:08:00.35+0200 4 documents
2015-04-22T11:08:00.35+0200 Metadata for test.resultado to dump\test\resulta
2015-04-22T11:08:00.35+0200 atav.metadata.json
2015-04-22T11:08:00.35+0200 test.system.profile to dump\test\system.profile
2015-04-22T11:08:00.36+0200 2658 documents
2015-04-22T11:08:00.36+0200 Metadata for test.system.profile to dump\test\ny
2015-04-22T11:08:00.36+0200 test.prueba to dump\test\prueba.bson
2015-04-22T11:08:00.36+0200 198 documents
2015-04-22T11:08:00.36+0200 Metadata for test.prueba to dump\test\prueba.net
2015-04-22T11:08:00.36+0200 atava.json
2015-04-22T11:08:00.36+0200 0.023+0200
2015-04-22T11:08:00.36+0200 test.prueba2 to dump\test\prueba2.bson
2015-04-22T11:08:00.36+0200 3 documents
2015-04-22T11:08:00.36+0200 Metadata for test.prueba2 to dump\test\prueba2.n
2015-04-22T11:08:00.36+0200 test.prueba3 to dump\test\prueba3.bson
2015-04-22T11:08:00.36+0200 197 documents
2015-04-22T11:08:00.36+0200 Metadata for test.prueba3 to dump\test\prueba3.n
2015-04-22T11:08:00.36+0200 atava.metadata.json
2015-04-22T11:08:00.36+0200 test.salida to dump\test\salida.bson
2015-04-22T11:08:00.36+0200 198 documents
2015-04-22T11:08:00.36+0200 Metadata for test.salida to dump\test\salida.net
2015-04-22T11:08:00.36+0200 atava.json
2015-04-22T11:08:00.36+0200 test.ensayo to dump\test\ensayo.bson
2015-04-22T11:08:00.36+0200 980000 documents
2015-04-22T11:08:00.36+0200 Metadata for test.ensayo to dump\test\ensayo.net
2015-04-22T11:08:00.36+0200 atava.metadata.json
C:\mongodb\bin>

```

Como resultado se conecta a la base de datos local en el puerto por defecto y el sistema guarda los archivos asociados a cada base de datos y colección (backup) en un directorio que por defecto se localiza en ./dump/[nombre\_base\_datos]/.

Se guardan los datos en archivos .bson, que son el formato interno que MongoDB utiliza para almacenar los documentos (figura 226).

**Figura 226.** Documentos almacenados.

| Nombre      | Fecha de modificación | Tipo                | Tamaño |
|-------------|-----------------------|---------------------|--------|
| admin       | 20/03/2015 9:36       | Carpeta de archivos |        |
| base        | 23/04/2015 22:32      | Carpeta de archivos |        |
| blog        | 23/04/2015 22:32      | Carpeta de archivos |        |
| m101        | 20/03/2015 9:36       | Carpeta de archivos |        |
| prueba      | 23/04/2015 22:32      | Carpeta de archivos |        |
| school      | 23/04/2015 22:32      | Carpeta de archivos |        |
| students    | 23/04/2015 22:32      | Carpeta de archivos |        |
| test        | 23/04/2015 22:32      | Carpeta de archivos |        |
| prueba.json | 20/03/2015 16:53      | Archivo JSON        | 5 KB   |

Las principales opciones de configuración son:

- La opción `--out` permite redirigir la salida de mongodump a un directorio especificado (figura 227).

**Figura 227.** Redirección de la salida de mongodump.

```
C:\mongod\bin>mongodump --out C:/Nuevo
connected to: 127.0.0.1
2015-04-24T18:38:01.180+0200 all dbs
2015-04-24T18:38:01.249+0200 DATABASE: base      to      C:/Nuevo\base
2015-04-24T18:38:01.255+0200      base.system.indexes to C:/Nuevo\base\system.indexes.json
2015-04-24T18:38:01.258+0200          1 documents
2015-04-24T18:38:01.260+0200      base.zips to C:/Nuevo\base\zips.json
2015-04-24T18:38:01.560+0200          29353 documents
2015-04-24T18:38:01.562+0200      Metadata for base.zips to C:/Nuevo\base\zips.netadata.json
2015-04-24T18:38:01.568+0200 DATABASE: blog      to      C:/Nuevo\blog
2015-04-24T18:38:01.573+0200      blog.system.indexes to C:/Nuevo\blog\system.indexes.json
2015-04-24T18:38:01.576+0200          7 documents
2015-04-24T18:38:01.577+0200      blog.autores to C:/Nuevo\blog\autores.json
2015-04-24T18:38:01.584+0200          1 documents
2015-04-24T18:38:01.586+0200      Metadata for blog.autores to C:/Nuevo\blog\autores.metadata.json
2015-04-24T18:38:01.594+0200      blog.posts to C:/Nuevo\blog\posts.json
2015-04-24T18:38:01.600+0200          13 documents
2015-04-24T18:38:01.603+0200      Metadata for blog.posts to C:/Nuevo\blog\posts.netadata.json
2015-04-24T18:38:01.611+0200      blog.post to C:/Nuevo\blog\post.json
2015-04-24T18:38:01.617+0200          0 documents
2015-04-24T18:38:01.618+0200      Metadata for blog.post to C:/Nuevo\blog\post.netadata.json
```

- Es posible realizar un backup de una base de datos y una colección determinada usando las opciones `--db` y `--collection` seguidas por sus nombres respectivos (figura 228).

**Figura 228.** Backup de una base de datos determinada.

```
C:\mongod\bin>mongodump --db multimedia --collection media
connected to: 127.0.0.1
2015-04-24T18:56:47.772+0200 DATABASE: multimedia          to      dump\multimedia
2015-04-24T18:56:47.775+0200   multimedia.media to dump\multimedia\media.bson
2015-04-24T18:56:47.779+0200           8 documents
2015-04-24T18:56:47.780+0200   Metadata for multimedia.media to dump\multimedia
\media.metadata.json
```

- Es posible realizar un backup de un servidor remoto utilizando las opciones --host y --port para indicar el nombre del equipo y el puerto donde se ejecuta el servidor de MongoDB (figura 229).

**Figura 229.** Backup de un servidor remoto.

```
C:\mongod\bin>mongodump --host Antonio --port 27019 --db multimedia --collection media
connected to: Antonio:27019
2015-04-24T19:03:55.266+0200 DATABASE: multimedia          to      dump\multimedia
2015-04-24T19:03:55.276+0200   multimedia.media to dump\multimedia\media.bson
2015-04-24T19:03:55.281+0200           8 documents
2015-04-24T19:03:55.284+0200   Metadata for multimedia.media to dump\multimedia
\media.metadata.json
```

Las opciones de configuración de mongodump se pueden consultar con mongodump –help (figura 230):

**Figura 230.** Opciones de mongodump.

```
C:\mongod\bin>mongodump --help
Export MongoDB data to BSON files.

Options:
  --help                               produce help message
  -v [ --verbose ]                      be more verbose (include multiple times
   for more verbosity e.g. -vvvv)
  --quiet                             silence all non error diagnostic
   messages
  --version                           print the program's version and exit
  -h [ --host ] arg                    mongo host to connect to <(set
   name|IP:port sets)>
  -s [ --port ] arg                   server port. Can also use --host
  --port arg                          hostname:port
  --ipv6                             enable IPv6 support (disabled by
   default)
  -u [ --username ] arg               username
  -p [ --password ] arg               password
  --authenticationDatabase arg        authentication source (defaults to dbname)
  --authenticationMechanism arg (<MONGODB-CR>
   authentication mechanism
   Service name to use when authenticating
   using GSSAPI/Kerberos
   Remote host name to use for purpose of
   GSSAPI/Kerberos authentication
   directly access mongod database files
   in the given path, instead of
   connecting to a mongod server - needs
   to lock the data directory, so cannot
   be used if a mongod is currently
   accessing the same path
   each db is in a separate directory
   (relevant only if dbpath specified)
   enable journaling (relevant only if
   dbpath specified)
   database to use
   execute some (some commands)
   output directory or "" for stdout
   json query
   Use oplog for point-in-time
   snapshotting
   try to recover a crashed database
   force table scan (do not use
   snapshot)
   Dump user and role definitions for the
   given database
```

### 3. Restauración de una copia de seguridad

Para restaurar una copia de seguridad se utiliza el comando mongorestore, y para ello:

- Es necesario que esté ejecutándose el servidor sobre el que se va a realizar la restauración (figura 231).

**Figura 231.** Ejecución del servidor de mongod.



```
C:\mongodb\bin\mongod.exe --help for help and startup options
2015-04-24T10:21:10.506+0200 [initandlisten] MongoDB starting : pid=5988 port=27
217 dbpath=C:\data\db\ 64-bit host=antonio
2015-04-24T10:21:10.595+0200 [initandlisten] targetMinOS: Windows 7/Windows Serv
er 2008 R2
2015-04-24T10:21:10.595+0200 [initandlisten] db version v2.6.2
2015-04-24T10:21:10.595+0200 [initandlisten] git version: a7d57ad22c382de02e9cb9
1bf903a00fd9ac9899
2015-04-24T10:21:10.596+0200 [initandlisten] build info: windows sys.getwindowsu
version[major=6, minor=1, build=7601, platform=2, service_pack='Service Pack 1']
BOOST_LIB_VERSION=1_49
2015-04-24T10:21:10.597+0200 [initandlisten] allocator: system
2015-04-24T10:21:10.597+0200 [initandlisten] options: {}
2015-04-24T10:21:10.740+0200 [initandlisten] journal dir=C:\data\db\journal
2015-04-24T10:21:10.744+0200 [initandlisten] recover : no journal files present,
let's proceed
2015-04-24T10:21:10.802+0200 [initandlisten]
2015-04-24T10:21:10.803+0200 [initandlisten] ** WARNING: mongod started without
--replSet yet 1 documents are present in local.system.replset
2015-04-24T10:21:10.803+0200 [initandlisten] Restart with --replSet
2015-04-24T10:21:10.804+0200 [initandlisten] 10000 documents in database, 0 other clients are connected
2015-04-24T10:21:10.804+0200 [initandlisten] **
The TTL collection moni
tor will not start because of this.
2015-04-24T10:21:10.804+0200 [initandlisten] **
For more info see https://dochub.mongodb.org/core/ttlcollections
```

- Se abre un terminal de comandos y se ejecuta el programa mongorestore, que se encuentra en el directorio bin en el que está instalado MongoDB (figura 232). Si se utiliza directamente, los datos se añaden al final de cada colección, por lo que se producen duplicados. Para evitar esto, se puede usar la opción –drop, que descarta cada colección antes de restaurar, reemplazando así los datos que había por los de la copia de seguridad.

**Figura 232.** Ejecución de mongorestore.

```

C:\mongod\bin>mongorestore --drop
connected to: 127.0.0.1:22111
2015-04-22T11:37:46.285+0200    dump\test\count.bson
2015-04-22T11:37:46.287+0200      going into namespace [test.count]
2015-04-22T11:37:46.212+0200      dropping
1 objects found
2015-04-22T11:37:46.230+0200          Creating index: < key: { _id: 1 }, name: "_id_">
2015-04-22T11:37:46.240+0200      dump\test\ensayo.bson
2015-04-22T11:37:46.245+0200      going into namespace [test.ensayo]
2015-04-22T11:37:46.252+0200      dropping
2015-04-22T11:37:46.092+0200      Progress: 2299826/7720198   94%
2015-04-22T11:37:46.240+0200      Creating index: < key: { _id: 1 }, name: "_id_">
2015-04-22T11:37:46.222+0200      Creating index: < key: { producto: 1 }, name: "producto_1">
2015-04-22T11:37:49.028+0200      Creating index: < key: { producto: 1, color: 1 }, name: "producto_1_color_1_fechafabricacion_1", ns: "test.ensayo" >
2015-04-22T11:37:51.142+0200      dump\test\nmap_reduce_result.bson
2015-04-22T11:37:51.143+0200      going into namespace [test.map_reduce_result]
2015-04-22T11:37:51.144+0200      dropping
3 objects Found
2015-04-22T11:37:51.170+0200      Creating index: < key: { _id: 1 }, name: "_id_">
2015-04-22T11:37:51.172+0200      dump\test\media.bson
2015-04-22T11:37:51.172+0200      going into namespace [test.media]
2015-04-22T11:37:51.189+0200      dropping
0 objects Found
2015-04-22T11:37:51.202+0200      Creating index: < key: { _id: 1 }, name: "_id_">
2015-04-22T11:37:51.206+0200      dump\test\prueba1.bson
2015-04-22T11:37:51.207+0200      going into namespace [test.prueba1]
2015-04-22T11:37:51.225+0200      dropping
198 objects Found
2015-04-22T11:37:51.251+0200      Creating index: < key: { _id: 1 }, name: "_id_">
2015-04-22T11:37:51.254+0200      dump\test\prueba2.bson
2015-04-22T11:37:51.259+0200      going into namespace [test.prueba2]
2015-04-22T11:37:51.271+0200      dropping
1 objects Found
2015-04-22T11:37:51.283+0200      Creating index: < key: { _id: 1 }, name: "_id_">
2015-04-22T11:37:51.289+0200      dump\test\prueba3.bson
2015-04-22T11:37:51.293+0200      going into namespace [test.prueba3]
2015-04-22T11:37:51.305+0200      dropping
197 objects Found
2015-04-22T11:37:51.318+0200      Creating index: < key: { _id: 1 }, name: "_id_">
2015-04-22T11:37:51.326+0200      dump\test\resultado.bson
2015-04-22T11:37:51.328+0200      going into namespace [test.resultado]
2015-04-22T11:37:51.340+0200      dropping
4 objects Found
2015-04-22T11:37:51.352+0200      Creating index: < key: { _id: 1 }, name: "_id_">
2015-04-22T11:37:51.354+0200      dump\test\salida.bson
2015-04-22T11:37:51.359+0200      going into namespace [test.salida]
2015-04-22T11:37:51.370+0200      dropping
2015-04-22T11:37:51.372+0200      Creating index: < key: { _id: 1 }, name: "_id_">
2015-04-22T11:37:51.397+0200      dump\test\system.profile.bson
2015-04-22T11:37:51.400+0200      dropping system.profile.bson
2015-04-22T11:37:51.411+0200      dump\test\users.bson
2015-04-22T11:37:51.414+0200      going into namespace [test.users]
2015-04-22T11:37:51.417+0200      dropping
1 objects Found
2015-04-22T11:37:51.424+0200      Creating index: < key: { _id: 1 }, name: "_id_">
2015-04-22T11:37:51.425+0200      dump\test\system.profile.bson
2015-04-22T11:37:51.426+0200      dropping system.profile.bson
2015-04-22T11:37:51.427+0200      dump\test\users.bson
2015-04-22T11:37:51.428+0200      going into namespace [test.users]
2015-04-22T11:37:51.429+0200      dropping

```

Como resultado, busca por defecto los backups de bases de datos que se encuentren en el directorio ./dump/[nombre\_base\_datos]/.

Las principales opciones de configuración son:

- Para restaurar backups desde un directorio específico, basta indicar el path al directorio en el que se encuentra el backup (figura 233).

**Figura 233.** Restauración en un directorio específico.

```
C:\mongod\bin>mongorestore --drop C:/Nuevo
connected to: 127.0.0.1
2015-04-24T19:24:46.012+0200 C:/Nuevo\multimedia\media.json
2015-04-24T19:24:46.019+0200      going into namespace [multimedia.media]
2015-04-24T19:24:46.020+0200      dropping
0 objects found
2015-04-24T19:24:46.046+0200      Creating index: < key: { _id: 1 }, name: "_id_" ,
ns: "multimedia.media" >
```

- Es posible realizar la restauración de un backup de una base de datos y una colección determinada usando las opciones --db y --collection seguidas por sus nombres respectivos. Además, hay que indicar el path a los datos que son restaurados (figura 234).

**Figura 234.** Restauración de una base de datos determinada.

```
C:\mongod\bin>mongorestore --collection media --db multimedia dump/multimedia/media.json
connected to: 127.0.0.1
2015-04-24T19:38:13.270+0200 dump/multimedia/media.json
2015-04-24T19:38:13.277+0200      going into namespace [multimedia.media]
Restoring to multimedia.media without dropping. Restored data will be inserted without raising errors; check your server log
0 objects found
2015-04-24T19:38:13.315+0200      Creating index: < key: { _id: 1 }, name: "_id_" ,
ns: "multimedia.media" >
```

- Es posible realizar la restauración en un servidor remoto utilizando las opciones --host y --port para indicar el nombre del equipo y el puerto donde se ejecuta el servidor de MongoDB (figura 235).

**Figura 235.** Restauración desde un servidor remoto.

```
C:\mongod\bin>mongorestore --drop --host Antonio --port 27019 --collection media
--db multimedia dump/multimedia/media.json
connected to: Antonio:27019
2015-04-24T19:44:06.432+0200 dump/multimedia/media.json
2015-04-24T19:44:06.432+0200      going into namespace [multimedia.media]
2015-04-24T19:44:06.439+0200      dropping
0 objects found
2015-04-24T19:44:06.469+0200      Creating index: < key: { _id: 1 }, name: "_id_" ,
ns: "multimedia.media" >
```

Las opciones de configuración de mongorestore se pueden consultar con mongorestore --help (figura 236).

**Figura 236.** Opciones de mongorestore.

```
mongorestore --help
Report BSON files into MongoDB.

Usage: mongorestore [options] [directory or filename to restore from]

Options:
  -h host      produce help message
  -v level     be more verbose (include multiple times)
  --quiet      ignore all diagnostic
  --version    print the program's version and exit
  -h t <host>  arg  mongo host to connect to (<host>
  --port arg   port)  port. Can also use --host
  --ipv6      portname:port
  -u <username> arg  default)  user
  -p <password> arg  password
  -c <collection> arg  database source (defaults to dbname)
  --authenticationMechanism arg (<MONGODB-CR>
  --gssapiServiceName arg (<mongodb>  service name to use when authenticating
  --gssapiHostName arg  using GSSAPI/Kerberos
  --dbpath arg   path to use for purpose of
  --directoryperdb  GSSAPI/Kerberos authentication
  --journal      instead of connecting to a mongod, needs
  --db <db>  arg  to specify the database name
  --collection <collection> arg  needs to connect to a mongod
  --check       before inserting
  --filter arg  filter to apply before inserting
  --drop        drop collection before import
  --noIndexReplay include option entries before the
  --oplogLimit arg  oplog limit (number of documents to replay)
  --keepIndexVersion
  --noIndexReplay
  --noOplogReplay
  --restoreDbFileAndRoles
  -v arg <-8>

  --journal      enable journaling (relevant only if
  --dbpath is set)
  --collection  collection to use (some commands)
  --db            need to be inserted
  --default      before inserting
  --filter       filter to apply before inserting
  --drop         drop collection before import
  --noIndexReplay include option entries before the
  --oplogLimit arg  oplog limit (number of documents to replay)
  --keepIndexVersion
  --noIndexReplay
  --noOplogReplay
  --restoreDbFileAndRoles
  -v arg <-8>
```

## 4. Exportación de datos

Para exportar datos se utiliza el comando mongoexport, y para ello:

- Es necesario que esté ejecutándose el servidor sobre el que se va a realizar la exportación (figura 237).

**Figura 237.** Ejecución del servidor mongod.

```
C:\mongodb\bin\mongod.exe
C:\mongodb\bin\mongod.exe --help-for-help-and-startup-options
2015-04-24T10:21:10.540+0200 [initandlisten] MongoDB starting : pid=5988 port=27
2015-04-24T10:21:10.540+0200 [initandlisten] targetMinOS: Windows 7?Windows Serv
2015-04-24T10:21:10.540+0200 [initandlisten] host=antonio
2015-04-24T10:21:10.540+0200 [initandlisten] db version v2.6.2
2015-04-24T10:21:10.595+0200 [initandlisten] git version: a7d57ad2e302de82e9cb9
2015-04-24T10:21:10.596+0200 [initandlisten] build info: windows sys.getwindowsu
ersion major=6 minor=1 build=7601 platform=2 service_pack='Service Pack 1'
2015-04-24T10:21:10.597+0200 [initandlisten] allocator: system
2015-04-24T10:21:10.597+0200 [initandlisten] options: {}
2015-04-24T10:21:10.740+0200 [initandlisten] journal dir=data\db\journal
2015-04-24T10:21:10.744+0200 [initandlisten] recover : no journal files present,
no recovery needed
2015-04-24T10:21:10.802+0200 [initandlisten] ** WARNING: mongod started without
replicet yet 1 documents are present in local.system.replset
2015-04-24T10:21:10.803+0200 [initandlisten] **          Restart with --replSet
2015-04-24T10:21:10.804+0200 [initandlisten] **          and no other clients are connected.
2015-04-24T10:21:10.804+0200 [initandlisten] **          The TTL collection moni
tor will not start because of this
2015-04-24T10:21:10.804+0200 [initandlisten] **          For more info see http://
dochub.mongodb.org/core/tl-collections
```

- Se abre un terminal de comandos y se ejecuta el programa mongoexport, que se encuentra en el directorio bin en el que está instalado MongoDB (figura 238). Es necesario indicar la base de datos con --db, la colección con --collection y el archivo destino con --out.

**Figura 238.** Ejecución de mongoexport.

```
C:\mongodb\bin>mongoexport --db multimedia --collection media --out C:/Nuevo/media.json
connected to: 127.0.0.1
exported 8 records
```

Como resultado se exportan los datos a un formato de datos especificado que por defecto es json (figura 239).

**Figura 239.** Exportación de datos a formato json.

```
[{"_id": {"$oid": "5324f040e0d0ef6d9f4555c1"}, "tipo": "DVD", "Titulo": "Matzka", "estreno": 1999, "act": "Luis Ospina"}, {"_id": {"$oid": "5324f3150e0d0ef6d9f4555c2"}, "tipo": "DVD", "Titulo": "Blade Runner", "estreno": 1982, "act": "Harrison Ford"}, {"_id": {"$oid": "5324f3250e0d0ef6d9f4555c3"}, "tipo": "DVD", "Titulo": "Batman", "estreno": 1999, "act": "Michael Keaton"}, {"_id": {"$oid": "5324f3360e0d0ef6d9f4555c4"}, "tipo": "DVD", "Titulo": "Superman", "estreno": 1999, "act": "Christopher Reeve"}, {"_id": {"$oid": "532504d20e0d0ef6d9f4555c5"}, "tipo": "DVD", "Titulo": "Blade Runner", "estreno": 1982, "act": "Harrison Ford"}, {"_id": {"$oid": "532504de0e0d0ef6d9f4555c6"}, "tipo": "DVD", "Titulo": "Batman", "estreno": 1989, "act": "Michael Keaton"}, {"_id": {"$oid": "532504e90e0d0ef6d9f4555c7"}, "tipo": "DVD", "Titulo": "Superman", "estreno": 1996, "act": "Christopher Reeve"}, {"_id": {"$oid": "532504e430e0d0ef6d9f4555c8"}, "tipo": "CD", "Artista": "Los piratas", "TituloCancion": "El rey del caribe"}]
```

Las principales opciones de configuración son:

- Para exportar datos a un archivo csv hay que utilizar las opciones --csv y --fields, indicando en esta última los campos de los documentos que son exportados. En la versión 3.0 se ha sustituido --csv por --type=csv (figuras 240 y 241).

**Figura 240.** Exportación a csv.

```
C:\mongodb\bin>mongoexport --db multimedia --collection media --csv --fields tipo,titulo --out C:/Nuevo/media.csv
connected to: 127.0.0.1
exported 8 records
```

**Figura 241.** Datos exportados a csv.

|   | tipo, titulo               |
|---|----------------------------|
| 1 | "CD"                       |
| 2 | "DVD"                      |
| 3 | "DVD"                      |
| 4 | "DVD"                      |
| 5 | "libro", "Java para todos" |
| 6 |                            |
| 7 |                            |
| 8 |                            |

- Se pueden filtrar los resultados que se quieren exportar utilizando la opción --query junto con una cadena que represente una condición como las que aparecen con el método find (). Debe ir encerrada entre comillas dobles y en el interior entre comillas simples (figuras 242 y 243).

**Figura 242.** Exportación con filtros.

```
C:\mongodb\bin>mongoexport --db multimedia --collection media --query "{tipo:'CD'}" --out C:/Nuevo/media.json
connected to: 127.0.0.1
exported 1 records
```

**Figura 243.** Datos exportados filtrados.

|   |                                                                              |
|---|------------------------------------------------------------------------------|
| 1 | { "_id" : { "oid" : "54dfa30e4fa9e7f2f218dde7" }, "tipo" : "CD", "Artista" : |
| 2 |                                                                              |

- Es posible realizar una exportación desde un servidor remoto utilizando las opciones -- host y -- port para indicar el nombre del equipo y el puerto donde se ejecuta el servidor de MongoDB (figura 244).

**Figura 244.** Exportación desde servidor remoto.

```
C:\mongodb\bin>mongoexport --host Antonio --port 27019 --db multimedia --collection media --query "{tipo:'CD'}" --out C:/Nuevo/media.json
connected to: Antonio:27019
exported 1 records
```

Las opciones de configuración de mongoexport se pueden consultar con mongoexport –help (figura 245):

**Figura 245.** Opciones de mongoexport.

```
C:\mongod\bin>mongoexport --help
Export MongoDB data to CSV, TSV or JSON files.

Options:
  --help
  -v [ --verbose ]                                produce help message
  --quiet   be more verbose <include multiple times
  --version                                       for more verbosity e.g. -vvvvv
  -h [ --host ] arg                               silence all non error diagnostic
  --port arg                                      messages
  --ipv6   print the program's version and exit
  -u [ --username ] arg                           mongo host to connect to < <set
  -p [ --password ] arg                           name>/s1,s2 for sets>
  --authenticationDatabase arg                   server port. Can also use --host
  --authenticationMechanism arg <=MONGODB-CR>    hostname:port
  --gssapiServiceName arg <=mongodb>             enable IPv6 support <disabled by
  --gssapiHostName arg                           default>
  --dbpath arg                                     username
  --directoryperdb                                password
  --journal   user source <defaults to dbname>
  --direct
  --fieldFile arg                                 authentication mechanism
  --query arg                                     Service name to use when authenticating
  --csv   using GSSAPI/Kerberos
  --out arg                                       Remote host name to use for purpose of
  --jsonArray                                     GSSAPI/Kerberos authentication
  --forceTableScan                                directly access mongod database files
  --skip arg <=0>                                 in the given path, instead of
  --limit arg <=0>                                connecting to a mongod server - needs
  --sort arg                                      to lock the data directory, so cannot
  --db arg   be used if a mongod is currently
  --collection arg                                accessing the same path
  --fields arg                                    each db is in a separate directory
  --fieldFile arg                                <relevant only if dbpath specified>
  --query arg                                     enable journaling <relevant only if
  --csv   dbpath specified>
  --out arg                                       database to use
  --jsonArray                                     collection to use <some commands>
  --forceTableScan                                comma separated list of field names
  --skip arg <=0>                                 e.g. -f name,age
  --limit arg <=0>                                file with field names - 1 per line
  --sort arg                                      query filter, as a JSON string, e.g.,
  --db arg   '<x:>{gt:1}>'
  --collection arg                                export to csv instead of json
  --fields arg                                    output file; if not specified, stdout
  --fieldFile arg                                is used
  --query arg                                     output to a json array rather than one
  --jsonArray                                     object per line
  --forceTableScan                                use secondaries for export if
  --skip arg <=0>                                 available, default true
  --limit arg <=0>                                force a table scan <do not use
  --sort arg                                      $snapshot>
  --db arg   documents to skip, default 0
  --collection arg                                limit the numbers of documents
  --fields arg                                    returned, default all
  --fieldFile arg                                sort order, as a JSON string, e.g.,
  --query arg                                     '<x:>1>'
```

## 5. Importación de datos

Para importar datos se utiliza el comando mongoimport, y para ello:

- Es necesario que esté ejecutándose el servidor sobre el que se va a realizar la exportación (figura 246).

**Figura 246.** Ejecución del servidor mongod.

```
C:\mongodb\bin\mongod.exe
2015-08-24T10:21:18.506+0200 help for help and startup options
2015-08-24T10:21:18.506+0200 [initandlisten] MongoDB starting : pid=5988 port=27017 dbpath=C:\data\db\ 64-bit host=DESKTOP-3H1V9D9\Windows 7\Windows Server 2008 R2 SP1
2015-08-24T10:21:18.509+0200 [initandlisten] targetBinOS: Windows 7\Windows Server 2008 R2 SP1
2015-08-24T10:21:18.509+0200 [initandlisten] git version: a9d59ad29e302de82e9cb9
2015-08-24T10:21:18.509+0200 [initandlisten] build info: windows sys.getwindowsversion 6.1.7601.16384
2015-08-24T10:21:18.509+0200 [initandlisten] allocator: system
2015-08-24T10:21:18.509+0200 [initandlisten] options: { "replSet": "rs0", "dbPath": "C:\data\db\", "journal": true }
2015-08-24T10:21:18.509+0200 [initandlisten] recover: 1 no journal files present, no recovery needed
2015-08-24T10:21:18.509+0200 [initandlisten] 
2015-08-24T10:21:18.509+0200 [initandlisten] ** WARNING: mongod started without --port or --bind_ip specified. Using system's reported port 27017.
2015-08-24T10:21:18.509+0200 [initandlisten] ** WARNING: mongod started without --replSet specified. Restart with --replSet unless you are doing maintenance and no other clients are connected.
2015-08-24T10:21:18.509+0200 [initandlisten] ** WARNING: The TTL collection monitor will not start because of this.
2015-08-24T10:21:18.509+0200 [initandlisten] ** For more info see http://dochub.mongodb.org/core/tlscollections
2015-08-24T10:21:18.509+0200 [initandlisten] 
```

- Se abre un terminal de comandos y se ejecuta el programa mongoimport, que se encuentra en el directorio bin en el que está instalado MongoDB. Es necesario indicar la base de datos con --db, la colección con --collection y el archivo de origen con -file (figuras 247 y 248).

**Figura 247.** Ejecución de mongoimport.

```
C:\mongodb\bin>mongoimport --db multimedia2 --collection media2 --file C:/Nuevo/media.json
connected to: 127.0.0.1
2015-08-24T21:51:34.498+0200 imported 1 objects
```

**Figura 248.** Base de datos importada.

```
C:\mongodb\bin>mongo
MongoDB shell version: 2.6.7
connecting to: test
> show databases
admin          <empty>
base           0.078GB
blog            0.078GB
ensayos         0.078GB
local           0.078GB
mi01            0.078GB
multimedia      0.078GB
multimedia2     0.078GB
prueba          0.078GB
school          0.078GB
students        0.078GB
test            0.078GB
>
```

Como resultado se importan los datos desde un fichero en formato json y lo carga en la colección y la base de datos indicada.

La utilidad de mongoimport permite importar datos desde tres formatos diferentes:

- CSV: cada línea representa un documento, y están separados por comas.
- TSV: similar a los ficheros csv, pero usa como delimitador un tabulador.
- JSON: contiene un bloque JSON por línea, que representa un documento.

Para indicar un tipo diferente de archivos json se debe especificar la opción `--type=nombre_tipo`, y hay que especificar el nombre de los campos por medio de la opción `--fields` o mediante la opción `--fieldFile`, que especifica un fichero donde se listan los nombres de los campos (uno por cada línea).

Las principales opciones de configuración son:

- La opción `--headerlines` se aplica a los archivos csv y tsv para indicar que se use la primera línea del archivo como la lista de los nombres de campos (figura 249).

**Figura 249.** Ejecución de mongoimport con opciones sobre la cabecera.

```
C:\mongodb\bin>mongoimport --db multimedia3 --collection media --type=csv --head
erline --file C:/Nuevo/media.csv
connected to: 127.0.0.1
2015-04-24T23:12:51.030+0200 imported 8 objects
```

- La opción `--ignoreblanks` no importa los campos vacíos. Si el campo es vacío, entonces no se crea una fila para ese elemento en el documento. En caso de no usar esa opción, se crea un elemento vacío con el nombre de la columna (figura 250).

**Figura 250.** Ejecución de mongoimport con opción --ignoreblanks.

```
C:\mongodb\bin>mongoimport --db multimedia2 --collection media2 --ignoreBlanks --file C:/Nuevo/media.json
connected to: 127.0.0.1
2015-04-24T22:35:40.661+0200 insertDocument :: caused by :: E11000 E11000 duplicate key error index: multimedia2.media2.$_id_ dup key: { : ObjectId('54dfa3be4fa
de7f2f218dde?') }
2015-04-24T22:35:40.677+0200 imported 1 objects
```

- La opción --drop elimina una colección y la vuelve a recrear con los datos que se han importado. En caso contrario, lo que hace es añadirlos a la colección (figura 251).

**Figura 251.** Ejecución de mongoimport con opción --drop.

```
C:\mongodb\bin>mongoimport --db multimedia2 --collection media2 --drop --file C:/Nuevo/media.json
connected to: 127.0.0.1
2015-04-24T22:36:36.736+0200 dropping: multimedia2.media2
2015-04-24T22:36:36.814+0200 imported 1 objects
```

- Es posible realizar una importación en un servidor remoto utilizando las opciones -- host y -- port para indicar el nombre del equipo y el puerto donde se ejecuta el servidor de MongoDB (figura 252).

**Figura 252.** Importación desde servidor remoto.

```
C:\mongodb\bin>mongoimport --host Antonio --port 27019 --db multimedia2 --collection media2 --drop --file C:/Nuevo/media.json
connected to: Antonio:27019
2015-04-24T23:03:35.293+0200 dropping: multimedia2.media2
2015-04-24T23:03:35.321+0200 imported 1 objects
```

Las opciones de configuración de mongoimport se pueden consultar con mongoimport –help (figura 253).

**Figura 253.** Opciones de mongoimport.

```

$ mongoimport --help
Import CSV, TSV or JSON data into MongoDB.

Then importing JSON documents, each document must be a separate line of the input file.

Example:
mongoimport --host myhost --db my_db --collection docs < mydocfile.json

Options:
--help
  -v [ --verbose ]           produce help message
  -q [ --quiet ]             log verboseness. Can include multiple times
                             for more verbosity e.g. -vvvvv
  --version                 silence all non error diagnostic
  -h [ --host ] arg          messages
                             print the program's version and exit
  -p [ --port ] arg          mongo host to connect to < (set
                             mongod port for sets>
  --port arg                server ports. Can also use --host
  --ipv6                    hostnames:port
                             or --host IPv6 support <disabled by
                             default>
  -u [ --username ] arg     username
  -p [ --password ] arg     password
  --authenticationDatabase arg use authentication source <defaults to dbname>
  --authenticationMechanism arg <-MONGODB-CR>
  --gssapiServiceName arg <-mongodbs> Service name to use when authenticating
  using GSSAPI/Kerberos
  --gssapiHostName arg      Remote host name to use for purpose of
  --dbpath arg               GSSAPI/Kerberos authentication
                             directly access mongod database files
                             in the given path instead of
                             connecting to a mongod server - needs
                             to lock the data directory, so cannot
                             be used if mongod is currently
                             accessing the dbpath
                             each db is in a separate directory
                             (relevant only if dbpath specified)
  --directoryperdb           Create an only if dbpath specified
  --journal                  multiple documents in dbpath
                             (relevant only if dbpath specified)
  --collection arg           database to use
  --db [ --db ] arg          collection to use (some commands)
  --collection [ --collection ] arg comma-separated list of field names
  --fields [ --fields ] arg  e.g. name,age
  --fieldFile arg            file with field names - 1 per line
  --ignoreBlanks             if given, empty fields in csv and tsv
  --type arg                 will be ignored
  --file arg                 type of file to import. default: json
  --drop                      File to import from if not specified
  --headerline               drop collection first
  --upsert                   first line in input file is a header
  --upsertFields arg         (CSV and TSV only)
  --insert or update objects that already
  --stopOnError               exist
  --upsertFields arg         command-separated fields for the query
                             part of the upsert. You should make
                             sure this is indexed
  --stopOnError               stop importing at first error rather
  --jsonArray                 than continuing
                             load a json array, not one item per
                             line. Currently limited to 1mb.

```

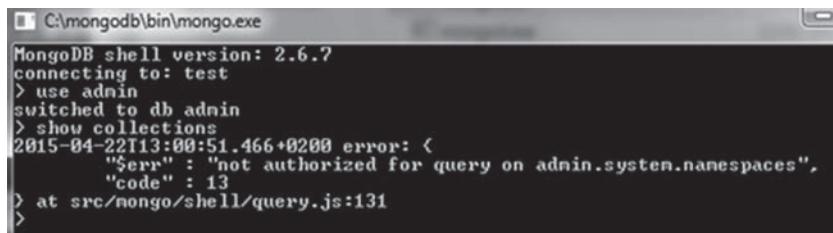
## 6. Creación de usuarios

Para crear un usuario administrador, entramos en la Shell de comandos y cambiamos a la base de datos «admin». A continuación se crea un usuario con el método db.createUser ({user: "Nombre usuario", pwd:"Password", roles: [rol1, rol2,...]}) donde rol= {role:" ", db:" "} y role será el role que el usuario tiene sobre la base de datos especificada (figura 254). En caso de no especificar una base de datos concreta, entonces se toman sobre la base de datos actual.

**Figura 254.** Creación de un usuario.

```
> db.createUser(
... {
...   user: "Antonio",
...   pwd: "Antonio",
...   roles:
...     [ { role: "readWrite", db: "multimedia"}, "dbAdmin"]
... }
... )
Successfully added user: {
  "user" : "Antonio",
  "roles" : [
    {
      "role" : "readWrite",
      "db" : "multimedia"
    },
    "dbAdmin"
}
>
```

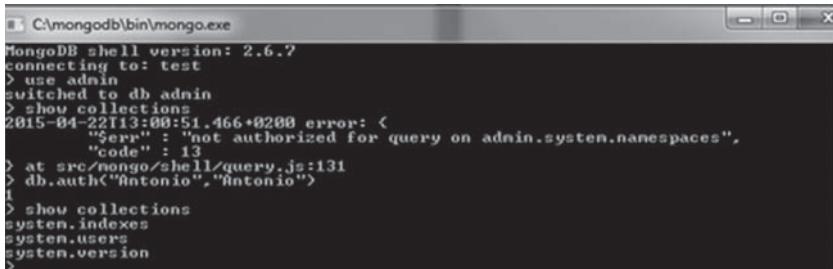
Para alterar la configuración del servidor y que admita la autenticación, se cierra el servidor, se reinicia con la opción --auth, es decir, mongod --auth, y se abre una Shell de comandos mongo. Ahora, por ejemplo, si se accede a la colección admin, y se trata de recuperar las colecciones con show collections, mostraría un error como el siguiente (figura 255):

**Figura 255.** Intento de acceder a la colección admin.


C:\mongodb\bin\mongo.exe

```
MongoDB shell version: 2.6.7
connecting to: test
> use admin
switched to db admin
> show collections
2015-04-22T13:00:51.466+0200 error: <
  "$err" : "not authorized for query on admin.system.namespaces",
  "code" : 13
> at src/mongo/shell/query.js:131
>
```

Este error se produce, dado que se trata de una sesión autenticada. Si se quieren ver las colecciones, hay que autenticarse con usuario que tenga permisos para leer en la base de datos correspondiente mediante el método con db.auth («Nombre Usuario», «Password») (figura 256):

**Figura 256.** Uso del método auth.


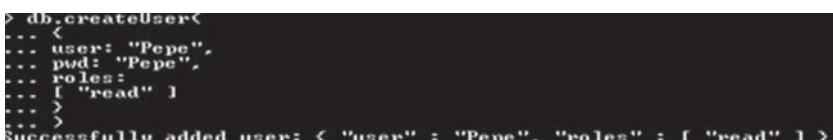
```
C:\mongodb\bin\mongo.exe
MongoDB shell version: 2.6.7
connecting to: test
> use admin
switched to db admin
> show collections
2015-04-22T13:00:51.466+0200 error: <
    "err": "not authorized for query on admin.system.namespaces",
    "code": 13
> at src/mongo/shell/query.js:131
> db.auth("Antonio","Antonio")
1
> show collections
system.indexes
system.users
system.version
>
```

Los usuarios del sistema se pueden mostrar consultando la colección «system.users» (figura 257).

**Figura 257.** Consulta de la colección «system.users».


```
> db.system.users.find().pretty()
{
    "_id" : "admin.Antonio",
    "user" : "Antonio",
    "db" : "admin",
    "credentials" : {
        "MONGODB-CR" : "bb909623f4fe8ad2dbe923de639dcf90"
    },
    "roles" : [
        {
            "role" : "readWrite",
            "db" : "multimedia"
        },
        {
            "role" : "dbAdmin",
            "db" : "admin"
        }
    ]
}
1
```

Cuando se crean usuarios que solo tienen permisos de lectura, el usuario tiene restricciones para realizar determinadas acciones (figura 258).

**Figura 258.** Creación de usuarios.


```
> db.createUser(
...   {
...     user: "Pepe",
...     pwd: "Pepe",
...     roles: [
...       { "read" }
...     ]
...   }
)
Successfully added user: { "user" : "Pepe", "roles" : [ "read" ] }
```

Cuando se autentica con este usuario, por ejemplo, no es posible ver el contenido de la colección system.users (figura 259).

**Figura 259.** Consulta de la colección system.users.

```
> use admin
switched to db admin
> db.auth("Pepe","Pepe")
1
> show collections
system.indexes
system.users
system.version
> db.system.users.find()
error: { "$err" : "not authorized for query on admin.system.users", "code" : 13
}
```

Los principales roles del sistema son los siguientes:

| Role      | Permisos                                                                                                                                                                                                                                                   |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| read      | Lectura sobre todas las colecciones que no son del sistema y las colecciones del sistema <code>system.indexes</code> , <code>system.js</code> y <code>system.namespaces</code> .                                                                           |
| readWrite | Los mismos privilegios que Read, y modificación de todas las colecciones que no son del sistema y sobre la colección del sistema <code>system.js</code> .                                                                                                  |
| dbAdmin   | Privilegios sobre las colecciones del sistema <code>system.indexes</code> , <code>system.namespaces</code> y <code>system.profile</code> , y algunos privilegios sobre colecciones que no son del sistema (entre ellos no se incluye la lectura completa). |
| dbOwner   | Combina los privilegios de los roles <code>readWrite</code> , <code>dbAdmin</code> y <code>userAdmin</code> .                                                                                                                                              |
| userAdmin | Permite crear y modificar roles de la base de datos actual. Indirectamente proporciona acceso superusuario a la base de datos o a un clúster. Es posible otorgar cualquier privilegio a cualquier usuario.                                                 |

Para obtener un listado exhaustivo de los roles y los privilegios otorgados, consultese la documentación de MongoDB (<<http://docs.mongodb.org/manual/reference/built-in-roles/>>).

## 7. Otras operaciones de administración

Otras operaciones de administración son las siguientes:

- 1) Obtener la versión del servidor mediante db.version () (figura 260).

**Figura 260.** Obtener la versión del servidor.

```
> db.version()
2.6.7
```

- 2) Obtener información sobre el estado del servidor mediante db.serverStatus () (figura 261).

**Figura 261.** Opciones de mongoimport.

```
> db.serverStatus()
{
  "host" : "Antonio",
  "version" : "2.6.7",
  "process" : "mongod",
  "pid" : NumberLong(2848),
  "uptime" : 1941,
  "lastHeartbeat" : NumberLong(193915),
  "optimeEstimate" : 180,
  "localTime" : ISODate("2015-04-25T10:46:40.755Z"),
  "asserts" : {
    "regular" : 0,
    "warning" : 0,
    "msg" : 0,
    "user" : 1,
    "rollovers" : 0
  },
  "backgroundFlushing" : {
    "flushes" : 3,
    "total_ms" : 238,
    "average_ms" : 79.33333333333333,
    "last_ms" : 39,
    "last_finished" : ISODate("2015-04-25T10:46:26.988Z")
  },
  "connections" : {
```

- 3) Parar el servidor desde la Shell de comandos con db.shutdownServer () desde la base de datos admin (figura 262).

**Figura 262.** Ejecución del comando db.shutdownServer ()�

```
> db.shutdownServer()
2015-04-25T12:48:40.947+0200 DBClientCursor::init call() failed
server should be down...
2015-04-25T12:48:40.952+0200 trying reconnect to 127.0.0.1:27017 <127.0.0.1> failed
2015-04-25T12:48:41.986+0200 warning: Failed to connect to 127.0.0.1:27017, reason: errno:10061 No se puede establecer una conexión ya que el equipo de destino renegó expresamente dicha conexión.
2015-04-25T12:48:41.988+0200 reconnect 127.0.0.1:27017 <127.0.0.1> failed failed
couldn't connect to server 127.0.0.1:27017 <127.0.0.1>, connection attempt failed
```

**4)** Validar una colección. Para ello hay que situarse en la base de datos que contiene la colección y usar el comando siguiente sobre la colección db.nombre\_colección.validate () (figura 263). Obtiene información sobre el estado de los datos e índices definidos.

**Figura 263.** Ejecución del método validate () sobre la colección.

```
> use multimedia
switched to db multimedia
> db.media.validate()
{
  "ns" : "multimedia.media",
  "firstExtent" : "0:d2000 ns:multimedia.media",
  "lastExtent" : "0:d2000 ns:multimedia.media",
  "extentCount" : 1,
  "datasize" : 1536,
  "records" : 8,
  "lastExtentSize" : 8192,
  "padding" : 1,
  "firstExtentDetails" : {
    "loc" : "0:d2000",
    "xnext" : "null",
    "xprev" : "null",
    "nsdiag" : "multimedia.media",
    "size" : 8192,
    "firstRecord" : "0:d20b0",
    "lastRecord" : "0:d26b0"
  },
  "deletedCount" : 1,
  "deletedSize" : 6352,
  "nindexes" : 1,
  "keysPerIndex" : {
    "multimedia.media.$_id_" : 8
  },
  "valid" : true,
  "errors" : [ ],
  "warning" : "Some checks omitted for speed. use {full:true} option to do more thorough scan."
}
> "ok" : 1
```

**5)** Si el método validate indica que hay problemas en los índices, estos se pueden reparar mediante el método reIndex () aplicado sobre la colección en la forma db.nombre\_colección.reIndex () (figura 264).

**Figura 264.** Ejecución del método reIndex () .

```
> db.media.reIndex()
{
  "nIndexesWas" : 1,
  "nIndexes" : 1,
  "indexes" : [
    {
      "key" : {
        "_id" : 1
      },
      "name" : "$_id",
      "ns" : "multimedia.media"
    }
  ],
  "ok" : 1
}
```

6) Si el método validate indica que hay problemas en los datos, estos se pueden reparar mediante el método repairDatabase () aplicado sobre la base de datos en la que se encuentra la colección en la forma db.repairDatabase (). También es posible hacer lo mismo en cuanto al servidor ejecutando el servidor con la opción --repair, es decir, mongod --repair (figura 265).

**Figura 265.** Ejecución del método repairDatabase.

```
> db.repairDatabase()
{ "ok" : 1 }
```

7) Existe una herramienta denominada «mongostat» (figura 266), que puede encontrarse en el directorio bin de la instalación de MongoDB, que proporciona información acerca de lo que está ocurriendo en el servidor. En particular son interesantes las siguientes columnas:

- Las primeras seis columnas ofrecen datos sobre el porcentaje el servidor realiza determinadas operaciones tales como insert, query, etc.
- %idx miss: porcentaje de consultas que no pudieron usar un índice. Un valor alto indica que se necesitarían añadir índices al sistema.
- conn: muestra cuántas conexiones están abiertas en el servidor. Un número alto indicaría un problema.
- % locked: muestra la cantidad de tiempo que el servidor tuvo bloqueadas las colecciones. Un número alto indica que se están realizando operaciones de bloqueo que sería mejor realizar durante el período de mantenimiento.

**Figura 266.** Ejecución de mongostat.

```
C:\mongodb\bin>mongostat
connected to: 127.0.0.1
insert query update delete getmore command flushes mapped vsize res faults
locked db idx miss % qps iow netIn netOut conn time
*0 *0 *0 *0 0:0 1:0 0 1.09g 2.33g 245n 0
..:0.1% 0 *0 0:0 0:0 62b 5k 1 13:12:35
*0 *0 *0 *0 0 0:0 1:0 0 1.09g 2.33g 245n 0
test:0.0% 0 0:0 0:0 0:0 62b 5k 1 13:12:36
*0 *0 *0 *0 0 0:0 1:0 0 1.09g 2.33g 245n 0
test:0.0% 0 0:0 0:0 0:0 62b 5k 1 13:12:38
*0 *0 *0 *0 0 0:0 1:0 0 1.09g 2.33g 245n 0
admin:0.0% 0 0:0 0:0 0:0 62b 5k 1 13:12:39
*0 *0 *0 *0 0 0:0 1:0 0 1.09g 2.33g 245n 0
test:0.0% 0 0:0 0:0 0:0 62b 5k 1 13:12:40
```

- 8) Se puede actualizar el servidor ejecutándolo con la opción –upgrade, pero, en tal caso, es necesario realizar algunas acciones previas de preparación tales como hacer un backup.

## 8. Ejercicios propuestos

1) Realizar un backup de las bases de datos que se tengan en algún directorio con permiso de escritura. Para ello, utilizar la opción –out, que permite redirigir la salida a un directorio diferente al que hay por defecto.

2) Añadir las bases de datos creadas en los ejercicios de los capítulos anteriores. Una vez cargadas, realizar un backup de la colección media que se encuentra en la base de datos multimedia. Para ello, utilizar las opciones --db y –collection, que permiten seleccionar una base de datos y una colección concreta.

3) Abrir una instancia de mongod en el puerto 27019, y desde un terminal de mongo sobre el mismo puerto añadir las bases de datos de los capítulos anteriores y repetir el ejercicio 2. Ahora, para poder hacer un backup, hay que indicarle explícitamente sobre qué host y en qué puerto debe realizarlo. Para ello, se debe

encontrar el nombre del host de la máquina, y usar las opciones `--host` y `-port`, que permiten indicar sobre qué máquina y en qué puerto debe realizarse el backup.

**4)** Restaurar el backup de las bases de datos que se realizó en el ejercicio 1. Para ello, junto al comando `mongostore`, se debe indicar el path al directorio en el que se encuentra el backup.

**5)** Considerar los backups que realizaron en el ejercicio 2. Recuperar los documentos de la base de datos multimedia y de la colección media. Para ello, utilizar las opciones `--db` y `-collection`, que permiten seleccionar una base de datos y una colección concreta.

**6)** Abrir una instancia de `mongod` en el puerto 27019, y desde un terminal de mongo sobre el mismo puerto restaurar las bases de datos de los capítulos anteriores y repetir el ejercicio 2. Ahora, para poder hacer una restauración, hay que indicarle explícitamente sobre qué host y en qué puerto debe realizarlo. Para ello, se debe encontrar el nombre del host de la máquina, y usar la opción `-host`, que permite indicar sobre qué máquina y en qué puerto debe realizarse la restauración.

**7)** Considerar la base de datos que se utilizó en el ejercicio 2 del capítulo sobre optimización. Crear de nuevo esa base de datos con la colección correspondiente. A continuación, realizar una exportación a un archivo de tipo csv, exportando únicamente los campos «`producto`» y «`color`». Para ello, utilizar las opciones `--csv` y `--fields`.

**8)** Considerar la base de datos anterior. A continuación realizar una exportación a un archivo de tipo csv, exportando el resultado de la consulta que se hizo en ese ejercicio, es decir, «`recupera las mesas de color rojo`». Para ello utilizar las opciones `--csv` y `-query`, donde se indicará como un documento en forma de cadena la condición de filtrado.

**9)** Abrir una instancia de mongod en el puerto 27019, y desde un terminal de mongo sobre el mismo puerto y repetir el ejercicio 1. Ahora, para poder hacer una exportación, hay que indicarle explícitamente sobre qué host y en qué puerto debe realizarlo. Para ello, se debe encontrar el nombre del host de la máquina, y usar las opciones --host y –port, que permiten indicar sobre qué máquina y en qué puerto debe realizarse la exportación.

**10)** Considerar la base de datos que se ha exportado en los ejercicios anteriores. A continuación, importar el archivo csv exportado a una colección denominada «ensayo2» en la base de datos «ensayos». Debe indicarse que obtenga los nombres de los campos a partir de la primera línea del archivo csv mediante la opción --headerline.

**11)** Crear un usuario administrador y un usuario con derechos de solo de lectura.

## Bibliografía

- Banker, K.** (2011). *MongoDB in action*. Manning Publications Co.
- Chodorow, K.** (2013). *MongoDB: the definitive guide*. O'Reilly Media, Inc.
- Hows, D.; Membrey, P.; Plugge, E.** (2014). *MongoDB Basics*. Apress.
- Membrey, P.; Plugge, E.; Hawkins, D.** (2010). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress.
- Sitio oficial MongoDB, «Administration». <<https://docs.mongodb.org/v2.6/administration/>>



## Anexo I. Instalación y puesta en marcha de MongoDB

### *Instalación de MongoDB*

En este anexo se muestra cómo instalar MongoDB en un entorno Windows. Las ilustraciones se han realizado utilizando la versión 2.6.7, aunque los pasos que se describen son comunes para otras versiones de MongoDB.

En primer lugar, hay que descargarse el ejecutable desde la página oficial de MongoDB: <<http://www.mongodb.org/downloads>>.

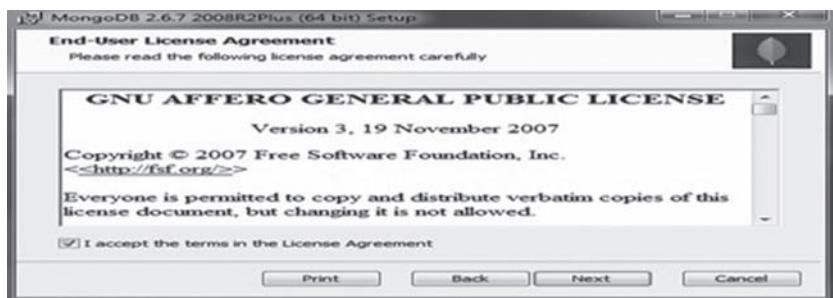
En esta página (figura 1), se elige la versión adecuada al sistema operativo que se tenga instalado en la máquina sobre la que se va a instalar MongoDB. El instalador que se obtiene es un ejecutable.

**Figura 1.** Ventana de descarga del instalador de MongoDB.



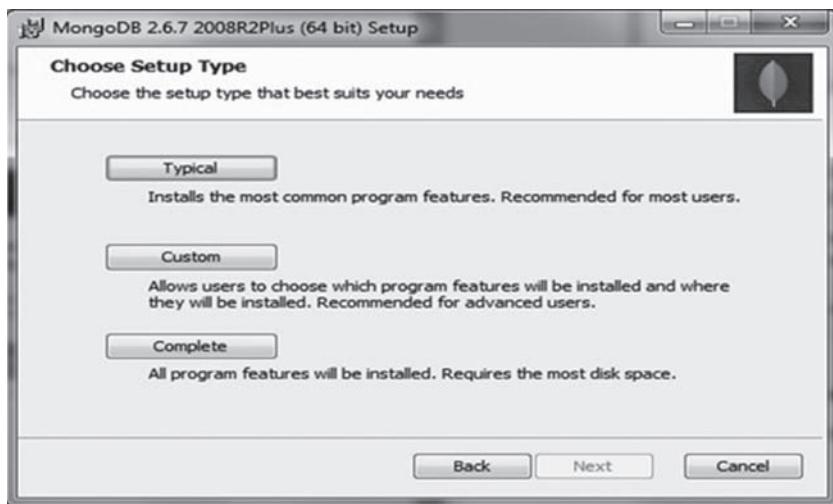
Una vez que se disponga del ejecutable, se hace doble clic sobre este y se siguen los pasos que indica el instalador pulsando Next en cada pantalla (figura 2).

**Figura 2.** Pantalla de instalación de MongoDB.



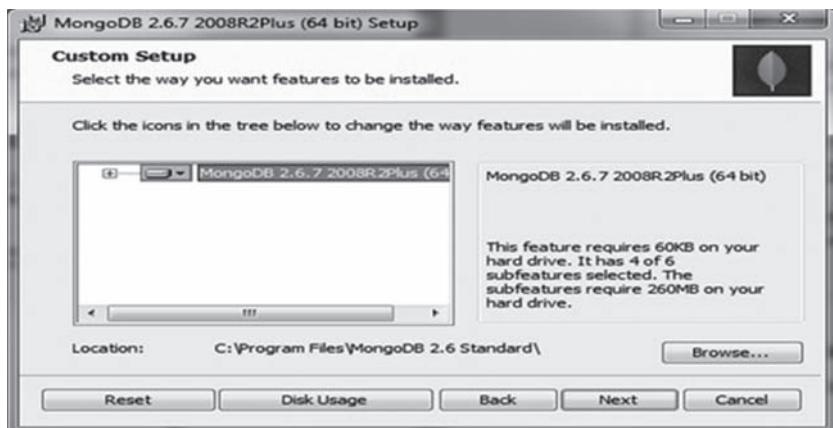
En la primera pantalla de instalación, hay que elegir entre varias opciones de configuración que ofrece el instalador. Se elegirá la opción «Custom» (figura 3).

**Figura 3.** Pantalla de elección del tipo de instalación.



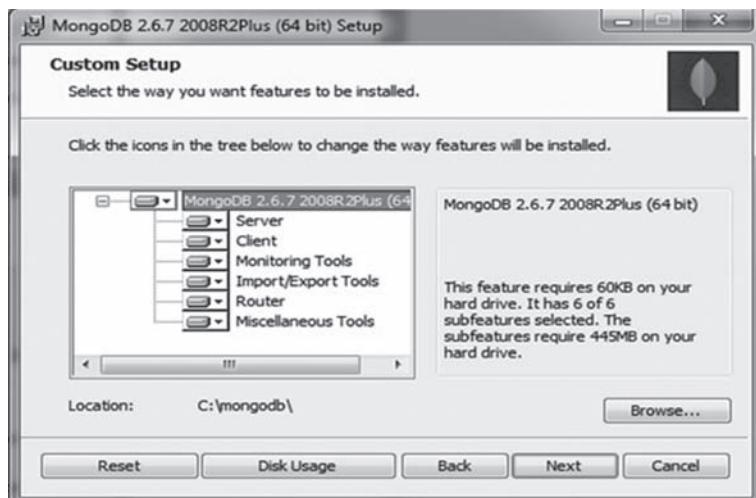
En la siguiente pantalla del instalador (figura 1.4), se puede configurar el directorio de instalación de la aplicación. Aunque puede instalarse en cualquier directorio, se va a realizar la instalación sobre el directorio C:\mongodb.

**Figura 4.** Pantalla de configuración del directorio de instalación.



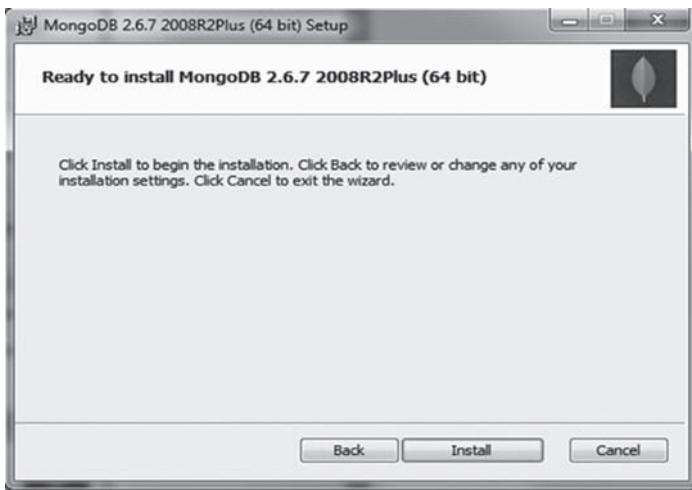
Una vez configurado el directorio de instalación en C: \mongodb, se eligen los programas que se quieren instalar (figura 5). Se pueden dejar seleccionados los que aparecen por defecto en el instalador. A continuación, se pulsa sobre Next para pasar a la siguiente pantalla.

**Figura 5.** Pantalla de selección de los programas que instalar.



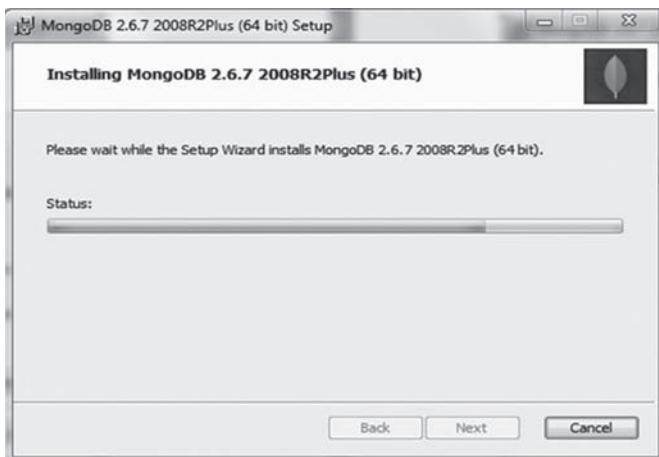
En la siguiente pantalla de instalación, se pulsa sobre Install para comenzar la instalación del programa (figura 6).

**Figura 6.** Pantalla de comienzo de instalación.



A continuación, comienza el proceso de instalación de la aplicación (figura 7).

**Figura 7.** Pantalla de instalación de MongoDB.



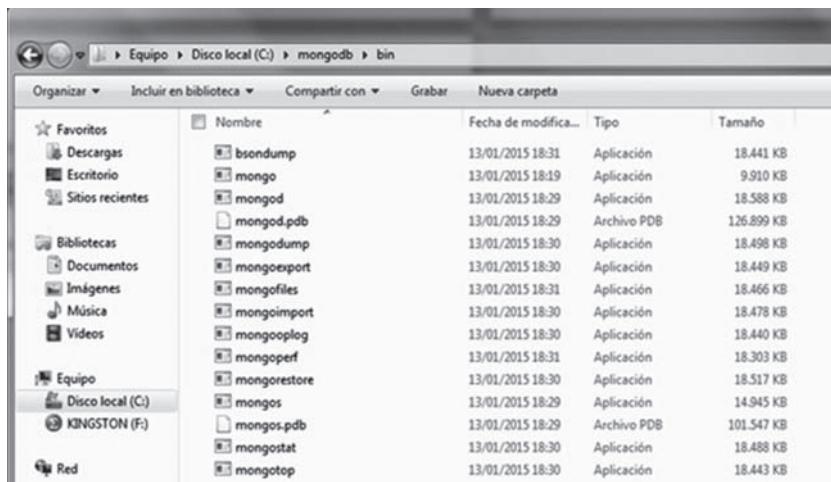
Una vez finalizada la instalación, aparece la pantalla (figura 8) de finalización, en la que se debe pulsar sobre el botón Finish.

**Figura 8.** Pantalla de final de instalación.



En el subdirectorio bin del directorio seleccionado para llevar a cabo la instalación (figura 9), se encuentran todas las aplicaciones que se han instalado.

**Figura 9.** Directorio bin con las aplicaciones instaladas.



Entre las aplicaciones instaladas, hay dos aplicaciones básicas para trabajar con MongoDB:

- **mongo:** permite usar la Shell o línea de comandos para interactuar con MongoDB. Desde esta Shell, se puede hacer cualquier operación sobre MongoDB.
- **Mongod:** es el servidor de la base de datos. Cuando se lanza el servidor, se pueden añadir diversas opciones tales como:

--dbpath: para indicar dónde se encuentra la base de datos.  
--version: para mostrar información sobre la versión de la base de datos.  
--sysinfo flag: para mostrar información sobre el sistema.  
--help: para mostrar todas las opciones posibles para lanzar el servidor.

Antes de poder ejecutar el servidor, es necesario crearse un directorio en el sistema que se usará para almacenar los archivos de la bases de datos. Por defecto, MongoDB intentará almacenar los archivos en el directorio C:\data\db. Es por ello que hay que crear esa estructura de directorios en el sistema (figura 10).

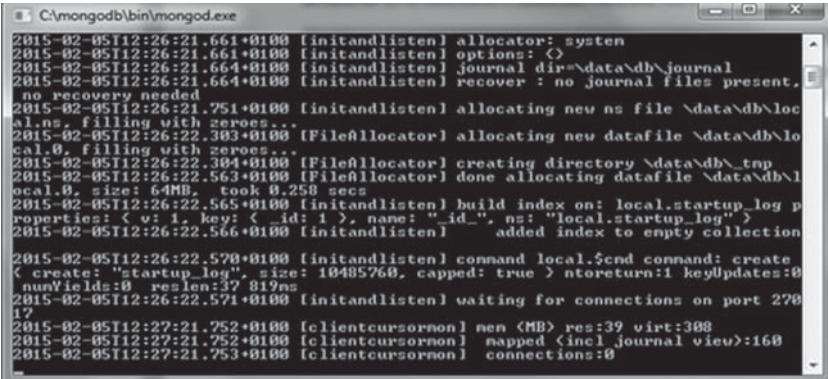
**Figura 10.** Directorio de almacenamiento de MongoDB.



A continuación, se puede ejecutar tanto el servidor como la Shell. Para ello, hay que situarse en el directorio bin donde se ha realizado la instalación. En primer lugar, se pulsa sobre el archivo denominado «**mongod**» para lanzar el servidor. MongoDB trabaja sobre el puerto 27017 e incorpora un servidor HTTP muy básico que funciona en el puerto 28017, de manera que, si se invoca en el navegador <http://localhost:28017>, se puede conseguir información administrativa acerca de la base de datos.

Hay que observar que se puede salir de mongod pulsando sobre Ctrl-C en la Shell en la que se está ejecutando el servidor (figura 1.11).

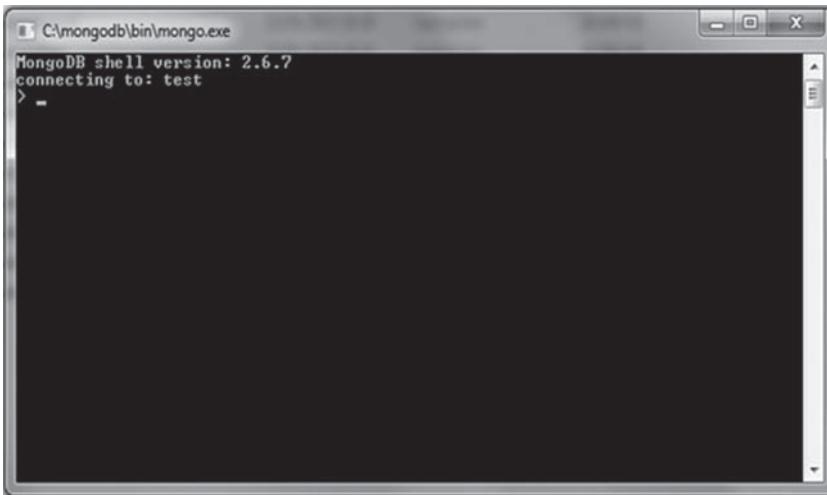
**Figura 11.** Shell en la que se ejecuta el servidor de MongoDB.



The screenshot shows a Windows Command Prompt window with the title 'C:\mongodb\bin\mongod.exe'. The window displays the log output of the MongoDB server (mongod) starting up. The text is as follows:

```
2015-02-05T12:26:21.661+0100 [initandlisten] allocator: system
2015-02-05T12:26:21.661+0100 [initandlisten] options: <>
2015-02-05T12:26:21.664+0100 [initandlisten] journal dir=\data\db\journal
2015-02-05T12:26:21.664+0100 [initandlisten] recover : no journal files present,
no recovery needed
2015-02-05T12:26:21.751+0100 [initandlisten] allocating new ns file \data\db\local.ns, filling with zeroes...
2015-02-05T12:26:22.303+0100 [FileAllocator] allocating new datafile \data\db\local.0, filling with zeroes...
2015-02-05T12:26:22.304+0100 [FileAllocator] creating directory \data\db\tmp
2015-02-05T12:26:22.563+0100 [FileAllocator] done allocating datafile \data\db\local.0, size: 64MB, took 0.258 secs
2015-02-05T12:26:22.565+0100 [initandlisten] build index on: local.startup_log properties: < v: 1, key: { _id: 1 }, name: "local.startup_log" >
2015-02-05T12:26:22.566+0100 [initandlisten] added index to empty collection
2015-02-05T12:26:22.570+0100 [initandlisten] command local.$cmd command: create
< create: "startup_log", size: 10485760, capped: true > nreturned:1 keyUpdates:0
numYields:0 reslen:37 819ns
2015-02-05T12:26:22.571+0100 [initandlisten] waiting for connections on port 27017
2015-02-05T12:27:21.752+0100 [clientcursorsmon] mem <MB> res:39 virt:308
2015-02-05T12:27:21.752+0100 [clientcursorsmon] mapped <incl. journal view>:160
2015-02-05T12:27:21.753+0100 [clientcursorsmon] connections:0
```

Una vez que se ha lanzado el servidor, se pulsa sobre el archivo denominado «**mongo**» para lanzar la Shell de comandos (figura 12).

**Figura 12.** Shell de comandos de MongoDB.A screenshot of a Windows command-line interface window titled "C:\mongodb\bin\mongo.exe". The window displays the MongoDB shell version 2.6.7 and a connection message: "connecting to: test". Below this, there is a single character input prompt "> -". The window has standard Windows-style scroll bars on the right side.

Cuando se ejecuta la Shell con la configuración por defecto, el servidor se conecta a una base de datos por defecto denominada «test». Una característica importante de MongoDB es que, cuando se intenta conectar a una base de datos que no existe, MongoDB la crea automáticamente.

Algunos de los comandos más útiles para empezar a trabajar sobre MongoDB son:

- show dbs: muestra los nombres de las bases de datos que existen.
- show collections: muestra las colecciones de la base de datos actual.
- show users: muestra los usuarios de la base de datos actual.
- use <db nombre>: establece la base de datos que se va a usar.

