NoSQL

Aitor Medrano `<<aitormedrano@gmail.com>>`

Tabla de contenidos

1. No Sólo SQL 5
1.1. Características 6
Esquemas dinámicos 6
Particionado 7
Auto-Sharding 8
Cuando Particionar 8
Replicación 9
Caché integrada 10
1.2. NoSQL vs. SQL 10
1.3. Implantando NoSQL 11
Casos de Uso 12
1.4. Modelos de Datos 12
Documental 13
Características 13
Casos de Uso 14
Clave-Valor 15
Casos de Uso 16
Basado en Columnas 17
Familias de Columnas 17
Operaciones 19
Casos de Uso 19
Grafos 20
Traversing 20
Casos de Uso 21
1.5. Consistencia 22
Sistemas Consistentes 22
Sistemas de Consistencia Eventual 22
1.6. Teorema de CAP 23
Clasificación según CAP 24
1.7. MongoDB 24
Instalación 26
Herramientas 27
1.8. Hola *MongoDB* 28
1.9. Ejercicios 30
(1 punto) Ejercicio 11. Cuestionario 30
(0.25 puntos) Ejercicio 12. Puesta en Marcha con *MongoDB* 30
(0.25 puntos) Ejercicio 13. Inserciones 30
2. MongoDB 32
2.1. BSON 32
2.2. Trabajando con el shell 33
2.3. ObjectId 34
2.4. Consultas 35
Criterios en consultas 37
Proyección de campos 39
Condiciones sobre objetos anidados 39

Condiciones compuestas con Y / O 40
Consultas sobre arrays 41
Conjunto de valores 42
Cursores 43
Contando Documentos 43
2.5. Actualizando documentos 44
Operadores de actualización 44
Actualización Múltiple 45
Actualizaciones sobre Arrays 46
Añadiendo elementos 47
Eliminando elementos 48
Operador posicional 48
2.6. Borrando Documentos 49
2.7. Control de Errores 49
2.8. MongoDB desde *Java* 50
`MongoClient` 51
`DBObject` 51
Inserción 52
Consultas 53
Criterios 54
Selección de campos 55
Campos anidados 55
Trabajando con `DBCursor` 56
Modificación 57
Borrado 57
2.9. Mapping de Objetos 58
2.10. Ejercicios 58
(1 punto) Ejercicio 21. Consultas desde `mongo` 58
(1 punto) Ejercicio 22. Modificaciones desde `mongo` 59
(1.5 puntos) Ejercicio 23. Operaciones desde *Java* 59
3. Rendimiento en *MongoDB* 62
3.1. Diseñando el Esquema 62
Referencias 62
Referencias Manuales 62
DBRef 63
Datos Embebidos 64
Relaciones 64
1:1 64
1:N 65
N:M 66
Jerárquicas 68
Rendimiento 68
3.2. GridFS 69
`mongofiles` 70
GridFS desde *Java* 71
Casos de Uso 73
3.3. Índices 74
Simples 76
Propiedades 78
Compuestos 79
Multiclave 80
Rendimiento 80
Plan de Ejecución 80

Hints ... 81
3.4. Colecciones Limitadas ... 82
3.5. Profiling ... 83
3.6. Ejercicios ... 85
(0.75 puntos) Ejercicio 31. Diseñando el Esquema ... 85
(0.75 puntos) Ejercicio 32. GridFS ... 85
(0.75 puntos) Ejercicio 33. Índices ... 86
(0.25 puntos) Ejercicio 34. Análisis del Log ... 86
4. Agregaciones y Escalabilidad ... 87
4.1. Agregaciones ... 87
4.2. *Pipeline* de Agregación ... 88
Operadores del *pipeline* ... 88
`$group` ... 89
`$sum` ... 91
`$avg` ... 91
`$addToSet` ... 92
`$push` ... 92
`$max` y `$min` ... 93
Doble `$group` ... 93
`$first` y `$last` ... 94
`$project` ... 94
`$match` ... 95
`$sort` ... 96
`$skip` y `$limit` ... 96
`$unwind` ... 97
Doble `$unwind` ... 98
De SQL al *Pipeline* de Agregaciones ... 99
Limitaciones ... 100
4.3. Agregaciones con *Java* ... 100
4.4. Replicación ... 101
Conjunto de Réplicas ... 101
Elementos de un Conjunto de Réplicas ... 101
oplog ... 102
Creando un Conjunto de Réplicas ... 103
Trabajando con las Réplicas ... 106
Tolerancia a Fallos ... 107
Proceso de Votación ... 108
Cantidad de Elementos ... 108
Comprobando la Tolerancia ... 109
Configuración Recomendada ... 110
Recuperación del Sistema ... 110
Consistencia en la Escritura ... 110
Niveles de consistencia ... 111
4.5. Replicación en *Java* ... 111
Reintento de Operaciones ... 112
Consistencia de Escritura ... 113
Preferencia de Lectura ... 114
`ReadPreference` ... 115
4.6. Particionado (*Sharding*) ... 115
Particionando con *MongoDB* ... 116
Sharded Cluster ... 116
Shard key ... 117
Preparando el *Sharding* con *MongoDB* ... 118

Habilitando el *Sharding* 120
Trabajando con el *Sharding* 122
4.7. Ejercicios 125
(0.75 puntos) Ejercicio 41. Agregaciones 125
(0.5 puntos) Ejercicio 42. Agregaciones en Java 126
(0.75 puntos) Ejercicio 43. Replicación 127
(0.5 puntos) Ejercicio 44. Sharding 127

1. No Sólo SQL

Si definimos **NoSQL** formalmente, podemos decir que se trata de un conjunto de tecnologías que permiten el procesamiento rápido y eficiente de conjuntos de datos dando la mayor importancia al rendimiento, la fiabilidad y la agilidad.

Si nos basamos en el acrónimo, el término se refiere a cualquier almacén de datos que no sigue un modelo relacional, los datos no son relacionales y por tanto no utilizan SQL como lenguaje de consulta. Así pues, los sistemas NoSQL se centran en sistemas complementarios a los SGBD relaciones, que fijan sus prioridades en la escalabilidad y la disponibilidad en contra de la atomicidad y consistencia de los datos.

NoSQL aparece como una necesidad debida al creciente volumen de datos sobre usuarios, objetos y productos que las empresas tienen que almacenar, así como la frecuencia con la que se accede a los datos. Los SGDB relacionales existentes no fueron diseñados teniendo en cuenta la escalabilidad ni la flexibilidad necesaria por las frecuentes modificaciones que necesitan las aplicaciones modernas; tampoco aprovechan que el almacenamiento a día de hoy es muy barato, ni el nivel de procesamiento que alcanzan las maquinas actuales.

Los diferentes tipos de bases de datos NoSQL existentes se pueden agrupar en cuatro categorías:

Clave-Valor

Los almacenes clave-valor son las bases de datos NoSQL más simples. Cada elemento de la base de datos se almacena con un nombre de atributo (o clave) junto a su valor. Los almacenes más conocidos son *Redis*, *Riak* y *Voldemort*. Algunos almacenes, como el caso de *Redis*, permiten que cada valor tenga un tipo (por ejemplo, `integer`) lo cual añade funcionalidad extra.

Documentales

Cada clave se asocia a una estructura compleja que se conoce como documento. Este puede contener diferentes pares clave-valor, o pares de clave-array o incluso documentos anidados, como en un documento JSON. Los ejemplos más conocidos son *MongoDB* y *CouchDB*.

Grafos

Los almacenes de grafos se usan para almacenar información sobre redes, como pueden ser conexiones sociales. Los ejemplos más conocidos son *Neo4J*, *FlockDB*, *InfiniteGraph* y HyperGraphDB.

Basada en columnas

Los almacenes basados en columnas como *BigTable*, *Hadoop*, *Cassandra* y *HBase* están optimizados para consultas sobre grandes conjuntos de datos, y almacenan los datos como columnas, en vez de como filas, de modo que cada fila puede contener un número diferente de columnas.

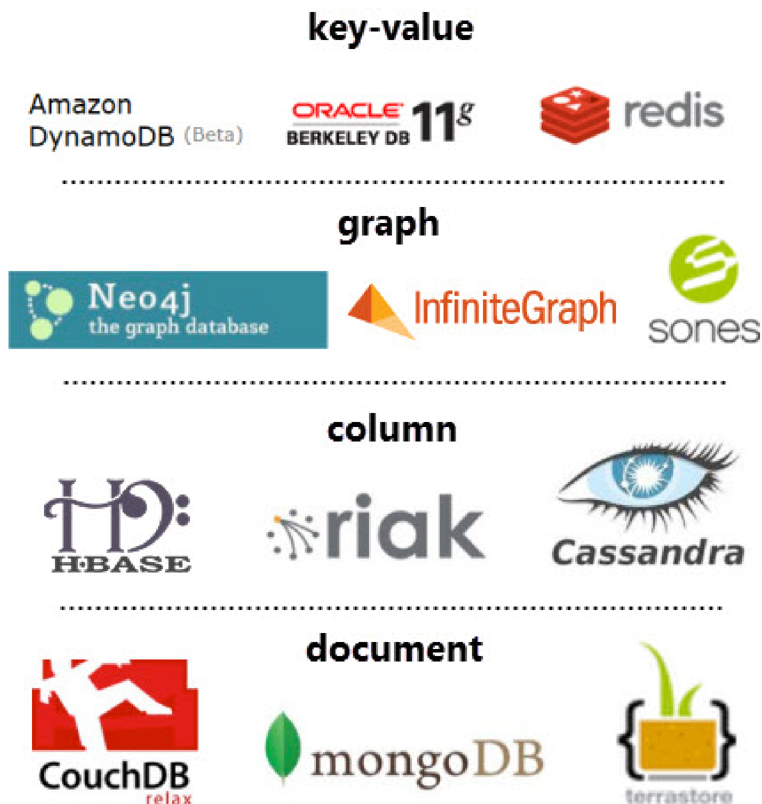


Figura 1. Sistemas NoSQL

Estos tipos los estudiaremos en detalle más adelante, entrando en profundidad tanto con *Redis* como en *MondoDB*.

1.1. Características

Si nos centramos en sus **beneficios** y comparamos con las base de datos relacionales, las bases de datos NoSQL son más escalables, ofrecen un rendimiento mayor y sus modelos de datos resuelven varios problemas que no se plantearon al definir el modelo relacional:

- Grandes volúmenes de datos estructurados, semi-estructurados y sin estructurar. Casi todas las implementaciones NoSQL ofrecen algún tipo de representación para datos sin esquema, lo que permite comenzar con una estructura y con el paso del tiempo, añadir nuevos campos, ya sean sencillos o anidados a datos ya existentes.
- Sprints ágiles, iteraciones rápidas y frecuentes *commits/pushes* de código, al emplear una sintaxis sencilla para la realización de consultas y la posibilidad de tener un modelo que vaya creciendo al mismo ritmo que el desarrollo.
- Arquitectura eficiente y escalable diseñada para trabajar con clusters en vez de una arquitectura monolítica y costosa. Las soluciones NoSQL soportan la escalabilidad de un modo transparente para el desarrollador.

Esquemas dinámicos

Las bases de datos relacionales requieren definir los esquemas antes de añadir los datos. Una base de datos SQL necesita saber de antemano los datos que vamos a para almacenar; por ejemplo, si nos centramos en los datos de un cliente, serían el nombre, apellidos, número de teléfono, etc…

Esto casa bastante mal con los enfoques de desarrollo ágil, ya que cada vez que añadimos nuevas funcionalidades, el esquema de la base de datos suele cambiar. De modo que si a mitad de desarrollo decidimos almacenar los productos favoritos de un cliente del cual guardábamos su dirección y números de teléfono, tendríamos que añadir una nueva columna a la base de datos y migrar la base de datos entera a un nuevo esquema.

Si la base de datos es grande, conlleva un proceso lento que implica parar el sistema durante un tiempo considerable. Si frecuentemente cambiamos los datos que la aplicación almacena (al usar un desarrollo iterativo), también tendremos períodos frecuentes de inactividad del sistema. Así pues, no hay un modo efectivo mediante una base de datos relacional, de almacenar los datos que están desestructurados o que no se conocen de antemano.

Las bases de datos *NoSQL* se construyen para permitir la inserción de datos sin un esquema predefinido. Esto facilita la modificación de la aplicación en tiempo real, sin preocuparse por interrupciones de servicio.

De este modo se consigue un desarrollo más rápido, integración de código más robusto y menos tiempo empleado en la administración de la base de datos.

Particionado

Dado el modo en el que se estructuran las bases de datos relacionales, normalmente escalan verticalmente - un único servidor que almacena toda la base de datos para asegurar la disponibilidad continua de los datos. Esto se traduce en costes que se incrementan rápidamente, con un límites definidos por el propio hardware, y en un pequeño número de puntos críticos de fallo dentro de la infraestructura de datos.

La solución es escalar horizontalmente, añadiendo nuevos servidores en vez de concentrarse en incrementar la capacidad de un único servidor. Este escalado horizontal se conoce como **Sharding** o *Particionado*.

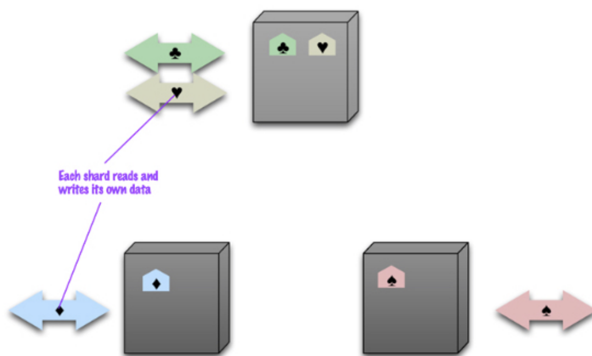


Figura 2. Particionado

El particionado no es único de las bases de datos NoSQL. Las bases de datos relacionales también lo soportan. Si en un sistema relacional queremos particionar los datos, podemos distinguir entre particionado:

- **Horizontal**: diferentes filas en diferentes particiones.
- **Vertical**: diferentes columnas en particiones distintas.

En el caso de las bases de datos *NoSQL*, el particionado depende del modelo de la base de datos:

- Los almacenes clave-valor y las bases de datos documentales normalmente se particionan horizontalmente.
- Las bases de datos basados en columnas se pueden particionar horizontal o verticalmente.

Escalar horizontalmente una base de datos relacional entre muchas instancias de servidores se puede conseguir pero normalmente conlleva el uso de SANs [1] y otras triquiñuelas para hacer que el hardware actúe como un único servidor.

Como los sistemas SQL no ofrecen esta prestación de forma nativa, los equipos de desarrollo se las tienen que ingeniar para conseguir desplegar múltiples bases de datos relacionales en varias máquinas. Para ello:

- Los datos se almacenan en cada instancia de base de datos de manera autónoma
- El código de aplicación se desarrolla para distribuir los datos y las consultas y agregar los resultados de los datos a través de todas las instancias de bases de datos
- Se debe desarrollar código adicional para gestionar los fallos sobre los recursos, para realizar *joins* entre diferentes bases de datos, balancear los datos y/o replicarlos, etc…

Además, muchos beneficios de las bases de datos como la integridad transaccional se ven comprometidos o incluso eliminados al emplear un escalado horizontal.

Auto-Sharding

Por contra, las bases de datos NoSQL normalmente soportan *auto-sharding*, lo que implica que de manera nativa y automáticamente se dividen los datos entre un número arbitrario de servidores, sin que la aplicación sea consciente de la composición del *pool* de servidores. Los datos y las consultas se balancean entre los servidores.

El particionado se realiza mediante un método consistente, como puede ser:

- Por **rangos** de su id: por ejemplo *"los usuarios del 1 al millón están en la partición 1"* o *"los usuarios cuyo nombre va de la A a la E"* en una partición, en otra de la M a la Q, y de la R a la Z en la tercera.
- Por **listas**: dividiendo los datos por la categoría del dato, es decir, en el caso de datos sobre libros, las novelas en una partición, las recetas de cocina en otra, etc..
- Mediante un función **hash**, la cual devuelve un valor para un elemento que determine a que partición pertenece.

Cuando Particionar

El motivo para particionar los datos se debe a:

- limitaciones de almacenamiento: los datos no caben en un único servidor, tanto a nivel de disco como de memoria RAM.
- rendimiento: al balancear la carga entre particiones las escrituras serán más rápidas que al centrarlas en un único servidor.
- disponibilidad: si un servidor esta ocupado, otro servidor puede devolver los datos. La carga de los servidores se reduce.

No particionaremos los datos cuando la cantidad sea pequeña, ya que el hecho de distribuir los datos conlleva unos costes que pueden no compensar con un volumen de datos insuficiente.

[1] Storage Area Networks

Tampoco esperaremos a particionar cuando tengamos muchísimos datos, ya que el proceso de particionado puede provocar sobrecarga del sistema.

La *nube* facilita de manera considerable este escalado, mediante proveedores como *Amazon Web Services* el cual ofrece virtualmente una capacidad ilimitada bajo demanda, y preocupándose de todas las tareas necesarias para la administración de la base de datos.

Los desarrolladores ya no necesitamos construir plataformas complejas para nuestras aplicaciones, de modo que nos podemos centrar en escribir código de aplicación. Una granja de servidores puede ofrecer el mismo procesamiento y capacidad de almacenamiento que un único servidor de alto rendimiento por mucho menos coste.

Replicación

La replicación mantiene copias idénticas de los datos en múltiples servidores, lo que facilita que las aplicaciones siempre funcionen y los datos se mantengan seguros, incluso si alguno de los servidores sufre algún problema.

La mayoría de las bases de datos NoSQL también soportan la replicación automática, lo que implica una alta disponibilidad y recuperación frente a desastres sin la necesidad de aplicaciones de terceros encargadas de ello. Desde el punto de vista del desarrollador, el entorno de almacenamiento es virtual y ajeno al código de aplicación.

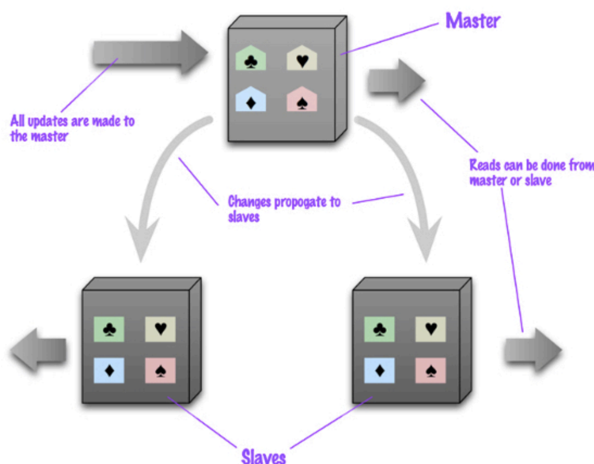
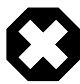


Figura 3. Replicación

La replicación de los datos se utiliza para alcanzar:

- **escalabilidad**, incrementando el rendimiento al poder distribuir las consultas en diferentes nodos, y mejorar la rendundancia al permitir que cada nodo tenga una copia de los datos.
- **disponibilidad**, ofreciendo tolerancia a fallos de hardware o corrupción de la base de datos. Al replicar los datos vamos a poder tener una copia de la base de datos, dar soporte a un servidor de datos agregados, o tener nodos a modo de copias de seguridad que pueden tomar el control en caso de fallo.
- **aislamiento** (la i en ACID - *isolation*), entendido como la propiedad que define cuando y cómo al realizar cambios en un nodo se propagan al resto de nodos. Si replicamos los datos podemos crear copias sincronizadas que separar procesos de la base de datos de producción, pudiendo ejecutar informes o copias de seguridad en nodos secundarios de modo que no tenga un impacto negativo en el nodo principal, así como ofrecer un sistema sencillo para separar el entorno de producción del de preproducción.



No hay que confundir la replicación (copia de los datos en varias máquinas) con la partición (cada máquina tiene un subconjunto de los datos). El entorno más seguro y con mejor rendimiento es aquel que tiene los datos particionados y replicados (cada maquina que tiene un subconjunto de los datos está replicada en 2 o más).

Caché integrada

Otra característica que comparten los sistemas NoSQL es que ofrecen un mecanismo de caché de datos integrado, de manera que se puede configurar los sistemas para que los datos se mantengan en memoria y se persistan de manera periodica. El uso de una caché conlleva que la consistencia de los datos no sea completa y podamos tener una consistencia eventual.

Existen diferentes productos que ofrecen un mecanismo de caché para los sistemas de bases de datos relacionales. Estos sistemas pueden incrementar el rendimiento de las lecturas de manera sustancial, pero no mejoran el rendimiento de las escrituras, y añaden un capa de complejidad al despliegue del sistema. Si en una aplicación predominan las lecturas, una cache distribuida como *MemCached* (http://www.memcached.org) puede ser una buena solución. Pero si en la aplicación las escrituras son más frecuentes, o se reparten al 50%, una caché distribuida puede no mejorar la experiencia global de los usuarios finales.

Muchas tecnologías de bases de datos relacionales tienen excelentes capacidades de caché integradas en sus soluciones, manteniendo en memoria los datos con mayor uso todo el tiempo posible y desechando la necesidad de una capa aparte a mantener.

1.2. NoSQL vs. SQL

A continuación vamos a comparar ambos sistemas:

Tabla 1. Comparación SQL vs NoSQL

	Bases de Datos SQL	Bases de Datos NoSQL
Tipos	Un tipo con pequeñas variantes	Muchos tipos distintos como almacenes clave-valor, base de datos documentales, basado en columnas o en grafos
Historia	Desarrollado en los 70 para tratar la primera hornada de aplicaciones que almacenaban datos	Desarrollado en el 2000 para ayudar a resolver las limitaciones de las bases de datos SQL, particularmente lo relacionado con la escalabilidad, replicación y almacenamiento de datos desestructurados
Ejemplos	MySQL, PostgreSQL, Oracle	MongoDB, Cassandra, HBase, Neo4j
Modelo de almacenamiento	Los registros individuales (por ejemplo, empleados) se almacenan como filas en tablas, donde cada columna almacena un atributo específico de ese registro (categoría, fecha de ingreso, etc..), similar a la forma tabular de una hoja de cálculo. Datos de tipos	Varía dependiendo del tipo de base de datos. Por ejemplo, en los almacenes clave-valor se trabaja de manera similar a las bases de datos relacionales, pero con solo 2 columnas ("clave" y "valor"), y usando el campo "valor" para almacenar información más

	Bases de Datos SQL	Bases de Datos NoSQL
	distintos se encuentran en tablas distintas, que se pueden unir al ejecutar consultas más complejas	compleja. En cambio, las bases de datos documentales descartan el modelo de tabla para almacenar toda la información relevante en un único documento con JSON, XML o cualquier otro formato que permita anidar los valores de manera jerarquica.
Esquemas	La estructura y los tipos de datos se conocen de antemano. Para almacenar información sobre un nuevo atributo se debe modificar la base de datos al completo, tiempo durante el cual la base de datos queda inactiva	Normalmente dinámica. Los registros pueden añadir información en caliente y almacenar información con diferente estructura en un mismo campo. En algunas bases de datos como los almacenes basados en columnas es un poco más costoso.
Escalado	Vertical: un único servidor que debe incrementar su potencia para tratar la demanda creciente. Se pueden repartir bases de datos SQL entre muchos servidores, pero requiere una configuración especial y desarrollo teniendo en cuenta esta configuración	Horizontal: para añadir capacidad, el administrador de bases de datos simplemente puede añadir más servidores virtuales o instancias en la nube. La base de datos se propaga de manera automática entre los diferentes servidores añadidos.
Modelo de Desarrollo	Repartido entre *open source* (p.ej. Postgres, MySQL) y software propietario (p.ej., Oracle, DB2)	*Open source*
Soporte de transacciones	Si, las modificaciones se pueden configurar para que se realizan todas o ninguna	En algunas circunstancias y ciertos niveles (a nivel de documento *vs* a nivel a de base de datos)
Manipulación de Datos	SQL	Mediante APIs orientadas a objetos
Consistencia	Configurable para consistencia alta	Depende del producto. Algunos ofrecen gran consistencia (p.ej. MongoDB) mientras otros ofrecen consistencia eventual (p.ej. Cassandra)

1.3. Implantando NoSQL

Normalmente, las empresas empezarán con una prueba de baja escalabilidad de una base de datos NoSQL, de modo que les permita comprender la tecnología asumiendo muy poco riesgo. La mayoría de las bases de datos NoSQL también son *open-source*, y por tanto se pueden probar sin ningún coste extra. Al tener unos ciclos de desarrollo más rápidos, las empresas pueden innovar con mayor velocidad y mejorar la experiencia de sus cliente a un menor coste.

Elegir la base de datos correcta para el proyecto es un tema importante. Se deben considerar las diferentes alternativas a las infraestructuras *legacy* teniendo en cuenta varios factores:

- la escalabilidad o el rendimiento más allá de las capacidades del sistema existente

- identificar alternativas viables respecto al software propietario
- incrementar la velocidad y agilidad del proceso de desarrollo

Así pues, al elegir un base de datos hemos de tener en cuenta las siguientes dimensiones:

- **Modelo de Datos**: A elegir entre un modelo documental, basado en columnas, de grafos o mediante clave-valor.
- **Modelo de Consultas**: Dependiendo de la aplicación, puede ser aceptable un modelo de consultas que sólo accede a los registros por su clave primaria. En cambio, otras aplicaciones pueden necesitar consultar por diferentes valores de cada registro. Además, si la aplicación necesita modificar los registros, la base de datos necesita consultar los datos por un índice secundario.
- **Modelo de Consistencia**: Los sistemas NoSQL normalmente mantienen múltiples copias de los datos para ofrecer disponibilidad y escalabilidad al sistema, lo que define la consistencia del mismo. Los sistemas NoSQL tienden a ser consistentes o eventualmente consistentes.
- **APIs**: No existe un estándar para interactuar con los sistemas *NoSQL*. Cada sistema presenta diferentes diseños y capacidades para los equipos de desarrollo. La madurez de un API puede suponer una inversión en tiempo y dinero a la hora de desarrollar y mantener el sistema NoSQL.
- **Soporte Comercial y de la Comunidad**: Los usuarios deben considerar la salud de la compañia o de los proyectos al evaluar una base de datos. El producto debe evolucionar y se mantenga para introducir nuevas prestaciones y corregir fallos. Una base de datos con una comunidad fuerte de usuarios:

  # permite encontrar y contratar desarrolladores con destrezas en el producto.
  # facilita encontrar información, documentación y ejemplos de código.
  # ayuda a las empresas a retener el talento.
  # favorece que otras empresas de software integren sus productos y participen en el ecosistema de la base de datos.

Casos de Uso

Una vez conocemos los diferentes sistemas y qué elementos puede hacer que nos decidamos por una solución u otra, conviene repasar los casos de uso más comunes:

- Si vamos a crear una aplicación web cuyo campos sean personalizables, usaremos una solución documental.
- Como una capa de caché, mediante un almacén clave-valor.
- Para almacenar archivos binarios sin preocuparse de la gestión de permisos del sistema de archivos, y porder realizar consultas sobre sus metadatos, ya sea mediante una solución documental o un almacén clave-valor.
- Para almacenar un enorme volumen de datos, donde la consistencia no es lo más importante, pero si la disponibilidad y su capacidad de ser distribuida.

1.4. Modelos de Datos

La principal clasificación de los sistemas de BBDD NoSQL se realiza respecto a los diferentes modelos de datos:

Documental

Mientras las bases de datos relacionales almacenan los datos en filas y columnas, las bases de datos documentales emplean documentos. Estos documentos utilizan una estructura JSON, ofreciendo un modo natural e intuitivo para modelar datos de manera similar a la orientación a objetos, donde cada documento es un objeto.

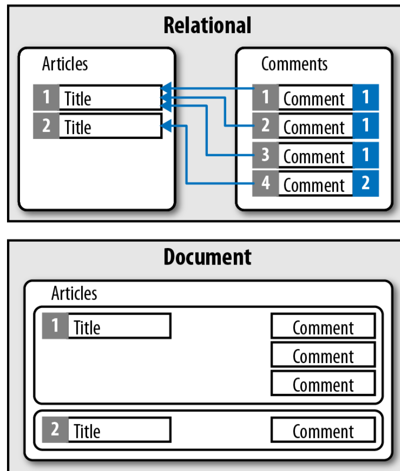


Figura 4. Representación de un documento

Los documentos se agrupan en colecciones o bases de datos, dependiendo del sistema, lo que permite agrupar documentos.

Los documentos contienen uno o más campos, donde cada campo contiene un valor con un tipo, ya sea cadena, fecha, binario o *array*. En vez de extender los datos entre múltiples columnas y tablas, cada registro y sus datos asociados se almacenan de manera unida en un único documento. Esto simplifica el acceso a los datos y reduce (y en ocasiones elimina) la necesidad de *joins* y transacciones complejas.

Características

En una base de datos documental, la noción de esquema es dinámico: cada documento puede contener diferentes campos. Esta flexibilidad puede ser útil para modelar datos desestructurados y polimórficos, lo que facilita la evolución del desarrollo al permitir añadir nuevos campos de manera dinámica.

Por ejemplo, podemos tener dos documentos que pertenecen a la misma colección, pero con atributos diferentes:

Datos de dos empleados

```
{
  "EmpleadoID": "BW1",
  "Nombre"    : "Bruce",
  "Apellido"  : "Wayne",
  "Edad"      : 35,
  "Salario"   : 10000000
}

{
  "EmpleadoID": "JK1",
```

```
    "Nombre"    : "Joker",
    "Edad"      : 34,
    "Salary"    : 5000000,
    "Direccion" : {
      "Lugar"   : "Asilo Arkham",
      "Ciudad"  : "Gotham"
    },
    "Proyectos" : [
      "desintoxicacion-virus",
      "top-secret-007"
    ]
  }
```

Normalmente, cada documento contiene un elemento clave, sobre el cual se puede obtener un documento de manera unívoca. De todos modos, las bases de datos documentales ofrecen un completo mecanismo de consultas, posibilitando obtener información por cualquier campo del documento. Algunos productos ofrecen opciones de indexado para optimizar las consultas, como pueden ser índices compuestos, dispersos, con tiempo de vida (TTL), únicos, de texto o geoespaciales.

Además, estos sistemas ofrecen productos que permiten analizar los datos, mediante funciones de agregación o implementación de *MapReduce*.

Respecto a la modificaciones, los documentos se pueden actualizar en una única sentencia, sin necesidad de dar rodeos para elegir los datos a modificar.

Casos de Uso

Las bases de datos documentales sirven para propósito general, válidos para un amplio abanico de aplicaciones gracias a la flexibilidad que ofrece el modelo de datos, lo que permite consultar cualquier campo y modelar de manera natural de manera similar a la programación orientada a objetos.

Entre los casos de éxito de estos sistemas cabe destacar:

- Sistemas de flujo de eventos: entre diferentes aplicaciones dentro de una empresa
- Gestores de Contenido, plataformas de Blogging: al almacenar los documentos mediante JSON, facilita la estructura de datos para guardar los comentarios, registros de usuarios, etc…
- Analíticas Web, datos en Tiempo Real: al permitir modificar partes de un documento, e insertar nuevos atributos a un documento cuando se necesita una nueva métrica
- Aplicaciones eCommerce: conforme las aplicaciones crecen, el esquema también lo hace

Si nos centramos en aquellos casos donde no conviene este tipo de sistemas podemos destacar:

- Sistemas operaciones con transacciones complejas
- Sistemas con consultas agregadas que modifican su estructura. Si los criterios de las consultas no paran de cambiar, acabaremos normalizando los datos.

Los productos más destacados son:

- *MongoDB*: http://www.mongodb.com
- *CouchDB*: http://couchdb.apache.org

Al final de la sesión y en el resto del módulo instalaremos, configuraremos y utilizaremos *MongoDB* en profundidad.

Clave-Valor

Un almacén clave-valor es una simple tabla *hash* donde todos los accesos a la base de datos se realizan a través de la clave primaria.

Desde una perspectiva de modelo de datos, los almacenes de clave-valor son los más básicos.

Su funcionamiento es similar a tener una tabla relacional con dos columnas, por ejemplo `id` y `nombre`, siendo `id` la columna utilizada como clave y `nombre` como valor. Mientras que en una base de datos en el campo `nombre ` sólo podemos almacenar datos de tipo cadena, en un almacén clave-valor, el valor puede ser de un dato simple o un objeto. Cuando una aplicación accede mediante la clave y el valor, se almacenan el par de elementos. Si la clave ya existe, el valor se modifica.

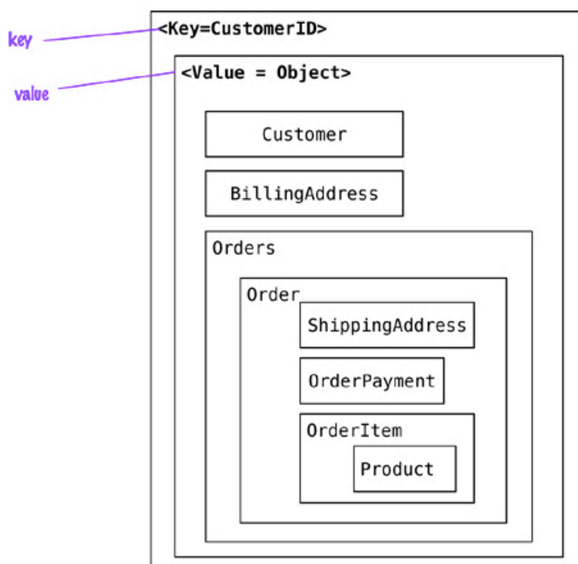


Figura 5. Representación de un almacén *key-value*

El cliente puede tanto obtener el valor por la clave, asignar un valor a una clave o eliminar una clave del almacén. El valor, sin embargo, es opaco al sistema, el cual no sabe que hay dentro de él, ya que los datos sólo se pueden consultar por la clave, lo cual puede ser un inconveniente. Así pues, la aplicación es responsable de saber qué hay almacenado en cada valor.

Interactuando con Riak medianta Java

```
Bucket bucket = getBucket(bucketName);   ❶
IRiakObject riakObject = bucket.store(key, value).execute();

IRiakObject riakObject = bucket.fetch(key).execute();
byte[] bytes = riakObject.getValue();
String value = new String(bytes);
```

❶ Riak utiliza el concepto de *bucket* (cubo) como una manera de agrupar claves, de manera similar a una tabla

Interactuando con Riak medianta HTTP

```
curl -v -X PUT http://localhost:8091/riak/heroes/ace -H "Content-Type:
 application/json" -d {"nombre" : "Batman", "color" : "Negro"}
```

Algunos almacenes clave-valor, como puede ser *Redis*, permiten almacenar datos con cualquier estructura, como por ejemplos listas, conjuntos, *hashes* y pueden realizar operaciones como intersección, unión, diferencia y rango.

Interactuando con Redis

```
SET nombre "Bruce Wayne"      //String
HSET heroe nombre "Batman"    // Hash – set
HSET heroe color "Negro"
SADD "heroe:amigos" "Robin" "Alfred"   // Set – create/update
```

Estas prestaciones hacen que *Redis* se extrapole a ámbitos ajenos a un almacén clave-valor. Otra característica que ofrecen algunos almacenes es que permiten crear un segundo nivel de consulta o incluso definir más de una clave para un mismo objeto.

Como los almacenes clave-valor siempre utilizan accesos por clave primaria, de manera general tienen un gran rendimiento y son fácilmente escalables.

Si queremos que su rendimiento sea máximo, pueden configurarse para que mantegan la información en memoria y que se serialice de manera periódica, a costa de tener una consistencia eventual de los datos.

Casos de Uso

Este modelo es muy útil para representar datos desestructurados o polimórficos, ya que no fuerzan ningún esquema más allá de los pares de clave-valor.

Entre los casos de uso de estos almacenes podemos destacar el almacenaje de:

- Información sobre la sesión de navegación (`sessionid`)
- Perfiles de usuario, preferencias
- Datos del carrito de la compra
- Cachear datos

Todas estas operaciones van a asociada a operaciones de recuperación, modificación o inserción de los datos de una sola vez, de ahí su elección.

En cambio, no conviene utilizar estos almacenes cuando queremos realizar:

- Relaciones entre datos
- Transacciones entre varias operaciones
- Consultas por los datos del valor
- Operaciones con conjuntos de claves

Los almacenes más empleados son:

- *Riak*: http://basho.com/riak/
- *Redis*: http://redis.io
- *Voldemort*: https://github.com/voldemort/voldemort implementación *open-source* de *Amazon DynamoDB* http://aws.amazon.com/dynamodb

Basado en Columnas

También conocidos como sistemas *Big Data* o tabulares. Su nombre viene tras la implementación de *Google* de *BigTable* (http://research.google.com/archive/bigtable.html), el cual consiste en columnas separadas y sin esquema, a modo de mapa de dos niveles.

Las bases de datos relacionales utilizan la fila como unidad de almacenamiento, lo que permite un buen rendimiento de escritura. Sin embargo, cuando las escrituras son ocasionales y es más comun tener que leer unas pocas columnas de muchas filas a la vez, es mejor utilizar como unidad de almacenamiento a grupos de columnas.

Un modelo basado en columnas se representa como una estructura agregada de dos niveles. El primer nivel formado por un almacén clave-valor, siendo la clave el identificador de la fila, y el valor un nuevo mapa con los datos agregados de la fila (familias de columnas). Los valores de este segundo nivel son las columnas. De este modo, podemos acceder a los datos de un fila, o a una determinada columna:

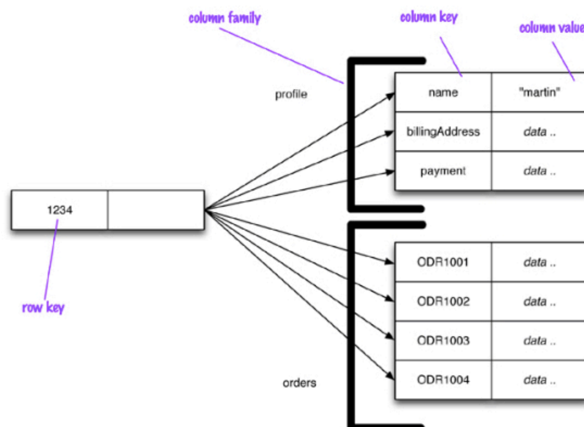


Figura 6. Representación de un almacén basado en columnas

Los almacenes basados en columnas utilizan un mapa ordenado multi-dimensional y distribuido para almacenar los datos. Están pensados para que cada fila tenga una gran número de columnas (del orden del millón), almacenando las diferentes versiones que tenga una fila (pudiendo almacenar del orden de miles de millones de filas).

Familias de Columnas

Una columna consiste en un pareja `name` - `value`, donde el nombre hace de clave. Además, contiene un atributo `timestamp` para poder expirar datos y resolver conflictos de escritura.

Ejemplo de columna

```
{
  name: "nombre",
  value: "Bruce",
```

```
  timestamp: 12345667890
}
```

Una fila es una colección de columnas agrupadas a una clave. Si agrupamos filas similares tendremos una **familia de columnas**:

```
// familia de columnas
{
  // fila
  "tim-gordon" : {
    nombre: "Tim",
    apellido: "Gordon",
    ultimaVisita: "2015/12/12"
  }
  // fila
  "bruce-wayne" : {
    nombre: "Bruce",
    apellido: "Wayne",
    lugar: "Gotham"
  }
}
```

Cada registro puede variar en el número de columnas con el que se almacena, y las columnas se pueden anidar dentro de otras formando **super-columnas**, donde el valor es un nuevo mapa de columnas.

```
{
  name: "libro:978-84-16152-08-7",
  value: {
    autor: "Grant Morrison",
    titulo: "Batman - Asilo Arkham",
    isbn: "978-84-16152-08-7"
  }
}
```

Cuando se utilizan super columnas para crear familias de columnas tendremos una familia de super columnas.

En resumen, las bases de datos basadas en columnas, almacenan los datos en familias de columnas como filas, las cuales tienen muchas columnas asociadas al identificador de una fila. Las familias de columnas son grupos de datos relacionados, a las cuales normalmente se accede de manera conjunta.

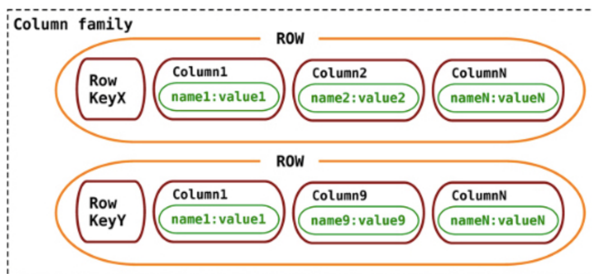


Figura 7. Familia de Columnas

Operaciones

A la hora de consultar los datos, éstos se pueden obtener por la clave primaria de la familia. Así pues, podemos obtener toda una familia, o la columna de una familia:

```
// Mediante Cassandra
GET Cliente['bruce-wayne'];  // familia
GET Cliente['bruce-wayne']['lugar']; // columna
```

Algunos productos ofrecen un soporte limitado para índices secundarios, pero con restricciones. Por ejemplo, *Cassandra* ofrece el lenguaje CQL similar a SQL pero sin *joins*, ni subconsultas donde las restricciones de `where` son sencillas:

```
SELECT * FROM Clientes
SELECT nombre,email FROM Clientes
SELECT nombre,email FROM Clientes WHERE lugar='Gotham'
```

Las actualizaciones se realizan en dos pasos: primero encontrar el registro y segundo modificarlo. En estos sistemas, una modificación puede suponer una reescritura completa del registro independientemente que hayan cambiado unos pocos *bytes* del mismo.

Casos de Uso

De manera similar a los almacenes clave-valor, el mercado de estos sistemas son las aplicaciones que sólo necesitan consultar los datos por un único valor. En cambio, estas aplicaciones centran sus objetivos en el rendimiento y la escalabilidad.

Entre los casos de uso destacamos:

- Sistemas de flujo de eventos: para almacenar estados de las aplicaciones o errores de las mismas.
- Gestores de Contenido, plataformas de Blogging: mediante familias de columnas podemos almacenar las entradas y las etiquetas, categorías, enlaces, *trackbacks* en columnas. Los comentarios se pueden almacenar en la misma fila o en otra base de datos.
- Contadores: para poder almacenar las visitas de cada visitante a cada apartado de un *site*

Si nos centramos en aquellos casos donde no conviene este tipo de sistemas podemos destacar:

- Sistemas operacionales con transacciones complejas
- Sistemas con consultas agregadas. Si los criterios de las consultas no paran de cambiar, acabaremos normalizando los datos.
- Prototipado inicial o sistemas donde el esquema no esté fijado de antemano, ya que las consultas dependen del diseño de las familias de columnas.

Los productos más destacados son:

- *HBase* : http://hbase.apache.org, el cual se basa en *Hadoop* - http://hadoop.apache.org
- *Cassandra* : http://cassandra.apache.org
- *Amazon SimpleDB*: http://aws.amazon.com/simpledb

Grafos

Las bases de datos de grafos almacenan entidades y las relaciones entre estas entidades. Las entidades se conocen como nodos, los cuales tienen propiedades. Cada nodo es similar a una instancia de un objeto. Las relaciones, también conocidas como vértices, a su vez tienen propiedades, y su sentido es importante.

Los nodos se organizan mediante relaciones que facilitan encontrar patrones de información existente entre los nodos. Este tipo de organización permite almacenar los datos una vez e interpretar los datos de diferentes maneras dependiendo de sus relaciones.

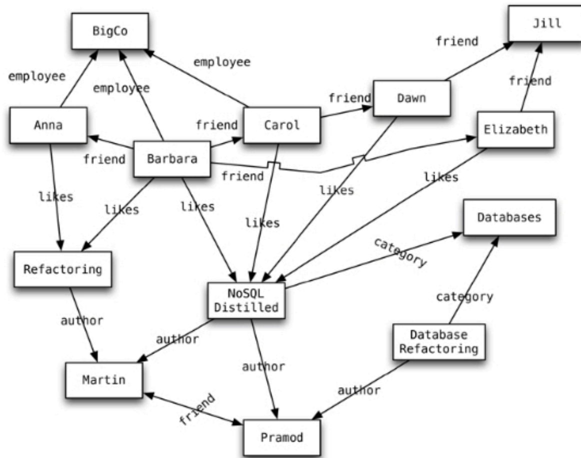


Figura 8. Estructura de Grafo

Los nodos son entidades que tienen propiedades, tales como el nombre. Por ejemplo, en el gráfico cada nodo tiene una propiedad nombre. También podemos ver que las relaciones tienen tipos, como *likes*, *author*, etc… Estas propiedades permiten organizar los nodos. Las relaciones pueden tener múltiples propiedades, y además tienen dirección, con lo cual si queremos incluir bidireccionalidad tenemos que añadir dos relaciones en sentidos opuestos.

Creando un grafo mediante Neo4J

```
Node martin = graphDb.createNode();
martin.setProperty("name", "Martin");
Node pramod = graphDb.createNode();
pramod.setProperty("name", "Pramod");

martin.createRelationshipTo(pramod, FRIEND);
pramod.createRelationshipTo(martin, FRIEND);
```

Los nodos permiten tener diferentes tipos de relaciones entre ellos y así representar relaciones entre las entidades del dominio, y tener relaciones secundarias para características como categoría, camino, árboles de tiempo, listas enlazas para acceso ordenado, etc… Al no existir un límite en el número ni en el tipo de relaciones que puede tener un nodo, todas se pueden representar en la misma base de datos.

Traversing

Una vez tenemos creado un grafo de nodos y relaciones, podemos consultar el grafo de muchas maneras; por ejemplo "obtener todos los nodos que trabajan para *Big Co* y que les

gusta *NoSQL Distilled*". Realizar una consulta se conoce como hacer un *traversing* (recorrido) del mismo.

Ejemplo de *Traversing* mediante Neo4J

```
Node martin = nodeIndex.get("name", "Martin").getSingle();
allRelationships = martin.getRelationships(Direction.INCOMING);
```

Una ventaja a destacar de las bases de datos basadas en grafos es que podemos cambiar los requisitos de *traversing* sin tener que cambiar los nodos o sus relaciones.

En las bases de datos de grafos, recorrer las relaciones es muy rápido, ya que no se calculan en tiempo de consulta, sino que se persisten como una relación, y por tanto no hay que hacer ningún cálculo.

En cambio, en una base de datos relacional, para crear una estructura de grafo se realiza para una relación sencilla (¿Quien es mi jefe?"). Para poder añadir otras relaciones necesitamos muchos cambios en el esquema y trasladar datos entre tablas. Además, necesitamos de antemano saber que consultar queremos realizar para modelar las tablas y las relaciones acorde a las consultas.

Así pues, estos sistemas ofrecen ricos modelos de consultas donde se pueden investigar las relaciones simples y complejas entre los nodos para obtener información directa e indirecta de los datos del sistemas. Los tipos de análisis que se realizan sobre estos sistema se ciñen a los tipos de relación existente entre los datos.

Casos de Uso

Mientras que el modelo de grafos no es muy intuitivo y tiene una importante curva de aprendizaje, se puede usar en un gran número de aplicaciones.

Su principal atractivo es que facilitan almacenar las relaciones entre entidades de una aplicación, como por ejemplo de una red social, o las intersecciones existentes entre carreteras. Es decir, se emplean para almacenar datos que se representan como nodos interconectados.

Por lo tanto, los casos de uso son:

- Datos conectados: redes sociales con diferentes tipos de conexiones entre los usuarios.
- Enrutamiento, entrega o servicios basados en la posición: si las relaciones almacenan la distancia entre los nodos, podemos realizar consultas sobre lugares cercanos, trayecto más corto, etc…
- Motores de recomendaciones: de compras, de lugares visitados, etc…

En cambio, no se recomienda su uso cuando necesitemos modificar todos o un subconjunto de entidades, ya que modificar una propiedad en todos los nodos es una operación compleja.

Los productos más destacados son:

- *Neo4j*: http://neo4j.com
- *FlockDB*: https://github.com/twitter/flockdb

- *HyperGraphDB*: http://www.hypergraphdb.org/index

1.5. Consistencia

En un sistema consistente, las escrituras de una aplicación son visibles en siguientes consultas. Con una consistencia eventual, las escrituras no son visibles inmediatamente.

Por ejemplo, en un sistema de control de *stock*, si el sistema es consistente, cada consulta obtendrá el estado real del inventario, mientras que si tiene consistencia eventual, puede que no sea el estado real en un momento concreto pero terminará siéndolo en breve.

Sistemas Consistentes

Cada aplicación tiene diferentes requisitos para la consistencia de los datos. Para muchas aplicaciones, es imprescindible que los datos sean consistentes en todo momento. Como los equipos de desarrollo han estado trabajo con un modelo de datos relacional durante décadas, este enfoque parece natural. Sin embargo, en otras ocasiones, la consistencia eventual es un traspiés aceptable si conlleva una mayor flexibilidad en la disponibilidad del sistema.

Las bases de datos documentales y de grafos pueden ser consistentes o eventualmente consistentes. Por ejemplo, *MongoDB* ofrece un consistencia configurable. De manera predeterminada, los datos son consistentes, de modo que todas las escrituras y lecturas se realizan sobre la copia principal de los datos. Pero como opción, las consultas de lectura, se pueden realizar con las copias secundarias donde los datos tendrán consistencia eventual. La elección de la consistencia se realiza a nivel de consulta.

Sistemas de Consistencia Eventual

Con los sistemas eventualmente consistentes, hay un período de tiempo en el que todas las copias de los datos no están sincronizados. Esto puede ser aceptable para aplicaciones de sólo-lectura y almacenes de datos que no cambian frecuentemente, como los archivos históricos. Dentro del mismo saco podemos meter las aplicaciones con alta tasa de escritura donde las lecturas sean poco frecuentes, como un archivo de *log*.

Un claro ejemplo de sistema eventualmente consistente es el servicio DNS, donde tras registrar un dominio, puede tardar varios días en propagar los datos a través de Internet, pero siempre están disponibles aunque contenga una versión antigua de los datos.

Respecto a las bases de datos NoSQL, los almacenes de clave-valor y los basados en columnas son sistemas eventualmente consistentes. Estos tienen que soportar conflictos en las actualizaciones de registros individuales.

Como las escrituras se pueden aplicar a cualquier copia de los datos, puede ocurrir, y no sería muy extraño, que hubiese un conflicto de escritura.

Algunos sistemas como *Riak* utilizan vectores de reloj para determinar el orden de los eventos y asegurar que la operación más reciente gana en caso de un conflicto.

Otros sistemas como *CouchDB*, retienen todos los valores conflictivos y permiten al usuario resolver el conflicto. Otro enfoque seguido por *Cassandra* sencillamente asume que el valor más grande es el correcto.

Por estos motivos, las escrituras tienden a comportarse bien en sistemas eventualmente consistentes, pero las actualizaciones pueden conllevar sacrificios que complican la aplicación.

1.6. Teorema de CAP

Propuesto por *Eric Brewer* en el año 2000, prueba que podemos crear una base de datos distribuida que elija dos de las siguientes tres características:

- **C**onsistencia: las escrituras son atómicas y todas las peticiones consecuentes obtienen el nuevo valor, independientemente del lugar de la petición
- Disponibilidad (***A**vailable*): la base de datos devolverá siempre un valor siempre que el servidor esté ejecutándose.
- Tolerancia a **P**articiones: el sistema funcionará incluso si la comunicación con un servidor se interrumpe de manera temporal, con lo cual divide los datos entre diferentes nodos.

En otras palabras, podemos crear un sistema de base de datos distribuido que sea consistente y tolerante a particiones (**CP**), un sistema que sea disponible y tolerante a particiones (**AP**), o un sistema que sea consistente y disponible (**CA**). Pero no es posible crear una base de datos distribuida que sea consistente, disponible y tolerante a particiones al mismo tiempo.

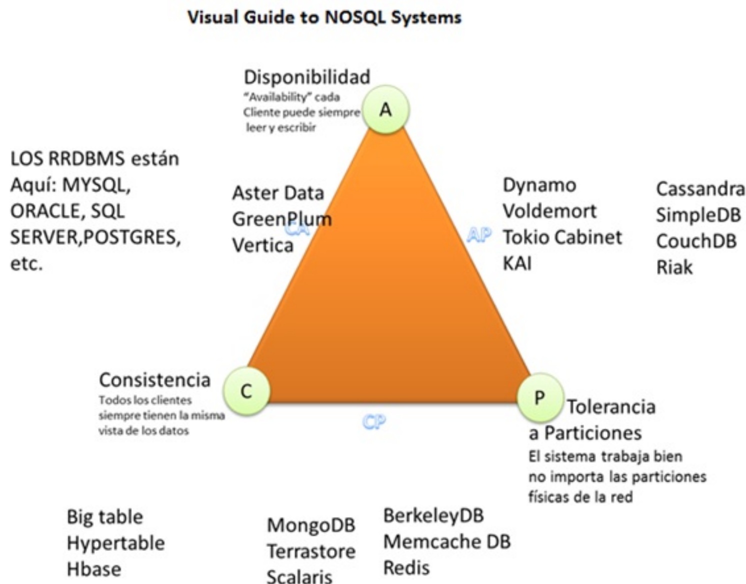
El teorema CAP es útil cuando consideramos el sistema de base de datos que necesitamos, ya que debemos decidir cual de las tres características vamos a descartar. La elección realmente se centra entre la disponibilidad y la consistencia, ya que la tolerancia a particiones es una decisión de arquitectura (sea o no distribuida).

Aunque el teorema dicte que si en un sistema distribuido elegimos disponibilidad no podemos tener consistencia, todavía podemos obtener consistencia eventual. Es decir, cada nodo siempre estará disponible para servir peticiones, aunque estos nodos no puedan asegurar que la información que contienen sea consistente (pero si bastante precisa), en algún momento lo será.

Algunas bases de datos tolerantes a particiones se pueden ajustar para ser más o menos consistentes o disponible a nivel de petición. Por ejemplo, *Riak* trabaja de esta manera, permitiendo a los clientes decidir en tiempo de petición que nivel de consistencia necesitan.

Clasificación según CAP

El siguiente gráfico muestra como dependiendo de estos atributos podemos clasificar los sistemas NoSQL:



Así pues, las bases de datos NoSQL se clasifican en:

- **CP**: Consistente y tolerantes a particiones. Tanto *MongoDB* como *HBase* son CP, ya que dentro de una partición pueden no estar disponibles para responder una determinada consulta (por ejemplo, evitando lecturas en los nodos esclavo), aunque son tolerantes a fallos porque cualquier nodo secundario se puede convertir en principal y asumir el rol del nodo caído.
- **AP**: Disponibles y tolerantes a particiones. *CouchDB* permite replicar los datos entre sus nodos aunque no garantiza la consistencia en ninguno de los sus servidores.
- **CA**: Consistentes y disponible. Por ejemplo, *Redis*, *PostreSQL* y *Neo4J* son CA, ya que no distribuyen los datos y por tanto la partición no es una restricción.

Lo bueno es que la gran mayoría de sistemas permiten configurarse para cambiar su tipo CAP, de manera que *MongoDB* pase de CP a AP, o *CouchDB* de AP a CP.

1.7. MongoDB

MongoDB (http://www.mongodb.org) es una de las bases de datos NoSQL más conocidas. Sigue un modelo de datos documental, donde los documentos se basan en JSON.

MongoDB destaca porque:

- Soporta esquemas dinámicos: diferentes documentos de una misma colección pueden tener atributos diferentes.
- No soporta *joins*, ya que no escalan bien.

- No soporta transacciones. Lo que en un SGDB puede suponer múltiples operaciones automáticas, con *MongoDB* se puede hacer en una sola operación al insertar/actualizar todo un documento de una sola vez.

Hay una serie de conceptos que conviene conocer antes de entrar en detalle:

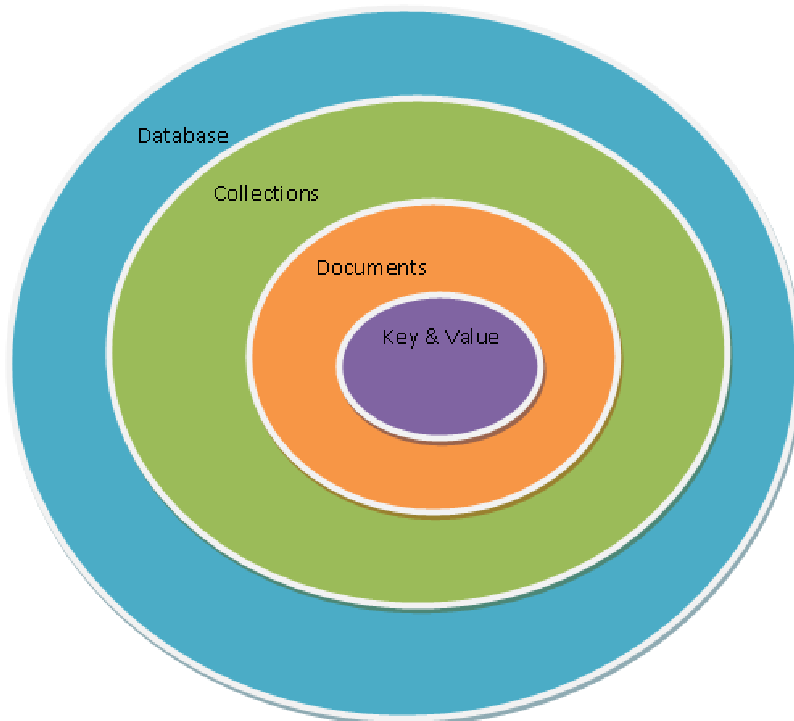


Figura 9. Elementos de MongoDB

- *MongoDB* tienen el mismo concepto de **base de datos** que un SGDB. Dentro de una instancia de MongoDB podemos tener 0 o más bases de datos, actuando cada una como un contenedor de alto nivel
- Una base de datos tendrá 0 o más colecciones. Una **colección** es muy similar a lo que entendemos como tabla dentro de un SGDB. *MongoDB* ofrece diferentes tipos de colecciones, desde las normales cuyo tamaño crece conforme lo hace el número de documentos, como las colecciones *capped*, las cuales tienen un tamaño predefinido y que pueden contener una cierta cantidad de información que se sustituirá por nueva cuando se llene.
- Las colecciones contiene 0 o más **documentos**, por lo que es similar a una fila o registro de un RDMS.
- Cada documento contiene 0 o más atributos, compuestos de parejas clave/valor. Cada uno de estos documentos no sigue ningún esquema, por lo que dos documentos de una misma colección pueden contener todos los atributos diferentes entre sí.
- *MongoDB* soporta **índices**, igual que cualquier SGDB, para acelerar la búsqueda de datos.
- Al realizar cualquier consulta, se devuelve un **cursor**, con el cual podemos hacer cosas tales como contar, ordenar, limitar o saltar documentos.

Así pues, tenemos que una base de datos va a contener varias colecciones, donde cada colección tendrá un conjunto de documentos:

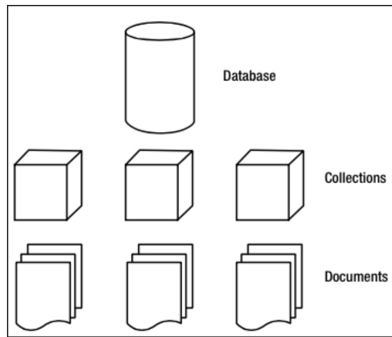


Figura 10. Modelo de MongoDB

Instalación

Desde http://www.mongodb.org/downloads podemos descargar la versión acorde a nuestro sistema operativo. Para instalar en Mac, lo más cómodo es utilizar la herramienta `brew`

```
brew install mongodb
```

Si tenemos un sistema de 32 bits, como es el caso de nuestra instalación en la máquina virtual, podemos trabajar con *MongoDB*, aunque no es recomendable ya que estamos restringiendo el tamaño de los archivos a 2GB. Así pues, se recomienda siempre instalar la versión de 64 bits sobre un sistema acorde.

Si queremos instalar nosotros los binarios a mano, necesitamos descargarlos y ponerlos en el path. Posteriormente, necesitaremos crear la carpeta para guardar las bbdd:

```
mkdir -p /data/db
chown `id -u` /data/db
```

Finalmente, si no hemos instalado *MongoDB* como un servicio, para arrancar el servidor en un terminal lanzaremos el demonio `mongod`.

```
aitormedrano — mongod — 80×24
MacBook-Air-de-Aitor:~ aitormedrano$ mongod
mongod --help for help and startup options
2015-02-27T10:01:28.446+0100 [initandlisten] MongoDB starting : pid=1351 port=27
017 dbpath=/data/db 64-bit host=MacBook-Air-de-Aitor.local
2015-02-27T10:01:28.447+0100 [initandlisten] db version v2.6.7
2015-02-27T10:01:28.447+0100 [initandlisten] git version: nogitversion
2015-02-27T10:01:28.447+0100 [initandlisten] build info: Darwin miniyosemite.loc
al 14.1.0 Darwin Kernel Version 14.1.0: Fri Dec  5 06:49:27 PST 2014; root:xnu-2
782.10.67~9/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_49
2015-02-27T10:01:28.447+0100 [initandlisten] allocator: tcmalloc
2015-02-27T10:01:28.447+0100 [initandlisten] options: {}
2015-02-27T10:01:28.546+0100 [initandlisten] journal dir=/data/db/journal
2015-02-27T10:01:28.546+0100 [initandlisten] recover : no journal files present,
 no recovery needed
2015-02-27T10:01:29.212+0100 [initandlisten] preallocateIsFaster=true 13.12
2015-02-27T10:01:29.247+0100 [initandlisten] waiting for connections on port 270
17
```

Figura 11. Lanzando el demonio mongod

Tanto los apuntes como la versión instalada en la máquina virtual son la **2.6**. El 3 de marzo se lanzó la versión 3.0 con mejoras en el rendimiento y la escalabilidad.

Por defecto, el demonio se lanza sobre el puerto 27017. Si accedemos a `http://localhost:27017` podremos ver que nos indica como estamos intentando acceder mediante HTTP a *MongoDB* mediante el puerto reservado al driver nativo.

Si no quisiéramos instalarlo, podríamos utilizar una solución *Cloud* que incluya *MongoDB*, como pueda ser MongoLab[2] o Amazon Web Services[3]. Más información en https://www.mongodb.com/partners/cloud

A continuación vamos a estudiar las diferentes herramientas que nos ofrece *MongoDB* para posteriormente todas las operaciones que podemos realizar.

Herramientas

Además del demonio y del cliente, *MongoDB* ofrece un conjunto de herramientas para interactuar con las bases de datos, permitiendo crear y restaurar copias de seguridad.

Si estamos interesados en introducir o exportar una colección de datos mediante JSON, podemos emplear los comandos `mongoimport` y `mongoexport`:

Importanto y Exportando datos

```
mongoimport -d nombreBaseDatos -c coleccion --file nombreFichero.json
mongoexport -d nombreBaseDatos -c coleccion nombreFichero.json
```

Estas herramientas interactúan con datos JSON y no sobre toda la base de datos.

Un caso particular y muy común es importar datos que se encuentran en formato CSV/TSV. Para ello, emplearemos el parámetro `--type csv`:

Importanto CSV/TSV - poblacionEspanya2013.tsv[4]

```
mongoimport --type tsv -d test -c poblacion --headerline --drop
 poblacionEspanya2013.tsv
```

Más información sobre importar y exportar datos en http://docs.mongodb.org/manual/core/import-export/

Antes que hacer un *export*, es más conveniente realizar un *backup* en binario mediante `mongodump`, el cual genera ficheros BSON. Estos archivos posteriormente se restauran mediante `mongorestore`.

Restaurando un copia de seguridad

```
mongodump -d nombreBaseDatos nombreFichero.bson
mongorestore -d nombreBaseDatos nombreFichero.bson
```

Más información sobre copias de seguridad en http://docs.mongodb.org/manual/core/backups/

2 https://mongolab.com
3 http://aws.amazon.com/es/
4 resources/nosql/poblacionEspanya2013.tsv

Si necesitamos transformar un fichero BSON a JSON (de binario a texto), tenemos el comando `bsondump`:

De BSON a JSON

```
bsondump file.bson > file.json
```

Otra herramienta es `mongostat` que permite visualizar el estado del servidor *MongoDB*, así como algunas estadísticas sobre su rendimiento. Esta herramienta la estudiaremos en la última sesión.

Para poder trabajar con *MongoDB* desde cualquier aplicación necesitamos un driver. *MongoDB* ofrece drivers oficiales para casi todos los lenguajes de programación actuales. Más información en http://docs.mongodb.org/ecosystem/drivers/

Por ejemplo, para poder trabajar con *Java*, mediante *Maven* podremos descargarlo mediante la siguiente dependencia:

```
<dependency>
  <groupId>org.mongodb</groupId>
  <artifactId>mongo-java-driver</artifactId>
  <version>2.13.1</version>
</dependency>
```

Aunque trabajaremos en detalle con Java en la siguiente sesión, en http://docs.mongodb.org/ecosystem/drivers/java/ podemos obtener información sobre el drivers y el soporte para las diferentes versiones de *MongoDB*.

Finalmente, una herramienta de terceros bastante utilizada es *RoboMongo* (http://robomongo.org), el cual extiende el shell y permite un uso más amigable.

En el curso nos vamos a centrar en el uso del shell y la conectividad de *MongoDB* mediante Java.

1.8. Hola *MongoDB*

Tras lanzar el demonio `mongod`, llega el momento de acceder mediante el cliente `mongo`, el cual funciona igual que un shell, de modo que con la fecha hacia arriba visualizaremos el último comando. El cliente utiliza *JavaScript* como lenguaje de interacción con la base de datos.

Al conectar con `mongo` si no le indicamos nada se conectará por defecto a la base de datos `test`. Si queremos conectar a una base de datos concreta, la pasaremos como parámetro:

Figura 12. Lanzando el cliente mongo

En cualquier momento podemos cambiar la base de datos activa mediante `use nombreBaseDatos`. Si la base de datos no existiese, *MongoDB* creará dicha base de datos. Esto es una verdad a medias, ya que la base de datos realmente se crea al insertar datos dentro de alguna colección.

Así pues, vamos a crear nuestra base de datos `expertojava`:

Accediendo a la base de datos `expertojava`

```
use expertojava
```

Una vez creada, podemos crear nuestra primera colección, que llamaremos `people`, e insertaremos un persona con nuestros datos personales mediante el método `insert`, al que le pasamos un objeto JSON:

Insertando una persona

```
db.people.insert({ nombre: "Aitor", edad: 37, profesion: "Profesor" })
```

Una vez insertada, sólo nos queda realizar una consulta para recuperar los datos y comprobar que todo funciona correctamente mediante el método `findOne`:

Recuperando una persona

```
db.people.findOne()
```

Lo que nos dará como resultado un objeto JSON que contiene un atributo `_id` además de los que le añadimos al insertar la persona:

Resultado de la consulta de la persona

```
{
  "_id" : ObjectId("53274f9883a7adeb6a573e64"),
  "nombre" : "Aitor",
  "edad" : 37,
  "profesion" : "Profesor"
}
```

Como podemos observar, todas las instrucciones van a seguir el patrón de `db.nombreColeccion.operacion()`.

Todas las operaciones que podemos realizar las veremos en la siguiente sesión. En todo caso, podemos consultar la documentación en http://docs.mongodb.org/manual/

1.9. Ejercicios

El repositorio a clonar es `java_ua/nosql-expertojava`.

Cada uno de los ejercicios se guardará en un fichero de texto que se almacenará en la raíz del repositorio, a no ser que se indique lo contrario.

(1 punto) Ejercicio 11. Cuestionario

En un fichero de texto o un documento con nombre `ej11.txt` / `ej11.odt`, responde a las siguientes cuestiones:

1. ¿Qué significa el prefijo *No* del acrónimo *NoSQL*?
2. ¿Un sistema puede soportar al mismo tiempo replicación y particionado?
3. Para los siguientes supuestos, indica qué modelo de datos emplearías y justifica tu respuesta:
    a. Enciclopedia de personajes de Comic
    b. Usuarios, perfiles, biblioteca, puntuaciones de una plataforma de gaming
    c. Información académica de un país (centros, alumnos, profesores, asignaturas, calificaciones, …)
4. Respecto al Teorema de CAP ¿Cómo puede *MongoDB* pasar de ser un sistema CP a AP?
5. ¿En qué consiste la "persistencia políglota"?

(0.25 puntos) Ejercicio 12. Puesta en Marcha con *MongoDB*

En esta sesión, vamos a centrarnos en comprobar la instalación que tenemos en nuestra máquina virtual y a trabajar con el shell de *MongoDB* para familiarizarnos con su uso y poder importar los datos necesarios.

Para ello, si no esta instalado como demonio, en un terminar arrancar el servicio `mongod`.

Posteriormente, tras abrir un terminal y arrancar el shell mediante `mongo`. Se pide comprobar:

- qué bases de datos existen.
- qué colecciones existen en la base de datos `expertojava`.

Tras esto, crear una nueva base de datos llamada `ejercicios`.

Anotad en un fichero de texto de nombre `ej12.txt` los comandos y resultados obtenidos.

(0.25 puntos) Ejercicio 13. Inserciones

Sobre la base de datos `ejercicios`, importa los datos que se encuentran en mongo_cities1000.json[5] en una colección denominada `cities`.

[5] resources/nosql/mongo_cities1000.json

Si analizas los datos, verás que se crean documentos con una estructura similar a esta:

- `name`: nombre de la ciudad.
- `country`: país.
- `timezone`: zona horaria.
- `population`: población.
- `location`: coordenadas compuestas de:
  - `longitude`: longitud
  - `latitude`: latitud

Por ejemplo:

```
> db.cities.findOne()
{
 "_id" : ObjectId("55264a34c25edd8055f20cba"),
 "name" : "Sant Julià de Lòria",
 "country" : "AD",
 "timezone" : "Europe/Andorra",
 "population" : 8022,
 "location" : {
  "longitude" : 42.46372,
  "latitude" : 1.49129
 }
}
```

Se pide insertar una nueva ciudad con los datos de la ciudad donde vives, pero poniendo como nombre de la ciudad tu nombre.

Escribe los comandos necesarios y el resultado en `ej13.txt`.

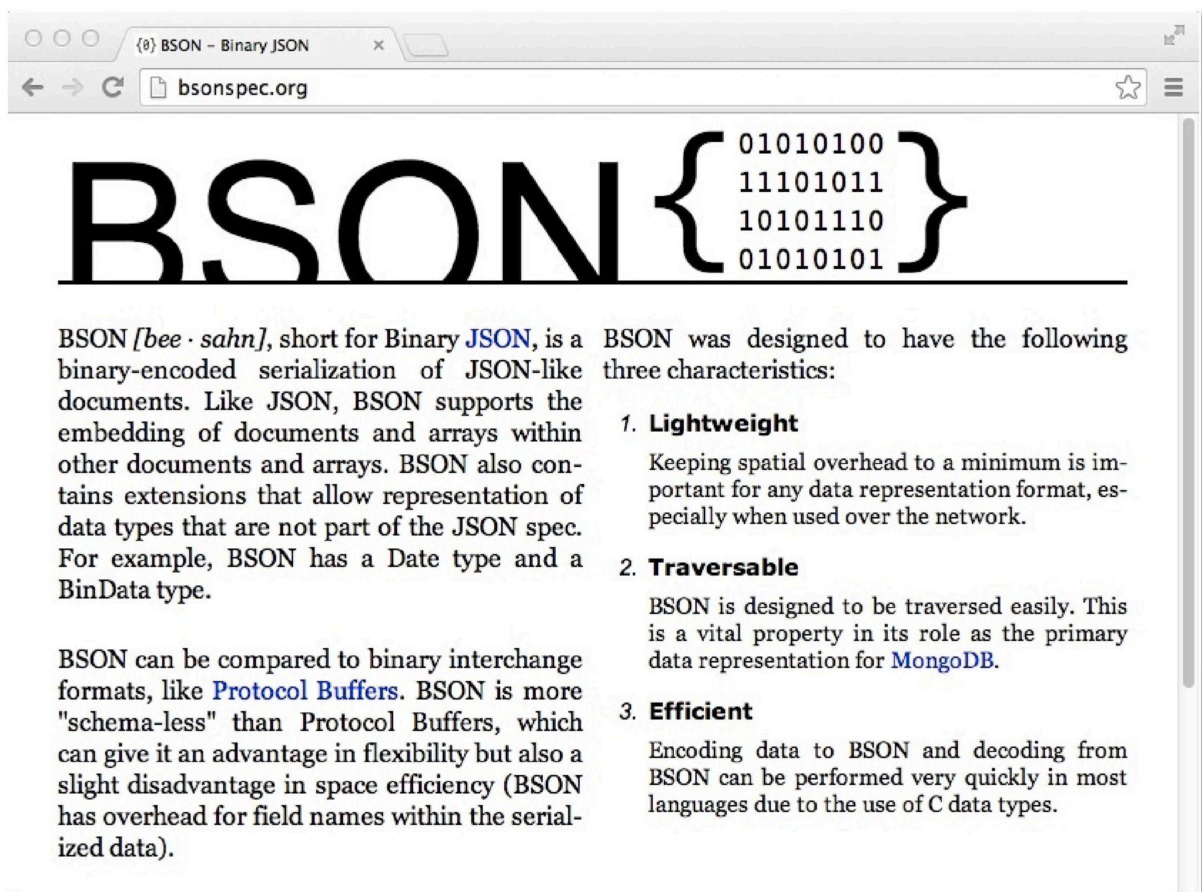
2. MongoDB

Ya hemos visto que *MongoDB* es una base de datos documental, que agrupa los documentos en colecciones.

En esta sesión estudiaremos la estructura de estos documentos, y como podemos interactuar con ellos.

2.1. BSON

Mediante *JavaScript* podemos crear objetos que se representan con JSON. Internamente, *MongoDB* almacena los documentos en BSON (*Binary JSON*). Podemos consultar la especificación en http://BSONSpec.org



BSON representa un superset de JSON, ya que:

1. Almacena datos en binario
2. Incluye un conjunto de tipos de datos no incluidos en JSON, como pueden ser `ObjectId`, `Date` o `BinData`.

Podemos consultar todos los tipos que soporta un objeto *BSON* en http://docs.mongodb.org/manual/reference/bson-types/

Ejemplo de objeto BSON

```
var yo = {
```

```
   nombre: "Aitor",
   apellidos: "Medrano",
   fnac: new Date("Oct 3, 1977"),
   hobbies: ["programación", "videojuegos", "baloncesto"],
   casado: true,
   hijos: 2,
   fechaCreacion = new Timestamp()
 }
```

Los documentos BSON tienen las siguientes restricciones:

- no pueden tener un tamaño superior a 16 MB.
- el atributo `_id` queda reservado para la clave primaria.
- los nombres de los campos no pueden empezar por `$`.
- los nombres de los campos no pueden contener el `.`.
- *MongoDB* no asegura que el orden de los campos se respete.

Si queremos validar si un documento JSON es válido, podemos usar http://jsonlint.com/. Hemos de tener en cuenta que sólo valida JSON y no BSON, por tanto nos dará errores en los tipos de datos propios de BSON.

2.2. Trabajando con el shell

Antes de entrar en detalles en las instrucciones necesarias para realizar las operaciones CRUD, veamos algunos comandos que nos serán muy utiles al interactuar con el shell:

Tabla 2. Comandos útiles dentro del cliente de *MongoDB*

Comando	Función
`show dbs`	Muestra el nombre de las bases de datos
`show collections`	Muestra el nombre de las colecciones
`db`	Muestra el nombre de la base de datos que estamos utilizando
`db.dropDatabase()`	Elimina la base de datos actual
`db.help()`	Muestra los comandos disponibles
`db.version()`	Muestra la versión actual del servidor

En el resto de la sesión vamos a hacer un uso intenso del shell de *MongoDB*. Por ejemplo, si nos basamos en el objeto definido en el apartado de BSON, podemos ejecutar las siguientes instrucciones:

Ejemplos de interacción con el shell

```
> db.people.insert(yo)          ❶
> db.people.find()              ❷
{ "_id" : ObjectId("53274f9883a7adeb6a573e64"), "nombre"
 : "Aitor", "apellidos" : "Medrano", "fnac" :
 ISODate("1977-10-02T23:00:00Z"), "hobbies" : [  "programación",
  "videojuegos",  "baloncesto" ], "casado" : true, "hijos"
 : 2, "fechaCreacion" : Timestamp(1425633249, 1) }
> yo.email = "aitormedrano@gmail.com"
```

```
aitormedrano@gmail.com
> db.people.save(yo)          ❸
> db.people.find()
{ "_id" : ObjectId("53274f9883a7adeb6a573e64"), "nombre"
 : "Aitor", "apellidos" : "Medrano", "fnac" :
 ISODate("1977-10-02T23:00:00Z"), "hobbies" : [  "programación",
  "videojuegos",  "baloncesto" ], "casado" : true, "hijos"
 : 2, "fechaCreacion" : Timestamp(1425633249, 1) }
{ "_id" : ObjectId("53274fca83a7adeb6a573e65"), "nombre"
 : "Aitor", "apellidos" : "Medrano", "fnac" :
 ISODate("1977-10-02T23:00:00Z"), "hobbies" : [  "programación",
  "videojuegos",  "baloncesto" ], "casado" : true, "hijos"
 : 2, "fechaCreacion" : Timestamp(1425633373, 1), "email"
 : "aitormedrano@gmail.com" }        ❹
> db.people.find().forEach(printjson)
```

❶ Si queremos insertar un documento en una colección, hemos de utilizar el método **`insert`** (http://docs.mongodb.org/master/reference/method/db.collection.insert/) pasándole como parámetro el documento que queremos insertar.

❷ **`find`** permite recuperar documentos

❸ **`save`** es similar a `insert`, pero si existe un documento con el mismo `ObjectId`, realizará un `update` (realmente un `upsert`)

❹ Hay dos documentos porque al guardar el segundo se le ha asignado un nuevo `ObjectId`. Además, los dos documentos no tienen el mismo número de campos, y la fechaCreación se ha actualizado con el *timestamp* actual.

Otros ejemplos tanto de `insert` como de `save` con objetos directos, sin necesidad de usar variables, serían:

Inserción y Guardado

```
db.people.insert({ nombre : "Aitor", edad : 37, profesion : "Profesor" })
db.people.save({ nombre : "Aitor", edad : 37, profesion : "Profesor" })
```

Autoevaluación

Al ejecutar las dos instrucciones anteriores sobre una colección vacía ¿Cuantos registros tendrá la colección? [6]

2.3. ObjectId

En *MongoDB*, el atributo `_id` es único dentro de la colección, y hace la función de clave primaria. Se le asocia un `ObjectId` (http://docs.mongodb.org/manual/reference/object-id/), el cual es un tipo BSON de 12 bytes que se crea mediante:

- el *timestamp* actual (*4 bytes*)
- un identificador de la máquina / *hostname* (*3 bytes*) donde se genera
- un identificador del proceso (*2 bytes*) donde se genera
- un número aleatorio (*3 bytes*).

Este objeto lo crea el driver y no *MongoDB*, por lo cual no deberemos considerar que siguen un orden concreto, ya que clientes diferentes pueden tener *timestamps* desincronizados. Lo

[6] 2, porque no comparten `ObjectId`

que sí que podemos obtener a partir del `ObjectId` es la fecha de creación del documento, mediante el método `getTimestamp()` del atributo `_id`.

Obteniendo la fecha de creación de un documento

```
> db.people.find()[0]._id
ObjectId("53274f9883a7adeb6a573e64")
> db.people.find()[0]._id.getTimestamp()
ISODate("2014-03-17T19:40:08Z")
```

Este identificador es **global, único e inmutable**. Esto es, no habrá dos repetidos y una vez un documento tiene un `_id`, éste no se puede modificar.

Si en la definición del objeto a insertar no ponemos el atributo identificador, *MongoDB* creará uno de manera automática. Si lo ponemos nosotros de manera explícita, *MongoDB* no añadirá ningún `ObjectId`. Eso sí, debemos asegurarnos que sea único (podemos usar números, cadenas, etc…).

Por lo tanto, podemos hacer esto:

Asignando un identificador al insertar

```
db.people.insert({_id:3, nombre:"Marina", edad:6 })
```

Cuidado con los tipos, ya que no es lo mismo insertar un atributo con `edad:6` (se considera el campo como entero) que con `edad:"6"`, ya que considera el campo como texto.

O también, si queremos podemos hacer que el `_id` de un documento sea un documento en sí, y no un entero, para ello, al insertarlo, podemos asignarle un objeto JSON al atributo identificador:

Insertanto un documento cuyo identificador es otro documento

```
db.people.insert({_id:{nombre:'Aitor', apellidos:'Medrano',
  twitter:'@aitormedrano'}, ciudad:'Elx'})
```

2.4. Consultas

Para recuperar los datos de una colección o un documento en concreto usaremos el método `find()`:

Ejemplo de consulta con `find()`

```
> db.people.find()
{ "_id" : ObjectId("53274f9883a7adeb6a573e64"), "nombre"
  : "Aitor", "apellidos" : "Medrano", "fnac" :
  ISODate("1977-10-02T23:00:00Z"), "hobbies" : [  "programación",
   "videojuegos",  "baloncesto" ], "casado" : true, "hijos" : 2 }
{ "_id" : ObjectId("53274fca83a7adeb6a573e65"), "nombre"
  : "Aitor", "apellidos" : "Medrano", "fnac" :
  ISODate("1977-10-02T23:00:00Z"), "hobbies" : [  "programación",
```

```
  "videojuegos",  "baloncesto" ], "casado" : true, "hijos" : 2, "email"
 : "aitormedrano@gmail.com" }
{ "_id" : 3, "nombre" : "Marina", "edad" : 6 }
```

El método `find()` sobre una colección devuelve un cursor a los datos obtenidos, el cual se queda abierto con el servidor y que se cierra automáticamente a los 10 minutos de inactividad o al finalizar su recorrido. Si hay muchos resultados, la consola nos mostrará un subconjunto de los datos (20).

Si queremos que salga más legible, podemos recorrer la consulta y mostrar una vista tabulada mediante `printjson`:

```
> db.people.find().forEach(printjson)
```

En cambio, si sólo queremos recuperar un documento hemos de utilizar `findOne()`:

Recuperando un único documento

```
> db.people.findOne()
{
  "_id" : ObjectId("53274f9883a7adeb6a573e64"),
  "nombre" : "Aitor",
  "apellidos" : "Medrano",
  "fnac" : ISODate("1977-10-02T23:00:00Z"),
  "hobbies" : [
    "programación",
    "videojuegos",
    "baloncesto"
  ],
  "casado" : true,
  "hijos" : 2
}
```

Se puede observar que al recuperar un documento con `findOne`, se muestra una vista formateada. Si queremos que esta vista se aplique a un documento encontrado con `find` podemos utilizar el sufijo `.pretty()`.

```
> db.people.find().pretty()
```

Preparando los ejemplos

Para los siguientes ejemplos, vamos a utilizar una colección de 800 calificaciones que han obtenido diferentes estudiantes en trabajos, exámenes o cuestionarios.

Para ello, importaremos la colección grades.json[7] mediante:

Importanto grades.json[8]

[7] resources/nosql/grades.json
[8] resources/nosql/grades.json

```
mongoimport -d expertojava -c grades --file grades.json
```

Un ejemplo de una calificación sería:

```
> db.grades.findOne()
{
  "_id" : ObjectId("50906d7fa3c412bb040eb577"),
  "student_id" : 0,
  "type" : "exam",
  "score" : 54.6535436362647
}
```

El campo `type` puede tomar los siguientes valores: *quiz*, *homework* o *exam*

Criterios en consultas

Al hacer una consulta, si queremos obtener datos mediante más de un criterio, en el primer parámetro del `find` podemos pasar un objeto JSON con los campos a cumplir (condición Y).

Consulta con dos condiciones

```
> db.grades.find({student_id:0, type:"quiz"})
```

Consejo de Rendimiento

Las consultas disyuntivas, es decir, con varios criterios u operador `$and`, deben filtrar el conjunto más pequeño cuanto más pronto posible.

Supongamos que vamos a consultar documentos que cumplen los criterios A, B y C. Digamos que el criterio A lo cumplen 40.000 documentos, el B lo hacen 9.000 y el C sólo 200. Si filtramos A, luego B, y finalmente C, el conjunto que trabaja cada criterio es muy grande.

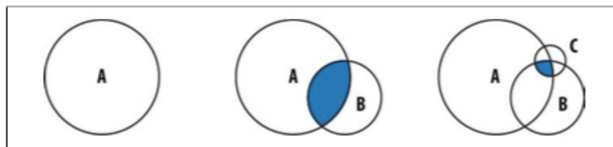


Figura 13. Restringiendo consultas AND de mayor a menor

En cambio, si hacemos una consulta que primero empiece por el criterio más restrictivo, el resultado con lo que se intersecciona el siguiente criterio es menor, y por tanto, se realizará más rápido.

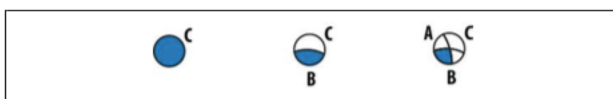


Figura 14. Restringiendo consultas AND de menor a mayor

MongoDB también ofrece operadores lógicos para los campos numéricos:

Tabla 3. Operadores lógicos

Comparador	Operador
menor que (<)	`$lt`
menor o igual que (≤)	`$lte`
mayor que (>)	`$gt`
mayor o igual que (≥)	`$gte`

Estos operadores se pueden utilizar de forma simultánea sobre un mismo campo o sobre diferentes campos, y se colocan como un nuevo documento en el valor del campo a filtrar, compuesto del operador y del valor a comparar:

Ejemplos de consultas con operadores relacionales

```
> db.grades.find({ score:{$gt:95} })
> db.grades.find({ score:{$gt:95, $lte:98}, type:"exam" })
> db.grades.find({ type:"exam", score:{$gte:65} })
```

Para los campos de texto, además de la comparación directa, podemos usar el operador `$ne` para obtener los documentos cuyo campos **no** tienen un determinado valor. Así pues, podemos usarlo para averiguar todas las calificaciones que no sean cuestionarios (*quiz*):

Consulta con *not equal*

```
> db.grades.find({type:{$ne:"quiz"}})
```

Mucho cuidado al usar polimorfismo y almacenar en un mismo campo un entero y una cadena, ya que al hacer comparaciones para recuperar datos, no vamos a poder mezclar cadenas con valores numéricos. Se considera un antipatrón el mezclar tipos de datos en un campo.

Las comparaciones de cadenas se realizan siguiendo el orden UTF8, similar a ASCII, con lo cual no es lo mismo buscar un rango entre mayúsculas que minúsculas.

Con cierto parecido a la condición de valor no nulo de las BBDD relacionales y teniendo en cuenta que la libertad de esquema puede provocar que un documento tenga unos campos determinados y otro no lo tenga, podemos utilizar el operador `$exists` si queremos averiguar si un campo existe (y por tanto tiene algún valor).

Consulta con condición de existencia de un campo

```
> db.grades.find({"score":{$exists:true}})
```

Pese a que ciertos operadores contengan su correspondiente operador negado, *MongoDB* ofrece el operador `$not`. Éste puede utilizarse conjuntamente con otros operadores para negar el resultado de los documentos obtenidos.

Por ejemplo, si queremos obtener todas las calificaciones que no sean múltiplo de 5, podríamos hacerlo así:

Ejemplo de consulta con negación

```
> db.grades.find({score:{$not: {$mod: [5,0]}}})
```

Finalmente, si queremos realizar consultas sobre partes de un campo de texto, hemos de emplear expresiones regulares. Para ello, tenemos el operador `$regexp` o, de manera más sencilla, indicando como valor la expresión regular a cumplir:

Por ejemplo, para buscar las personas cuyo nombre contenga la palabra `Aitor`:

Ejemplo de consulta con expresión regular

```
> db.people.find({nombre:/Aitor/})
> db.people.find({nombre:/aitor/i})
> db.people.find({nombre: {$regex:/aitor/i}})
```

Ya vimos en el módulo de *JavaScript* la flexibilidad y potencia que ofrecen las expresiones regulares. Para profundizar en su uso mediante *MongoDB* podéis obtener más información sobre el operador `$regex` en http://docs.mongodb.org/manual/reference/operator/query/regex/#op._S_regex

Otros operadores

Algunos operadores que conviene citar aunque su uso es más bien ocasional son:

- Si queremos recuperar documentos que dependan del tipo de campo que contiene, podemos preguntar con `$type` http://docs.mongodb.org/manual/reference/operator/query/type/
- El operador `$where` permite introducir una expresión *JavaScript* http://docs.mongodb.org/manual/reference/operator/query/where/

Proyección de campos

Las consultas realizadas hasta ahora devuelven los documentos completos. Si queremos que devuelva un campo o varios campos en concreto, hemos de pasar un segundo parámetro de tipo JSON con aquellos campos que deseamos mostrar con el valor `true` o `1`. Destacar que si no se indica nada, por defecto siempre mostrará el campo `_id`

```
> db.grades.findOne({student_id:3},{score:true});
{ "_id" : ObjectId("50906d7fa3c412bb040eb583"), "score" : 92.6244233936537
 }
```

Por lo tanto, si queremos que no se muestre el `_id`, lo podremos a `false` o `0`:

```
> db.grades.findOne({student_id:3},{score:true, _id:false});
{ "score" : 92.6244233936537 }
```

Condiciones sobre objetos anidados

Si queremos acceder a campos de subdocumentos, siguiendo la sintaxis de JSON, se utiliza la notación punto. Esta notación permite acceder al campo de un documento anidado, da igual el nivel en el que esté y su orden respecto al resto de campos.

Por ejemplo, supongamos que tenemos un catálogo de productos de una tienda electrónica, el cual es similar al siguiente documento:

```
{
  "producto" : "Condensador de Fluzo",
  "precio" : 100000000000,
  "reviews" : [
    {
      "usuario" : "emmett",
      "comentario" : "¡Genial!",
      "calificacion" : 5
    },{
      "usuario" : "marty" ,
      "comentario" : "¡Justo lo que necesitaba!",
      "calificacion" : 4
    } ]
}
```

Para acceder al usuario de una revisión usaríamos la propiedad `reviews.usuario`.

Por ejemplo, para averiguar los productos que cuestan más de 10.000 y que tienen una calificación igual a 5 o superior haríamos:

```
> db.catalogo.find({"precio":{$gt:10000},"reviews.calificacion":{$gte:5}})
```

Condiciones compuestas con Y / O

Para usar la conjunción o la disyunción, tenemos los operadores `$and` y `$or`. Son operadores prefijo, de modo que se ponen antes de las subconsultas que se van a evaluar. Estos operadores trabajan con arrays, donde cada uno de los elementos es un documento con la condición a evaluar, de modo que se realiza la unión entre estas condiciones, aplicando la lógica asociada a AND y a OR.

```
> db.grades.find({ $or:[ {"type":"exam"}, {"score":{$gte:65}} ]})
> db.grades.find({ $or:[ {"score":{$lt:50}}, {"score":{$gt:90}} ]})
```

Realmente el operador `$and` no se suele usar porque podemos anidar en la consulta 2 criterios, al poner uno dentro del otro. Así pues, estas dos consultas hacen lo mismo:

Ejemplos consultas conjunciones con y sin `$and`

```
> db.grades.find({ type:"exam", score:{$gte:65} })
> db.grades.find({ $and:[ {type:"exam"}, {score:{$gte:65}} ] })
```

Consejo de Rendimiento

Las consultas conjuntivas, es decir, con varios criterios excluyentes u operador `$or`, deben filtrar el conjunto más grande cuanto más pronto posible.

Supongamos que vamos a consultar los mismos documentos que cumplen los criterios A (40.000 documentos), B (9.000 documentos) y C (200 documentos).

Si filtramos C, luego B, y finalmente A, el conjunto de documentos que tiene que comprobar *MongoDB* es muy grande.

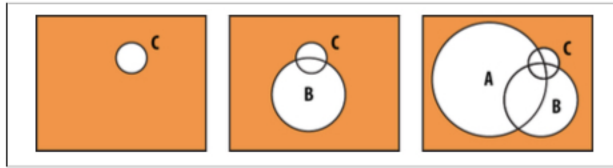


Figura 15. Restringiendo consultas OR de menor a myor

En cambio, si hacemos una consulta que primero empiece por el criterio menos restrictivo, el conjunto de documentos sobre el cual va a tener que comprobar siguientes criterios va a ser menor, y por tanto, se realizará más rápido.

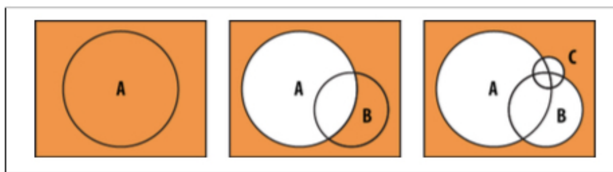


Figura 16. Restringiendo consultas OR de mayor a menor

También podemos utilizar el operado `$nor`, que no es más que la negación de `$or` y que obtendrá aquellos documentos que no cumplan ninguna de las condiciones.

Autoevaluación

Que obtendríamos al ejecutar la siguiente consulta: [9]

```
> db.grades.find({ score:{$gte:65}, $nor:[ {type:"quiz"},
  {type:"homework"} ] })
```

Finalmente, si queremos indicar mediante un array los diferentes valores que puede cumplir un campo, podemos utilizar el operador `$in`:

```
> db.grades.find({ type:{$in:["quiz","exam"]}})
```

Por supuesto, también existe su negación mediante `$nin`.

Consultas sobre arrays

Si trabajamos con arrays, vamos a poder consultar el contenido de una posición del mismo tal como si fuera un campo normal, siempre que sea un campo de primer nivel, es decir, no sea un documento embebido dentro de un array.

Si queremos filtrar teniendo en cuenta el número de ocurrencias del array, podemos utilizar:

- `$all` para filtrar ocurrencias que tienen todos los valores del array, es decir, los valores pasados a la consulta serán un subconjunto del resultado. Puede que devuelva los mismos, o un array con más campos (el orden no importa)

[9] documentos con una calificación superior o igual a 65 y no que no sean de tipo "quiz" ni "homework"

- `$in`, igual que SQL, para obtener las ocurrencias que cumple con alguno de los valores pasados (similar a usar `$or` sobre un conjunto de valores de un mismo campo). Si queremos su negación, usaremos `$nin`, para obtener los documentos que no cumplen ninguno de los valores.

Por ejemplo, si queremos obtener las personas que dentro de sus amistades se encuentre *Juan* **y** *David*, y respecto a sus hobbies estén el *footing* **o** el *baloncesto*, tendríamos:

Ejemplo consulta con `$all` y `$in`

```
> db.people.find( {amistades: {$all: ["Juan", "David"]}, hobbies: {$in:
  ["footing", "baloncesto"]}} )
```

Si el array contiene documentos y queremos filtrar la consulta sobre los campos de los documentos del array, tenemos que utilizar `$elemMatch`. Más información en http://docs.mongodb.org/manual/reference/operator/projection/elemMatch/

Si lo que nos interesa es la cantidad de elementos que contiene un array, emplearemos el operador `$size`.

Por ejemplo, para obtener las personas que tienen 3 hobbies haríamos:

Ejemplo consulta con `$size`

```
> db.people.find( {hobbies : {$size : 3}} )
```

Finalmente, a la hora de proyectar los datos, si no estamos interesados en todos los valores de un campo que es un array, podemos restringir el resultado mediante el operador `$slice`:

Así pues, si quisieramos obtener las personas que tienen mas de un hijo, y que de esas personas, en vez de mostrar todos sus hobbies, mostrase los dos primeros, haríamos:

Ejemplo con `$slice`

```
> db.people.find( {hijos: {$gt:1}}, {hobbies: {$slice:2}} )
```

Más información en http://docs.mongodb.org/manual/reference/operator/projection/slice/

Conjunto de valores

Igual que en SQL, a partir de un colección, si queremos obtener todos los diferentes valores que existen en un campo, utilizaremos el método **`distinct`**

```
> db.grades.distinct('type')
[ "exam", "quiz", "homework" ]
```

Si queremos filtrar los datos sobre los que se obtienen los valores, le pasaremos un segundo parámetro con el criterio a aplicar:

```
> db.grades.distinct('type', { score: { $gt: 99.9 } } )
[ "exam" ]
```

Cursores

Al hacer una consulta en el shell se devuelve un cursor. Este cursor lo podemos guardar en un variable, y partir de ahí trabajar con él como haríamos mediante *Java*. Si `cur` es la variable que referencia al cursor, podremos utilizar los siguientes métodos:

Tabla 4. Métodos de uso de cursores

Método	Uso	Lugar de ejecución
`cur.hasNext()`	`true` / `false` para saber si quedan elementos	Cliente
`cur.next()`	Pasa al siguiente documento	Cliente
`cur.limit(numElementos)`	Restringe el número de resultados a *numElementos*	Servidor
`cur.sort({campo:1})`	Ordena los datos por *campo* `1` ascendente o `-1` o descendente	Servidor
`cur.skip(numElementos)`	Permite saltar *numElementos* con el cursor	Servidor

La consulta no se ejecuta hasta que el cursor comprueba o pasa al siguiente documento (`next` / `hasNext`), por ello que tanto `limit` como `sort` (ambos modifican el cursor) sólo se pueden realizar antes de recorrer cualquier elemento del cursor.

Tras realizar una consulta con `find`, realmente se devuelve un cursor. Un uso muy habitual es encadenar una operación de `find` con `sort` y/o `limit` para ordenar el resultado por uno o más campos y posteriormente limitar el número de documentos a devolver.

Así pues, si quisiéramos obtener la calificación con la nota más alta, podríamos hacerlo así:

```
> db.grades.find({ type:'homework'}).sort({score:-1}).limit(1)
```

Por ejemplo, si queremos paginar las notas de 10 en 10, a partir de la tercera página, podríamos hacer algo así:

```
> db.grades.find().sort({score:-1}).limit(10).skip(20);
```

Autoevaluación

> A partir de la colección *grades*, escribe un consulta que obtenga los documentos de tipo "exam" ordenados descendentemente y que obtenga los documentos de 51 al 70. [10]

Contando Documentos

Para contar el número de documentos, en vez de `find` usaremos el método `count`. Por ejemplo:

```
> db.grades.count({type:"exam"})
```

[10] `db.grades.find({"type":"exam"}).sort({"score":-1}).skip(50).limit(20)`

```
> db.grades.find({type:"exam"}).count()
> db.grades.count({type:"essay", score:{$gt:90}})
```

Como se puede observar en el ejemplo, también lo podemos utilizar como método de un cursor.

2.5. Actualizando documentos

Para actualizar (y fusionar datos), se utiliza el método `update` con 2 parámetros: el primero es la consulta para averiguar sobre qué documentos, y en el segundo parámetro, los campos a modificar.

Modificando un documento

```
> db.people.update({nombre:"Steve Jobs"},{nombre:"Domingo Gallardo",
  salario: 1000000})
```

`update` hace un **reemplazo** de los campos, es decir, si en el origen había 100 campos y en el `update` sólo ponemos 2, el resultado sólo tendrá 2 campos. ¡Cuidado que puede ser muy peligroso!

Si cuando vamos a actualizar, en el criterio de selección no encuentra el documento sobre el que hacer los cambios, no se realiza ninguna acción.

Si quisiéramos que en el caso de no encontrar nada insertase un nuevo documento, acción conocida como **upsert** (***up**date* + *in**sert***), hay que pasarle un tercer parámetro al método con el objeto `{upsert:true}`

Ejemplo *upsert*

```
db.people.update({nombre:"Domingo Gallardo"},{name:"Domingo Gallardo",
  twitter: '@domingogallardo'}, {upsert: true})
```

Otra manera de realizar un *upsert* es mediante la operación `save`, que ya hemos visto anteriormente. Así pues, si reescribimos la consulta anterior tendríamos (siempre y cuando considerasemos que el `nombre` actúa como el campo `_id`):

Ejemplo *save*

```
db.people.save({nombre:"Domingo Gallardo"},{name:"Domingo Gallardo",
  twitter: '@domingogallardo'})
```

Si no indicamos el valor `_id`, el comando `save` asume que es una inserción e inserta el documento en la colección.

Operadores de actualización

MongoDB ofrece un conjunto de operadores para simplificar la modificación de campos.

Para evitar el reemplazo, hay que usar la variable `$set` (si el campo no existe, se creará).

Por ejemplo, para modificar el salario haríamos:

Ejemplo `$set`

```
> db.people.update({nombre:"Aitor Medrano"},{ $set:{salario: 1000000} })
```

Mediante `$inc` podemos incrementar el valor de una variable.

En cambio, si queremos incrementar el salario haríamos:

Ejemplo `$inc`

```
> db.people.update({nombre:"Aitor Medrano"},{ $inc:{salario: 1000} })
```

Para eliminar un campo de un documento, usaremos el operador `$unset`.

De este modo, para eliminar el campo `twitter` de una persona haríamos:

Ejemplo `$unset`

```
> db.people.update({nombre:"Aitor Medrano"},{ $unset:{twitter: ''} })
```

Otros operadores que podemos utilizar son `$mul`, `$min`, `$max` y `$currentDate`. Podemos consultar todos los operadores disponibles en http://docs.mongodb.org/manual/reference/operator/update/

Autoevaluación

Tras realizar la siguiente operación sobre una colección vacía:

```
> db.people.update({nombre:'yo'}, {'$set':{'hobbies':
['gaming', 'sofing']}}, {upsert: true} );
```

¿Cuál es el estado de la colección? [11]

Realmente podemos dividir las actualizaciones en cuatro tipos:

- reemplazo completo
- modificar un campo
- hacer un *upsert*
- o actualizar múltiples documentos

Actualización Múltiple

Un aspecto que no hemos comentado y el cual es muy importante es que, si a la hora de actualizar la búsqueda devuelve más de un resultado, la actualización sólo se realiza sobre el primer resultado obtenido.

Para modificar múltiples documentos, en el tercer parámetro indicaremos `{multi: true}`. ¡Esta es una diferencia sustancial respecto a SQL!

Por ejemplo, para incrementar todas las calificaciones de los exámenes en un punto haríamos:

[11] Al estar la colección vacía, insertará un nuevo registro

Ejemplo Actualización Múltiple

```
> db.grades.update({type:'exam'}, {'$inc':{'score':1}}, {multi: true} );
```

Cuando se hace una actualización múltiple, *MongoDB* no la realiza de manera atómica (no soporta transacciones *isolated*), lo que provoca que se puedan producir pausas (*pause yielding*). Cada documento si es atómico, por lo que ninguno se va a quedar a la mitad.

MongoDB ofrece el método `findAndModify` para encontrar y modificar un documento de manera atómica, y así evitar que, entre la búsqueda y la modificación, el estado del documento se vea afectado. Además, devuelve el documento modificado. Un caso de uso muy común es para contadores y casos similares.

Encontrar y Modificar de manera atómica - `findAndModify`

```
> db.grades.findAndModify({
  query: { student_id: 0, type: "exam"},
  update: { $inc: { score: 1 } },
  new: true
})
```

Por defecto, el documento devuelto será el resultado que ha encontrado con la consulta. Si queremos que nos devuelva el documento modificado con los cambios deseados, necesitamos utilizar el parámetro `new` a `true`. Si no lo indicamos o lo ponemos a `false`, tendremos el comportamiento por defecto.

Para el resto de opciones que ofrece `findAndModifiy` se recomienda consultar la documentación (http://docs.mongodb.org/master/reference/method/db.collection.findAndModify/)

Finalmente, un caso particular de las actualizaciones es la posibilidad de renombrar un campo mediante el operador `$rename`:

Renombrando un campo con `$rename`

```
> db.people.update( { _id: 1 }, { $rename:
  { 'nickname': 'alias', 'cell': 'movil' } } )
```

Podemos consultar todas las opción de configuración de una actualización en http://docs.mongodb.org/manual/reference/method/db.collection.update/

Actualizaciones sobre Arrays

Para trabajar con arrays necesitamos nuevos operadores que nos permitan tanto introducir como eliminar elementos de una manera más sencilla que sustituir todos los elementos del array.

Los operadores que podemos emplear para trabajar con arrays son:

Operador	Propósito
`$push`	Añade un elemento
`$pushAll`	Añade varios elementos
`$addToSet`	Añade un elemento sin duplicados

Operador	Propósito
`$pull`	Elimina un elemento
`$pullAll`	Elimina varios elementos
`$pop`	Elimina el primer o el último

Preparando los ejemplos

Para trabajar con los arrays, vamos a suponer que tenemos una colección de *enlaces* donde vamos a almacenar un documento por cada site, con un atributo *tags* con etiquetas sobre el enlace en cuestión

```
> db.enlaces.insert( {titulo:"www.google.es", tags:
["mapas", "videos"] })
```

De modo que tendríamos el siguiente objeto:

```
{
 "_id" : ObjectId("54f9769212b1897ae84190cf"),
 "titulo" : "www.google.es",
 "tags" : [
  "mapas", "videos"
 ]
}
```

Añadiendo elementos

Si queremos añadir un elemento, usaremos el operador `$push`. Si queremos añadir varios elementos de una sola vez, usaremos `$pushAll`.

```
> db.enlaces.update({titulo:"www.google.es"},{$push: {tags:"blog"}})
> db.enlaces.update({titulo:"www.google.es"},{$pushAll: {tags:
["calendario","email","mapas"]}})
```

Al hacer estar modificación, el resultado del documento sería:

```
{
  "_id" : ObjectId("54f9769212b1897ae84190cf"),
  "titulo" : "www.google.es",
  "tags" : [
    "mapas",
    "videos",
    "blog",
    "calendario",
    "email",
    "mapas"
  ]
}
```

Tanto `$push` como `$pushAll` no tienen en cuenta lo que contiene el array, por tanto, si un elemento ya existe, se repetirá y tendremos duplicados. Si queremos evitar los duplicados, usaremos `$addToSet`:

```
> db.enlaces.update({titulo:"www.google.es"},{$addToSet:
  {tags:"buscador"}})
```

Si queremos añadir más de un campo a la vez sin duplicados, debemos anidar el operador `$each`:

```
> db.enlaces.update({titulo:"www.google.es"},{$addToSet: {tags: { $each:
["drive", "traductor"] }  }})
```

Eliminando elementos

En cambio, si queremos eliminar elementos de un array, usaremos el operador `$pull`:

```
> db.enlaces.update({titulo:"www.google.es"},{$pull: {tags:"traductor"}})
```

Similar al caso anterior, con `$pullAll`, eliminaremos varios elementos de una sola vez:

```
> db.enlaces.update({titulo:"www.google.es"},{$pullAll: {tags:
["calendario","email"]}})
```

Otra manera de eliminar elementos del array es mediante `$pop`, el cual elimina el primero (`-1`) o el último (`1`) elemento del array:

```
> db.enlaces.update({titulo:"www.google.es"},{$pop: {tags:-1}})
```

Operador posicional

Por último, tenemos el operador posicional, el cual se expresa con el símbolo `$` (http://docs.mongodb.org/master/reference/operator/update/positional/) y nos permite modificar el elemento que ocupa una determinada posición del array.

Supongamos que tenemos las calificaciones de los estudiantes (colección `students`) en un documento con una estructura similar a la siguiente:

```
{ "_id" : 1, "grades" : [ 80, 85, 90 ] }
```

y queremos cambiar la calificación de `80` por `82`. Mediante el operador posicional haremos:

Modificando un array con el operador posicional

```
> db.students.update( { _id: 1, grades: 80 }, { $set: { "grades.$" : 82
  } } )
```

De manera similar, si queremos modificar parte de un documento el cual forma parte de un array, debemos usar la notación punto tras el `$`:

Por ejemplo, supongamos que tenemos estas calificación de un determinado alumno:

Ejemplo de calificación de un alumno, las cuales forman parte de un objeto dentro de un array

```
{ "_id" : 4, "grades" :
  [ { grade: 80, mean: 75, std: 8 },
    { grade: 85, mean: 90, std: 5 },
    { grade: 90, mean: 85, std: 3 } ] }
```

Podemos observar como tenemos cada calificación como parte de un objeto dentro de un array. Si queremos cambiar el valor de `std` a `6` de la calificación cuya nota es `85`, haremos:

Modificando un elemento de un objeto dentro de un array

```
> db.students.update( { _id: 4, "grades.grade": 85 }, { $set: { "grades.
$.std" : 6 } } )
```

Es decir, el `$` referencia al documento que ha cumplido el filtro de búsqueda.

Podemos consultar toda la documentación disponible sobre estos operadores en http://docs.mongodb.org/manual/reference/operator/update-array/

2.6. Borrando Documentos

Para borrar, usaremos el método `remove`, el cual funciona de manera similar a `find`. Si no pasamos ningún parámetro, borra toda la colección documento a documento. Si le pasamos un parámetro, éste será el criterio de selección de documentos a eliminar.

```
> db.people.remove({nombre:"Domingo Gallardo"})
```

Al eliminar un documento, no podemos olvidar que cualquier referencia al documento que existe en la base de datos seguirá existiendo. Por este motivo, manualmente también hay que eliminar o modificar esas referencias.

Si queremos borrar toda la colección, es más eficiente usar el método `drop`, ya que también elimina los índices.

```
> db.people.drop()
```

Recordad que eliminar un determinado campo de un documento no se considera un operación de borrado, sino una actualización mediante el operador `$unset`.

2.7. Control de Errores

En versiones anteriores a la 2.6, si queremos averiguar qué ha sucedido, y si ha fallado conocer el motivo, deberemos ejecutar el siguiente comando con `getLastError` (http://docs.mongodb.org/master/reference/command/getLastError/):

Control de Errores

```
> db.runCommand({getLastError:1})
```

Para ello, ejecutaremos la sentencia después de haber realizado una operación, para obtener información sobre la última operación realizada.

Si la última operación ha sido una modificación mediante un `update` podremos obtener el número de registros afectados, o si es un `upsert` podremos obtener si ha insertado o modificado el documento. Finalmente, en el caso de una operación de borrado, podemos obtener el número de documentos eliminados.

Desde la versión 2.6, *MongoDB* devuelve un objeto `WriteResult` con información del número de documentos afectados (`nInserted`), y en el caso de un error, un documento en la propiedad `writeError`:

```
> db.people.insert({"_id":"error","nombre":"Pedro Casas", "edad":38})
WriteResult({ "nInserted" : 1 })
> db.people.insert({"_id":"error","nombre":"Pedro Casas", "edad":38})
WriteResult({
 "nInserted" : 0,
 "writeError" : {
  "code" : 11000,
  "errmsg" : "insertDocument :: caused by :: 11000 E11000 duplicate key
 error index: expertojava.people.$_id_  dup key: { : \"error\" }"
 }
})
```

Más información en http://docs.mongodb.org/master/reference/method/db.collection.insert/#writeresult

A continuación vamos a estudiar como realizar todas estas operación mediante el driver *Java* que ofrece *MongoDB*.

2.8. MongoDB desde *Java*

Para interactuar desde *Java* con *MongoDB* disponemos de diferentes alternativas:

- Trabajar directamente con el driver *Java*
- Utilizar un abstracción JPA

En nuestro caso, nos vamos a centrar en el uso del driver.

Para descargar el driver, tal como vimos en la unidad anterior, hemos de utilizar la siguiente dependencia *Maven*:

```
<dependency>
  <groupId>org.mongodb</groupId>
  <artifactId>mongo-java-driver</artifactId>
  <version>2.13.1</version>
</dependency>
```

Todas las clases explicadas a continuación pertenecen al paquete `com.mongodb`. Toda la información del API de MongoDB la podemos encontrar en http://api.mongodb.org/java/current/ e información del driver en http://mongodb.github.io/mongo-java-driver/2.13/

`MongoClient`

Para conectarnos desde *Java*, tenemos que crear un `MongoClient`, el cual gestiona internamente un *pool* de conexiones. Su constructor se sobrecarga para permitir la conexión a una URI, a un determinado puerto o a un conjunto de réplicas. Podemos consultar todas estas opciones en http://api.mongodb.org/java/2.13/com/mongodb/MongoClient.html

A partir de un `MongoClient`, podremos obtener una `DB` y de ésta una `DBCollection`:

Ejemplo de conexión a MongoDB con Java (`HolaMongoDB.java`)

```
MongoClient cliente = new MongoClient();  ❶

DB db = cliente.getDB("expertojava");
DBCollection col = db.getCollection("people");

System.out.println("doc:" + col.findOne());  ❷
```

❶ `MongoClient` realiza la conexión con la base de datos. El constructor por defecto se conecta con `localhost` al puerto `27017`. Además, lanza una `UnknownHostException` cuando no encuentra un servidor funcionando

❷ `DBCollection` nos permite interactuar con la colección, y sobre ella realizaremos las operaciones CRUD.

Sobre un `MongoClient` podemos destacar los siguientes métodos:

- `getDB(String nombre)` → recupera la base de datos indicada
- `dropDatabase(String nombre)` → elimina la base de datos indicada
- `getDatabaseNames()` → obtiene el nombre de las bases de datos existentes

Sobre una `DB` podemos destacar los siguientes métodos:

- `getCollection(String nombre)` → recupera la colección indicada
- `command(DBObject obj)` → ejecuta un comnado
- `createCollection(String col)` → crea una nueva colección sobre la DB activa
- `dropDatabase()` → elimina la base de datos activa
- `getCollectionNames()` → obtiene el nombre de las colecciones existentes
- `getLastError()` → obtiene el último error, si lo hay, de la operación previa (*deprecated*)
- `shutdownServer()` → detiene el servidor

`DBObject`

Para representar un documento JSON se utiliza el interfaz `DBObject`, el cual se emplea como parámetro para la mayoría de operaciones. Su funcionamiento es similar a un mapa donde las claves están ordenadas.

Para crear un documento necesitamos una instancia de `BasicDBObject`. Por ejemplo, podremos crear un documento del siguiente modo:

Rellenando un `BasicDBObject`

```
BasicDBObject doc = new BasicDBObject();
doc.put("nombre", "Aitor Medrano");
doc.put("fnac", new Date(234832423));
doc.put("casado", true);
doc.put("hijos", 2);
doc.put("hobbies",
 Arrays.asList("programación","videojuegos", "baloncesto")); ❶
doc.put("direccion", new BasicDBObject("calle", "Mayor")  ❷
  .append("ciudad", "Elx")
  .append("cp", "03206"));
```

❶ Los arrays realmente son implementaciones de `BasicDBList`, el cual es una lista de `DBObject`

❷ Además de usar el método `put` para añadir un atributo a un objeto, podemos crear un objeto con su constructor de clave/valor, o mediante el método `append` para concatenar un objeto al existente.

Además, las operaciones para realizar consultas devuelven objetos `DBObject` o bien son listas (`List<DBObject>`). Para acceder a los campos de un `DBObject` emplearemos el método `get`:

Obteniendo datos a partir de un `DBObject`

```
Persona p = new Persona(); ❶
p.setNombre((String) obj.get("nombre"));
p.setFnac((Date) obj.get("fnac"));
p.setHijos((Integer) obj.get("hijos"));

BasicDBList hobbies = (BasicDBList) obj.get("hobbies");
p.setHobbies(hobbies.toArray(new String[0])); ❷
```

❶ Supongamos que tenemos una clase de modelo `Persona` compuesta únicamente de *getters/setters* sobre las propiedades del objeto

❷ Dentro de la clase `Persona`, tenemos la siguiente declaración `String[] hobbies`

Para realizar operaciones, tras conectar del `MongoClient` una `DB`, y de la `DB` una `DBCollection` podemos realizar las operaciones de inserción, consulta, modificación y borrado.

Inserción

Para insertar datos emplearemos el método `coleccion.insert(DBObject objeto)`.

Tras insertar el objeto, *MongoDB* rellenará automáticamente la clave `_id`

Autoevaluación

¿Funcionará el segundo `insert`? [12]

[12] Sí, porque la llamada a `removeField` borrará la clave añadida por el driver en el primer `insert`

```
MongoClient client = new MongoClient();
DB db = client.getDB("expertojava");
DBCollection people = db.getCollection("people");
DBObject doc = new BasicDBObject("nombre", "Aitor
 Medrano")
  .append("twitter", "@aitormedrano");

try {
  people.insert(doc);      // primer insert
  doc.removeField("_id");  // elimina el campo "_id"
  people.insert(doc);      // segundo insert
} catch (Exception e) {
  e.printStackTrace();
}
```

Consultas

Para hacer consultas utilizaremos el método `find` o `findOne`, de manera similar al uso desde el *shell*. Hay que destacar que cuando hacemos una consulta con `find` recuperamos un `DBCursor`, el cual funciona como un iterador y que nos permite recorrer los documentos encontrados.

A continuación tenemos un ejemplo de su uso:

```
MongoClient client = new MongoClient();
DB db = client.getDB("expertojava");
DBCollection coleccion = db.getCollection("pruebas");
collection.drop(); ❶

// insertamos 10 documentos con un número aleatorio
for (int i = 0; i < 10; i++) {
  coleccion.insert(new BasicDBObject("numero", new
 Random().nextInt(100)));
}

System.out.println("Primero:");
DBObject uno = coleccion.findOne(); // Encuentra uno
System.out.println(uno);

System.out.println("\nTodos: ");
DBCursor cursor = coleccion.find(); // Encuentra todos
try {
  while (cursor.hasNext()) {
    DBObject otro = cursor.next();
    System.out.println(otro);
  }
} finally {
  cursor.close(); ❷
}

System.out.println("\nTotal:" + coleccion.count());
```

❶ Antes de rellenar la colección, la vaciamos para siempre partir de cero.

❷ Es recomendable cerrar el cursor tras finalizar su uso

Criterios

Supongamos que partimos de una colección con los siguientes datos:

```
MongoClient cliente = new MongoClient();
DB db = cliente.getDB("expertojava");
DBCollection coleccion = db.getCollection("pruebas");
coleccion.drop();

// insertamos 10 documentos con 2 números aleatorios
for (int i = 0; i < 10; i++) {
  coleccion.insert(
    new BasicDBObject("x", new Random().nextInt(2))
      .append("y", new Random().nextInt(100)));
}
```

Para añadir criterios a las consultas, podemos hacerlo de dos maneras:

1. Usando el objeto `QueryBuilder`, el cual ofrece diferentes métodos asociados a los operadores lógicos y aritméticos, y que permite hacer consultas a más alto nivel, lo que desacopla al driver de la sintaxis de *MongoDB*.

   ```
   QueryBuilder builder =
    QueryBuilder.start("x").is(0).and("y").greaterThan(10).lessThan(90);
   long cantidadBuilder = coleccion.count(builder.get());
   ```

   Más información en http://api.mongodb.org/java/2.13/com/mongodb/QueryBuilder.html

2. Añadiendo las condiciones de manera similar a como se realiza mediante el shell creando `BasicDBObject`

   ```
   DBObject query = new BasicDBObject("x", 0).append("y", new
    BasicDBObject("$gt", 10).append("$lt", 90));
   long cantidadQuery = coleccion.count(query);
   ```

En ambos casos, le podemos pasar tanto el `DBObject` como el `QueryBuilder` a los métodos `find`:

```
System.out.println("\nConsultas: ");
DBCursor cursor = coleccion.find(builder.get());  ❶
try {
  while (cursor.hasNext()) {
    DBObject cur = cursor.next();
    System.out.println(cur);
  }
} finally {
  cursor.close();
}
```

❶ A partir de un `QueryBuilder`, mediante el método `get()` obtenemos un `DBObject`

La versión 3.0 ha introducido nuevos filtros para facilitar el filtrado de campos, como `eq()`, `gt()`, `and()`, etc… Más información en http://api.mongodb.org/java/current/com/mongodb/client/model/Filters.html

Selección de campos

Para elegir que datos queremos proyectar y que aparezcan como resultado de la consulta, el método `find` permite que indiquemos con un segundo parámetro los campos deseados mediante un `BasicDBObject` poniendo como nombre los nombres de los atributos y como valores `true`/`false` dependiendo de si queremos que se devuelvan o no.

```
DBObject query =
 QueryBuilder.start("x").is(0).and("y").greaterThan(10).lessThan(70).get();
DBObject proyeccion = new BasicDBObject("y", true).append("_id", false);
  ❶

DBCursor cursor = coleccion.find(query, proyeccion);  ❷
try {
  while (cursor.hasNext()) {
    DBObject cur = cursor.next();
    System.out.println(cur);
  }
} finally {
  cursor.close();
}
```

❶ Proyecta el atributo `y` y no muestra el `_id`
❷ Al método `find()` le pasamos tanto la consulta como la proyección

Autoevaluación

Dada una variable `alumnos` de tipo `DBCollection`,

```
alumnos.find(new
 BasicDBObject("tlfno", 1).append("_id", 0))
alumnos.find(new BasicDBObject("tlfno", 1))
alumnos.find(new BasicDBObject(), new
 BasicDBObject("tlfno", 1).append("_id", 0))
alumnos.find(new
 BasicDBObject("tlfno", 1).append("_id", 0), new
 BasicDBObject()
```

¿Cual de las anteriores instrucciones nos permitirá obtener todos los documentos pero recuperando únicamente el campo `tlfno` ? [13]

Campos anidados

Cuando tenemos un documento que forma parte del valor del atributo de otro documento, usaremos la notación `.` para navegar y descender un nivel.

[13] la correcta es la 3ª instrucción, ya que el primer parámetro del `find` indica que quiere todos los documentos, y con el segundo ya fija los campos a devolver

```
// insertamos 10 documentos con puntos de inicio y fin aleatorios
for (int i = 0; i < 10; i++) {
  coleccion.insert( ❶
    new BasicDBObject("_id", i)
    .append("inicio", new BasicDBObject("x", rand.nextInt(90)).append("y",
 rand.nextInt(90)))
    .append("fin", new BasicDBObject("x", rand.nextInt(90)).append("y",
 rand.nextInt(90)))
  );
}

QueryBuilder builder = QueryBuilder.start("inicio.x").greaterThan(50); ❷

DBCursor cursor = coleccion.find(builder.get(), new
 BasicDBObject("inicio.y", true).append("_id", false)); ❸
```

❶ Crea 10 documentos del tipo `{ "_id" : 0 , "inicio" : { "x" : 28 , "y" : 46} , "fin" : { "x" : 37 , "y" : 51}}`

❷ La consulta filtra por el campo anidado `inicio.x`

❸ La proyección sólo muestra el campo anidado `inicio.y`

Autoevaluación

Con el siguiente fragmento de código, ¿Qué piensas que sucederá si en la colección existe un documento que cumple con la consulta pero no que no tiene una clave llamada `medio.url` ? [14]

```
DBObject encuentraUrlPorTipoMedio(DBCollection videos,
 String tipoMedio) {
  DBObject query = new BasicDBObject("medio.tipo",
 mediaType);
  DBObject proyeccion = new BasicDBObject("medio.url",
 true);

  return videos.findOne(query, proyeccion);
}
```

1. Lanzará una excepción
2. Devolverá un documento vacío
3. Devolverá un documento que contiene un único campo que contiene el `_id` del documento
4. No hay suficiente información para responder

Trabajando con `DBCursor`

Del mismo modo que con el *shell*, podemos utilizar los métodos `sort`, `skip` y `limit` sobre un `DBCursor`.

Suponiendo que tenemos los mismos datos del ejemplo anterior:

[14] lanzará una excepción ya que no puede aplicar la proyección

Ejemplo de métodos sobre un cursor -

```
DBCursor cursor = coleccion.find().sort(new
 BasicDBObject("inicio.x", 1).append("inicio.y", -1)).skip(2).limit(5);
```

Modificación

Si queremos modificar un documento, el *driver* nos ofrece el método `update` con diferentes sobrecargas:

- `update(DBObject origen, DBOject destino)` → para cambiar un documento por otro, o aplicar un operador sobre *destino* y actuar conforme indique el operador
- `update(DBObject origen, DBOject destino, boolean upsert, boolean multiple)` → igual que el anterior, más la posibilidad de indicar de si hacemos un *upsert* o si la actualización es múltiple.

Ejemplo de actualización en Java -

```
List<String> nombres =
 Arrays.asList("Laura", "Pedro", "Ana", "Sergio", "Helena");
for (String nombre : nombres) {
  coleccion.insert(new BasicDBObject("_id", nombre));
}

coleccion.update(new BasicDBObject("_id", "Laura"), new
 BasicDBObject("hermanos", 2)); ❶

coleccion.update(new BasicDBObject("_id", "Laura"), new
 BasicDBObject("$set", new BasicDBObject("edad", 34))); ❷

coleccion.update(new BasicDBObject("_id", "Laura"), new
 BasicDBObject("sexo", "F")); ❸

coleccion.update(new BasicDBObject("_id", "Emilio"), new
 BasicDBObject("$set", new BasicDBObject("edad", 36)), true, false); ❹

coleccion.update(new BasicDBObject(), new BasicDBObject("$set", new
 BasicDBObject("titulo", "Don")), false, true);  ❺
```

❶ Le asigna a `Laura` 2 hermanos
❷ Le añade la edad, pero manteniendo el resto de atributos
❸ Realiza un reemplazo completo con lo que `Laura` solo tiene el atributo `sexo`
❹ Realiza un *upsert* con lo que inserta una nueva persona
❺ Realiza una actualización múltiple, con lo que todas las personas tendrán el atributo `"titulo":"Don"`

Borrado

Para borrar un documento usaremos el método `remove(DBObject obj)` sobre la colección:

```
coleccion.remove(new BasicDBObject("_id", "Sergio"));
```

Podéis consultar un ejemplo completo de CRUD en http://www.javahotchocolate.com/notes/mongodb-crud.html

2.9. Mapping de Objetos

Hasta ahora, el mapeo de objetos POJO a JSON lo estamos realizando a mano, atributo por atributo. Otras opciones alternativas es automatizar el *mapping* con herramientas como:

- **Jackson** (https://github.com/FasterXML/jackson), y en particular **MongoJack** (http://mongojack.org), que facilitan la conversión de BSON a objetos Java.
- **Morphia** (https://github.com/mongodb/morphia/wiki): framework *ORM* ligero, similar a *Hibernate*, para automatiza el *mapping* de manera automática mediante anotaciones.
- **Spring Data MongoDB**: *wrapper* que facilita la conexión y simplifica el uso de consultas: (http://projects.spring.io/spring-data-mongodb/)

En esta sesión no vamos a entrar en detalle en estas herramientas por falta de tiempo, pero cabe destacar que ofrecen una serie de ventajas que conviene conocer:

- Desarrollo más ágil que con mapeo manual.
- Anotación unificada entre todas las capas.
- Manejo de tipos amigables, por ejemplo, para cambios de tipos de `long` a `int` de manera transparente.
- Posibilidad de incluir mapeos diferentes entre la base de datos y las capas del servidor web para transformar los formatos como resultado de una llamada REST.

Otra solución flexible es **Hibernate OGM** (http://hibernate.org/ogm/), con soporte para *Infinispan*, *Ehcache*, *MongoDB* y *Neo4j*. Más información sobre *Hibernate OGM* y *MongoDB* en http://docs.jboss.org/hibernate/ogm/4.1/reference/en-US/html/ogm-mongodb.html

Finalmente, si nos decidimos por acceder via el driver directamente y empleamos EJB para ofrecer una capa de servicios, es conveniente encapsular el cliente dentro de un *Singleton*. Podemos ver un ejemplo completo en http://www.codingpedia.org/ama/how-to-connect-to-mongodb-from-a-java-ee-stateless-application/

2.10. Ejercicios

En esta sesión, vamos a centrarnos en utilizar los comandos aprendidos para interactuar con los datos de la base de datos `ejercicios`.

Posteriormente, mediante *Java* también interactuaremos con estos datos.

(1 punto) Ejercicio 21. Consultas desde `mongo`

Escribe la operación necesaria y el resultado para averiguar:

1. Número de ciudades.
2. Datos de la ciudad de `Elx`.
3. Población de la ciudad de `Vergel`.
4. Cantidad de ciudades en España `({"country":"ES"})`.

5. Datos de las ciudades españolas con más de 1.000.000 de habitantes.
6. Cantidad de ciudades de Andorra `({"country":"AN"})` y España.
7. Listado con el nombre y la población de las 10 ciudades más pobladas.
8. Nombre de las distintas zonas horarias en España.
9. Ciudades españolas que su zona horaria no sea `Europe/Madrid`.
10. Ciudades españolas que comiencen por `Ben`
11. Ciudades que su zona horaria sea `Atlantic/Canary` o `Africa/Ceuta`.
12. Nombre y población de las tres ciudades europeas más pobladas.
13. Cantidad de ciudades españolas cuya coordenadas de longitud estén comprendidas entre `-0.1` y `0.1`.

Escribe los comandos necesarios y el resultado en `ej21.txt` comando necesario.

(1 punto) Ejercicio 22. Modificaciones desde `mongo`

Escribe la operación necesarias para:

1. Modifica la población de tu ciudad a 1.000.000
2. Incrementa la población de `Elx` en 666 personas.
3. Reduce la cantidad de todas las ciudades de Andorra en 5 personas.
4. Modifica la ciudad de `Gibraltar` para que sea española (tanto el país como la zona horaria).
5. Modifica todas las ciudades y añade un atributo `tags` que contenga un array vacío.
6. Modifica todas las ciudades españolas y añade al atributo `tags` el valor `sun`.
7. Modifica el valor de `sun` de la ciudad `A Coruña` y sustitúyelo por `rain`.
8. Renombra en las ciudades de Andorra, el atributo `population` por `poblacion`.
9. Elimina las coordenadas de `Gibraltar`.
10. Elimina tu entrada

Escribe los comandos necesarios y el resultado en `ej22.txt` comando necesario.

(1.5 puntos) Ejercicio 23. Operaciones desde *Java*

Los siguientes ejercicios se basan en el uso de *Java*. Para ello, los ejercicios estarán dentro del paquete `es.ua.expertojava.nosql`, en una clase nombrada como `ConsultasEjercicios`.

En base a los datos sobre ciudad almacenados en la colección `cities` de la base de datos `ejercicios`, usaremos la siguiente clase `Ciudad`:

```
public class Ciudad {
  private String name;
  private String country;
  private String timezone;
```

```
    private long population;
    private float longitude;
    private float latitude;

    public String getName() {
      return name;
    }
    public void setName(String name) {
      this.name = name;
    }

    public String getCountry() {
      return country;
    }
    public void setCountry(String country) {
      this.country = country;
    }

    public String getTimezone() {
      return timezone;
    }
    public void setTimezone(String timezone) {
      this.timezone = timezone;
    }

    public long getPopulation() {
      return population;
    }

    public void setPopulation(long population) {
      this.population = population;
    }

    public float getLongitude() {
      return longitude;
    }
    public void setLongitude(float longitude) {
      this.longitude = longitude;
    }

    public float getLatitude() {
      return latitude;
    }
    public void setLatitude(float latitude) {
      this.latitude = latitude;
    }
}
```

Para poder interactuar con este objeto, deberás crear dos métodos privados que se encarguen del *mapping* entre BSON y el objeto Java:

- `Ciudad mapDBObject2Ciudad(DBObject dbo)`

  # Al asociar la población desde un `DBObject` a una propiedad *Java* de tipo `long`, *MongoDB* en ocasiones devuelve un entero y en otras un entero largo. Para evitar problemas de *casting* podemos hacer:

```
ciudad.setPopulation(((Number) dbo.get("population")).longValue());
```

- `DBObject mapCiudad2DBOject(Ciudad ciudad)`

Una vez creado estos métodos, añadiremos los siguientes métodos para interactuar con los datos:

- `void insertaCiudad(Ciudad ciudad)`: A partir de una *ciudad*, inserta los datos en la colecciones `cities`.
- `List<Ciudad> listarCiudades()`: Obtiene todas las ciudades de la colección.
- `List<Ciudad> listarCiudades(String pais)`: Obtiene todos las ciudades de un determinado país.
- `List<String> listarPaises()`: Obtiene un listado de los paises (sin repeticiones).

3. Rendimiento en *MongoDB*

En esta unidad vamos a estudiar como diseñar el esquema, así como el uso de índices y otras herramientas avanzadas para mejorar el rendimiento.

3.1. Diseñando el Esquema

MongoDB es una base de datos documental, no relacional, donde el esquema no se debe basar en el uso de claves ajenas/*joins*, ya que no existen.

A la hora de diseñar un esquema, si nos encontramos que el esquema esta en 3FN o si cuando hacemos consultas (recordad que no hay *joins*) estamos teniendo que realizar varias consultas de manera programativa (primero acceder a una tabla, con ese `_id` ir a otra tabla, etc.…) es que no estamos siguiendo el enfoque adecuado.

MongoDB no soporta transacciones, ya que su enfoque distribuido dificultaría y penalizaría el rendimiento. En cambio, sí que asegura que las operaciones sean atómicas. Los posibles enfoques para solucionar la falta de transacciones son:

1. Restructurar el código para que toda la información esté contenida en un único documento.
2. Implementar un sistema de bloqueo por software (semáforo, etc…).
3. Tolerar un grado de inconsistencia en el sistema.

Dependiendo del tipo de relación entre dos documentos, normalizaremos los datos para minimizar la redundancia pero manteniendo en la medida de lo posible que mediante operaciones atómicas se mantenga la integridad de los datos. Para ello, bien crearemos referencias entre dos documentos o embeberemos un documento dentro de otro.

Referencias

Las aplicaciones que emplean *MongoDB* utilizan dos técnicas para relacionar documentos:

- Referencias **Manuales**
- Uso de **DBRef**

Referencias Manuales

De manera similar a una base de datos relacional, se almacena el campo `_id` de un documento en otro documento a modo de clave ajena. De este modo, la aplicación realiza una segunda consulta para obtener los datos relaciones. Estas referencias son sencillas y suficientes para la mayoría de casos de uso.

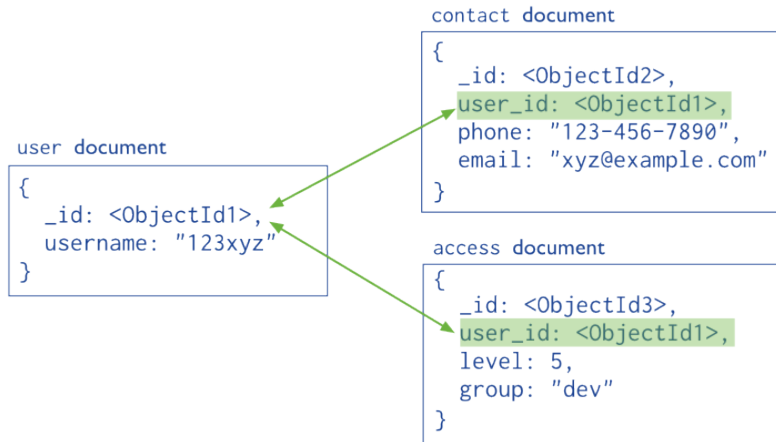


Figura 17. Referencias manuales

Por ejemplo, si nos basamos en el gráfico anterior, podemos conseguir referenciar estos objetos del siguiente modo:

Ejemplo de referencia manual - Usuario/Contacto

```
var idUsuario = ObjectId();

db.usuario.insert({
  _id: idUsuario,
  nombre: "123xyz"
});

db.contacto.insert({
  usuario_id: idUsuario,
  telefono:  "123 456 7890",
  email: "xyz@ejemplo.com"
});
```

DBRef

Son referencias de un documento a otro mediante el valor del campo `_id`, el nombre de la colección y, opcionalmente, el nombre de la base de datos. Estos objetos siguen una convención para representar un documento mediante la notación `{ "$ref" : <nombreColeccion>, "$id" : <valorCampo_id>, "$db" : <nombreBaseDatos> }`.

Al incluir estos nombres, las *DBRef* permite referenciar documentos localizados en diferentes colecciones.

Así pues, si reescribimos el código anterior mediante *DBRef* tendríamos que el contacto queda de la siguiente manera:

Ejemplo de DBRef - Usuario/Contacto

```
db.contacto.insert({
  usuario_id: new DBRef("usuario", idUsuario),
  telefono:  "123-456-7890",
  email: "xyz@example.com"
});
```

De manera similar a las referencias manuales, mediante consultas adicionales se obtendrán los documentos referenciados.

Muchos *drivers* (incluido el de *Java*, mediante la clase `DBRef`) contienen métodos auxiliares que realizan las consultas con referencias *DBRef* automáticamennte.

Desde la propia documentación de *MongoDB*, recomiendan el uso de referencias manuales, a no ser de que dispongamos documentos de una colección que referencian a documentos que se encuentran en varias colecciones diferentes.

Datos Embebidos

En cambio, si dentro de un documento almacenamos los datos mediante sub-documentos, ya sea dentro de un atributo o un array, podremos obtener todos los datos mediante un único acceso.

Figura 18. Datos Embebidos

Generalmente, emplearemos datos embebidos cuando tengamos:

- relaciones *"contiene"* entre entidades, entre relaciones de documentos "uno a uno" o "uno a pocos".
- relaciones "uno a muchos" entre entidades. En estas relaciones los documentos hijo (o "muchos") siempre aparecen dentro del contexto del padre o del documento "uno".

Los datos embebidos ofrecen mejor rendimiento al permitir obtener los datos mediante una única operación, así como modificar datos relacionados en una sola operación atómica de escritura.

Un aspecto a tener en cuenta es que un documento *BSON* puede contener un máximo de 16MB. Si quisiéramos que un atributo contenga más información, tendríamos que utilizar el API de *GridFS* que veremos más adelante.

Relaciones

Vamos a estudiar en detalle cada uno de los tipos de relaciones, para intentar clarificar cuando es conveniente utilizar referencias o datos embebidos.

1:1

Cuando existe una relación 1:1, como pueda ser entre `Persona` y `Curriculum`, o `Persona` y `Direccion` hay que embeber un documento dentro del otro, como parte de un atributo.

Ejemplo relación 1:1 - Persona/Dirección

```
{
  nombre: "Aitor",
  edad: 37,
  direccion: {
    calle: "Mayor",
    ciudad: "Elx"
  }
}
```

La principal ventaja de este planteamiento es que mediante una única consulta podemos obtener tanto los detalles del usuario como su dirección.

Un par de aspectos que nos pueden llevar a no embeberlos son:

- la frecuencia de acceso. Si a uno de ellos se accede raramente, puede que convenga tenerlos separados para liberar memoria.
- el tamaño de los elementos. Si hay uno que es mucho más grande que el otro, o uno lo modificamos muchas más veces que el otro, para que cada vez que hagamos un cambio en un documento no tengamos que modificar el otro será mejor separarlos en documentos separados.

Pero siempre teniendo en cuenta la atomicidad de los datos, ya que si necesitamos modificar los dos documentos al mismo tiempo, tendremos que embeber uno dentro del otro.

1:N

Vamos a distinguir dos tipos:

- **1 a muchos**, como puede ser entre `Editorial` y `Libro`. Para este tipo de relación es mejor usar referencias entre los documentos:

Ejemplo relación 1:N - Editorial

```
{
  _id: 1,
  nombre: "O'Reilly",
  pais: "EE.UU."
}
```

Ejemplo relación 1:N - Libro

```
{
  _id: 1234,
  titulo: "MongoDB: The Definitive Guide",
  autor: [ "Kristina Chodorow", "Mike Dirolf" ],
  numPaginas: 216,
  editorial_id: 1,
}
{
  _id: 1235,
  titulo: "50 Tips and Tricks for MongoDB Developer",
  autor: "Kristina Chodorow",
```

```
  numPaginas: 68,
  editorial_id: 1,
}
```

- **1 a pocos**, como por ejemplo, dentro de un blog, la relación entre `Mensaje` y `Comentario`. En este caso, la mejor solución es crear un array dentro de la entidad 1 (en nuestro caso, `Mensaje`). De este modo, el `Mensaje` contiene un array de `Comentario`:

Ejemplo relación 1:N - Mensaje/Comentario

```
{
  titulo: "La broma asesina",
  url: "http://es.wikipedia.org/wiki/Batman:_The_Killing_Joke",
  text: "La dualidad de Batman y Joker",
  comentarios: [
    {
      autor: "Bruce Wayne",
      fecha: ISODate("2015-04-01T09:31:32Z"),
      comentario: "A mi me encantó"
    },
    {
      autor: "Bruno Díaz",
      fecha: ISODate("2015-04-03T10:07:28Z"),
      comentario: "El mejor"
    }
  ]
}
```

Hay que tener siempre en mente la restricción de los 16 MB de BSON. Si vamos a embeber muchos documentos y estos son grandes, hay que vigilar no llegar a dicho tamaño.

En ocasiones las relaciones 1 a muchos se traducen en documentos embebidos cuando la información que nos interesa es la que contiene en un momento determinado. Por ejemplo, dentro de `Pedido`, el precio de los productos debe embeberse, ya que si en un futuro se modifica el precio de un producto determinado debido a una oferta, el pedido realizado no debe modificar su precio total.

Del mismo modo, al almacenar la dirección de una persona, también es conveniente embeberla. No queremos que la dirección de envío de un pedido se modique si un usuario modifica sus datos personales.

N:M

Más que relaciones muchos a muchos, suelen ser relaciones pocos a pocos, como por ejemplo, `Libro` y `Autor`, o `Profesor` y `Estudiante`.

Supongamos que tenemos libros de la siguiente manera y autores con la siguiente estructura:

Ejemplo relación N:N - Libro

```
{
  _id: 1,
  titulo: "La historia interminable",
```

```
  anyo: 1979
}
```

Ejemplo relación N:M - Autor

```
{
  _id: 1,
  nombre: "Michael Ende",
  pais: "Alemania"
}
```

Podemos resolver estas relaciones de tres maneras:

1. Siguiendo un enfoque relacional, empleando un documento como la entidad que agrupa con referencias manuales a los dos documentos.

   Ejemplo relación N:M - Autor/Libro

   ```
   {
     autor_id: 1,
     libro_id: 1
   }
   ```

   Este enfoque se desaconseja porque necesita tres consultas para obtener toda la información.

2. Mediante 2 documentos, cada uno con un array que contenga los ids del otro documento (*2 Way Embedding*). Hay que tener cuidado porque podemos tener problemas de inconsistencia de datos si no actualizamos correctamente.

   Ejemplo relación N:N - Libro referencia a Autor

   ```
   {
     _id: 1,
     titulo: "La historia interminable",
     anyo: 1979,
     autores: [1]
   },{
     _id: 2,
     titulo: "Momo",
     anyo: 1973,
     autores: [1]
   }
   ```

   Ejemplo relación N:M - Autor referencia a Libro

   ```
   {
     _id: 1,
     nombre: "Michael Ende",
     pais: "Alemania",
     libros: [1,2]
   }
   ```

3. Embeber un documento dentro de otro (*One Way Embedding*). Por ejemplo:

Ejemplo relación N:M - Autor embebido en Libro

```
{
  _id: 1,
  titulo: "La historia interminable",
  anyo: 1979,
  autores: [{nombre:"Michael Ende", pais:"Alemania"}]
},{
  _id: 2,
  titulo: "Momo",
  anyo: 1973,
  autores: [{nombre:"Michael Ende", pais:"Alemania"}]
}
```

En principio este enfoque no se recomienda porque el documento puede crecer mucho y provocar anomalías de modificaciones donde la información no es consistente. Si se opta por esta solución, hay que tener en cuenta que si un documento depende de otro para su creación (por ejemplo, si metemos los profesores dentro de los estudiantes, no vamos a poder dar de alta a profesores sin haber dado de alta previamente a un alumno).

A modo de resumen, en las relaciones N:M, hay que establecer el tamaño de N y M. Si N como máximo vale 3 y M 500000, entonces deberíamos seguir un enfoque de embeber la N dentro de la M (*One Way Embedding*).

En cambio, si N vale 3 y M vale 5, entonces podemos hacer que ambos embeban al otro documento (*Two Way Embedding*).

Más información en http://docs.mongodb.org/manual/applications/data-models-relationships/

Jerárquicas

Si tenemos que modelar alguna entidad que tenga hijos y nos importa las relaciones padre-hijos (categoría-subcategoría), podemos tanto embeber un array con los hijos de un documento (*children*), como embeber un array con los padres de un documento (*ancestors*)

Más información en http://docs.mongodb.org/manual/applications/data-models-tree-structures/

Rendimiento

De modo general, si vamos a realizar más lecturas que escrituras, es más conveniente denormalizar los datos para usar datos embebidos y así con sólo una lectura obtengamos más información. En cambio, si realizamos muchas inserciones y sobretodo actualizaciones, será conveniente usar referencias con dos documentos.

El mayor beneficio de embeber documentos es el rendimiento, sobretodo el de lectura. El acceso a disco es la parte más lenta, pero una vez la aguja se ha colocado en el sector adecuado, la información se obtiene muy rápidamente (alto ancho de banda). El hecho de que toda la información a recuperar esté almacenada de manera secuencial, mediante documentos embebidos, favorece que el rendimiento de lectura sea muy alto, ya que sólo se hace un

acceso a la BBDD. Por lo tanto, si la consistencia es secundaria, duplicar los datos (pero de manera limitada) no es una mala idea, ya que el espacio en disco es más barato que el tiempo de computación.

Es por ello, que un planteamiento inicial a la hora de modelar los datos es basarse en unidades de aplicación, entendiendo como unidad una petición al *backend*, ya sea el *click* de un botón o la carga de los datos para un gráfico. Así pues, cada unidad de aplicación se debería poder conseguir con una única consulta, y por tanto, en gran medida los datos estarán embebidos.

Si lo que necesitamos es consistencia de datos, entonces hay que normalizar y usar referencias. Esto conllevará que al modificar un documento, al estar normalizado los datos serán consistentes, aunque necesitemos dos o más lecturas para obtener la información deseada.

Hay que tener en cuenta que no debemos hacer *joins* en las lecturas. En todo caso, si tenemos redundancia, las realizaremos en las escrituras.

Estos y más consejos en *6 Rules of Thumb for MongoDB Schema Design*: http://blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1

3.2. GridFS

Tal como comentamos anteriormente, los documentos BSON tienen la restricción de que no pueden ocupar más de 16 MB. Si necesitamos almacenar *Blobs*, hemos de utilizar GridFS, el cual es una utilidad que divide un *Blob* en partes para crear una colección y poder almacenar más información.

Así pues, en vez de almacenar un fichero en un único documento, *GridFS* divide el fichero en partes, o trozos (*chunks*), y almacena cada uno de estos trozos en un documento separado. Por defecto, *GridFS* limita el tamaño de cada trozo a 256KB.

Para ello, utiliza dos colecciones para almacenar los archivos:

- La colección `chunks` almacena los trozos de los ficheros
- Mientra que la colección `files` almacena los metadatos de los ficheros.

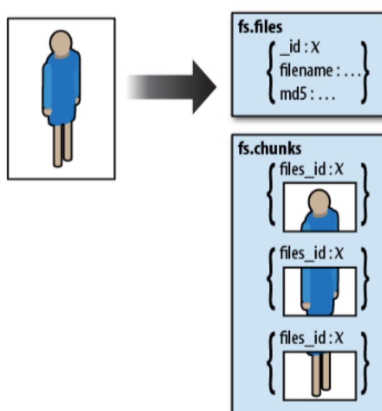


Figura 19. GridFS

Estas colecciones se crean en el espacio de nombres `fs`. Este nombre se puede modificar, por ejemplo, si queremos almacenar diferentes tipos de archivos, es decir, por una lado imágenes y por otro vídeos.

Cuando se consulta un almacén *GridFS* por un fichero, el driver o el cliente unirá los trozos tal como necesite. Se pueden hacer consultas sobre ficheros almacenados con *GridFS*. También se puede acceder a información de secciones arbitrarias de los ficheros, lo que nos permite saltar a la mitad de un archivo de sonido o video.

`mongofiles`

Para interactuar con los archivos almacenados desde la consola utilizaremos el comando `mongofiles`.

Si queremos visualizar todos los ficheros que tenemos en nuestra base de datos usaremos la opción `list`:

Listando los archivos GridFS mediante `mongofiles`

```
$ mongofiles list
```

Al estar vacía la base de datos no se mostrará nada. Aunque en el día a día no utilizaremos `mongofiles` para interactuar con los archivos, si que es muy útil para explorar y probar los archivos almacenados.

Una vez que creamos un archivo, podemos usar la herramienta para explorar los archivos y trozos creados.

Para incluir un archivo, se realiza mediante la opción de `put`:

Por ejemplo, para añadir el archivo `video.mp4` haríamos:

Insertando un archivo en GridFS mediante `mongofiles`

```
$ mongofiles put video.mp4
connected to: 127.0.0.1
added file: { _id: ObjectId('550957b83f627a4bb7f28bc8'), filename:
 "video.mp4", chunkSize: 261120, uploadDate: new Date(1426675642227), md5:
 "b7d51c0c83ef61ccf69f223eded44797", length: 39380552 }
done!
```

Podemos observar que tras la inserción, obtenemos un documento que contiene:

- `chunkSize`: el tamaño de cada trozo
- `length`: tamaño del fichero
- `uploadDate`: fecha de creación del fichero en *MongoDB*

Si ahora comprobamos los archivos disponibles tendremos:

Listando los archivos GridFS mediante `mongofiles`

```
$ mongofiles list
connected to: 127.0.0.1
video.mp4 39380552
```

Estas operaciones de consulta también la podemos realizar directamente realizando una consulta a la colección `fs.files` mediante el comando `db.fs.files.find()` desde `mongo`:

Resultado de la colección `fs.files`

```
> db.fs.files.find()
{ "_id" : ObjectId("550957b83f627a4bb7f28bc8"), "filename"
 : "video.mp4", "chunkSize" : 261120, "uploadDate"
 : ISODate("2015-03-18T10:47:22.227Z"), "md5"
 : "b7d51c0c83ef61ccf69f223eded44797", "length" : 39380552 }
```

Si queremos obtener información sobre los trozos de un archivo (*chunks*), consultaremos la colección `fs.chunks` añadiendo un filtro para que no nos muestre la información en binario:

Resultado de la colección `fs.chunks`

```
> db.fs.chunks.find({}, {"data":0})
{ "_id" : ObjectId("550957b856eb8d804bc96fb8"), "files_id" :
 ObjectId("550957b83f627a4bb7f28bc8"), "n" : 0 }
{ "_id" : ObjectId("550957b956eb8d804bc96fb9"), "files_id" :
 ObjectId("550957b83f627a4bb7f28bc8"), "n" : 1 }
...
{ "_id" : ObjectId("550957ba56eb8d804bc9704e"), "files_id" :
 ObjectId("550957b83f627a4bb7f28bc8"), "n" : 150 }
```

Toda interacción con *GridFS* se debe realizar a través de un driver para evitar incongruencias en los datos.

Otras operaciones que podemos realizar con `mongofiles` son:

- `search` : busca una cadena en el archivo
- `delete` : elimina un archivo de *GridFS*
- `get` : obtiene un archivo

Todas las opciones de `mongofiles` se pueden consultar en http://docs.mongodb.org/manual/reference/program/mongofiles/

GridFS desde *Java*

El driver *Java* de *MongoDB* ofrece la clase `GridFS` para interactuar con los archivos. A partir de dicha clase vamos a poder almacenar archivos en *MongoDB*.

A continuación, se muestra mediante código *Java* como leer un archivo de vídeo y almacenarlo en la base de datos:

Ejemplo de inserción en *GridFS*

```
MongoClient cliente = new MongoClient();
DB db = cliente.getDB("expertojava");
FileInputStream inputStream = null;

GridFS videos = new GridFS(db);

try {
  inputStream = new FileInputStream("video.mp4"); ❶
} catch (FileNotFoundException e) {
```

```
    System.out.println("No puedo abrir el fichero");
    System.exit(1);
  }

  GridFSInputFile video = videos.createFile(inputStream, "video.mp4");
    ❷

  // Creamos algunos metadatos para el vídeo
  BasicDBObject meta = new BasicDBObject("descripcion", "Prevención de
   riesgos laborales");
  List<String> tags = new ArrayList<String>();
  tags.add("Prevención");
  tags.add("Ergonomía");
  meta.append("tags", tags);

  video.setMetaData(meta);  ❸
  video.save();  ❹

  System.out.println("Object ID: " + video.get("_id"));
```

❶ Leemos el archivo desde el sistema de archivos
❷ Creamos un objeto `GridFS` que referencia al archivo
❸ Le asociamos al archivo metadatos mediante una lista de cadenas
❹ Almacena el archivo en la colección

Al ejecutar este fragmento de código, la colección `fs.files` contendrá un documento similar al siguiente donde podemos observar que se ha añadido una propiedad ` metadata`:

```
{ "_id" : ObjectId("553a78edd4c66e72c890472b"), "chunkSize"
 : NumberLong(261120), "length" : NumberLong(39380552), "md5"
 : "b7d51c0c83ef61ccf69f223eded44797", "filename"
 : "video.mp4", "contentType" : null, "uploadDate" :
 ISODate("2015-04-24T17:10:05.356Z"), "aliases" : null, "metadata"
 : { "descripcion" : "Prevención de riesgos laborales", "tags" :
 [ "Prevención", "Ergonomía" ] } }
```

Si lo que queremos es obtener un archivo que tenemos almacenado en la base de datos para guardarlo en un fichero haríamos:

Ejemplo de lectura de *GridFS*

```
MongoClient cliente = new MongoClient();
DB db = cliente.getDB("expertojava");

GridFS videos = new GridFS(db);

// Buscamos un fichero
GridFSDBFile gridFile = videos.findOne(new
 BasicDBObject("filename", "video.mp4")); ❶

FileOutputStream outputStream = new FileOutputStream("video_copia.mp4");
gridFile.writeTo(outputStream); ❷

// Buscamos varios ficheros
```

```
List <GridFSDBFile> ficheros = videos.find(new
 BasicDBObject("descripción", "Prueba")); ❸

for (GridFSDBFile fichero: ficheros) {
  System.out.println(fichero.getFilename());
}
```

❶ Buscamos en *GridFS* un archivo por su nombre el cual se almacena en un objeto `GridFSDBFile`
❷ Escribimos el contenido del `gridFile` en un nuevo archivo
❸ Al obtener varios elementos, el resultado de la búsqueda se almacena en `List<GridFSDBFile>`

Casos de Uso

El motivo principal de usar *GridFS* se debe a superar la restricción de los 16MB de los documentos BSON.

Además, hay casos donde es preferible almacenar los archivos de vídeo y audio en una base de datos en vez de en el sistema de archivos, ya sea para almacenar metadatos de los archivos, acceder a ellos desde aplicaciones ajenos al sistemas de archivos o replicar el contenido para ofrecer una alta disponibilidad.

Otro caso importante es cuando tenemos contenido generado por el usuario como grandes informes o datos estáticos que no suelen cambiar y que cuestan mucho de generar. En vez de generarlos con cada petición, se pueden ejecutar una vez y almacenarlos como un documento. Cuando se detecta un cambio en el contenido estático, se vuelve a generar el informe en la próxima petición de los datos.

Si el sistema de archivos no siempre está disponible, también podemos evaluar *GridFS* como una alternativa viable. También podemos aprovechar que los fichero se almacenan en trozos y usar estos trozos para almacenar parte del archivo que interesa, como puede ser el contenido MD5 de los datos.

Si nos centramos en sus inconvenientes, tenemos que tener claro que hay pérdida de rendimiento respecto a acceder al sistema de archivos. Por ello, se recomienda crear una prueba de concepto en el sistema a desarrollar antes de implementar la solución.

Hay que tener en cuenta que *GridFS* almacena los datos en múltiples documentos, con lo que una actualización atómica no es posible. Si tenemos claro que el contenido es inferior a 16 MB, que es el caso de la mayoría de contenido generado por el usuario, podemos dejar de lado *GridFS*, y usar directamente documentos BSON los cuales aceptan datos binarios.

Guardado datos binarios en un documento BSON - DatosBinario.java

```
String recurso = "cartel300.png";
byte[] imagenBytes = leerDatosBinarios(recurso);

DBObject doc = new BasicDBObject("_id", 1);
doc.put("nombreFichero", recurso);
doc.put("tamanyo", imagenBytes.length);
doc.put("datos", imagenBytes);
coleccion.insert(doc);

byte[] leerDatosBinarios(String recurso) throws IOException {
```

```
  InputStream in =
 Thread.currentThread().getContextClassLoader().getResourceAsStream(recurso);
  if (in != null) {
    int available = in.available();
    byte[] bytes = new byte[available];
    in.read(bytes);
    return bytes;
  } else {
    throw new IllegalArgumentException("Recurso " + recurso + " no
 encontrado");
  }
}
```

3.3. Índices

Los índices son una parte importante de la gestión de bases de datos. Un índice en una base de datos es similar a un índice de un libro; permite saltar directamente a la parte del libro en vez de tener que pasar las páginas buscando el tema o la palabra que nos interesa.

En el caso de *MongoDB*, un índice es una estructura de datos que almacena información sobre los valor de determinados campos de los documentos de una colección. Esta estructura permite recorrer los datos y ordenarlos de manera muy rápida. Así pues, los índices se utilizan tanto al buscar un documento como al ordenar los datos de una consulta.

Preparando los ejemplos

Para los siguientes ejemplos, vamos a utilizar una colección de 200 estudiantes con las calificaciones que han obtenido en diferentes trabajos, exámenes o cuestionarios.

Para ello, importaremos la colección students.json[15] mediante:

```
mongoimport -d expertojava -c students --file students.json
```

Un ejemplo de una estudiante sería:

```
> db.students.findOne()
{
  "_id" : 0,
  "name" : "aimee Zank",
  "scores" : [
    {
      "type" : "exam",
      "score" : 1.463179736705023
    }, {
      "type" : "quiz",
      "score" : 11.78273309957772
    }, {
      "type" : "homework",
      "score" : 6.676176060654615
    }, {
```

[15] resources/nosql/students.json

```
        "type" : "homework",
        "score" : 35.8740349954354
      }
    ]
  }
```

Para comprobar el impacto del uso de índices, vamos a empezar con un ejemplo para ver cómo de rápido puede hacerse una consulta que tiene un índice respecto a uno que no lo tiene.

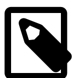
Para analizar el plan de ejecución de una consulta, podemos emplear el método `explain()` (http://docs.mongodb.org/v2.6/reference/method/cursor.explain/) sobre un cursor:

```
> db.students.find({"name" : "Kaila Deibler"}).explain()
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 2,
  "nscannedObjects" : 200,
  "nscanned" : 200,
  "nscannedObjectsAllPlans" : 200,
  "nscannedAllPlans" : 200,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 755,
  "indexBounds" : {

  },
  "server" : "MacBook-Air-de-Aitor.local:27017"
}
```

El plan de ejecución devuelve mucha información (http://docs.mongodb.org/v2.6/reference/method/cursor.explain/#explain-output-fields-core), pero nos vamos a centrar en unos pocos atributos para analizar el resultado.

Se puede observar mediante la propiedad `cursor` que ha utilizado un `BasicCursor`, el cual es un tipo de cursor que indica que no se ha usado ningún índice en la consulta, por lo que ha realizado un escaneado completo (*full scan*), de manera que se han recorrido todos los documentos de la colección. Esta consulta sólo devuelve un par de documentos (`n`) pero ha tenido que escanear los 200 existentes (`nscannedObjects`).



Cuando vamos a buscar un elemento, es mucho mas rápido hacer un `findOne` que `find`, porque mientras `find` recorre toda la colección, con `findOne` en cuanto encuentre un documento, el cursor se detendrá.

Por defecto, el campo `_id` esta indexado. Así pues, vamos a buscar el mismo documento de antes, pero ahora mediante el campo indexado:

```
> db.students.find({_id:30}).explain()
{
  "cursor" : "BtreeCursor _id_",
  "isMultiKey" : false,
```

```
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 3,
  "indexBounds" : {
   "start" : {
    "_id" : 30
   },
   "end" : {
    "_id" : 30
   }
  },
  "server" : "MacBook-Air-de-Aitor.local:27017"
 }
```

Ahora *MongoDB* sólo ha escaneado el documento que ha devuelto, y ha utilizado un cursor `Btree`. Al haber utilizado un índice, hemos evitado tener que mirar en más documentos. Esta consulta se ha realizado más rápidamente (y a mayor número de documentos más se nota la diferencia). Por supuesto, no siempre vamos a buscar por su `_id`, así que vamos a ver como crear nuevos índices.

Toda la información sobre el uso de índices con *MongoDB* se encuentra en http://docs.mongodb.org/manual/core/indexes/

Simples

Para crear un índice hemos de utilizar el método `ensureIndex({atributo:orden})`

Si queremos crear un índice sobre la propiedad `name` en orden ascendente haríamos lo siguiente:

```
> db.students.ensureIndex( {name:1} )
"createdCollectionAutomatically" : false,
"numIndexesBefore" : 1,
"numIndexesAfter" : 2,
"ok" : 1
```

Si ahora volvemos a ejecutar la consulta por nombre, comprobaremos como ahora ya utiliza un cursor `Btree` y que no ha tenido que recorrer todos los documentos.

```
> db.students.find({"name" : "Kaila Deibler"}).explain()
{
  "cursor" : "BtreeCursor name_1",
  "isMultiKey" : false,
  "n" : 2,
  "nscannedObjects" : 2,
  "nscanned" : 2,
  "nscannedObjectsAllPlans" : 2,
```

```
  "nscannedAllPlans" : 2,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 757,
  "indexBounds" : {
    "name" : [
      [
      "Kaila Deibler",
      "Kaila Deibler"
      ]
    ]
  },
  "server" : "MacBook-Air-de-Aitor.local:27017",
  "filterSet" : false
}
```

Un aspecto a considerar de los índices es que aceleran mucho las búsquedas, pero ralentizan las inserciones/modificaciones y hace que la información ocupe más espacio en disco. Por ello, deberemos considerar añadir índices a las colecciones donde el número de lecturas sea mayor que el de escrituras. Si sucede al revés, el uso de índices puede provocar un deterioro en el rendimiento.

El orden de los índices (`1` para ascendente, `-1` para descendente) no importa para un índice sencillo, pero si que tendrá un impacto en los índices compuestos cuando se utilizan para ordenar o con una condición de rango.

Por supuesto, podemos crear índices sobre propiedades que forman parte de un array. Así pues, podemos crear un índice sobre el tipo de calificación que tiene un estudiante mediante:

```
db.students.ensureIndex( {scores.type:1} )
```

Pero además también podemos crear un índice sobre todas las calificaciones, lo cual indexa cada elemento del array, con lo que podemos buscar por cualquier objeto del array. Este tipo de índices se conocen como **multiclave**:

```
> db.students.ensureIndex( {scores:1} )
```

Toda la información relativa a los índices creados se almacenan en la colección `system.indexes`, la cual podremos consultar.

Además, también podemos obtener los índices de una determinada colección mediante el método `getIndexes()`.

Finalmente, para borrar un índice emplearemos el método `dropIndex(atributo)`.

Así pues, tenemos que algunas de las operaciones relacionadas con los índices más importantes son:

```
db.system.indexes.find() // muestra los índices existentes
```

```
db.students.getIndexes() // muestra los índices de la colección students
db.students.dropIndex( {"name":1} ) // borra el índice que existe sobre la
 propiedad name
```

Propiedades

Al crear un índice podemos pasarle algunas opciones como segundo parámetro, como puede ser:

- `unique:true` : Permite crear indices que sólo permiten valores únicos en una propiedad. No puede haber valores repetidos y una vez creado no permitirá insertar valores duplicados.

```
> db.students.ensureIndex( {students_id:1}, {unique:1} )
```

El índice sobre `_id` es único aunque al visualizar el indice no nos diga que lo sea, ya que no permite que se inserten dos `_id` iguales.

- `dropDups:true` : Al crear un índice único, podemos obligar a limpiar los datos para deshacernos de los duplicados. El inconveniente principal es que no se sabe cuales han sido eliminados, ya que no sigue ningún criterio establecido, por lo que sólo se recomienda su uso para casos excepcionales.

```
> db.students.ensureIndex( {students_id:1}, {unique:1, dropDups:1} )
```

Autoevaluación

Si elegimos utilizar la opción `dropDups` al crear un índice único ¿Qué hará *MongoDB* con los documentos que entran en conflicto con un entrada existente? [16]

1. Moverlos a una colección de archivo.
2. Eliminar el atributo que provoca el conflicto para que pueda ser indexada y permanecer en la colección
3. Eliminarlos pero registrar los documentos borrados en un fichero del directorio de log.
4. Borrarlos por siempre jamás.

- Si queremos añadir un índice sobre una propiedad que no aparece en todos los documentos, necesitamos crear un *Spare Index* mediante `spare:true` , el cual se crea para el conjunto de clave que tienen valores.

```
> db.students.ensureIndex( {size:1}, {spare:1} )
```

Si hacemos una consulta sobre una propiedad que tiene asociado un *Spare Index*, nos van a aparecer menos resultados, ya que sólo mostrará aquellos que tengan valores, y no tendrá en cuenta los documentos que tengan dicho campo sin crear.

[16] la 4ª opción, es decir, borrará los documentos que provoquen conflictos

Autoevaluación

Suponemos que tenemos los siguientes documentos en una colección llamada 'people' con los siguientes documentos:

```
> db.people.find()
{ "_id" : ObjectId("50a464fb0a9dfcc4f19d6271"), "nombre"
 : "Juan", "cargo" : "Técnico" }
{ "_id" : ObjectId("50a4650c0a9dfcc4f19d6272"), "nombre"
 : "Pedro", "cargo" : "CEO" }
{ "_id" : ObjectId("50a465280a9dfcc4f19d6273"), "nombre"
 : "Sandra" }
```

Y hay un índice definido del siguiente modo:

```
> db.people.ensureIndex( {cargo:1}, {spare:1} )
```

Si realizamos la siguiente consulta, ¿Qué documentos aparecerán y por qué? [17]

```
> db.people.find({cargo:null})
```

1. Ningún documento, ya que la consulta utiliza el índice y no puede haber documentos cuyo *cargo* sea nulo
2. Ningún documento, ya que la consulta de `cargo:null` sólo encuentra documentos que de manera explícita tienen el *cargo* a nulo, independientemente del índice.
3. El documento de *Sandra*, ya que la consulta no utilizará el índice
4. Todos los documentos de la colección, ya que todos los documentos cumplen `cargo:null`
5. El documento de *Sandra*, ya que el comando `ensureIndex` no se ejecutará sobre este documento.

Compuestos

Si queremos aplicar un índice sobre más de una propiedad, podemos crear índices compuestos, indicando las propiedades separadas por coma:

```
> db.students.ensureIndex({name:1,scores.type:1})
```

Es importante destacar que el orden de los índices importa, y mucho. Si hacemos una consulta que sólo utilice el atributo `scores.type`, este índice no se va a utilizar.

No confundir los indices compuestos con hacer 2 o más índices sobre diferentes propiedades.

[17] la primera opción, ya que no puede haber documentos con propiedades nulas en un índice *spare*

Si creamos un índice sobre los campos `A`, `B`, `C`, el índice se va a utilizar para las búsquedas sobre `A`, sobre la dupla `A`, `B` y sobre el trio `A`, `B`, `C`. Es decir, los índices se usan con los subconjuntos por la izquierda (prefijos) de los índices compuestos.

Si tenemos varios índices candidatos a la hora de ejecutar, el optimizador de consultas de *MongoDB* los usará en paralelo y se quedará con el resultado del primero que termine. Más información en http://docs.mongodb.org/manual/core/query-plans/

Multiclave

Cuando se indexa una propiedad que es un array se crea un índice multiclave para todos los valores del array de todos los documentos. El uso de estos índices son lo que hacen que las consultas sobre documentos embebidos funcionen tan rápido.

```
> db.students.ensureIndex({"teachers":1})
> db.students.find({"teachers":{"$all":[1,3]}})
```

Se pueden crear índices tanto en propiedades básicas, como en propiedades internas de un array, mediante la notación de `.`:

```
> db.students.ensureIndex({"addresses.phones":1})
```

Sólo se pueden crear índices compuestos multiclave cuando sólo una de las propiedades del índice compuesto es un array; es decir, no puede haber dos propiedades array en un índice compuesto. Hay que tener cuidado ya que no se va a quejar al crearlo, solo al insertar, porque no va a poder indexar arrays paralelos.

Rendimiento

Por defecto, los índices se crean en *foreground*, de modo que al crear un índice se van a bloquear a todos los *writers*. Si queremos crearlos en *background* para no penalizar las escrituras (es más lento, de 2 a 5 veces) lo indicaremos con un segundo parámetro:

```
> db.students.ensureIndex( { twitter: 1}, {background: true} )
```

Las creación de índices en *background* sólo se puede realizar de uno en uno, y aunque el servidor admita más peticiones, el shell de `mongo` donde se crea el índice queda bloqueado.

Algunos de los operadores que no utilizan los índices eficientemente son los operadores `$where`, `$nin` y `$exists`. Cuando estos operadores se emplean en una consultar hay que tener en mente un posible cuelo de botella cuando el tamaño de los datos incremente.

Plan de Ejecución

Al explicar los índices ya hemos visto que podemos obtener información sobre la operación realizada mediante el método `.explain()`.

El atributo `indexOnly` me dice si toda la información que quiero recuperar se encuentra en el índice. Este atributo va a depender de los campos que quiera que me devuelva la consulta, si son un subconjunto del índice utilizado.

Más información en http://docs.mongodb.org/manual/reference/method/cursor.explain/

Los índices tienen que caber en memoria. Si están en disco, pese a ser algorítmicamente mejores que no tener, al ser más grandes que la RAM disponible, no se obtienen beneficios por la penalización de la paginación.

Para averiguar el tamaño de los índices (en bytes):

```
> db.students.stats()  // obtiene estadísticas de la colección
> db.students.totalIndexSize()  // obtiene el tamaño del índice
```

Mucho cuidado con los índices *multikeys* porque crecen mucho y si el documento tiene que moverse en disco, el cambio supone tener que cambiar todos los puntos de índice del array.

Aunque sea más responsabilidad de un DBA, los desarrolladores debemos saber si el índice va a caber en memoria. Si no van a caber es mejor no usarlos.

Si vemos que no usamos un índice o que su rendimiento es peor, podemos borrarlos con `dropIndex`.

```
> db.students.dropIndex('nombreDeIndice')
```

Autoevaluación

Hemos actualizado un documento con una clave llamada *etiquetas* que provoca que el documento tenga que moverse a disco. Supongamos que el documento contiene 100 etiquetas en él y que el array de etiquetas está indexada con un índice multiclave.

¿Cuantos puntos de índice tienen que actualizarse en el índice para acodomar el movimiento? [18]

Hints

Si en algún momento queremos forzar el uso de un determinado índice al realizar una consulta, necesitaremos usar el método `.hint({campo:1})`

Si queremos que se utilice el índice asociado a la propiedad `twitter`:

```
> db.people.find({nombre:"Aitor
 Medrano",twitter:"aitormedrano"}).hint({twitter:1}})
```

Si por algún motivo no queremos usar índices, le pasaremos el operador `$natural` al método `hint()`.

```
> db.people.find({nombre:"Aitor
 Medrano",twitter:"aitormedrano"}).hint({$natural:1}})
```

[18] 100, es decir, todos los valores existentes en el índice multiclave

Si usamos un *hint* sobre un índice *spare* y no hay documentos a devolver con dicho índice porque todos sus campos son nulos, la consulta no devolverá nada, aunque haya documentos que sin dicho índice si cumplen los criterios.

Hay que destacar que los operadores `$gt`, `$lt`, `$ne` … provocan un uso ineficiente de los índices, ya que la consulta tiene que recorrer toda la colección de índices. Si hacemos una consulta sobre varios atributos y en uno de ellos usamos `$gt`, `$lt` o similar, es mejor hacer un *hint* sobre el resto de atributos que sí tienen una selección directa.

Por ejemplo, supongamos que en la coleccion de calificaciones quieseramos obtener los exámenes con un calificación comprendida entre 95 y 98.

```
> db.grades.find({ score:{$gt:95, $lte:98}, type:"exam" })
```

Para esta consulta, suponiendo que tenemos un índice tanto en `score` como en `type`, sería conveniente hacer el *hint* sobre el `type`

```
> db.grades.find({ score:{$gt:95, $lte:98}, type:"exam" }).hint('type')
```

Otros tipos de índices

- **Geoespaciales**, para trabajar con coordenadas → http://docs.mongodb.org/manual/applications/geospatial-indexes/
- **De texto**, para realizar búsquedas dentro de un campo → http://docs.mongodb.org/manual/core/index-text/
- **Hash**, centrado en el uso de *sharding* → http://docs.mongodb.org/manual/core/index-hashed/

3.4. Colecciones Limitadas

Una colección limitada (*capped collection*) es una colección de tamaño fijo, donde se garantiza el orden natural de los datos, es decir, el orden en que se insertaron.

Una vez se llena la colección, se eliminan los datos más antiguos, y los datos más nuevos se añaden al final, de manera similar a un *buffer* circular, asegurando que el orden natural de la colección sigue el orden en el que se insertaron los registros.

Este tipo de colecciones se utilizan para logs y auto-guardado de información, ya que su rendimiento es muy alto para inserciones.

Se crean de manera explícita mediante el método `createCollection`, pasándole el tamaño en *bytes* de la colección. Por ejemplo, si queremos cerar una colección para auditar datos de 20 KB haríamos:

```
> db.createCollection("auditoria", {capped:true, size:20480})
```

Los documentos que se añaden a una colección limitada se pueden modifican, pero no pueden crecer en tamaño. Si sucede, la modificación fallará. Además, tampoco se pueden eliminar documentos de la colección. Para ello, hay que borrar toda la colección (*drop*) y volver a crearla.

También podemos limitar el número de elementos que se pueden añadir a la colección mediante el parámetro `max:` en la creación de la colección. Sin embargo, hay que asegurarse de disponer de suficiente espacio en la colección para los elementos que queremos añadir. Si la colección se llena antes de que el número de elementos se alcance, se eliminará el elemento más antiguo de la colección.

Si retomamos el ejemplo anterior, pero fijamos su máximo a 100 elementos, crearíamos la colección del siguiente modo:

Creando una colección limitada a 100 documentos

```
> db.createCollection("auditoria", {capped:true, size:20480, max:100})
```

El *shell* de *MongoDB* ofrece la utilidad `validate()` para visualizar la cantidad de espacio utilizado por cada colección, ya sea limitada o no. Para comprobar el estado de la colección anterior haríamos:

Validando el estado de una colección

```
> db.auditoria.validate()
```

Si queremos consultar los datos de una colección limitada, por su idiosincracia, los resultados aparecerán en el orden de inserción. Si queremos obtenerlos en orden inverso, le tenemos que pasar el operador `$natural` al método `sort()`:

Consultando una colección limitada a la inversa

```
> db.auditoria.find().sort({ $natural:-1 })
```

Finalmente, si queremos averiguar si una colección es limitada, lo haremos mediante el método `isCapped()`:

Comprobando si una colección es limitada

```
> db.auditoria.isCapped()
true
```

Más información en http://docs.mongodb.org/v2.6/core/capped-collections/

3.5. Profiling

MongoDB trae integradas varias herramientas para el control del rendimiento.

Por ejemplo, la colección `db.system.profile` auditará las consultas ejecutadas. Podemos indicar el nivel de las consultas a auditar mediante tres niveles: `0` (ninguna), `1` (consultas lentas), `2` (todas las consultas)

Si queremos que se auditen todas las consultas, lo indicaremos del siguiente modo:

Auditando todas las consultas

```
> db.setProfilingLevel(2)
```

El método `setProfilingLevel()` también admite un segundo parámetro para indicar el numero mínimo de milisegundos de las consultas para ser auditadas.

Por defecto, *MongoDB* automáticamente escribe en el log las consultas que tardan más de 100ms.

Si queremos indicar estas propiedades al arrancar el demonio, le pasaremos los parámetros `--profile` y/o `--slowms`:

```
mongod --profile=1 --slowms=15
```

Si en algún momento queremos consultar tanto el nivel como el estado del *profiling*, podemos utilizar los métodos `db.getProfilingLevel()` y `db.getProfilingStatus()`.

Sobre los datos auditados, podemos hacer `find` sobre `db.system.profile` y filtrar por los campos mostrados:

```
> db.system.profile.find({ millis : { $gt : 1000 } }).sort({ts : -1})
> db.system.profile.find().limit(10).sort( { ts : -1 } ).pretty()
```

Dentro de estas consultas algunos campos significativos son:

- `op`: tipo de operación, ya sea `command`, `query`, `insert`, …
- `millis`: tiempo empleado en la operación
- `ts`: *timestamp* de la operación

Podéis consultar todos los campos disponibles en http://docs.mongodb.org/manual/reference/database-profiler/

Otras herramientas para controlar el rendimiento que se ejecutan en un terminal, son:

- **mongotop** → similar a la herramienta `top` de UNIX, muestra el tiempo empleado por *MongoDB* en las diferentes colecciones, indicando tanto el tiempo empleado en lectura como en escrituras. Para ello, si queremos se ejecute cada tres segundos, en un terminal:

```
mongotop 3
```

```
MacBook-Air-de-Aitor:1314 aitormedrano$ mongotop 3
connected to: 127.0.0.1

                          ns       total        read       write
2014-04-06T09:51:16
           local.system.users         0ms         0ms         0ms
         local.system.replset         0ms         0ms         0ms
         local.system.indexes         0ms         0ms         0ms
            local.startup_log         0ms         0ms         0ms
           jtech.system.users         0ms         0ms         0ms
         jtech.system.indexes         0ms         0ms         0ms
                 jtech.grades         0ms         0ms         0ms

                          ns       total        read       write
2014-04-06T09:51:19
                 jtech.grades         1ms         1ms         0ms
           local.system.users         0ms         0ms         0ms
         local.system.replset         0ms         0ms         0ms
```

Figura 20. Ejemplo de `mongotop`

Más información en : http://docs.mongodb.org/manual/reference/program/mongotop/

- **mongostat** → muestra el número de operaciones por cada tipo que se realizan por segundo a nivel de servidor, lo que nos da una instantánea de los que está haciendo el servidor.

```
MacBook-Air-de-Aitor:1314 aitormedrano$ mongostat 5
connected to: 127.0.0.1
insert  query update delete getmore command flushes mapped  vsize    res faults  locked db
idx miss %      qr|qw   ar|aw  netIn netOut  conn       time
    *0     *0     *0     *0      0     0|0       0   160m  2.73g    35m      0  test:0.0%
         0      0|0     0|0    12b   598b     2   11:53:34
    *0     *0     *0     *0      0     1|0       0   160m  2.73g    35m      0  test:0.0%
         0      0|0     0|0   136b   739b     2   11:53:39
    *0     *0     *0     *0      0     0|0       0   160m  2.73g    35m      0 jtech:0.0%
         0      0|0     0|0    39b     2k     2   11:53:44
    *0     *0     *0     *0      0     0|0       0   160m  2.73g    35m      0      .:0.0%
         0      0|0     0|0    65b   653b     2   11:53:49
```

Figura 21. Ejemplo de `mongostat`

Una buena columna a vigilar es `idx miss %`, la cual muestra los índices perdidos, es decir, aquellas consultas que han causado paginación y en vez de obtener los datos de memoria han tenido que acceder a disco.

Más información en : http://docs.mongodb.org/manual/reference/program/mongostat/

3.6. Ejercicios

(0.75 puntos) Ejercicio 31. Diseñando el Esquema

A partir de la base de datos empleada en el proyecto de integración, se pide rediseñarla pero siguiendo un enfoque de base de datos documental, empleando referencias manuales y/o documentos embebidos dependiendo del caso.

En base a las unidades de aplicación implementadas en el proyecto, se pide crear una base de datos denominada `proyecto` donde se incluyan las colecciones necesarias.

Además, cada colección debe incluir dos o más documentos.

Una vez diseñada, creada y tras insertar los datos, se ha de exportar la base de datos mediante `mongoexport` en un archivo denominado `ej31.json`.

(0.75 puntos) Ejercicio 32. GridFS

Crear una clase llamada `GridFSEjercicios`, la cual contenga los siguientes métodos:

- Un método para almacenar ficheros en *GridFS*, a partir de una ruta, una cadena con la descripción y una lista con nombres de etiquetas:

```
String insertaFichero(String nombreFichero, String descripcion,
 List<String> etiquetas)
```

- Un método para recuperar de *GridFS* un archivo por su nombre y cree una copia del mismo en otro archivo

```
void recuperaFichero(String nombreFichero, String nombreDestino)
```

- Un método que permita recuperar un listado de nombres de ficheros con aquellos que el atributo `tags` de los metadatos contenga una determinada etiqueta

```
List<String> recuperaNombreFicherosPorTag(String etiqueta)
```

(0.75 puntos) Ejercicio 33. Índices

En este ejercicio vamos optimizar la base de datos de `ejercicios` importada en la primera sesión.

Las consultas que más se realizan sobre la colección `cities` son:

- Recuperar una ciudad por su nombre
- Recuperar las 5 ciudades más pobladas de un determinado país
- Recuperar el nombre del país y la ciudad de las 3 ciudades más pobladas de una determinada zona horaria.

Se pide crear los índices adecuados para que estas consultas se ejecuten de manera óptima.

Por lo tanto, el archivo `ej33.txt` debe contener:

- los comandos empleados para comprobar los planes de ejecución antes y después de crear los índices necesarios
- comandos necesarios para crear los índices elegidos
- una explicación de la elección del tipo de índice elegido en cada caso.
- resultado de obtener todos los índices de la colección `cities` una vez creados todos los índices elegidos
- tamaño de los índices y requisitos de hardware del servidor necesarios para dar soporte a estos índices

(0.25 puntos) Ejercicio 34. Análisis del Log

En este ejercicio vamos a analizar el log que ha generado una instancia de *MongoDB*.

Para ello, debemos importar el archivo sysprofile.json[19] con el siguiente comando:

```
mongoimport -d ejercicios -c profile < sysprofile.json
```

Ahora, consulta los datos del *profiler*, buscando todas las consultas de la colección `students` dentro de la base de datos `school2`, ordenadas por latencia descendentemente.

Se pide averiguar cual es la duración (en ms) de la operación que ha tardado más en ejecutarse.

Escribe la consulta necesaria y la respuesta en un archivo de texto denominado `ej34.txt`.

[19] resources/nosql/sysprofile.json

4. Agregaciones y Escalabilidad

4.1. Agregaciones

Para poder agrupar datos y realizar cálculos sobre éstos, *MongoDB* ofrece diferentes alternativas:

1.- Mediante operaciones *Map-reduce* con operación `mapreduce()` (http://docs.mongodb.org/manual/core/map-reduce/)

2.- Mediante operaciones de agrupación sencilla, como pueden ser las operaciones `count()`, `distinct()` o `group()`. Esta última operación permite realizar una serie de cálculos sobre elementos filtrado, de manera similar a *Map-reduce*, pero más sencillo y limitado, obteniendo un array de elementos agrupados.

La firma completa es `group({ key, reduce, initial })` donde sus parámetros definen:

- `key`: atributo por el que se van a agrupar los datos
- `initial`: define un valor base para cada grupo de resultados. Normalmente se inicializa
- `reduce`: función que agrupa los elementos similar. Recibe dos parámetros, el documento actual (`item`) sobre el cual se itera, y el objeto contador agregado (`prev`).

Por ejemplo, si quisieramos saber cuantas personas tienen diferente cantidad de hijos y cuantos hay de cada tipo haríamos:

```
> db.people.group( {
  key: { hijos: true },
  reduce: function ( item, prev ) {
    prev.total += 1;
  },
  initial: { total : 0 }
} )
```

Con lo que obtendríamos (dependiendo de los datos) que hay dos personas que tienen 2 hijos, y una persona que no tiene el atributo hijo definido:

```
[ { "hijos" : 2, "total" : 2 },
  { "hijos" : null, "total" : 1 } ]
```

Hay que destacar que la función `group()` no funciona en entornos *sharded*, donde habría que utilizar la función `mapreduce()` o el framework de agregación.

Podemos ver un ejemplo sencillo en http://docs.mongodb.org/manual/core/single-purpose-aggregation/#group y más información en http://docs.mongodb.org/manual/reference/method/db.collection.group

2.- Mediante el uso del *Aggregation Framework*, basado en el uso de *pipelines*, el cual permite realizar diversas operaciones sobre los datos. Este framework forma parte de *MongoDB* desde la versión 2.2, ofrece más posibilidades que la operación *group* y además permite su uso con *sharding*.

Para ello, a partir de una colección, mediante el método `aggregate` le pasaremos un array con las fases a realizar:

```
db.productos.aggregate([
  {$group:
    {_id:"$fabricante", numProductos:{$sum:1}}
  },
  {$sort: {numProductos:-1}}
])
```

4.2. *Pipeline* de Agregación

Las agregaciones usan un *pipeline*, conocido como *Aggregation Pipeline*, de ahí el uso de un array con `[ ]` donde cada elemento es una fase del *pipeline*, de modo que la salida de una fase es la entrada de la siguiente:

```
db.coleccion.aggregate([op1, op2, ... opN])
```

El resultado del pipeline es un documento y por lo tanto está sujeto a la restricción de BSON, que limita su tamaño a 16MB

En la siguiente imagen se resumen los pasos de una agrupación donde primero se eligen los elementos que vamos a agrupar mediante `$match` y posteriormente se agrupan con `$group` para hacer `$sum` sobre el total:

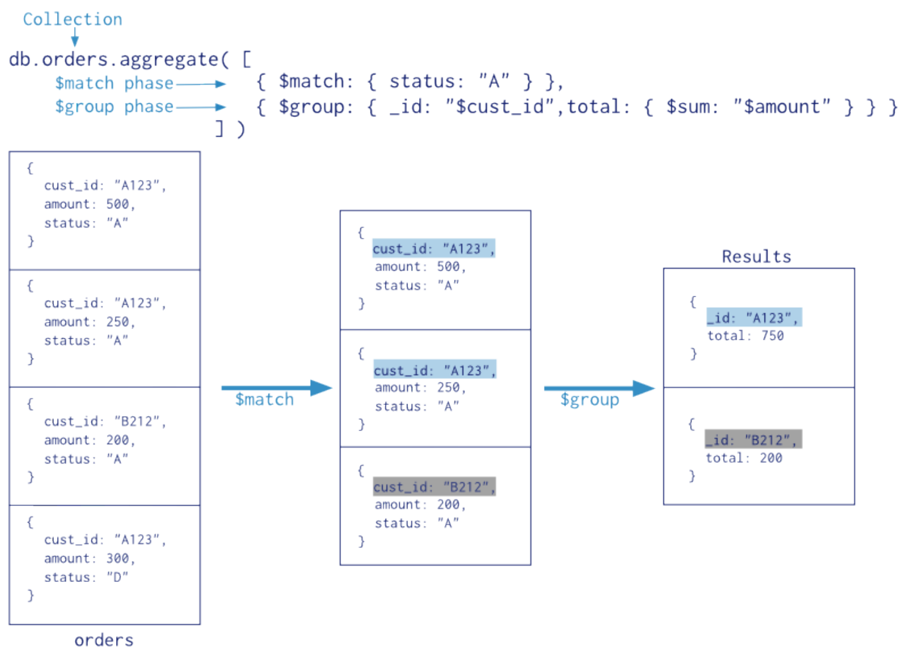


Figura 22. Ejemplo de pipeline con `$match` y `$group`

Operadores del *pipeline*

Antes de nada destacar que las fases se pueden repetir, por lo que una consulta puede repetir operadores.

Más información en: http://docs.mongodb.org/manual/reference/operator/aggregation/

A continuación vamos a estudiar todos estos operadores:

Tabla 5. Operadores del pipeline

Operador	Descripción	Cardinalidad
$project	Proyección de campos, es decir, propiedades en las que estamos interesados. También nos permite modificar un documento, o crear un subdocumento (*reshape*)	1:1
$match	Filtrado de campos, similar a *where*	N:1
$group	Para agrupar los datos, similar a *group by*	N:1
$sort	Ordenar	1:1
$skip	Saltar	N:1
$limit	Limitar los resultados	N:1
$unwind	Separa los datos que hay dentro de un array	1:N

Preparando los ejemplos

Para los siguientes ejemplos, vamos a utilizar una colección de productos (productos.js[20]) de un tienda de electrónica con las características y precios de los mismos.

Un ejemplo de un producto sería:

```
> db.productos.findOne()
{
  "_id" : ObjectId("5345afc1176f38ea4eda4787"),
  "nombre" : "iPad 16GB Wifi",
  "fabricante" : "Apple",
  "categoria" : "Tablets",
  "precio" : 499
}
```

Para cargar este archivo desde la consola podemos realizar:

```
mongo < productos.js
```

`$group`

Agrupa los documentos con el propósito de calcular valores agregrados de una colección de documentos. Por ejemplo, podemos usar `$group` para calcular la media de páginas visitas de manera diaria.

[20] resources/nosql/productos.js

La salida de $group esta desordenada

La salida de `$group` depende de como se definan los grupos. Se empieza especificando un identificador (por ejemplo, un campo `_id`) para el grupo que creamos con el pipeline. Para este campo `_id`, podemos especificar varias expresiones, incluyendo un único campo proveniente de un documento del pipeline, un valor calculado de una fase anterior, un documento con muchos campos y otras expresiones válidas, tales como constantes o campos de subdocumentos. También podemos usar operadores de `$project` para el campo `_id`.

Cuando referenciemos al valor de un campo lo haremos poniendo un `$` delante del nombre del campo. Así pues, para referenciar al fabricante de un producto lo haremos mediante `$fabricante`.

```
> db.productos.aggregate([{$group:
  {
    _id: { "empresa": "$fabricante" },
    total: { $sum:1 }
  }
}])
{ "_id" : { "empresa" : "Amazon" }, "total" : 2 }
{ "_id" : { "empresa" : "Sony" }, "total" : 1 }
{ "_id" : { "empresa" : "Samsung" }, "total" : 2 }
{ "_id" : { "empresa" : "Google" }, "total" : 1 }
{ "_id" : { "empresa" : "Apple" }, "total" : 4 }
```

También podemos agrupar más de un atributo, de tal modo que tengamos un `_id` compuesto. Por ejemplo:

```
> db.productos.aggregate([{$group:
  {
    _id: {
      "empresa": "$fabricante",
      "tipo" : "$categoria" },
    total: {$sum:1}
  }
}])
{ "_id" : { "empresa" : "Amazon", "tipo" : "Tablets" }, "total" : 2 }
{ "_id" : { "empresa" : "Google", "tipo" : "Tablets" }, "total" : 1 }
{ "_id" : { "empresa" : "Apple", "tipo" : "Portátiles" }, "total" : 1 }
{ "_id" : { "empresa" : "Sony", "tipo" : "Portátiles" }, "total" : 1 }
{ "_id" : { "empresa" : "Samsung", "tipo" : "Tablets" }, "total" : 1 }
{ "_id" : { "empresa" : "Samsung", "tipo" : "Smartphones" }, "total" : 1 }
{ "_id" : { "empresa" : "Apple", "tipo" : "Tablets" }, "total" : 3 }
```

Cada expresión de `$group` **debe** especificar un campo `_id`.

Además del campo `_id`, la expresión `$group` puede incluir campos calculados. Estos otros campos deben utilizar uno de los siguientes acumuladores.

Tabla 6. Operadores / Acumuladores de $group

Nombre	Descripción
$addToSet	Devuelve un array con todos los valores únicos para los campos seleccionados entre cada documento del grupo (sin repeticiones)
$first	Devuelve el primer valor del grupo. Se suele usar después de ordenar.
$last	Devuelve el último valor del grupo. Se suele usar después de ordenar.
$max	Devuelve el mayor valor de un grupo
$min	Devuelve el menor valor de un grupo.
$avg	Devuelve el promedio de todos los valores de un grupo
$push	Devuelve un array con todos los valores del campo seleccionado entre cada documento del grupo (puede haber repeticiones)
$sum	Devuelve la suma de todos los valores del grupo

A continuación vamos a ver ejemplos de cada uno de estos acumuladores.

`$sum`

El operador `$sum` acumula los valores y devuelve la suma.

Por ejemplo, para obtener el montante total de los prodyctos agrupados por fabricante, haríamos:

Agrupación con `$sum`

```
> db.productos.aggregate([{
  $group: {
    _id: {
      "empresa":"$fabricante"
    },
    totalPrecio: {$sum:"$precio"}
  }
}])
{ "_id" : { "empresa" : "Amazon" }, "totalPrecio" : 328 }
{ "_id" : { "empresa" : "Sony" }, "totalPrecio" : 499 }
{ "_id" : { "empresa" : "Samsung" }, "totalPrecio" : 1014.98 }
{ "_id" : { "empresa" : "Google" }, "totalPrecio" : 199 }
{ "_id" : { "empresa" : "Apple" }, "totalPrecio" : 2296 }
```

`$avg`

Mediante `$avg` podemos obtener el promedio de los valores de un campo numérico.

Por ejemplo, para obtener el precio medio de los productos agrupados por categoría, haríamos:

```
> db.productos.aggregate([{
  $group: {
    _id: {
      "categoria":"$categoria"
```

```
    },
    precioMedio: {$avg:"$precio"}
  }
}])
{ "_id" : { "categoria" : "Portátiles" }, "precioMedio" : 499 }
{ "_id" : { "categoria" : "Smartphones" }, "precioMedio" : 563.99 }
{ "_id" : { "categoria" : "Tablets" }, "precioMedio" : 396.4271428571428 }
```

`$addToSet`

Mediante `addToSet` obtendremos un array con todos los valores únicos para los campos seleccionados entre cada documento del grupo (sin repeticiones).

Por ejemplo, para obtener para cada empresa las categorías en las que tienen productos, haríamos:

```
> db.productos.aggregate([{
  $group: {
    _id: {
     "fabricante":"$fabricante"
    },
    categorias: {$addToSet:"$categoria"}
  }
}])
{ "_id" : { "fabricante" : "Amazon" }, "categorias" : [ "Tablets" ] }
{ "_id" : { "fabricante" : "Sony" }, "categorias" : [ "Portátiles" ] }
{ "_id" : { "fabricante" : "Samsung" }, "categorias" :
 [ "Tablets", "Smartphones" ] }
{ "_id" : { "fabricante" : "Google" }, "categorias" : [ "Tablets" ] }
{ "_id" : { "fabricante" : "Apple" }, "categorias" :
 [ "Portátiles", "Tablets" ] }
```

`$push`

Mediante `$push` también obtendremos un array con todos los valores para los campos seleccionados entre cada documento del grupo, pero con repeticiones. Es decir, funciona de manera similar a `$addToSet` pero permitiendo elementos repetidos.

Por ello, si reescribimos la consulta anterior pero haciendo uso de `$push` obtendremos categorías repetidas:

```
> db.productos.aggregate([{
  $group: {
    _id: {
      "empresa":"$fabricante"
    },
    categorias: {$push:"$categoria"}
  }
}])
{ "_id" : {"empresa" : "Amazon"}, "categorias" : ["Tablets", "Tablets"] }
{ "_id" : {"empresa" : "Sony"}, "categorias" : ["Portátiles"] }
{ "_id" : {"empresa" : "Samsung"}, "categorias" :
 ["Smartphones", "Tablets"] }
{ "_id" : {"empresa" : "Google"}, "categorias" : ["Tablets"] }
```

```
{ "_id" : {"empresa" : "Apple"}, "categorias" :
 ["Tablets", "Tablets", "Tablets", "Portátiles"] }
```

`$max` y `$min`

Los operadores `$max` y `$min` permiten obtener el mayor y el menor valor, respectivamente, del campo por el que se agrupan los documentos.

Por ejemplo, para obtener el precio del producto más caro que tiene cada empresa haríamos:

```
> db.productos.aggregate([{
  $group: {
    _id: {
      "empresa":"$fabricante"
    },
    precioMaximo: {$max:"$precio"},
    precioMinimo: {$min:"$precio"},
  }
}])
{ "_id" : { "empresa" : "Amazon" }, "precioMaximo" : 199, "precioMinimo"
 : 129 }
{ "_id" : { "empresa" : "Sony" }, "precioMaximo" : 499, "precioMinimo"
 : 499 }
{ "_id" : { "empresa" : "Samsung" }, "precioMaximo"
 : 563.99, "precioMinimo" : 450.99 }
{ "_id" : { "empresa" : "Google" }, "precioMaximo" : 199, "precioMinimo"
 : 199 }
{ "_id" : { "empresa" : "Apple" }, "precioMaximo" : 699, "precioMinimo"
 : 499 }
```

Doble `$group`

Si queremos obtener el resultado de una agrupación podemos aplicar el operador `$group` sobre otro `$group`.

Por ejemplo, para obtener el precio medio de los precios medios de los tipos de producto por empresa haríamos:

```
> db.productos.aggregate([
  {$group: {
    _id: {
      "empresa":"$fabricante",
      "categoria":"$categoria"
      },
    precioMedio: {$avg:"$precio"} ❶
    }
  },
  {$group: {
    _id: "$_id.empresa",
    precioMedio: {$avg: "$precioMedio"}  ❷
    }
  }
])
{ "_id" : "Samsung", "precioMedio" : 507.49 }
```

```
{ "_id" : "Sony", "precioMedio" : 499 }
{ "_id" : "Apple", "precioMedio" : 549 }
{ "_id" : "Google", "precioMedio" : 199 }
{ "_id" : "Amazon", "precioMedio" : 164 }
```

❶ Precio medio por empresa y categoría
❷ Precio medio por empresa en base al precio medio anterior

`$first` y `$last`

Estos operadores devuelven el valor resultante de aplicar la expresión al primer/último elemento de un grupo de elementos que comparten el mismo grupo por clave.

Por ejemplo, para obtener para cada empresa, cual es el tipo de producto que más tiene y la cantidad de dicho tipo haríamos:

```
> db.productos.aggregate([
  {$group: {  ❶
    _id: {
      "empresa": "$fabricante",
      "tipo" : "$categoria" },
    total: {$sum:1}
    }
  },
  {$sort: {"total":-1}},
  {$group: {
      _id:"$_id.empresa",
      producto: {$first: "$_id.tipo"  ❷
      },
      cantidad: {$first:"$total"}
    }
  }
])
{ "_id" : "Samsung", "producto" : "Tablets", "cantidad" : 1 }
{ "_id" : "Sony", "producto" : "Portátiles", "cantidad" : 1 }
{ "_id" : "Amazon", "producto" : "Tablets", "cantidad" : 2 }
{ "_id" : "Google", "producto" : "Tablets", "cantidad" : 1 }
{ "_id" : "Apple", "producto" : "Tablets", "cantidad" : 3 }
```

❶ Agrupamos por empresa y categoría de producto
❷ Al agrupar por empresa, elegimos la categoría producto que tiene más unidades

Más información en http://docs.mongodb.org/manual/reference/operator/aggregation/first/ y http://docs.mongodb.org/manual/reference/operator/aggregation/last/

`$project`

Si queremos realizar una proyección sobre el conjunto de resultados y quedarnos con un subconjunto de los campos usaremos el operador `$project`. Como resultado obtendremos el mismo número de documentos, y en el orden indicado en la proyección.

La proyección dentro del *framework* de agregación es mucho más potente que dentro de las consultas normales. Se emplea para:

- renombrar campos.

- introducir campos calculados en el documento resultante, mediante `$add`, `$substract`, `$multiply`, `$divide` o `$mod`
- transformar campos a mayúsculas `$toUpper` o minúsculas `$toLower`, concatenar campos mediante `$concat` u obtener subcadenas con `$substr`.
- transformar campos en base a valores obtenidos a partir de una condición mediante expresiones lógicas con los operadores de comparación vistos en las consultas.

```
> db.productos.aggregate([
  {$project:
    {
      _id:0,
      'empresa': {$toUpper:"$fabricante"},  ❶
      'detalles': { ❷
        'categoria': "$categoria",
        'precio': {"$multiply": ["$precio", 1.1]}  ❸
      },
      'elemento':'$nombre'  ❹
    }
  }
])
{ "empresa" : "APPLE", "detalles" : { "categoria" : "Tablets", "precio"
 : 548.9000000000001 }, "elemento" : "iPad 16GB Wifi" }
{ "empresa" : "APPLE", "detalles" : { "categoria" : "Tablets", "precio"
 : 658.9000000000001 }, "elemento" : "iPad 32GB Wifi" }
```

❶ Transforma un campo y lo pasa a mayúsculas
❷ Crea un documento anidado
❸ Incrementa el precio el 10%
❹ Renombra el campo

Más información en http://docs.mongodb.org/manual/reference/operator/aggregation/project/

`$match`

Se utiliza principalmente para filtrar los documentos que pasarán a la siguiente etapa del *pipeline* o a la salida final.

Por ejemplo, para seleccionar sólo las tabletas haríamos:

```
> db.productos.aggregate([{$match:{categoria:"Tablets"}}])
```

Aparte de igualar un valor a un campo, podemos emplear los operadores usuales de consulta, como `$gt`, `$lt`, `$in`, etc…

Se recomienda poner el operador `match` al principio del *pipeline* para limitar los documentos a procesar en siguientess fases. Si usamos este operador como primera fase podremos hacer uso de los indices de la colección de una manera eficiente.

Así pues, para obtener la cantidad de *Tablets* de menos de 500 euros haríamos:

```
> db.productos.aggregate([
```

```
  {$match:
    {categoria:"Tablets",
    precio: {$lt: 500}}},
  {$group:
    {_id: {"empresa":"$fabricante"},
    cantidad: {$sum:1}}
  }]
)
{ "_id" : { "empresa" : "Amazon" }, "cantidad" : 2 }
{ "_id" : { "empresa" : "Samsung" }, "cantidad" : 1 }
{ "_id" : { "empresa" : "Google" }, "cantidad" : 1 }
{ "_id" : { "empresa" : "Apple" }, "cantidad" : 1 }
```

Más información en http://docs.mongodb.org/manual/reference/operator/aggregation/match/

`$sort`

El operador `$sort` ordena los documentos recibidos por el campo y el orden indicado por la expresión indicada al *pipeline*.

Por ejemplo, para ordenar los productos por precio descendentemente haríamos:

```
> db.productos.aggregate({$sort:{precio:-1}})
```

El operador `$sort` ordena los datos en memoria, por lo que hay que tener cuidado con el tamaño de los datos. Por ello, se emplea en las últimas fases del pipeline, cuando el conjunto de resultados es el menor posible.

Si retomamos el ejemplo anterior, y ordenamos los datos por el precio total tenemos:

```
> db.productos.aggregate([
  {$match:{categoria:"Tablets"}},
  {$group:
    {_id: {"empresa":"$fabricante"},
    totalPrecio: {$sum:"$precio"}}
  },
  {$sort:{totalPrecio:-1}}  ❶
])
{ "_id" : { "empresa" : "Apple" }, "totalPrecio" : 1797 }
{ "_id" : { "empresa" : "Samsung" }, "totalPrecio" : 450.99 }
{ "_id" : { "empresa" : "Amazon" }, "totalPrecio" : 328 }
{ "_id" : { "empresa" : "Google" }, "totalPrecio" : 199 }
```

❶ Al ordenar los datos, referenciamos al campo que hemos creado en la fase de `$group`

Más información en http://docs.mongodb.org/manual/reference/operator/aggregation/sort/

`$skip` y `$limit`

El operador `$limit` únicamente limita el número de documentos que pasan a través del *pipeline*.

El operador recibe un número como parámetro:

```
> db.productos.aggregate([{$limit:3}])
```

Este operador no modifica los documentos, sólo restringe quien pasa a la siguiente fase.

De manera similar, con el operador `$skip`, saltamos un número determinado de documentos:

```
> db.productos.aggregate([{$skip:3}])
```

El orden en el que empleemos estos operadores importa, y mucho, ya que no es lo mismo saltar y luego limitar, donde la cantidad de elementos la fija `$limit`:

```
> db.productos.aggregate([{$skip:2},{$limit:4}])
{ "_id" : ObjectId("54ffff889836d613eee9a6e7"), "nombre" : "iPad 64GB
 Wifi", "categoria" : "Tablets", "fabricante" : "Apple", "precio" : 699 }
{ "_id" : ObjectId("54ffff889836d613eee9a6e8"), "nombre" : "Galaxy
 S3", "categoria" : "Smartphones", "fabricante" : "Samsung", "precio"
 : 563.99 }
{ "_id" : ObjectId("54ffff889836d613eee9a6e9"), "nombre" : "Galaxy Tab
 10", "categoria" : "Tablets", "fabricante" : "Samsung", "precio" : 450.99
 }
{ "_id" : ObjectId("54ffff889836d613eee9a6ea"), "nombre"
 : "Vaio", "categoria" : "Portátiles", "fabricante" : "Sony", "precio"
 : 499 }
```

En cambio, si primero limitamos y luego saltamos, la cantidad de elementos se obtiene de la diferencia entre el límite y el salto:

```
> db.productos.aggregate([{$limit:4},{$skip:2}])
{ "_id" : ObjectId("54ffff889836d613eee9a6e7"), "nombre" : "iPad 64GB
 Wifi", "categoria" : "Tablets", "fabricante" : "Apple", "precio" : 699 }
{ "_id" : ObjectId("54ffff889836d613eee9a6e8"), "nombre" : "Galaxy
 S3", "categoria" : "Smartphones", "fabricante" : "Samsung", "precio"
 : 563.99 }
```

Más información en http://docs.mongodb.org/manual/reference/operator/aggregation/limit/ y http://docs.mongodb.org/manual/reference/operator/aggregation/skip/

`$unwind`

Este operador es muy interesante y se utiliza solo con operadores array. Al usarlo con un campo array de tamaño N en un documento, lo transforma en N documentos con el campo tomando el valor individual de cada uno de los elementos del array.

Si retomamos el ejemplo de la segunda sesión donde actualizabamos una colección de enlaces, teníamos un enlace con la siguiente información:

```
> db.enlaces.findOne()
{
 "_id" : ObjectId("54f9769212b1897ae84190cf"),
 "titulo" : "www.google.es",
 "tags" : [
```

```
    "mapas",
    "videos",
    "blog",
    "calendario",
    "email",
    "mapas"
  ]
}
```

Podemos observar como el campo `tags` contiene 6 valores dentro del array (con un valor repetido). A continuación vamos a *desenrollar* el array:

```
> db.enlaces.aggregate(
  {$match:{titulo:"www.google.es"}},
  {$unwind:"$tags"})
{ "_id" : ObjectId("54f9769212b1897ae84190cf"), "titulo"
 : "www.google.es", "tags" : "mapas" }
{ "_id" : ObjectId("54f9769212b1897ae84190cf"), "titulo"
 : "www.google.es", "tags" : "videos" }
{ "_id" : ObjectId("54f9769212b1897ae84190cf"), "titulo"
 : "www.google.es", "tags" : "blog" }
{ "_id" : ObjectId("54f9769212b1897ae84190cf"), "titulo"
 : "www.google.es", "tags" : "calendario" }
{ "_id" : ObjectId("54f9769212b1897ae84190cf"), "titulo"
 : "www.google.es", "tags" : "email" }
{ "_id" : ObjectId("54f9769212b1897ae84190cf"), "titulo"
 : "www.google.es", "tags" : "mapas" }
```

Así pues hemos obtenido 6 documentos con el mismo `_id` y `titulo`, es decir, un documento por elemento del array.

De este modo, podemos realizar consultas que sumen/cuenten los elementos del array. Por ejemplo, si queremos obtener las 3 etiquetas que más aparecen en todos los enlaces haríamos:

```
> db.enlaces.aggregate([
  {"$unwind":"$tags"},
  {"$group":
   {"_id":"$tags",
    "total":{$sum:1}
   }
  },
  {"$sort":{"total":-1}},
  {"$limit": 3}
])
{ "_id" : "mapas", "total" : 3 }
{ "_id" : "email", "total" : 2 }
{ "_id" : "calendario", "total" : 1 }
```

Doble `$unwind`

Si trabajamos con documentos que tienen varios arrays, podemos necesitar desenrollar los dos array. Al hacer un doble `unwind` se crea un producto cartesiano entre los elementos de los 2 arrays.

Supongamos que tenemos los datos del siguiente inventario de ropa:

```
> db.inventario.drop();
> db.inventario.insert({'nombre':"Camiseta", 'tallas':
["S", "M", "L"], 'colores':['azul', 'blanco', 'naranja', 'rojo']})
> db.inventario.insert({'nombre':"Jersey", 'tallas':
["S", "M", "L", "XL"], 'colores':['azul', 'negro', 'naranja', 'rojo']})
> db.inventario.insert({'nombre':"Pantalones", 'tallas':
["32x32", "32x30", "36x32"], 'colores':
['azul', 'blanco', 'naranja', 'negro']})
```

Para obtener un listado de cantidad de pares talla/color haríamos:

```
> db.inventario.aggregate([
  {$unwind: "$tallas"},
  {$unwind: "$colores"},
  {$group:
    { '_id': {'talla': '$tallas', 'color': '$colores'},
    'total' : {'$sum': 1}
    }
  }
])
{ "_id" : { "talla" : "XL", "color" : "rojo" }, "total" : 1 }
{ "_id" : { "talla" : "XL", "color" : "negro" }, "total" : 1 }
{ "_id" : { "talla" : "L", "color" : "negro" }, "total" : 1 }
{ "_id" : { "talla" : "M", "color" : "negro" }, "total" : 1 }
...
```

De SQL al *Pipeline* de Agregaciones

Ya hemos visto que el *pipeline* ofrece operadores para realizar la misma funcionalidad de agrupación que ofrece SQL.

Si relacionamos los comandos SQL con el *pipeline* de agregaciones tenemos las siguientes equivalencias:

Tabla 7. Equivalencia con SQL

SQL	*Pipeline* de Agregaciones
WHERE	`$match`
GROUP BY	`$group`
HAVING	`$match`
SELECT	`$project`
ORDER BY	`$sort`
LIMIT	`$limit`
SUM()	`$sum`
COUNT()	`$sum`

Podemos encontrar ejemplos de consultas SQL transformadas al *pipeline* en http://docs.mongodb.org/manual/reference/sql-aggregation-comparison/

Limitaciones

Hay que tener en cuenta las siguiente limitaciones:

- En versiones anteriores a la 2.6, el *pipeline* devolvía en cada fase un objeto BSON, y por tanto, el resultado estaba limitado a 16MB
- Las fases tienen un límite de 100MB en memoría. Si una fase excede dicho límite, se producirá un error. En este caso, hay que habilitar el uso de disco mediante `allowDiskUse` en las opciones de la agregación. Más información en http://docs.mongodb.org/manual/reference/method/db.collection.aggregate

4.3. Agregaciones con *Java*

Para realizar agregaciones mediante *Java* emplearemos el método `aggregate(List<DBObject>)` el cual accepta una lista con el *pipeline* de la consulta.

Vamos a traducir la siguiente consulta:

```
>  db.productos.aggregate([
  {$match:{categoria:"Tablets"}},
  {$group:
    {_id: {"empresa":"$fabricante"},
    totalPrecio: {$sum:"$precio"}}
  },
  {$sort:{totalPrecio:-1}}
])
```

Así pues, una vez conectados a la base de datos correspondiente y a la colección `productos`, vamos a crear tres objetos (uno por cada fase), y se los pasaremos como una lista al método `aggregate()`:

```
DBObject match = new BasicDBObject("$match", new
 BasicDBObject("categoria", "Tablets"));
DBObject group = new BasicDBObject("$group",
  new BasicDBObject("_id", new BasicDBObject("empresa", "$fabricante"))
  .append("totalPrecio", new BasicDBObject("$sum", "$precio")));
DBObject sort = new BasicDBObject("$sort", new BasicDBObject("totaPrecio",
 -1));

AggregationOutput output = coleccion.aggregate(Arrays.asList(match, group,
 sort));   ❶

Iterable<DBObject> datos = output.results();
for (DBObject doc : datos) {   ❷
  System.out.println(doc);
}
```

❶ Creamos una lista a partir de los `DBObject`
❷ Recorremos el conjunto de datos

El resultado es un objeto `AggregationOutput`, el cual contiene el método `results()` que nos devuelve un iterador el cual podemos recorrer.

4.4. Replicación

Un aspecto muy importante de *MongoDB* es que soporta la replicación de los datos de forma nativa mediante el uso de conjuntos de réplicas.

Conjunto de Réplicas

En *MongoDB* se replican los datos mediante un conjunto de réplicas. Un **Conjunto de Réplicas** (*Replica Set*) es un grupo de servidores (nodos `mongod`) donde hay uno que ejerce la función de **primario** y por tanto recibe las peticiones de los clientes, y el resto de servidores hace de **secundarios**, manteniendo copias de los datos del primario.

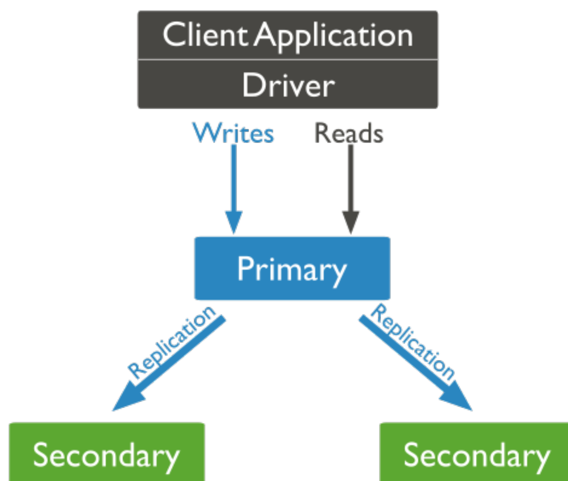


Figura 23. Conjunto de Réplicas

Si el nodo primario se cae, los secundarios eligen un nuevo primario entre ellos mismos, en un proceso que se conoce como votación. La aplicación se conectará al nuevo primario de manera transparente. Cuando el antiguo nodo primario vuelva en sí, será un nuevo nodo secundario.

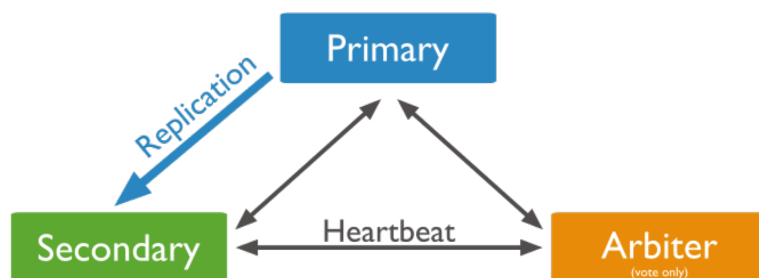


Figura 24. Arbitraje de un secundario

Al usar replicación, si un servidor se cae, siempre vamos a poder obtener los datos a partir de otros servidores del conjunto. Si los datos de un servidor se dañan o son inaccesibles, podemos crear una nueva copia desde uno de los miembros del conjunto.

Elementos de un Conjunto de Réplicas

Los tipos de nodos que podemos encontrar en un conjunto de réplica son:

- Regular: Es el tipo de nodo más común.

Primario: Acepta todas las operaciones de escritura de los clientes. Cada conjunto de réplicas tendrá sólo un primario, y como sólo un miembro acepta operaciones de escritura, ofrece consitencia estricta para todas las lecturas realizadas desde él.

Secundario: Los secundarios replican el *oplog* primario y aplican las operaciones a sus conjuntos de datos. De este modo, los nodos secundarios son un espejo del primario. Si el primario deja de estar disponible, el conjunto de replica elegirá a un secundario para que sea el nuevo primario, mediante un proceso de votación.

Por defecto, los clientes realizan las lecturas desde el nodo primario. Sin embargo, los clientes pueden indicar que quieren realizar lecturas desde los nodos secundarios.

Es posible que al realizar lecturas de un nodo secundario la información que se obtenga no refleje el estado del nodo primario.

- **Árbitro**: se emplea sólo para votar. No contiene copia de los datos y no se puede convertir en primario. Los conjuntos de réplica pueden tener árbitros para añadir votos en las elecciones de un nuevo primario. Siempre tienen un voto, y permiten que los conjuntos de replica tengan un número impar de nodos, sin la necesidad de tener un miembro que replique los datos. Además, no requieren hardware dedicado.

No ejecutar un árbitro en sistemas que también ejecutan los miembros primarios y secundarios del conjunto de réplicas.

Sólo añadir un árbitro a un conjunto con un número par de miembros.

Si se añade un árbitro a un conjunto con un número impar de miembros, el conjunto puede sufrir un empate.

- Retrasado (*delayed*): nodo que se emplea para la recuperación del sistema ante un fallo. Para ello, hay que asignar la propiedad `priority:0`. Este nodo nunca será un nodo primario.
- Oculto: empleado para analíticas del sistema.

oplog

Para soportar la replicación, el nodo primario almacena todos los cambios en su **oplog**.

De manera simplificada, el *oplog* es un diario de todos los cambios que la instancia principal realiza en las bases de datos con el propósito de replicar dichos cambios en un nodo secundario para asegurar que las dos bases de datos sean idénticas.

El servidor principal mantiene el *oplog*, y el secundario consulta al principal por nuevas entradas que aplicar a sus propias copias de las bases de datos replicadas.

El *oplog* crea un *timestamp* para cada entrada. Esto permite que un secundario controle la cantidad de información que se ha modificado desde una lectura anterior, y qué entradas necesita transferir para ponerse al día. Si paramos un secundario y lo reiniciamos más adelante, utilizará el *oplog* para obtener todos los cambios que ha perdido mientras estaba *offline*.

El *oplog* se almacena en una colección limitada (*capped*) y ordenada de un tamaño determinado. La opción `oplogSize` define en MB el tamaño del archivo. Para un sistema de 64 bits con comportamiento de lectura/escritura normales, el `oplogSize` debería ser

de al menos un 5% del espacio de disco disponible. Si el sistema tiene más escrituras que lecturas, puede que necesitemos incrementar este tamaño para asegurar que cualquier nodo secundario pueda estar *offline* una cantidad de tiempo razonable sin perder información.

Más información de *oplog* en http://docs.mongodb.org/manual/core/replica-set-oplog/

Creando un Conjunto de Réplicas

A la hora de lanzar una instancia, podemos indicarle mediante parámetros opcionales la siguiente información:

- `--dbpath` : ruta de la base de datos
- `--port` : puerto de la base de datos
- `--replSet` : nombre del conjunto de réplicas
- `--fork` : indica que se tiene que crear en un hilo
- `--logpath` : ruta para almacenar los archivos de *log*.

Normalmente, cada instancia `mongod` se coloca en un servidor físico y todos en el puerto estándar.

Como ejemplo vamos a crear un conjunto de tres réplicas. Para ello, arrancaremos tres instancias distintas pero que comparten el mismo conjunto de réplicas. Además, en vez de hacerlo en tres máquinas distinas, lo haremos en tres puertos diferentes:

Las carpeta que se crean tienen que tener los mismos permisos que `mongod` . Si no existiesen, las tenemos que crear previamente.

Script de creación del conjunto de réplicas - (creaConjuntoReplicas.sh[21])

```
#!/bin/bash
mkdir -p /data/db/rs1 /data/db/rs2 /data/db/rs3 /data/logs
mongod --replSet replicaExperto --logpath /data/logs/rs1.log --dbpath /
data/db/rs1 --port 27017 --oplogSize 64 --smallfiles --fork
mongod --replSet replicaExperto --logpath /data/logs/rs2.log --dbpath /
data/db/rs2 --port 27018 --oplogSize 64 --smallfiles --fork
mongod --replSet replicaExperto --logpath /data/logs/rs3.log --dbpath /
data/db/rs3 --port 27019 --oplogSize 64 --smallfiles --fork
```

Y lo lanzamos desde el shell mediante:

Lanzando la creación de la réplica

```
bash < creaConjuntoReplicas.sh
```

Al lanzar el *script*, realmente estamos creando las réplicas, por lo que obtendremos que ha creado hijos y que esta a la espera de conexiones:

[21] resources/nosql/creaConjuntoReplicas.sh

Resultado de crear el conjunto de replicas

```
about to fork child process, waiting until server is ready for
 connections.
forked process: 1811
child process started successfully, parent exiting
about to fork child process, waiting until server is ready for
 connections.
forked process: 1814
child process started successfully, parent exiting
about to fork child process, waiting until server is ready for
 connections.
forked process: 1817
child process started successfully, parent exiting
```

Una vez lanzados las tres réplicas, tenemos que enlazarlas.

Así pues, nos conectaremos al *shell* de `mongo`. Puede ser que necesitemos indicar que nos conectamos al puerto adecuado:

```
mongo --port 27017
```

Para comprobar su estado emplearemos el comando `rs.status()`:

```
> rs.status()
{
  "startupStatus" : 3,
  "info" : "run rs.initiate(...) if not yet done for the set",
  "ok" : 0,
  "errmsg" : "can't get local.system.replset config from self or any seed
 (EMPTYCONFIG)"
}
```

Dentro del *shell* de `mongo`, los comandos que trabajan con réplicas comienzan por el prefijo `rs.`. Mediante `rs.help()` obtendremos la ayuda de los métodos disponibles,

A continuación, crearemos un documento con la configuración donde el `_id` tiene que ser igual al usado al crear la réplica, y el array de `members` contiene las replicas creadas donde los puertos han de coincidir.

Configurando el conjunto de réplicas

```
> config = { _id: "replicaExperto", members:[
  { _id : 0, host : "localhost:27017"},
  { _id : 1, host : "localhost:27018"},
  { _id : 2, host : "localhost:27019"}
]};
```

Si en los miembros ponemos `slaveDelay: numSeg` podemos retrasar un nodo respecto al resto (también deberemos indicar que `priority :`

> `0` para que no sea un nodo principal). Más información en http://docs.mongodb.org/manual/core/replica-set-delayed-member/

Tras crear el documento de configuración, podemos iniciar el conjunto mediante:

```
> rs.initiate(config)
{
  "info" : "Config now saved locally.  Should come online in about a
 minute.",
  "ok" : 1
}
```

Si ahora volvemos a consultar el estado de la réplica tendremos:

```
> rs.status()
{
  "set" : "replicaExperto",
  "date" : ISODate("2015-05-02T15:52:20Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "localhost:27017",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 572,
      "optime" : Timestamp(1430581402, 1),
      "optimeDate" : ISODate("2015-05-02T15:43:22Z"),
      "electionTime" : Timestamp(1430581412, 1),
      "electionDate" : ISODate("2015-05-02T15:43:32Z"),
      "self" : true
    },
    {
      "_id" : 1,
      "name" : "localhost:27018",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 536,
      "optime" : Timestamp(1430581402, 1),
      "optimeDate" : ISODate("2015-05-02T15:43:22Z"),
      "lastHeartbeat" : ISODate("2015-05-02T15:52:19Z"),
      "lastHeartbeatRecv" : ISODate("2015-05-02T15:52:18Z"),
      "pingMs" : 0,
      "syncingTo" : "localhost:27017"
    },
    {
      "_id" : 2,
      "name" : "localhost:27019",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 536,
      "optime" : Timestamp(1430581402, 1),
      "optimeDate" : ISODate("2015-05-02T15:43:22Z"),
```

```
        "lastHeartbeat" : ISODate("2015-05-02T15:52:19Z"),
        "lastHeartbeatRecv" : ISODate("2015-05-02T15:52:19Z"),
        "pingMs" : 0,
        "syncingTo" : "localhost:27017"
      }
    ],
    "ok" : 1
  }
```

La próxima vez que lancemos las réplicas ya no deberemos configurarlas. Así pues, el proceso de enlazar e iniciar las réplicas sólo se realiza una vez.

Trabajando con las Réplicas

Una vez que hemos visto que las tres réplicas están funcionando, vamos a comprobar como podemos trabajar con ellas.

Para ello, nos conectamos al nodo principal (al ser el puerto predeterminado, podemos omitirlo):

```
$ mongo --port 27017
```

Al conectarnos al nodo principal, nos aparece como símbolo del *shell* el nombre del conjunto de la réplica seguido de dos puntos y `PRIMARY` si nos hemos conectado al nodo principal, o `SECONDARY` en caso contrario.

```
replicaExperto:PRIMARY>
```

Para saber si nos hemos conectado al nodo correcto, mediante `rs.isMaster()` obtendremos el tipo del nodo (propiedad `ismaster`) e información sobre el resto de nodos:

```
replicaExperto:PRIMARY> rs.isMaster()
{
  "setName" : "replicaExperto",
  "setVersion" : 1,
  "ismaster" : true,
  "secondary" : false,
  "hosts" : [
    "localhost:27017",
    "localhost:27019",
    "localhost:27018"
  ],
  "primary" : "localhost:27017",
  "me" : "localhost:27017",
  "maxBsonObjectSize" : 16777216,
  "maxMessageSizeBytes" : 48000000,
  "maxWriteBatchSize" : 1000,
  "localTime" : ISODate("2015-05-02T16:02:42.291Z"),
  "maxWireVersion" : 2,
  "minWireVersion" : 0,
  "ok" : 1
}
```

Ahora que sabemos que estamos en el nodo principal, vamos a insertar datos.

Para ello, vamos a insertar 100 documentos:

Insertamos 100 documentos sobre `replicaExperto:PRIMARY`

```
for (i=0; i<1000; i++) {
  db.pruebas.insert({num: i})
}
```

Estos 1000 documentos se han insertado en el nodo principal, y se han replicado a los secundarios. Para comprobar la replicación, abrimos un nuevo terminal y nos conectamos a un nodo secundario:

```
$ mongo --port 27018
replicaExperto:SECONDARY>
```

Si desde el nodo secundario intentamos consultar el total de documentos de la colección obtendremos un error:

```
replicaExperto:SECONDARY> db.pruebas.count()
count failed: { "note" : "from execComand", "ok" : 0, "errmsg" : "not
 master"}
```

El error indica que no somos un nodo primario y por lo tanto no podemos leer de él. Para permitir lecturas en los nodos secundarios, mediante `rs.slaveOk()` le decimos a `mongo` que sabemos que nos hemos conectado a un secundairo y admitimos la posibilidad de obtener datos obsoletos.

```
replicaExperto:SECONDARY> rs.slaveOk()
replicaExperto:SECONDARY> db.pruebas.count()
1000
```

Pero que podamos leer no significa que podamos escribir. Si intentamos escribir en un nodo secundario obtendremos un error:

```
replicaExperto:SECONDARY> db.pruebas.insert({num : 1001})
{"writeError": { "code" : undefined, "errmsg" : "not master"}}
```

Tolerancia a Fallos

Cuando un nodo primario no se comunica con otros miembros del conjunto durante más de 10 segundos, el conjunto de réplicas intentará, de entre los secundarios, que un miembro se convierta en el nuevo primario.

Para ello se realiza un proceso de **votación**, de modo que el nodo que obtenga el mayor número de votos se erigirá en primario. Este proceso de votación se realiza bastante rápido (menos de 3 segundos), durante el cual no existe ningún nodo primario y por tanto la réplica no acepta escrituras y todos los miembros se convierten en nodos de sólo-lectura.

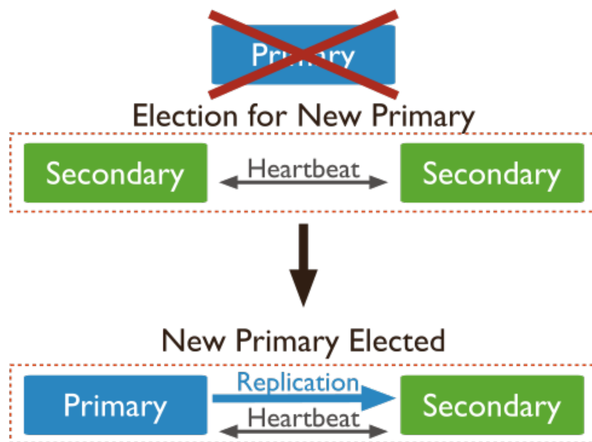


Figura 25. Elección de un nuevo primario

Proceso de Votación

Cuando un nodo secundario no puede contactar con su nodo primario, contactará con el resto de miembros y les indicará que quiere ser elegido como primario. Es decir, cada nodo que no encuentre un primario se nominará como posible primario, de modo que un nodo no nomina a otro a ser primario, únicamente vota sobre una nominación ya existente.

Antes de dar su voto, el resto de nodos comprobarán:

- si ellos tienen conectividad con el primario
- si el nodo que solicita ser primario tienen una replica actualizada de los datos. Todas las operaciones replicadas están ordenadas por el *timestamp* ascendentemente, de modo los candidatos deben tener operaciones posteriores o iguales a cualquier miembro con el que tengan conectividad.
- si existe algún nodo con una prioridad mayor que debería ser elegido.

Si algún miembro que quiere ser primario recibe una mayoría de *"sís"* se convertirá en el nuevo primario, siempre y cuando no haya un servidor que vete la votación. Si un miembro la veta es porque conoce alguna razón por la que el nodo que quiere ser primario no debería serlo, es decir, ha conseguido contactar con el antiguo primario.

Una vez un candidato recibe una mayoría de *"sís"*, su estado pasará a ser primario.

Cantidad de Elementos

En la votación, se necesita una mayoría de nodos para elegir un primario, ya que una escritura se considera segura cuando ha alcanzado a la mayoría de los nodos. Esta mayoría se define como **más de la mitad** de todos los nodos del conjunto. Hay que destacar que la mayoría no se basa en los elementos que queden en pie o estén disponibles, sino en el conjunto definido en la configuración del conjunto.

Por lo tanto, es importante configurar el conjunto de una manera que siempre se puede elegir un nodo primario. Por ejemplo, en un conjunto de cinco nodos, si los nodos 1, 2 y 3 están en un centro de datos y los miembros 4 y 5 en otro, debería haber casi siempre una mayoría disponible en el primer centro de datos (es más probable que se pierda la conexión de red entre centros de datos que dentro de ellos).

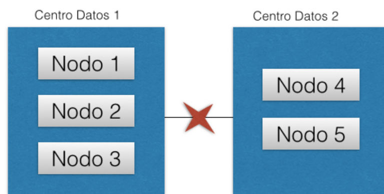


Figura 26. Elección de un nuevo primario

Por lo tanto, una configuración que hay que evitar es aquella compuesta por dos elementos: uno primario y uno secundario. Si uno de los dos miembros deja de estar disponible, el otro miembro no puede verlo. En esta situación, ninguna parte de la partición de red tiene una mayoría, con lo que acabaríamos con dos secundarios.

Por ello, el número mínimo de nodos es 3, para que al realizar una nueva elección se pueda elegir un nuevo nodo.

Comprobando la Tolerancia

Para comprobar esto, desde el nodo primario vamos a detenerlo:

```
replicaExperto:PRIMARY> db.adminCommand({"shutdown" : 1})
```

Otra posibilidad en vez de detenerlo es degradarlo a nodo secundario:

```
replicaExperto:PRIMARY> rs.stepDown()
```

Si pasamos al antiguo nodo secundario, y le preguntamos si es el principal obtendremos:

```
replicaExperto:SECONDARY> rs.isMaster()
{
  "setName" : "replicaExperto",
  "setVersion" : 1,
  "ismaster" : false,
  "secondary" : true,
  "hosts" : [
    "localhost:27018",
    "localhost:27019",
    "localhost:27017"
  ],
  "primary" : "localhost:27019",
  "me" : "localhost:27018",
  "maxBsonObjectSize" : 16777216,
  "maxMessageSizeBytes" : 48000000,
  "maxWriteBatchSize" : 1000,
  "localTime" : ISODate("2015-03-24T21:55:27.382Z"),
  "maxWireVersion" : 2,
  "minWireVersion" : 0,
  "ok" : 1
}
```

Si nos fijamos en la propiedad `primary`, veremos que tenemos un nuevo primario.

Configuración Recomendada

Se recomiendan dos configuraciones:

1. Mediante una mayoría del conjunto en un centro de datos. Este planteamiento es bueno si tenemos un *data center* donde queremos que siempre se aloje el nodo primario de la réplica. Siempre que el centro de datos funcione normalmente, habrá un nodo primario. Sin embargo, si el centro primario pierde la conectividad, el centro de datos secundario no podrá elegir un nuevo primario.
2. Mediante el mismo número de servidores en cada centro de datos, más un servidor que rompe la igualdad en una tercera localización. Este diseño es conveniente cuando ambos centros de datos tienen el mismo grado de confiabilidad y robustez.

Recuperación del Sistema

Si en un conjunto de réplicas se cae el primario y hay escrituras que se han pasado al *oplog* de modo que los otros nodos no las han replicado, cuando el nodo primario vuelva en sí como secundario y se sincronice con el primario, se dará cuenta que hay operaciones de escritura pendientes y las pasará a *rollback*, para que si se desean se apliquen manualmente.

Para evitar este escenario, se necesita emplear consistencia en la escritura, de manera que hasta que la escritura no se haya replicado en la mayoría de los nodos no se considere como una escritura exitosa.

Consistencia en la Escritura

Ya hemos visto que tanto las lecturas como las escrituras se realizan de manera predeterminada en el nodo primario.

Las aplicaciones pueden decidir que las escrituras vayan al nodo primario pero las lecturas al secundario. Esto puede provocar que haya lecturas caducas, con datos obsoletos, pero como beneficio podemos escalar el sistema.

La replicación es un proceso asíncrono. En el período de tiempo en el que el sistema de votación sucede, no se completa ninguna escritura.

MongoDB garantiza la consistencia en la escritura, haciendo que sea un sistema consistente. Para ello, ofrece un sistema que garantiza que una escritura ha sido exitosa. Dependiendo del nivel de configuración de la consistencia, las inserciones, modificaciones y borrados pueden tardar más o menos. Si reducimos el nivel de consistencia, el rendimiento será mejor, a costa de poder obtener datos obsoletos u perder datos que no se han terminado de serializar en disco. Con un nivel de consistencia más alto, los clientes esperan tras enviar una operación de escritura a que *MongoDB* les confirme la operación.

Los valores que podemos configurar se realizan mediante las siguientes opciones:

- `w`: indica el número de servidores que se han de replicar para que la inserción devuelva un ACK.
- `j`: indica si las escrituras se tienen que trasladar a un diario de bitácora (*journal*)
- `wtimeout`: indica el límite de tiempo a esperar como máximo, para prevenir que una escritura se bloquee indefinidamente.

Niveles de consistencia

Con estas opciones, podemos configurar diferentes niveles de consistencia son:

- Sin confirmación: `w:0`
- Con confirmación: `w:1`
- Con diario: `w:1, j:true`. Cada inserción primero se escribe en el diario y posteriormente en el directorio de datos.
- Con confirmación de la mayoría: `w:"majority"`

Estas opciones se indican como parámetro final en las operaciones de insercion y modificación de datos. Por ejemplo:

Insertando un documento indicando el nivel de consistencia

```
db.pruebas.insert(
  {num : 1002},
  {writeConcern: {w:"majority", wtimeout: 5000}}
)
```

Más información en http://docs.mongodb.org/manual/core/write-concern/ y https://www.youtube.com/watch?v=49BPAY1Yb5w

4.5. Replicación en *Java*

Para conectar a una réplica desde *Java*, en el constructor del `MongoClient` le pasaremos como parámetro una lista con los elementos del conjunto:

Ejemplo de conexión a un Conjunto de Réplicas

```
MongoClient cliente = new MongoClient(Arrays.asList(   ❶
  new ServerAddress("localhost", 27017),
  new ServerAddress("localhost", 27018),
  new ServerAddress("localhost", 27019)));

// Resto de código similar a si nos conectamos a un sólo servidor
DBCollection coleccion =
 cliente.getDB("expertojava").getCollection("pruebas");
coleccion.drop();

for (int i = 0; i < 1000; i++) {
  coleccion.insert(new BasicDBObject("_id", i));
  System.out.println("Insertado documento: " + i);
}
```

❶ Creamos la lista con ayuda de `Arrays.asList()`

Autoevaluación

Si dejamos un nodo fuera de la réplica dentro de la lista de conexión del driver ¿Qué pasará? [22]

[22] la opción correcta es la 2ª, es decir, si ponemos un nodo de manera correcta, la replica cargará todos los nodos que la forman

> - La aplicación no utilizará el nodo
> - El nodo perdido será descubierto siempre y cuando en la lista haya un nodo válido
> - El nodo perdido se empleará para lectura, pero no para escrituras
> - El nodo perdido se utilizará para escrituras, pero no para lecturas

Cuando nos conectamos a una réplica, en el cliente le pasamos una lista de nodos (`ServerAdress`) a conectarnos. Aunque podemos no indicar todos los nodos del conjunto, es conveniente ponerlos.

A partir de la lista de servidores, conocida como lista de semillas (*seed list*), la clase `MongoClient` averigua los detalles del servidor a partir de los metadatos de la replica, obtiendo todos los nodos que forman la réplica, así como quien es el primario en dicho momento. Es decir, si nos dejamos algún nodo fuera, el *driver* descubrirá la configuración del resto de nodos del conjunto y realizará las operaciones sobre los nodos a los no que no nos hemos conectado de manera explicita.

Además, `MongoClient` mantendrá en todo momento un hilo en *background* con la réplica para estar informado de cualquier cambio que suceda en el estado de la misma.

Reintento de Operaciones

En el caso de operaciones que insertan o modifican los datos, puede que sea conveniente realizar varios intentos con las operaciones para dar tiempo a que el conjunto de réplicas se recupere:

Ejemplo de Reintento de Operación

```
MongoClient client = new MongoClient(Arrays.asList(
  new ServerAddress("localhost", 27017),
  new ServerAddress("localhost", 27018),
  new ServerAddress("localhost", 27019)));

DBCollection coleccion =
 client.getDB("expertojava").getCollection("pruebas");
coleccion.drop();

for (int i = 0; i < Integer.MAX_VALUE; i++) {
  for (int intentos = 0; intentos <= 2; intentos++) {
    try {
      coleccion.insert(new BasicDBObject("_id", i));
      System.out.println("Documento insertado: " + i);
      break;
    } catch (MongoException.DuplicateKey e) {
      System.out.println("Documento ya insertado: " + i);
    } catch (MongoException e) {  ❶
      System.out.println(e.getMessage());
      System.out.println("Reintentando");
      Thread.sleep(5000); ❷
    }
  }
  Thread.sleep(500);
}
```

❶ Si mientras estamos escribiendo sobre la réplica, el nodo primario se cae, el driver lanzará una `MongoException`. Por ello, es conveniente capturar la excepción, y si la operación que estamos realizando es una inserción, reintentarla.

❷ Esperamos 5 segundos a que la réplica se recupere y un secundario se convierta en primario

Para poder probar este ejemplo, tras lanzar la clase Java, comenzará a insertar documentos sin parar. En un terminal nos conectamos al nodo primario y lo degradamos mediante `rs.stepDown()` para que se produzca una votación y se elija un nuevo primario:

```
replicaExperto:PRIMARY> rs.stepDown()
2015-05-03T09:41:52.619+0200 DBClientCursor::init call() failed
2015-05-03T09:41:52.621+0200 Error: error doing query: failed at src/
mongo/shell/query.js:81
2015-05-03T09:41:52.623+0200 trying reconnect to 127.0.0.1:27018
 (127.0.0.1) failed
2015-05-03T09:41:52.626+0200 reconnect 127.0.0.1:27018 (127.0.0.1) ok
replicaExperto:SECONDARY>
```

Si comprobamos la salida del sistema, veremos como el sistema se recupera de una caída del sistema y reintenta la inserción:

```
...
Documento insertado: 39855
Operation on server localhost:27018 failed
Reintentando
Documento insertado: 39856
...
```

Consistencia de Escritura

En *Java*, para indicar el nivel de consistencia, se realiza mediante el método `setWriteConcern(nivel)`, siendo nivel uno de los valores de la enumeración `WriteConcern`, la cual puede tomar diferentes valores, siendo los más importantes:

- `WriteConcern.UNACKNOWLEDGED`
- `WriteConcern.ACKNOWLEDGED`
- `WriteConcern.JOURNALED`
- `WriteConcern.MAJORITY`

Este nivel se puede indicar a nivel de:

- cliente: `cliente.setWriteConcern(nivel)`
- de base de datos: : `db.setWriteConcern(nivel)`
- de colección: `coleccion.setWriteConcern(nivel)`
- o con cada inserción : `cliente.insert(doc, nivel)`

Ejemplo de `WriteConcern`

```
MongoClient cliente = new MongoClient(Arrays.asList(
  new ServerAddress("localhost", 27017),
  new ServerAddress("localhost", 27018),
  new ServerAddress("localhost", 27019)));

cliente.setWriteConcern(WriteConcern.JOURNALED);

DB db = cliente.getDB("expertojava");
db.setWriteConcern(WriteConcern.ACKNOWLEDGED);
DBCollection coleccion = db.getCollection("pruebas");
coleccion.setWriteConcern(WriteConcern.MAJORITY);

coleccion.drop();

DBObject doc = new BasicDBObject("_id", 1);

coleccion.insert(doc);

try {
  coleccion.insert(doc, WriteConcern.UNACKNOWLEDGED);
} catch (MongoException e) {
  System.out.println(e.getMessage());
}
```

Preferencia de Lectura

La preferencia de lectura permite indicar al driver a qué servidores podemos enviar consultas.

Como regla general, enviar peticiones de lectura a nodos secundarios es una mala decisión.

Los posibles preferencias de lectura llevan asociadas situaciones en las que tiene sentido realizar la lectura desde un nodo secundario. Así pues, las preferencias se dividen en:

- `primary`: valor por defecto. Todas las lecturas se realizan en el nodo primario. Si se cae este nodo, el sistema no acepta lecturas. Es el único modo que garantiza los datos más recientes, ya que todas las escrituras pasan por él.
- `primaryPreferred`: realiza la lectura de nodos secundarios cuando el primario ha caído. Si el nodo primario está en pie siempre se realizará la lecturas sobre él.
- `nearest`: en ocasiones los nodos secundarios están más cercanos al cliente. En este caso, podemos acceder a los datos con la menor latencia posible, sin tener en cuenta si accedemos a un primario o a un secundario.
- `secondary`: envía las lecturas a nodos secundarios. Si no hubiese ningún nodo secundario disponible, la lectura fallará. Se emplea en aplicaciones que sólo quieren utilizar el nodo primario para escrituras, o bien en aplicaciones que quieres sacar estadísticas de los datos y no quieren cargar el nodo principal.
- `secondaryPreferred`: envía las lecturas a nodos secundarios, pero si no hubiese ninguno, la enviaría al primario.

Las lecturas de los nodos secundarios toman sentido cuando tenemos datos en los que la cantidad de operaciones de lectura son completamente predominantes sobre las escrituras.

`ReadPreference`

Para configurar estos posibilidades mediante *Java* emplearemos la clase `ReadPreference`, la cual ofrece métodos para configurar los valores vistos.

Se puede configura con granularidad de:

- cliente: `client.setReadPreference()`
- base de datos: `db.setReadPreference()`
- colección: `col.setReadPreference()`
- operación: `cursor.setReadPreference()`

Ejemplo de `ReadPreference`

```
MongoClient cliente = new MongoClient(Arrays.asList(
  new ServerAddress("localhost", 27017),
  new ServerAddress("localhost", 27018),
  new ServerAddress("localhost", 27019)));
cliente.setReadPreference(ReadPreference.primary());

DB db = cliente.getDB("expertojava");
db.setReadPreference(ReadPreference.primary());
DBCollection coleccion = db.getCollection("pruebas");
coleccion.setReadPreference(ReadPreference.primaryPreferred());

DBCursor cursor =
 coleccion.find().setReadPreference(ReadPreference.nearest()); ❶
try {
  while (cursor.hasNext()) {
    System.out.println(cursor.next());
  }
} finally {
  cursor.close();
}
```

❶ El driver elige el servidor al que enviar la consulta mediante un ping a la máquina para averiguar el tiempo empleado (ms).

Más información en http://docs.mongodb.org/manual/core/read-preference/ y http://docs.mongodb.org/manual/reference/read-preference

4.6. Particionado (*Sharding*)

Ya vimos en la primera sesión que dentro del entorno de las bases de datos, particionar consiste en dividir los datos entre múltiples máquinas. Al poner un subconjunto de los datos en cada máquina, vamos a poder almacenar más información y soportar más carga sin necesidad de máquinas más potentes, sino una mayor cantidad de máquinas más modestas (y mucho más baratas).

El *Sharding* es una técnica que fragmenta los datos de la base de datos horizontalmente agrupándolos de algún modo que tenga sentido y que permita un direccionamiento más rápido.

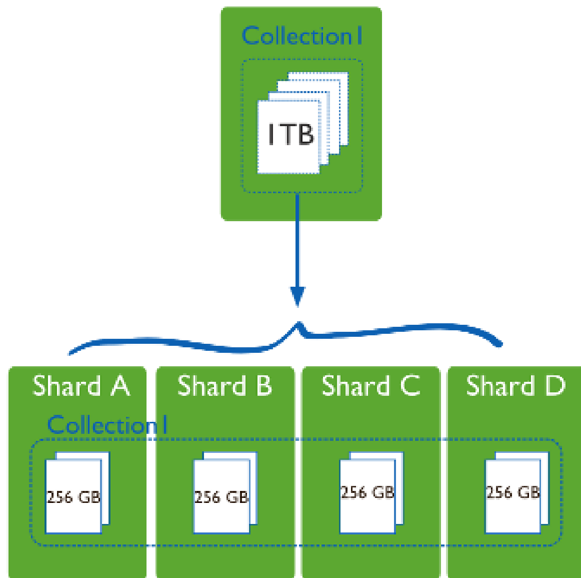


Figura 27. Sharding

Por lo tanto, estos *shards* (fragmentos) pueden estar localizados en diferentes bases de datos y localizaciones físicas.

El *Sharding* no tiene por qué estar basado únicamente en una colección y un campo, puede ser a nivel de todas las colecciones. Por ejemplo podríamos decir *"todos los datos de usuarios cuyo perfil está en los Estados Unidos los redirigimos a la base de datos del servidor en Estados Unidos, y todos los de Asia van a la base de datos de Asia"*.

Particionando con *MongoDB*

MongoDB implementa el *sharding* de forma nativa y **automática** (de ahí el término de *auto-sharding*), siguiendo un enfoque **basado en rangos**.

Para ello, divide una colección entre diferentes servidores, utilizando `mongos` como router de las peticiones entre los *sharded clusters*.

Esto favorece que el desarrollador ignore que la aplicación no se comunica con un único servidor, balanceando de manera automática los datos y permitiendo incrementar o reducir la capacidad del sistema a conveniencia.

Antes de plantearse hacer *auto-sharding* sobre nuestros datos, es conveniente dominar cómo se trabaja con *MongoDB* y el uso de conjuntos de réplica.

Sharded Cluster

El particionado de *MongoDB* permite crear un cluster de muchas máquinas, dividiendo a nivel de colección y poniendo un subconjunto de los datos de la colección en cada uno de los fragmentos.

Los componentes de un *sharded clusters* son:

***Shards* (Fragmentos)**
: Cada una de las máquinas del cluster, que almacena un subconjunto de los datos de la colección. Cada *shard* es una instancia de `mongod` o un conjunto de réplicas. En un entorno de producción, todos los *shards* son conjuntos de réplica.

Servidores de Configuracion

Cada servidor de configuración es una instancia de `mongod` que almacena metadatos sobre el cluster. Los metadatos mapean los trozos con los *shards*, definiendo qué rangos de datos definen un trozo (*chunk*) de la colección, y qué trozos se encuentran en un determinado *shard*.

En entornos de producción se aconseja tener 3 servidores de configuración ya que si sólo tuviésemos uno, al producirse una caída el cluster quedaría inaccesible.

Enrutadores

Cada router es una instancia `mongos` que enruta las lecturas y escrituras de las aplicaciones a los *shards*. Las aplicaciones no acceden directamente a los *shards*, sino al *router*. Estos enrutadores funcionan de manera similar a una tabla de contenidos, que nos indica donde se encuentran los datos. Una vez recopilados los datos de los diferentes *shards*, se fusionan y se encarga de devolverlos a la aplicación.

En entornos de producción es común tener varios *routers* para balancear la carga de los clientes.

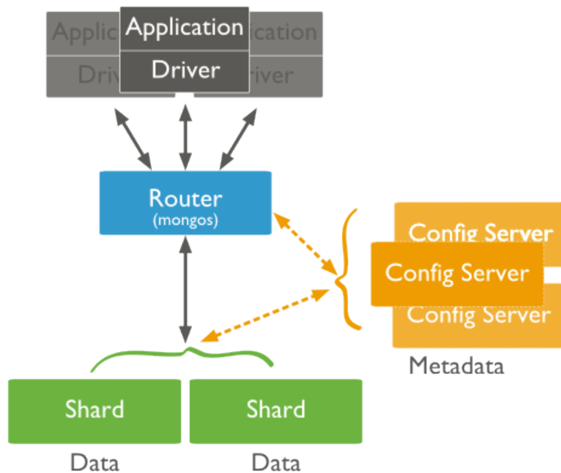


Figura 28. Componentes de un *sharded cluster*

Autoevaluación

Supongamos que queremos ejecutar múltiples routers `mongos` para soportar la redundancia. ¿Qué elemento asegurará la tolerancia a fallos y cambiará de un `mongos` a otro dentro de tu aplicación? [23]

- `mongod`
- `mongos`
- Driver
- Los servidores de configuración de *sharding*

Shard key

Para que *MongoDB* sepa cómo dividir una colección en trozos, hay que elegir una **shard key**, normalmente el identificador del documento, por ejemplo, `student_id`. Este identificador es la clave del *chunk* (por lo hace la misma función que una clave primaria).

[23] El driver se encarga de manera transparente de conectar al router adecuado, y cambiar un router por otro si al que estamos conectado se cae

Para las búsquedas, borrados y actualizaciones, al emplear la *shard key*, *mongos* sabe a que *shard* enviar la petición. En cambio, si la operación no la indica, se hará un *broadcast* a todas los shards para averiguar donde se encuentra.

Por eso, toda inserción debe incluir la *shard key*. En el caso de tratarse de una clave compuesta, la inserción debe contener la clave completa.

Entre los aspectos a tener en cuenta a la hora de elegir una *shard key* cabe destacar que debe:

- Tener una alta cardinalidad, para asegurar que los documentos puedan dividirse en los distintos fragmentos. Por ejemplo, si elegimos un *shard key* que solo tiene 3 valores posibles y tenemos 5 fragmentos, no podríamos separar los documentos en los 5 fragmentos al solo tener 3 valores posibles para separar. Cuantos más valores posibles pueda tener la clave de fragmentación, más eficiente será la división de los trozos entre los fragmentos disponibles.
- Tener un alto nivel de aleatoriedad. Si utilizamos una clave que siga un patrón incremental como una fecha o un ID, conllevará que al insertar documentos, el mismo fragmento estará siendo utilizando constantemente durante el rango de valores definido para él. Esto provoca que los datos estén separados de una manera óptima, pero pondrá siempre bajo estrés a un fragmento en períodos de tiempo mientras que los otros posiblemente queden con muy poca actividad (comportamiento conocido como *hotspotting*).

Una solución a las claves que siguen patrones incrementales es aplicar una funcion *hash* y crear una clave *hasheada* que si tiene un alto nivel de aleatoriedad.

Más consejos sobre como elegir la *shard key* en http://techinsides.blogspot.com.es/2013/09/keynote-concerns-how-to-choose-mongodb.html

Finalmente, destacar que toda *shard key* debe tener un índice asociado.

Preparando el *Sharding* con *MongoDB*

Para comenzar, vamos a crear un particionado en dos instancias en las carpetas `/data/s1/db` y `/data/s2/db`. Los logs los colocaremos en `/data/logs` y crearemos un servidor para la configuración de los metadatos del *shard* en `/data/con1/db`:

```
mkdir -p /data/s1/db /data/s2/db /data/logs /data/conf1/db
chown `id -u` /data/s1/db /data/s2/db /data/logs /data/conf1/db
```

A continuación, arrancaremos un proceso `mongod` por cada uno de los *shards* y un tercero para la base de datos de configuración. Finalmente, también lanzaremos un proceso `mongos`:

Script de creación del Shard - (creaShard.sh[24])

```
mongod --shardsvr --dbpath  /data/s1/db --port 27000 --logpath /data/logs/
sh1.log --smallfiles --oplogSize 128 --fork
mongod --shardsvr --dbpath  /data/s2/db --port 27001 --logpath /data/logs/
sh2.log --smallfiles --oplogSize 128 --fork
mongod --configsvr --dbpath  /data/conf1/db --port 25000 --logpath  /data/
logs/config.log --fork
```

[24] resources/nosql/creaShard.sh

```
mongos --configdb localhost:25000 --logpath  /data/logs/mongos.log --fork
```

El cual lanzaremos mediante

```
bash < creaShard.sh
```

Una vez creado, arrancaremos un shell del `mongo`, y observaremos como se lanza `mongos`:

```
$ mongo
MongoDB shell version: 2.6.7
connecting to: test
mongos>
```

Finalmente, configuraremos el *shard* mediante el método `sh.addShard(URI)`, obteniendo confirmación tras cada cada uno:

```
mongos> sh.addShard("localhost:27000")
{ "shardAdded" : "shard0000", "ok" : 1 }  ❶
mongos> sh.addShard("localhost:27001")
{ "shardAdded" : "shard0001", "ok" : 1 }
```

❶ El valor de la propiedad `shardAdded` nos devuelve el identificado unívoco de cada *shard*.

De manera similar que con el conjunto de réplicas se emplean el prefijo `rs`, para interactuar con los componentes implicados en el *sharding* se emplea `sh`. Por ejemplo, mediante `sh.help()` obtendremos la ayuda de los métodos disponibles.

Así pues, en este momento tenemos montada un *shard* con:

- dos instancias de `mongod` para almacenar datos en los puertos 27000 y 27001 (*shards*)
- una instancia `monogd` en el puerto 25000 (servidor de configuración) encargada de almacenar los metadatos del shard, a la cual sólo se deberían conectar el proceso `mongos` o los drivers para obtener información sobre el *shard* y la *shard key*
- y un proceso `mongos` (*enrutador*), encargado de aceptar las peticiones de los clientes y enrutar las peticiones al *shard* adecuado.

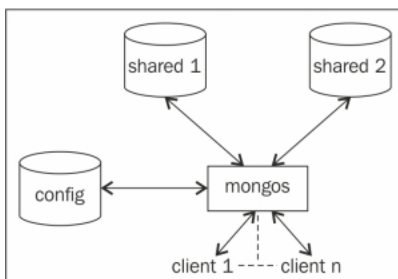


Figura 29. Shard con dos máquinas

Si comprobamos el estado del *shard* podremos comprobar como tenemos dos shards, con sus identificadores y URIs:

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
 "_id" : 1,
 "version" : 4,
 "minCompatibleVersion" : 4,
 "currentVersion" : 5,
 "clusterId" : ObjectId("551d6d8d87642203b426ab00")
}
  shards:
 {  "_id" : "shard0000",  "host" : "localhost:27000" }
 {  "_id" : "shard0001",  "host" : "localhost:27001" }
  databases:
 {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
```

En un entorno de producción, en vez de tener dos *shards*, habrá un conjunto de réplicas para asegurar la alta disponibilidad. Además, tendremos tres servidores de configuración para asegurar la disponibilidad de éstos. Del mismo modo, habrá tantos procesos `mongos` creados para un *shard* como conexiones de clientes.

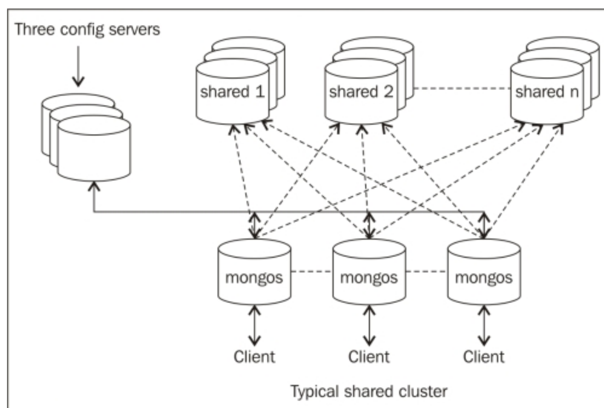


Figura 30. Sharding en un entorno de Producción

En init_sharded_replica.sh[25] podéis comprobar como crear *sharding* sobre un conjunto de réplicas.

Habilitando el *Sharding*

Una vez hemos creado la estructura necesaria para soportar el *sharding* vamos a insertar un conjunto de datos para posteriormente particionarlos.

Para ello, vamos a insertar cien mil usuarios en una colección:

```
mongos> use expertojava
switched to db expertojava
mongos> for (var i=0; i<100000; i++) {
  db.usuarios.insert({"login":"usu" + i,"nombre":"nom" +
 i*2, "fcreacion": new Date()});
}
```

[25] resources/nosql/init_sharded_replica.sh

```
mongos> db.usuarios.count()
100000
```

Como podemos observar, interactuar con `mongos` es igual a hacerlo con `mongo`.

Ahora mismo no sabemos en qué cual de los dos *shards* se han almacenado los datos. Además, estos datos no están particionados, es decir residen en sólo uno de los *shards*.

Para habilitar el *sharding* a nivel de base de datos y que los datos se repartan entre los fragmentos disponibles, ejecutaremos el comando `sh.enableSharding(nombreDB)` :

```
mongos> sh.enableSharding("expertojava")
```

Si volvemos a comprobar el estado del *shard*, tenemos que se ha creado la nueva base de datos que contiene la propiedad `"partitioned" : true`, la cual nos informa que esta fragmentada.

Antes de habilitar el *sharding* para una determinada colección, tenemos que crear un índice sobre la *shard key*:

```
mongos> db.usuarios.ensureIndex({"login": 1})
{
 "raw" : {
  "localhost:27000" : {
   "createdCollectionAutomatically" : false,
   "numIndexesBefore" : 1,
   "numIndexesAfter" : 2,
   "ok" : 1
  }
 },
 "ok" : 1
}
```

Una vez habilitado el *shard* ya podemos fragmentar la colección:

```
mongos> sh.shardCollection("expertojava.usuarios", {"login": 1}, false)
```

El método `shardCollection` particiona una colección a partir de una *shard key*. Para ello, recibe tres parámetros:

1. nombre de la colección, con nomenclatura de `nombreBD.nombreColección`
2. nombre del campo para fragmentar la colección, es decir, el *shard key*. Uno de los requisitos es que esta clave tengo una alta cardinalidad. Si tenemos una propiedad con una cardinalidad baja, podemos hacer un *hash* de la propiedad mediante `{"login": "hashed"}`. Como en nuestro caso hemos utilizado un campo con valores únicos hemos puesto `{"login": 1}`.
3. booleano que indica si el valor utilizado como *shard key* es único. Para ello, el índice que se crea sobre el campo debe ser del tipo `unique`.

Este comando divide la colección en *chunks*, la cual es la unidad que utiliza *MongoDB* para mover los datos. Una vez que se ha ejecutado, *MongoDB* comenzará a balancear la colección

entre los *shards* del *cluster*. Este proceso no es instantáneo. Si la colección contiene un gran conjunto de datos puede llevar horas completar el balanceo.

Si ahora volvemos a comprobar el estado del *shard* obtendremos:

```
mongos> sh.status()
--- Sharding Status ---
sharding version: {
  "_id" : 1,
  "version" : 4,
  "minCompatibleVersion" : 4,
  "currentVersion" : 5,
  "clusterId" : ObjectId("551d6d8d87642203b426ab00")
}
shards:
  {  "_id" : "shard0000",  "host" : "localhost:27000" }
  {  "_id" : "shard0001",  "host" : "localhost:27001" }
databases:
  {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
  {  "_id" : "expertojava",  "partitioned" : true,  "primary"
 : "shard0000" }
  expertojava.usuario
    shard key: { "login" : 1 }
    chunks: ❶
      shard0000 1
    { "login" : { "$minKey" : 1 } } -->> { "login" : { "$maxKey" : 1 } }
 on : shard0000 Timestamp(1, 0) ❷
```

❶ la propiedad `chunks` muestra la cantidad de *trozos* que alberga cada partición. Así, pues en este momento tenemos 1 *chunk*

❷ Para cada uno de los fragmentos se muestra el rango de valores que alberga cada *chunk*, así como en que *shard* se ubica.

Las claves `$minKey` y `$maxKey` son similares a menos infinito y más infinito, es decir, no hay ningún valor por debajo ni por encima de ellos. Es decir, indican los topes de la colección.

Trabajando con el *Sharding*

En este momento, el shard esta creado pero todos los nodos residen en un único fragmento dentro de un partición. Vamos a volver a insertar 100.000 usuarios más a ver que sucede.

```
mongos> for (var i=100000; i<200000; i++) {
  db.usuarios.insert({"login":"usu" + i,"nombre":"nom" +
 i*2, "fcreacion": new Date()});
}
mongos> db.usuarios.count()
200000
```

Si ahora comprobamos el estado del *shard*, los datos se deberían haber repartido entre los *shards* disponibles:

```
mongos> sh.status()
--- Sharding Status ---
```

```
sharding version: {
  "_id" : 1,
  "version" : 4,
  "minCompatibleVersion" : 4,
  "currentVersion" : 5,
  "clusterId" : ObjectId("5548e331555a41e2253350cc")
}
  shards:
    {  "_id" : "shard0000",  "host" : "localhost:27000" }
    {  "_id" : "shard0001",  "host" : "localhost:27001" }
  databases:
    {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
    {  "_id" : "expertojava",  "partitioned" : true,  "primary"
 : "shard0000" }
  expertojava.usuarios
    shard key: { "login" : 1 }
    chunks:
      shard0001 2 ❶
      shard0000 2
    { "login" : { "$minKey" : 1 } } -->> { "login" : "usu0" } on :
 shard0001 Timestamp(3, 0)
    { "login" : "usu0" } -->> { "login" : "usu167405" } on : shard0000
 Timestamp(3, 1)
    { "login" : "usu167405" } -->> { "login" : "usu99999" } on : shard0000
 Timestamp(2, 3)
    { "login" : "usu99999" } -->> { "login" : { "$maxKey" : 1 } } on :
 shard0001 Timestamp(2, 0)
```

❶ Con estos datos se ha forzado a balancear los mismos entre los dos fragmentos, habiendo en cada uno de ellos dos tros, de ahí los 4 *chunks*

Si ahora realizamos una consulta y obtenemos su plan de ejecución veremos como se trata de una consulta que se ejecuta en paralelo:

```
mongos> db.usuarios.find({"login":"usu12345"}).explain()
{
  "clusteredType" : "ParallelSort",
  "shards" : {
    "localhost:27000" : [
    {
      "cursor" : "BtreeCursor login_1",
      "isMultiKey" : false,
      "n" : 1,
      "nscannedObjects" : 1,
      "nscanned" : 1,
      "nscannedObjectsAllPlans" : 1,
      "nscannedAllPlans" : 1,
      "scanAndOrder" : false,
      "indexOnly" : false,
      "nYields" : 0,
      "nChunkSkips" : 0,
      "millis" : 0,
      "indexBounds" : {
        "login" : [
          [
          "usu12345",
          "usu12345"
```

```
          ]
        ]
      },
      "server" : "MacBook-Air-de-Aitor.local:27000",
      "filterSet" : false
    }
    ]
  },
  "cursor" : "BtreeCursor login_1",
  "n" : 1,
  "nChunkSkips" : 0,
  "nYields" : 0,
  "nscanned" : 1,
  "nscannedAllPlans" : 1,
  "nscannedObjects" : 1,
  "nscannedObjectsAllPlans" : 1,
  "millisShardTotal" : 0,
  "millisShardAvg" : 0,
  "numQueries" : 1,
  "numShards" : 1,
  "indexBounds" : {
    "login" : [
      [
        "usu12345",
        "usu12345"
      ]
    ]
  },
  "millis" : 1
}
```

El objeto devuelto se compone de dos partes, con el contenido de un plan de ejecución anidado dentro de otro plan de ejecución. El plan exterior se refiere a la información obtenida por `mongos`, mientras que los internos son los realizados por cualquier *shard* empleado en la consulta, en nuestro caso, `localhost:27000`

Si en vez de obtener un documento concreto, obtenemos el plan de ejecución de obtener todos los documentos tendremos:

```
mongos> db.usuarios.find().explain()
{
  "clusteredType" : "ParallelSort",
  "shards" : {
  "localhost:27000" : [
      {
        "cursor" : "BasicCursor",
        "isMultiKey" : false,
        "n" : 199999,
        "nscannedObjects" : 199999,
        "nscanned" : 199999,
        "nscannedObjectsAllPlans" : 199999,
        "nscannedAllPlans" : 199999,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 1562,
        "nChunkSkips" : 0,
```

```
        "millis" : 299,
        "server" : "MacBook-Air-de-Aitor.local:27000",
        "filterSet" : false
      }
    ],
    "localhost:27001" : [
      {
        "cursor" : "BasicCursor",
        "isMultiKey" : false,
        "n" : 1,
        "nscannedObjects" : 1,
        "nscanned" : 1,
        "nscannedObjectsAllPlans" : 1,
        "nscannedAllPlans" : 1,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 0,
        "server" : "MacBook-Air-de-Aitor.local:27001",
        "filterSet" : false
      }
    ]
  },
  "cursor" : "BasicCursor",
  "n" : 200000,
  "nChunkSkips" : 0,
  "nYields" : 1562,
  "nscanned" : 200000,
  "nscannedAllPlans" : 200000,
  "nscannedObjects" : 200000,
  "nscannedObjectsAllPlans" : 200000,
  "millisShardTotal" : 299,
  "millisShardAvg" : 149,
  "numQueries" : 2,
  "numShards" : 2,
  "millis" : 299
}
```

Así pues, si en una consulta no le enviamos la *shard key* como criterio, `mongos` enviará la consulta a cada *shard*. Si la consulta contiene la *shard key*, la consulta se enruta directamente al *shard* apropiado.

4.7. Ejercicios

(0.75 puntos) Ejercicio 41. Agregaciones

A partir de la colección `cities` de la base de datos `ejercicios`, escribe los comandos necesarios y el resultado en `ej41.txt` para obtener la información de las siguientes consultas mediante el *pipeline* de agregación:

- `Nombre` y `población` de las tres ciudades españolas con más habitantes. Resultado:

```
{ "nombre" : "Madrid", "poblacion" : 3255944 }
{ "nombre" : "Barcelona", "poblacion" : 1621537 }
```

```
{ "nombre" : "Valencia", "poblacion" : 814208 }
```

- `País`, `población` y cantidad de `ciudades` de dicho país, ordenados de mayor a menor población. Resultado:

```
{ "poblacion" : 282839031, "ciudades" : 5568, "pais" : "CN" }
{ "poblacion" : 272149640, "ciudades" : 3350, "pais" : "IN" }
{ "poblacion" : 223341111, "ciudades" : 14566, "pais" : "US" }
...
```

- `País`, `población`, `ratio` entendido como el resultado de dividir la población del país entre el número de ciudades, `ciudadMasPoblada` en mayúsculas y `pobCiudadMasPoblada` (población de la ciudad más poblada) ordenados por el ratio de población/ciudades:

```
{ "ciudadMasPoblada" : "Singapore", "pobCiudadMasPoblada"
 : 3547809, "pais" : "SG", "ratio" : 3547809 }
{ "ciudadMasPoblada" : "Hong Kong", "pobCiudadMasPoblada"
 : 7012738, "pais" : "HK", "ratio" : 2260092.75 }
{ "ciudadMasPoblada" : "Macau", "pobCiudadMasPoblada" : 520400, "pais"
 : "MO", "ratio" : 520400 }
...
```

(0.5 puntos) Ejercicio 42. Agregaciones en Java

Modifica la clase `ConsultasEjercicios` creada en el ejercicio 23, y añade un método con la siguiente definición:

```
List<Pais> listarPaisesMenosPoblados(int limite)
```

Este método obtendrá una lista con los países menos poblados, restringiendo la cantidad de países a los indicados por el `limite`.

Para ello, necesitarás crear una clase `Pais` similar a la siguiente:

```
public class Pais {
  private String name;
  private long population;

  public Pais(String name, long population) {
    this.name = name;
    this.population = population;
  }

  public long getPopulation() {
    return population;
  }
  public void setPopulation(long population) {
    this.population = population;
  }
```

```
    public String getName() {
      return name;
    }
    public void setName(String name) {
      this.name = name;
    }
  }
```

(0.75 puntos) Ejercicio 43. Replicación

Se pide crear una conjunto de 4 réplicas de nombre `ejer43` en la cual insertaremos los datos de las ciudades de la primera sesión.

El script de creación del conjunto de réplicas se almacenará en `ejer43creacion.sh`, y los comandos para inicializar el conjunto en `ejer43init.js`.

Una vez cargado los datos, obtener el estado del conjunto y almacenar el comando y el resultado en `ejer43estado.txt`.

A continuación, crea una clase *Java* llamada `ReplicaEjercicios`, la cual trabaje con el conjunto recién creado y contenga:

- Un método para guardar una ciudad en la réplica, con consistencia en la escritura de la mayoría del conjunto de réplicas:

```
void insertaCiudad(Ciudad ciudad)
```

- Un método para recuperar una ciudad a partir de su nombre permitiendo la lectura de nodos secundarios si el primario ha caído:

```
Ciudad recuperaCiudad(String nombre)
```

(0.5 puntos) Ejercicio 44. Sharding

Se pide crear un *shard* con tres servidores e importar las ciudades en el *shard*.

Para ello, particionar los datos por el nombre de la ciudad.

El script de creación del conjunto de réplicas se almacenará en `ejer44creacion.sh`, y los comandos para inicializar el sharding en `ejer44init.js`.

Una vez cargado los datos, obtener el estado del sharding y almacenar el comando y el resultado en `ejer44estado1.txt`.

Tras ello, vaciar la colección y volver a importar los datos. Una vez importados, obtener el estado del *sharding* y almacenar el comando y el resultado en `ejer44estado2.txt`