

- 1. Características de los sistemas NoSQL
- 2. Ventajas de los sistemas NoSQL
- 3. Sistemas de bases de datos NoSQL
- 4. Base de datos MongoDB. Introducción
 - 4.1 Características de MongoDB
 - 4.2 Estructuras JSON
 - 4.3 Tipos de datos
 - 4.4 Instalación y ejecución
- 5. MongoDB. Operaciones básicas
 - 5.1 Insertar
 - 5.2 Actualizar documentos
 - 5.3 Consultar documentos
 - 5.4 Eliminar documentos
- 6. Consultas en MongoDB
 - 6.1 Uso de proyecciones
 - 6.2 Uso de filtros
 - 6.3 Consultas sobre arrays
- 7. Consultas avanzadas en MongoDB
 - 7.1 Métodos de find
 - 7.2 Operaciones de agregación
- 8. Importar y exportar
 - 8.1 Exportar colecciones
 - 8.2 Importar colecciones
- 9. Relaciones entre documentos
 - 9.1 Documentos embebidos
 - 9.2 Documentos referenciados
 - 9.3 ¿Qué diseño elegir?
- 10. Programación Java sobre MongoDB
 - 10.1 Crear la conexión
 - 10.2 Acceder a una base de datos
 - 10.3 Acceder a una colección
 - 10.4 Crear un documento
 - 10.5 Insertar documentos
 - 10.6 Consultar la colección
 - 10.7 Consultar la colección especificando un filtro
 - 10.8 Contar documentos en una colección
 - 10.9 Actualizar documentos
 - 10.10 Eliminar documentos
 - 10.11 Consultas avanzadas

1. Características de los sistemas NoSQL

Los Sistemas de Bases de Datos NoSQL proponen una estructura de **almacenamiento más versátil** y tratan de solucionar los problemas de **escalabilidad** que se producen en las bases de datos relacionales.

Sus características principales son:

- Los datos no tienen una definición de atributos fija.
- Permiten almacenar estructuras de datos complejas
- Facilitan el intercambio de información entre sistemas mediante XML o JSON
- Priorizan el rendimiento frente a la fiabilidad de las operaciones
- No utilizan lenguaje SQL o similares
- Almacenamiento basado en parejas clave-valor
- Escalabilidad horizontal de servidores mediante una arquitectura distribuida

2. Ventajas de los sistemas NoSQL

- Se ejecutan en máquinas con pocos recursos
- Cuando el volumen de información y de operaciones aumenta, basta con añadir nuevos nodos (nuevas máquinas). Se distribuye el trabajo entre ellos.
- Permiten manejar grandes volúmenes de datos con alto rendimiento
- Minimizan los problemas de cuellos de botella que se pueden producir en otros sistemas.
- Responden muy rápido y eficientemente a la activación y desactivación de los servicios.

3. Sistemas de bases de datos NoSQL

- **Cassandra**: usado por Facebook, Twitter, Deeg
- **Hbase**: Yahoo, Adobe.
- **Redis**: Flickr, Instagram.
- **Neo4j**: Infojobs
- **MongoDB**: LinkedIn, Ebay, The New York Times
- **BaseX**
- **eXist**

4. Base de datos MongoDB. Introducción

4.1 Características de MongoDB



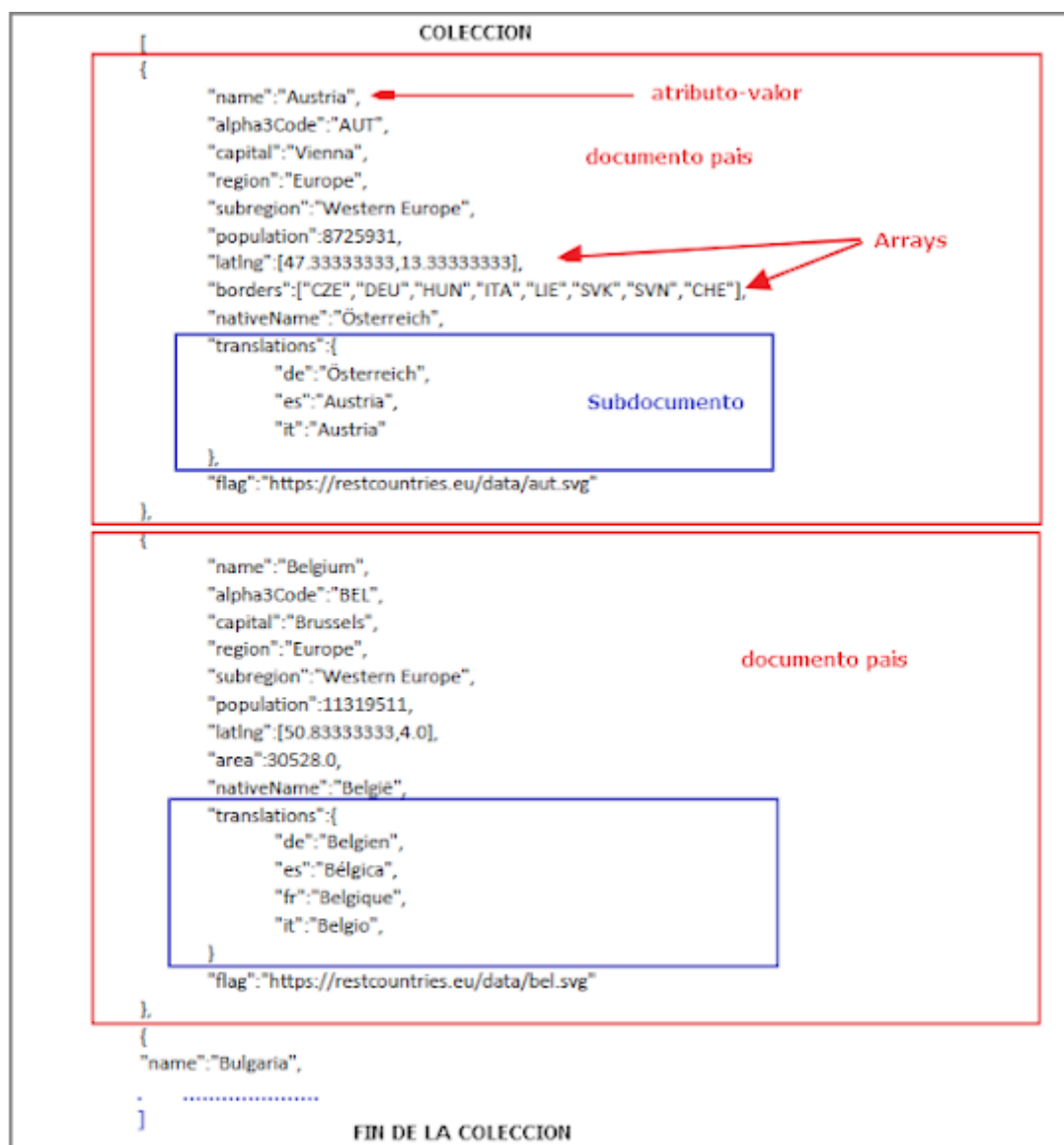
- Base de datos documental (orientada a documentos).
- Permite almacenar cualquier contenido sin obedecer a un esquema o modelo previo.
- Almacena contenidos con formato JSON en una representación interna binaria llamada BSON.

- Opera a una alta velocidad y las consultas son sencillas.

La estructura de almacenamiento de los contenidos en MongoDB es totalmente distinto al de los SGBD Relacionales. Usaremos una terminología de estructuras de almacenamiento nueva pero que tiene una cierta similitud con la terminología usada en SGBD Relacionales:

Modelo Relacional	Modelo en MongoDB
Base de datos	Base de datos
Tabla	Colección
Fila de una tabla	Documento
Columna	Campo o atributo
Índice	Índice
Join (relación de clave ajena)	Referencia o documento embebido

4.2 Estructuras JSON



4.3 Tipos de datos

Simple	Complejos
Integer	Arrays
Double	Object
String	Binary data
Boolean	Object ID
Date	Expresiones regulares
Null	

```
{
  "_id" : ObjectId("610f0bbf08763faa10209344"),           // Object ID
  "dni" : "12345678Z",                                   // String
  "nombre" : "Pepe",                                     // String
  "fechaNacimiento" : ISODate("1995-05-01T00:00:00.000Z"), // Date
  "matriculado" : true,                                   // Boolean
  "clases" : [                                           // Array
    "IF01",
    "IF03"
  ],
  "modulos" : [                                          // Array
    {                                                     // Object
      "nombre" : "Acceso a datos",
      "horas" : 7.0
    },
    {                                                     // Object
      "nombre" : "Programacion de servicios y procesos",
      "horas" : 4.0
    }
  ]
}
```

4.4 Instalación y ejecución

- Si queremos instalar MongoDB en nuestro equipo podemos seguir el manual:
 - Windows:
 - <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/>
 - Linux:
 - <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/>
- Nosotros vamos a utilizar **Docker** en la máquina virtual
- Instalación de Docker en Linux:
 - <https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository>
- Instalación de Docker-compose:
 - <https://docs.docker.com/compose/install/#install-compose>

Para ejecutar el cliente: **mongosh -u root**

5. MongoDB. Operaciones básicas

- Comando para obtener cual es la base de datos actual de trabajo

```
> db
test
```

- Comando para abrir otra base de datos de trabajo (si no existe la crea):

```
> use centroDB
switched to db centroDB
```

- Crear dos documentos Alumno. Cada alumno tiene el nombre, apellidos, nota_media y curso

```
> alumno1={nombre: "Ana Isabel", apellidos: "Camacho Soto", curso: "DAM1",
nota_media:7.5}
{
  "nombre" : "Ana Isabel",
  "apellidos" : "Camacho Soto",
  "curso" : "DAM1",
  "nota_media" : 7.5
}
> alumno2={nombre: "Mario", apellidos: "Pacheco Lombana", curso:"DAM1",
nota_media:6.3}
...
```

- Tenemos actualmente en memoria dos **documentos** llamados alumno1 y alumno2. Pero no están almacenados (si se cierra la conexión se pierden los documentos).
- Almacenar los documentos anteriores en una **colección** llamada **alumnos**

```
> db.alumnos.insertOne(alumno1)
{
  "acknowledged" : true,
  "insertedId" : ObjectId("61150d8f6e8e4f409cebe683")
}
> db.alumnos.insertOne(alumno2)
...
```

- La sintaxis de la instrucción significa: **En la base de datos de trabajo (db), en su colección alumnos llama al método insertOne para añadir o insertar el documento que se pasa como parámetro**

(alumno1 en la primera instrucción).

- Dado que la colección alumnos no existía, se ha **creado** en la primera instrucción.
- La colección ya queda almacenada en disco. Los documentos ahora sí quedan guardados si se cierra la conexión.
- **No se respeta un esquema o modelo de datos.** Por ejemplo, podemos añadir a la colección alumnos un nuevo documento que no tenga los 4 campos que tenían alumno1 y alumno2:

```
> db.alumnos.insertOne({ nombre: "Marta", apellidos: "Arribas Prieto",  
fechaBaja: "2024-01-26"})
```

- Además de poder insertar un documento que tiene atributos distintos a los de los documentos anteriores, vemos que se ha podido añadir el documento a la colección sin crear un documento con un nombre determinado. El documento creado se dice que es **anónimo**.
- Por último, vamos a ver el contenido completo de la **colección alumnos**. Para ello usaremos el **método find()** de la colección:

```
> db.alumnos.find()  
[  
  {  
    _id: ObjectId('65a0fa425a4685ccc0c27311'),  
    nombre: 'Ana Isabel',  
    apellidos: 'Camacho Soto',  
    curso: 'DAM1',  
    nota_media: 7.5  
  },  
  {  
    _id: ObjectId('65a0fa4b5a4685ccc0c27312'),  
    nombre: 'Mario',  
    apellidos: 'Pacheco Lombana',  
    curso: 'DAM1',  
    nota_media: 6.3  
  },  
  {  
    _id: ObjectId('65a0faa35a4685ccc0c27313'),  
    nombre: 'Marta',  
    apellidos: 'Arribas Prieto',  
    fechaBaja: '2024-01-26'  
  }  
]
```

- Vemos que los tres documentos tienen un atributo **_id** (identificador del documento). El valor de ese atributo lo ha asignado el servidor MongoDB automáticamente.

5.1 Insertar

- Ya hemos visto que para insertar un documento en una colección, se usa el método **insertOne** de la colección. Hay otros métodos para insertar:
 - **insertMany**: para insertar varios documentos en la misma instrucción.
 - **insert**: permite insertar uno o varios documentos.
 - **save**: permite insertar un documento o modificarlo si su `_id` ya existe.
- Actualmente **se recomienda no usar** insert y save.

Ejemplo1: insertar en alumnos los datos de la alumna Jennifer Martínez Prieto, de DAM1 y con nota media 4.65.

```
> db.alumnos.insertOne( { nombre: "Jennifer", apellidos: "Martínez Prieto",
curso: "DAM1", nota_media: 4.65 } )
```

Ejemplo2: insertar en alumnos a Javier Díaz Iturbe, de DAW1 y con nota media 6.7 y con `_id` 101.

```
> db.alumnos.insertOne(
{
  _id:101,
  nombre: "Javier",
  apellidos: "Díaz Iturbe",
  curso: "DAW1",
  nota_media:6.7
}
)
```

Ejemplo 3: insertar en alumnos los datos de Raquel San Román Avilés, de DAW1, nota 5.35, id 102; de Ana de la Pinta Ruiz, DAW2, nota 4.75, id 201 y edad 20; de David Valle Pérez , DAW2, nota 5.75, id 202 y módulos cursados DIW, DWES y DWEC.

```
> db.alumnos.insertMany([
  { _id: 102, nombre: "Raquel", apellidos: "San Román Avilés", curso:
"DAW1", nota_media: 5.35 },
  { _id: 201, nombre: "Ana", apellidos: "Pinta de la, Ruiz", curso: "DAW2",
nota_media: 4.75, edad: 20 },
  { _id: 202, nombre: "David", apellidos: "Valle Pérez", curso: "DAW2",
nota_media: 5.75, modulos: ["DIW","DWES","DWEC"] }
])
```

En los parámetros de **insertMany** se envuelven los documentos entre llaves. También vemos como se asigna un **array** de valores al atributo modulos.

5.2 Actualizar documentos

- Vamos a ver dos ejemplos de modificación, el primero usa **updateOne** para modificar un solo documento y el otro usa **updateMany** para modificar varios documentos.

Ejemplo 5: modificar la nota del alumno con id 101 para que sea 6,80.

```
> db.alumnos.updateOne(
  { _id: 101 }, //Filtro
  { $set: { nota_media: 6.8} } //Modificación (SET)
)
```

Ejemplo 6: incrementar en un punto la nota de los alumnos de DAW2.

```
> db.alumnos.updateMany(
  { curso: "DAW2" },
  { $inc: {nota_media: 1} }
)
```

- Operadores de actualización <https://docs.mongodb.com/manual/reference/operator/update/>

Hoja04_MongoDB_01

5.3 Consultar documentos

- Vamos a ver cuatro ejemplos de consultas. Más adelante profundizaremos mucho más en la sintaxis de las consultas.

Ejemplo 7: obtener todos los datos de los alumnos de DAM1.

```
> db.alumnos.find( { curso: "DAM1" } )
```

Ejemplo 8: obtener nombre y apellidos de todos los alumnos

```
> db.alumnos.find( {}, { _id:0, nombre: 1, apellidos: 1} )
```

Ejemplo 9: obtener todos los datos de los alumnos de DAM1, excepto el id.

```
> db.alumnos.find( {curso: "DAM1"}, { _id:0} )
```

Ejemplo 10: obtener todos los datos de los alumnos de DAW1 con nota media superior o igual a 6.

```
> db.alumnos.find( {curso: "DAW1", nota_media : {$gte : 6 }} )
```


5.4 Eliminar documentos

- Vamos a ver dos ejemplos de eliminación, en los dos vamos a usar **deleteMany** para eliminar varios documentos.
- En el primero podríamos usar también **deleteOne**. El método deleteOne elimina sólo el primero de los eliminables.

Ejemplo 11: Eliminar el documento correspondiente a la alumna Marta Arribas Prieto.

```
> db.alumnos.deleteMany( {nombre : "Marta", apellidos: "Arribas Prieto"} )
{ "acknowledged" : true, "deletedCount" : 1 }
```

Ejemplo 12: eliminar los documentos de los alumnos del curso DAM1.

```
> db.alumnos.deleteMany( { curso: "DAM1" } )
{ "acknowledged" : true, "deletedCount" : 3 }
```

6. Consultas en MongoDB

- Para consultar documentos de una colección se usa el método **find**. Fundamentalmente, este método admite tres sintaxis:
 - **find()**: devuelve el contenido de todos los documentos de la colección. Devuelve todos sus campos o atributos.
 - **find({filtro})**: devuelve el contenido de todos los atributos de los documentos de la colección que cumplen la condición del filtro.
 - **find({filtro},{proyección})**: devuelve el contenido de los atributos que se indiquen en la proyección para los documentos que cumplan la condición de filtro.
- En los ejemplos siguientes de códigos de consultas vamos a usar una colección **inventario** que puedes incluir en una base de datos **pruebas**.
- Creamos la colección con 5 documentos:

```
db.inventario.insertMany([
  { elemento: "journal", cantidad: 25, dim: { h: 14, w: 21, um: "cm" },
    estado: "A" },
  { elemento: "notebook", cantidad: 50, dim: { h: 8.5, w: 11, um: "in" },
    estado: "A" },
  { elemento: "paper", cantidad: 100, dim: { h: 8.5, w: 11, um: "in" },
    estado: "D" },
  { elemento: "planner", cantidad: 75, dim: { h: 22.85, w: 30, um: "cm" },
    estado: "C" },
  { elemento: "postcard", cantidad: 45, dim: { h: 10, w: 15.25, um: "cm" }
```

```

}, estado: "A" },
  { elemento: "postcard", cantidad: 45, dim: { h: 10, w: 15.25, um: "cm"
}, estado: "A" },
  { elemento: "album", dim: { h: 12, w: 12, um: "in" }, estado: "B" },
  { elemento: "pencil", cantidad: 100, dim: null, estado: "D" }
]);

```

- Fíjate que en cada documento hay un subdocumento *dim* con atributos *alto(h)*, *ancho (w)* y *unidad de medida (um)*.

6.1 Uso de proyecciones

- Siempre que no se pase un segundo parámetro al método **find**, se devuelven todos los atributos de una colección. Veamos dos ejemplos que no requieren indicar una proyección.

Ejemplo 1: obtener todos los datos de todos los documentos de inventario

```
> db.inventario.find()
```

Ejemplo 2: obtener todos los datos de los documentos de inventario cuyo elemento sea paper. Se usa un filtro de igualdad.

```
> db.inventario.find( { elemento: "paper" } )
```

- La sintaxis para especificar en una proyección los atributos o campos que se van a mostrar es:
 - **{atributo1 : 1, atributo2 : 1,}**
- La sintaxis para especificar en una proyección los atributos o campos que no se van a mostrar es:
 - **{atributo1 : 0, atributo2 : 0,}**
- **En la proyección no se pueden especificar a la vez atributos que se quieren mostrar y atributos que no se quieren mostrar.**
- Con la primera sintaxis se muestra también el atributo **_id** (aunque no esté en la lista de atributos puestos a 1). Para que no se muestre ese atributo hay que indicarlo poniéndole un 0 en la condición de proyección.

Ejemplo 3: obtener los valores de **_id**, **elemento**, **cantidad** de todos los documentos de inventario

```
> db.inventario.find( {}, { elemento: 1, cantidad: 1 } )
```

Ejemplo 4: obtener los valores de **elemento** y **cantidad** de todos los documentos de inventario.

```
> db.inventario.find( {}, { _id: 0, elemento: 1, cantidad: 1 } )
```

Ejemplo 5: obtener los valores de los atributos de todos los datos de los elementos con estado A exceptuando _id y estado.

```
> db.inventario.find( {estado: "A"}, { _id: 0, estado: 0 } )
```

6.2 Uso de filtros

- Un filtro establece las condiciones que deben cumplir los documentos que se van a obtener como resultado.
- En un filtro se pueden tener varias condiciones de selección (selectores) que se podrán relacionar con otros selectores lógicos para realizar operaciones AND, OR y NOT.

Condiciones de igualdad:

- Se establecen con la sintaxis
 - **atributo:valor**
- Que equivale a poner en una instrucción SQL
 - **WHERE atributo=valor**

Ejemplo 6: obtener todos los elementos con estado A.

```
> db.inventario.find( {estado: "A"} )
```

Ejemplo 7: obtener todos los elementos cuya unidad de medida de dimensión sea cm

```
> db.inventario.find( { "dim.um": "cm" })
```

*Dado que um es un atributo de un subdocumento dim, su notación debe ser anidación de objetos **dim.um** y se debe escribir entre comillas.*

Ejemplo 8: obtener todos los elementos con estado A y cuya unidad de medida de dimensión sea cm.

```
> db.inventario.find( { estado: "A", "dim.um": "cm" })
```

*Las condiciones de un filtro separadas por coma implican **casi siempre** una operación AND.*

Selectores de comparación

- En toda consulta que no sea de igualdad, se deben usar en el filtro los operadores de selección o **selectores**. La sintaxis para usar un selector es la siguiente:

- **atributo : {selector : valor }**

- El selector indica la **operación** que se realiza entre el atributo y el valor. Hay selectores que requieren una lista de valores, en cuyo caso, todos esos valores se separan con comas y se encierran entre corchetes.
- Por ejemplo, para obtener los elementos con cantidad superior a 60:

```
db.inventario.find( { cantidad : { $gt : 60 } } );
```

- Y para obtener los elementos con estado A, C o D:

```
db.inventario.find( { estado : { $in : ["A", "C", "D"] } } );
```

Selectores de comparación

Selector	Comparación realizada
\$gt	Si valor de atributo es mayor que un valor
\$gte	Si valor de atributo es mayor o igual que un valor
\$lt	Si valor de atributo es menor que un valor
\$lte	Si valor de atributo es menor o igual que un valor
\$in	Si valor de atributo está incluido en lista de valores
\$nin	Si valor de atributo no está incluido en lista de valores
\$ne	Si valor de atributo distinto de valor

Ejemplo 9: obtener todos los elementos con estado A y cuya cantidad sea inferior a 50.

```
> db.inventario.find( { estado: "A", cantidad: { $lt: 50 } })
```

Ejemplo 10: obtener todos los elementos con estado A, B o C y con ancho en cm superior o igual a 21.

```
> db.inventario.find( { estado: { $in: ["A", "B", "C"] }, "dim.um": "cm",  
  "dim.w": { $gte: 21 } })
```

Ejemplo 11: obtener todos los elementos cuyo nombre del elemento es mayor que "p".

```
> db.inventario.find( { elemento: { $gt: "p" } })
```

Ejemplo 12: obtener todos los elementos con cantidad comprendida entre 40 y 60 unidades.

```
> db.inventario.find( { cantidad: { $gt: 40 }, cantidad: { $lt: 60 } } )
```

Verás que esta solución **no es válida** (en casos en que se usa el mismo atributo en dos comparaciones, solo se hace la última)

Veremos después la solución correcta usando `$and`.

Selectores lógicos

Selector	Operación realizada
<code>\$and</code>	Realiza la operación AND entre una lista de expresiones
<code>\$or</code>	Realiza la operación OR entre una lista de expresiones
<code>\$not</code>	Realiza la operación NOT sobre la expresión que se indique

- Sintaxis:
 - `$and : [expr1, expr2, ...]`
 - `$or : [expr1, expr2, ...]`
 - `$not : {expr1}`

Ejemplo 13: obtener todos los elementos con cantidad comprendida entre 40 y 60 unidades.

```
> db.inventario.find({ $and : [ {cantidad : { $gte: 40 }}, {cantidad : { $lte: 60 } } ] })
```

Ejemplo 14: obtener todos los elementos con cantidad no comprendida entre 40 y 60 unidades.

```
> db.inventario.find({ $or : [ {cantidad : { $lt: 40 }}, {cantidad : { $gt: 60 } } ] })
```

Ejemplo 15: obtener todos los elementos con estado A o D y cantidad superior a 70 unidades.

```
> db.inventario.find({ $and : [ { $or : [{ estado: 'A' }, { estado: 'D' } ] }, {cantidad : { $gt: 70 } } ] })
```

```
> db.inventario.find({ $and : [ {estado: { $in: ['A', 'D'] }}, {cantidad : { $gt: 70 } } ] })
```

Ejemplo 16: obtener todos los elementos con cantidad no superior a 60 unidades.

```
> db.inventario.find({cantidad: {$not : {$gt: 60}}})
```

También se puede plantear, mucho más fácil y mejor, obteniendo los elementos con cantidad menor o igual a 60 unidades.

```
> db.inventario.find({cantidad: {$lte: 60}})
```

Pero hay una diferencia: En la primera solución se muestran elementos que tienen cantidad a valor null o que no tienen el atributo cantidad. En la segunda solución se muestran los que tienen atributo cantidad y, de ellos, los que en cantidad tienen un valor y es inferior o igual que 60.

Selectores de elemento

Selector	Operación realizada
\$exists	Comprueba si cada documento contiene o no el atributo indicado
\$type	Comprueba si cada documento contiene un atributo de un determinado tipo

Ejemplo 17: obtener todos los elementos que no tienen atributo cantidad.

```
> db.inventario.find( {cantidad: {$exists: false}} )
```

Ejemplo 18: obtener todos los elementos que tienen atributo cantidad.

```
> db.inventario.find( {cantidad: {$exists: true}} )
```

Ejemplo 19: obtener todos los elementos que tienen atributo dim con valores de tipo object (subdocumento).

```
> db.inventario.find( {dim: {$type : "object"}} )
```

Selectores de evaluación

Selector	Operación realizada
\$expr	Permite usar expresiones de agregación para seleccionar documentos
\$regex	Obtiene documentos cuyos valores cumplen con una expression regular o patrón
\$text	Obtiene documentos filtrados por una búsqueda de texto

Ejemplo 20: obtener todos los elementos cuyo nombre comienza por p.

```
> db.inventario.find( {elemento: {$regex : "^p.*"}} )
```

Ejemplo 21: obtener todos los elementos cuyo nombre comienza por pa o por Pa

```
> db.inventario.find( {$or: [{elemento: {$regex : "^pa.*"}}, {elemento: {$regex: "^Pa*"}} ] } )
```

Ejemplo 22: obtener todos los elementos cuyo nombre termina en r.

```
> db.inventario.find( {elemento: {$regex : ".*r$"}} )
```

Ejemplo 23: obtener todos los elementos cuyo nombre contiene el texto ANN sin diferenciar mayúsculas y minúsculas.

```
> db.inventario.find( {elemento: {$regex : "ANN", $options: "i"}} )
```

6.3 Consultas sobre arrays

Selectores de arrays

Selector	Operación realizada
\$all	Selecciona documentos que tienen un array que contiene todos los elementos que se indiquen en la consulta
\$elemMatch	Selecciona documentos que tienen un array en el que alguno de sus elementos cumple con los criterios que se establezcan.
\$size	Selecciona documentos que tienen un array con el tamaño indicado en la consulta

Para los ejemplos que vamos a usar, crea la siguiente colección **inventario2**.

```
db.inventario2.insertMany([
  { elemento: "journal", cantidad: 25, colores: ["black", "red"], dim_cm:
[ 14, 21 ] },
  { elemento: "notebook", cantidad: 50, colores: ["red", "black"], dim_cm:
[ 14, 21 ] },
  { elemento: "paper", cantidad: 100, colores: ["red", "black", "green"],
dim_cm: [ 14, 21 ] },
  { elemento: "planner", cantidad: 75, colores: ["black", "red", "blue"],
dim_cm: [ 22.85, 30 ] },
  { elemento: "postcard", cantidad: 45, colores: ["blue"], dim_cm: [ 10,
15.25 ] }
]);
```

Ejemplo 24: obtener los elementos que incluyen el color red.

```
> db.inventario2.find( { colores: "red" } )
```

Ejemplo 25: obtener los elementos que tengan sólo los colores red y black y en este orden.

```
> db.inventario2.find( { colores: ["red", "black"]} )
```

Ejemplo 26: obtener los elementos que tengan los colores black y red.

```
> db.inventario2.find( { colores: { $all: ["red", "black"] } } )
```

Ejemplo 27: obtener los elementos que tengan 3 colores.

```
> db.inventario2.find( { colores: { $size: 3 } } )
```

Ejemplo 28: obtener los elementos que tienen alguna de sus dimensiones mayores que 22 y menores que 30.

```
> db.inventario2.find( { dim_cm : { $elemMatch: { $gt: 22, $lt: 30 } } } )
```

Ejemplo 29: obtener los elementos que tienen alguna de sus dimensiones mayores que 20 .

```
> db.inventario2.find( { dim_cm : { $elemMatch: { $gt: 20 } } } )
```

Hoja04_MongoDB_02

7. Consultas avanzadas en MongoDB

7.1 Métodos de find

- A los resultados obtenidos con **find** se le pueden aplicar varios métodos modificadores. Puedes verlos ejecutando **db.inventario.find().help()**

Método	Funcionamiento
--------	----------------

Método	Funcionamiento
sort({campo: 1 -1})	Ordenar los resultados por un campo
limit(n)	Mostrar sólo n documentos (los primeros)
skip(n)	Mostrar desde el documento n
count()	Total de documentos que devuelve la consulta
size()	Total de documentos que muestra la consulta

Ejemplo 1: obtener los nombres y cantidades de los elementos de inventario ordenando por cantidad de mayor a menor.

```
> db.inventario.find({}, {_id: 0, elemento: 1, cantidad: 1}).sort({cantidad: -1})
```

Ejemplo 2: obtener el nombre del elemento que mayor cantidad tiene en inventario

```
> db.inventario.find({}, {_id: 0, elemento: 1, cantidad: 1}).sort({cantidad: -1}).limit(1)
```

Ejemplo 3: obtener el nombre y cantidad disponible de los 3 elementos con menores cantidades en inventario.

```
> db.inventario.find({cantidad: {$exists: true}}, {_id: 0, elemento: 1, cantidad: 1}).sort({cantidad:1}).limit(3)
```

Ejemplo 4: obtener el nombre y cantidad del elemento que ocupa la cuarta posición por cantidad disponible en inventario.

```
> db.inventario.find({cantidad: {$exists: true}}, {_id: 0, elemento: 1, cantidad: 1}).sort({cantidad:-1}).skip(3).limit(1)
```

Ejemplo 5: obtener cuantos elementos con cantidad disponible superior a 40 hay cargados en inventario.

```
> db.inventario.find({cantidad: {$gt: 40}}).count()
```

Ejemplo 6: obtener los elementos y cantidades del inventario ordenados por cantidad ascendentemente y de forma que no salgan los dos primeros.

```
> db.inventario.find({}, {_id: 0, elemento: 1, cantidad: 1}).sort({cantidad:1}).skip(2)
```

Ejemplo 7: obtener cuantos documentos genera la consulta anterior.

```
> db.inventario.find({}, {_id: 0, elemento: 1, cantidad: 1}).sort({cantidad:1}).skip(2).count()
```

Ejemplo 8: obtener cuantos elementos se han mostrado en la consulta del ejemplo 6.

```
> db.inventario.find({}, {_id: 0, elemento: 1, cantidad: 1}).sort({cantidad:1}).skip(2).size()
```

7.2 Operaciones de agregación

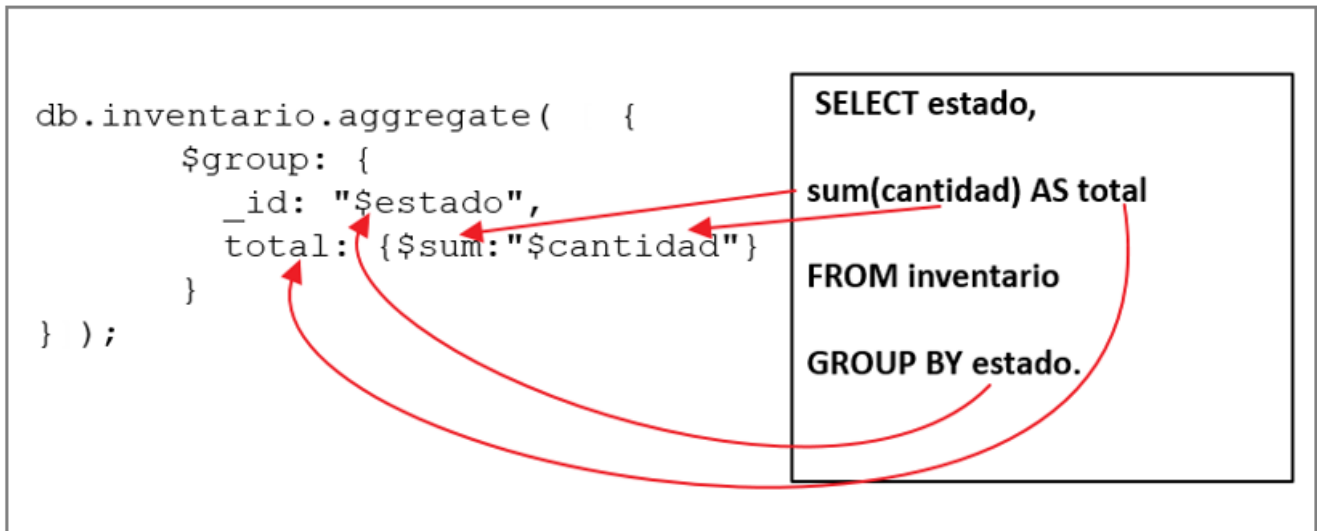
- MongoDB dispone de múltiples funciones de agregado. Se trata de funciones que realizan **operaciones de agregación**, que procesan documentos y obtienen nuevos resultados, calculados o transformados.
- Hay tres formas de realizar operaciones de agregación:
 - Agregación **pipeline**
 - Agregación **map-reduce**
 - Agregación de **propósito único**
- Únicamente vamos a ver la agregación **pipeline**. De todo lo que se puede hacer con esta agregación, sólo vamos a ver las agrupaciones de documentos que comparten el valor de uno o varios atributos.

Funciones de agrupamiento

Método	Funcionamiento
\$sum	Devuelve la suma de un grupo de valores numéricos ignorando los que no sean numéricos en el grupo.
\$avg	Devuelve la media de un grupo de valores numéricos ignorando los que no sean numéricos.
\$first	Devuelve el primer valor de un grupo
\$last	Devuelve el último valor de un grupo
\$max	Devuelve el valor máximo de un grupo o de un array
\$min	Devuelve el valor mínimo de un grupo o de un array

- Para realizar consultas con operaciones de agregación se usa el método **aggregate** de una colección y no el método find. Vamos a ver con un ejemplo como se realiza.

- Queremos obtener la cantidad total de elementos de inventario que están en cada uno de los estados (A, B, C,..).
- Esto en SQL sería algo como:
 - `SELECT estado, sum(cantidad) FROM inventario GROUP BY estado.`



- **Las referencias a atributos dentro de aggregate deben estar entrecomilladas y con el nombre del atributo precedido de \$.**

Ejemplo 9: obtener los cursos que hay en la colección alumnos de la base de datos **centroDB**.

```

> db.alumnos.aggregate ( {
  $group: {
    _id: "$curso"
  }
})

```

Ejemplo 10: obtener cuantos alumnos de la colección alumnos hay matriculados en cada curso.

```

> db.alumnos.aggregate ( {
  $group: {
    _id: "$curso",
    numero_alumnos: {$sum: 1}
  }
})

```

Ejemplo 11: obtener cuantos alumnos de la colección alumnos hay matriculados y la nota media en cada curso.

```

> db.alumnos.aggregate ( {
  $group: {
    _id: "$curso",
    numero_alumnos: {$sum: 1},

```

```
    media: {$avg: "$nota_media"}
  }
})
```

Ejemplo 12: obtener el primer y último alumno matriculado en cada curso.

```
> db.alumnos.aggregate ( {
  $group: {
    _id: "$curso",
    primero: {$first: "$apellidos"},
    ultimo: {$last: "$apellidos"}
  }
})
```

```
> db.alumnos.aggregate ( {
  $group: {
    _id: "$curso",
    primero: {$first: {$concat: ["$apellidos", " ", " ", "$nombre"]}},
    ultimo: {$last: {$concat: ["$apellidos", " ", " ", "$nombre"]}}
  }
})
```

Ejemplo 13: obtener la máxima y mínima nota en cada curso.

```
> db.alumnos.aggregate ( {
  $group: {
    _id: "$curso",
    maxima_nota: {$max: "$nota_media"},
    minima_nota: {$min: "$nota_media"}
  }
})
```

Ejemplo 14: obtener el último alumno por orden alfabético de cada curso.

```
> db.alumnos.aggregate ( {
  $group: {
    _id: "$curso",
    ultimo: {$max: "$apellidos"},
  }
})
```

8. Importar y exportar

8.1 Exportar colecciones

Exportar el contenido de la colección ccaa de la base de datos geografía a un fichero JSON llamado ccaa_completo.json en la ruta actual de trabajo.

```
mongoexport -u usuario --db geografia --collection ccaa --out  
ccaas_completo.json
```

Exportar el contenido de la colección ccaa de la base de datos geografía a un fichero JSON pretty llamado ccaa_completo.json en la ruta actual de trabajo.

```
mongoexport -u usuario --db geografia --collection ccaa --out  
ccaas_completo.json --pretty
```

Exportar el contenido de la colección ccaa de la base de datos geografía a un fichero JSON llamado ccaa_completo.json en la ruta actual de trabajo como un array de documentos

```
mongoexport -u usuario --db geografia --collection ccaa --out  
ccaas_completo.json --jsonArray
```

Exportar el contenido de la colección ccaa de la base de datos geografía a un fichero JSON pretty llamado ccaa_completo.json en la ruta actual de trabajo como un array de documentos

```
mongoexport -u usuario --db geografia --collection ccaa --out  
ccaas_completo.json --jsonArray --pretty
```

8.2 Importar colecciones

Importar el contenido de un fichero JSON llamado ccaa_completo.json en la ruta actual de trabajo a una colección ccaa de la base de datos geografía. El fichero NO está estructurado como un array de documentos. Es una lista de documentos que simplemente está separados por espacios o por espacios y ENTER.

```
mongoimport -u usuario --db geografia --collection ccaa --file  
ccaas_completo.json
```

Importar el contenido de un fichero JSON llamado ccaa_completo.json en la ruta actual de trabajo a una colección ccaa de la base de datos geografía. El fichero contiene un array de documentos

```
mongoimport -u usuario --db geografia --collection ccaa --file
ccaa_completo.json --jsonArray
```

9. Relaciones entre documentos

- Hay dos modos de referenciar documentos en MongoDB:
 - Documentos **embebidos**
 - Documentos **referenciados**

9.1 Documentos embebidos

- Hace referencia a guardar uno o varios documentos JSON dentro de otro
- Se guarda como valor de una de sus propiedades
- Por ejemplo, tenemos un documento patron y dos documentos address:

```
// patron document
{
  _id: "joe",
  name: "Joe Bookreader"
}
// address documents
{
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}
{
  street: "1 Some Other Street",
  city: "Boston",
  state: "MA",
  zip: "12345"
}
```

- Podríamos usar el siguiente documento embebido:

```
{
  "_id": "joe",
  "name": "Joe Bookreader",
  "addresses": [
    {
      "street": "123 Fake Street",
      "city": "Faketon",
      "state": "MA",
```

```

        "zip": "12345"
      },
      {
        "street": "1 Some Other Street",
        "city": "Boston",
        "state": "MA",
        "zip": "12345"
      }
    ]
  }
}

```

9.2 Documentos referenciados

- En contraposición de los documentos embebidos, en un documento JSON **se guarda el valor de una única propiedad**, en lugar del documento completo.
- Normalmente, se guarda el valor de una propiedad que **identifique de manera unívoca al documento referenciado** (por ejemplo su `_id`)
- En el ejemplo anterior tendríamos algo así:

```

// patron document
{
  _id: "joe",
  name: "Joe Bookreader"
}
// address documents
{
  patron_id: "joe", // referencia al documento patron
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}
{
  patron_id: "joe",
  street: "1 Some Other Street",
  city: "Boston",
  state: "MA",
  zip: "12345"
}

```

- También podríamos haber hecho:

```

// patron document
{
  _id: "joe",
  name: "Joe Bookreader",
  addresses: [100, 200]
}

```

```
}  
// address documents  
{  
  _id: 100,  
  street: "123 Fake Street",  
  city: "Faketon",  
  state: "MA",  
  zip: "12345"  
}  
{  
  _id: 200,  
  street: "1 Some Other Street",  
  city: "Boston",  
  state: "MA",  
  zip: "12345"  
}
```

9.3 ¿Qué diseño elegir?

Documentos embebidos

- **Ventajas**
 - Al recuperar un documento traemos toda la información relacionada en otras colecciones
- **Inconvenientes**
 - Mayores tiempos al realizar consultas
 - Actualizaciones más costosas si hay que modificar algún documento embebido (puede aparecer en varios sitios)

Documentos referenciados

- **Ventajas**
 - Consultas más rápidas
 - Actualizaciones más rápidas
- **Inconvenientes**
 - Consultas más complicadas. Habrá que consultar los identificadores que relacionan los documentos en otras colecciones y hacer consultas adicionales
- Así pues, si **no vamos a realizar muchas actualizaciones y no tenemos muchos usuarios**, quizá nos interese usar documentos **embebidos**.
- Si tenemos **muchos usuarios** y queremos realizar **muchísimas consultas**, quizás sea mejor documentos **referenciados**, ya que es más rápido.

10. Programación Java sobre MongoDB

- Para conectarnos a una base de datos MongoDB desde Java tenemos que utilizar el controlador adecuado.

- Ya que estamos utilizando **Maven** en nuestros proyectos, añadiremos lo siguiente al fichero **pom.xml**:

```
<dependencies>
  <dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-sync</artifactId>
    <version>4.11.1</version>
  </dependency>
</dependencies>
```

10.1 Crear la conexión

- Desde la versión 3.7 se recomienda utilizar el método **create** de la clase **MongoClients**
- Aunque el método está sobrecargado podemos utilizar el siguiente:

```
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoClient;

MongoClient cliente =
MongoClients.create("mongodb://usuario:password@nombreHost:27017");
```

- Nos devolverá una instancia de **MongoClient**. Representa un grupo de conexiones a la base de datos

10.2 Acceder a una base de datos

- Una vez que tengamos una instancia MongoClient conectada a una implementación de MongoDB, usaremos el método **getDatabase()** especificando el nombre de la base de datos.
- Si no existe, MongoDB crea la base de datos la primera vez que almacena datos para esa base de datos.
- El siguiente ejemplo accede a la base de datos mydb:

```
MongoDatabase baseDatos = cliente.getDatabase("mydb");
```

10.3 Acceder a una colección

- Una vez que tengamos una instancia MongoDatabase, usaremos el método **getCollection** para acceder a una colección especificando su nombre.
- Si una colección no existe, MongoDB crea la colección cuando almacena datos para esa colección por primera vez.
- Por ejemplo, accedemos a la colección llamada **prueba** en la base de datos mydb:

```
MongoCollection<Document> coleccion = database.getCollection("prueba");
```

10.4 Crear un documento

- Para crear un documento usaremos la clase **Document** del paquete **org.bson**
- Por ejemplo, para crear el siguiente elemento JSON:

```
{
  "name" : "MongoDB",
  "type" : "database",
  "count" : 1,
  "versions": [ "v7.0", "v6.0", "v5.0", "v4.4", "v4.2", "v4.0", "v3.6",
    "v3.4", "v3.2", "v3.0" ],
  "info" : { x : 203, y : 102 }
}
```

- Crearemos un objeto Document con un campo y un valor y usaremos el método **append** para ir añadiendo campos y valores adicionales al objeto.
- El valor puede ser otro objeto Document para especificar un documento incrustado:

```
Document doc = new Document("name", "MongoDB")
    .append("type", "database")
    .append("count", 1)
    .append("versions", Arrays.asList("v7.0", "v6.0", "v5.0",
    "v4.4", "v4.2", "v4.0", "v3.6", "v3.4", "v3.2", "v3.0"))
    .append("info", new Document("x", 203).append("y", 102));
```

10.5 Insertar documentos

Insertar un documento

- Para insertar un solo documento en la colección, puede utilizar el método **insertOne()** de la colección

```
coleccion.insertOne(doc);
```

Insertar varios documentos

- Para insertar varios documentos, se usa el método **insertMany()** de la colección que recibe una lista de documentos.
- El siguiente ejemplo creará documentos con un bucle y se añaden a una lista de objetos Document:

```
List<Document> documentos = new ArrayList<Document>();
for (int i = 0; i < 100; i++) {
    documentos.add(new Document("i", i));
}
```

- Para insertar estos documentos en la colección, el método `insertMany()` recibe la colección:

```
coleccion.insertMany(documentos);
```

10.6 Consultar la colección

- Para consultar la colección, se utiliza el método **find()** de la colección.
- Se puede llamar al método sin argumentos para consultar todos los documentos de una colección o pasar un **filtro** para consultar documentos que coincidan con los criterios del filtro.
- El método `find()` devuelve una instancia de **FindIterable()** que proporciona una interfaz para encadenar otros métodos.

Buscar el primer documento de la colección

- Para devolver el primer documento de la colección, se llama al método `find()` sin ningún parámetro y después al método **first()**
- Si la **colección está vacía**, la operación devuelve un valor **nulo**.
- El siguiente ejemplo imprime el primer documento encontrado en la colección:

```
Document documento = coleccion.find().first();
System.out.println(documento.toJson());
```

- Se imprimirá el siguiente documento:

```
{ "_id" : { "$oid" : "551582c558c7b4fbacf16735" },
  "name" : "MongoDB",
  "type" : "database",
  "count" : 1,
  "info" : { "x" : 203, "y" : 102 } }
```

- Una vez, que tenemos un `Document`, para obtener el valor de un atributo `count` (entero), podemos usar:

```
int count=documento.getInteger("count", -1);
// si no hay atributo carga -1
```

- O también:

```
int count=(Integer)documento.get("count");
```

- Para obtener el valor del atributo name:

```
String nom=(String)documento.get("name");
```

- O bien:

```
String nom= documento.getString("name");
```

- Para obtener el contenido del subdocumento info:

```
Document docInfo=(Document)documento.get("info");
```

Buscar todos los documentos de una colección

- Para recuperar todos los documentos de la colección, usaremos el método find() sin ningún parámetro.
- Para iterar a través de los resultados, encadenaremos el método **iterator()** al find().
- Recuperar todos los documentos de la colección e imprimirlos:

```
MongoCursor<Document> cursor = coleccion.find().iterator();
try {
    while (cursor.hasNext()) {
        System.out.println(cursor.next().toJson());
    }
} finally {
    cursor.close();
}
```

- Aunque está permitido, deberemos **evitar** el for de cada elemento. Puede no funcionar correctamente si se modifica la colección

```
for (Document doc : coleccion.find()) {
    System.out.println(doc.toJson());
}
```

10.7 Consultar la colección especificando un filtro

- Para consultar documentos que cumplan determinadas condiciones se usará el método **find()** **pasándole un objeto de filtro**.
- Para facilitar la creación de objetos filtro se usa la clase **Filters** del paquete `com.mongodb.client.model`
- Los objetos Filter se pueden crear con métodos de filtro. Por ejemplo, podemos usar:

Método de filtro	Descripción
<code>eq(atributo,valor)</code>	Compara si el valor del atributo es igual a valor
<code>ne(atributo,valor)</code>	Compara si el valor del atributo no es igual a valor
<code>gt(atributo,valor)</code>	Compara si el valor del atributo es mayor que valor
<code>and(cond1, cond2, ...)</code>	Operación AND entre todas las condiciones de filtro indicadas
<code>or(cond1, cond2, ...)</code>	Operación OR entre todas las condiciones de filtro indicadas
<code>not(cond)</code>	Operación NOT condición

Obtener un solo documento que coincida con un filtro

- Por ejemplo, para encontrar el primer documento donde el campo `i` tiene el valor 71, pasamos un objeto de filtro **eq** para especificar la condición de igualdad:

```
documento = coleccion.find(eq("i", 71)).first();
System.out.println(documento.toJson());
```

Obtener todos los documentos que coincidan con un filtro

- El siguiente ejemplo devuelve e imprime todos los documentos donde `"i" > 50`:

```
coleccion.find(gt("i", 50))
    .forEach(doc -> System.out.println(doc.toJson()));
```

Para especificar un filtro para un rango, como $50 < i \leq 100$, puede usar el **and**:

```
coleccion.find(and(gt("i", 50), lte("i", 100)))
    .forEach(doc -> System.out.println(doc.toJson()));
```

10.8 Contar documentos en una colección

- Para contar el número de documentos de una colección, se puede utilizar el método **countDocuments()** de la colección.
- Debería imprimirse 101 (100 insertados mediante insertMany más 1 insertado mediante el insertOne).

```
System.out.println(coleccion.countDocuments());
```

10.9 Actualizar documentos

- Para actualizar los documentos de una colección se pueden utilizar los métodos **updateOne** y **updateMany**.
- Los métodos recibirán:
 - Un **filtro** para determinar el documento o documentos que se actualizarán. Para especificar un filtro vacío (y actualizar todos los documentos de una colección) se utilizará un Document vacío.
 - Un **documento de actualización** que especifica las modificaciones.
- Para representar la operación de actualización, se usan métodos de modificación equivalentes a los ya vistos (set, unset, inc, addToSet, ...).
- Los métodos de actualización devuelven un **UpdateResult** que proporciona información sobre la operación, incluido el número de documentos modificados por la actualización.
- Un objeto de este tipo, dispone de los métodos **getMatchedCount()** y **getModifiedCount()** que devuelven cuántos documentos cumplían la condición de filtro de update y cuántos fueron realmente modificados.

Actualizar un sólo documento

- Para actualizar como máximo un sólo documento se utilizar **updateOne**
- El siguiente ejemplo actualiza el primer documento que cumple el filtro i es igual a 10 y establece el valor de i en 110:

```
coleccion.updateOne(eq("i", 10), set("i", 110));
```

Actualizar varios documentos

- Para actualizar todos los documentos que coinciden con el filtro usamos el método **updateMany**.
- El siguiente ejemplo incrementa el valor de i en 100 para todos los documentos donde i es menor que 100:

```
UpdateResult updateResult = coleccion.updateMany(lt("i", 100), inc("i", 100));
System.out.println(updateResult.getModifiedCount());
```

10.10 Eliminar documentos

- Para eliminar documentos de una colección se puede utilizar los métodos **deleteOne** y **deleteMany**.
- Debemos pasar un **filtro** para determinar el documento o documentos a eliminar.
- También podemos especificar un filtro vacío (para borrar todos los documentos), utilizando un objeto **Document** vacío.
- Los métodos de eliminación devuelven un **DeleteResult** que proporciona información sobre la operación, incluido el número de documentos eliminados.

Eliminar un sólo documento que coincide con un filtro

- Para eliminar como máximo un solo documento que coincida con el filtro, usaremos el método **deleteOne**:
- El siguiente ejemplo elimina como máximo un documento que cumple con el filtro i igual 110:

```
coleccion.deleteOne(eq("i", 110));
```

Eliminar todos los documentos que coinciden con un filtro

- Para eliminar todos los documentos que coincidan con el filtro utilizamos el método **deleteMany**.
- El siguiente ejemplo elimina todos los documentos donde i es mayor o igual a 100:

```
DeleteResult resultado = coleccion.deleteMany(gte("i", 100));  
System.out.println(resultado.getDeletedCount());
```

Hoja04_MongoDB_05

10.11 Consultas avanzadas

- Como ya hemos visto, las consultas se realizan con el método **find** de Collection
- Pero también se puede usar el método **aggregate** si se trata de consultas de agregación.
- Tanto el método find como el método aggregate devuelven un objeto FindIterable.
- Al objeto FindIterable se le pueden aplicar unos métodos. Los principales son:

Método	Descripción
first()	Devuelve el primer Document del resultado
iterator()	Devuelve un iterador (MongoCursor) de Document con todo el resultado que podemos recorrer con hasNext y next.
into(ArrayList a)	Carga los resultados en un ArrayList
limit(int n)	Del resultado generado, limita a n documents

Método	Descripción
projection(Bson bson)	Del resultado generado selecciona los atributos indicados en bson
forEach(action)	Para cada document del resultado realiza una acción
sort(criterio)	Ordena los documents del resultado

first()

```
Document doc=coleccion.find(eq("i","200")).first();
if(doc==null)
    System.out.println("No existe el documento");
else {
    System.out.println("Documento : "+(Integer)doc.get("i"));
}
```

iterator()

```
MongoCursor<Document> cursor=coleccion.find().iterator();
while( cursor.hasNext()){
    Document doc=cursor.next();
    System.out.println("Documento " +(Integer)doc.get("i"));
}
```

into(T Clase)

```
ArrayList<Document> lista = coleccion.find().into(new ArrayList<Document>());
```

limit(num)

```
MongoCursor<Document> cursor = coleccion.find().limit(3).iterator();
```

projection(include(atrib1, atrib2, ...))

```
MongoCursor<Document> cursor =
coleccion.find().projection(include("nombre", "edad")).iterator();
```

projection(exclude(atrib1, atrib2, ...))


```
MongoCursor<Document> cursor =  
coleccion.find().projection(exclude("_id")).iterator();
```

projection(fields(include(atrib1, ...),exclude(atrib2, ...)))

```
MongoCursor<Document> cursor = coleccion.find().projection(  
    fields(include("nombre", "edad"), excludeId())).iterator();
```

sort(ascending(atributos))

```
MongoCursor<Document> cursor =  
coleccion.find().sort(ascending("nombre")).iterator();
```

sort(descending(atributos))

```
MongoCursor<Document> cursor = coleccion.find().sort(descending("nombre",  
"edad")).iterator();
```

Hoja04_MongoDB_06

Hoja04_MongoDB_07